Compiler-Based Code-Improvement Techniques

KEITH D. COOPER, KATHRYN S. MCKINLEY, and LINDA TORCZON

Since the earliest days of compilation, code quality has been recognized as an important problem [18]. A rich literature has developed around the issue of improving code quality. This paper surveys one part of that literature: code transformations intended to improve the running time of programs on uniprocessor machines.

This paper emphasizes transformations intended to improve code quality rather than analysis methods. We describe analytical techniques and specific data-flow problems to the extent that they are necessary to understand the transformations. Other papers provide excellent summaries of the various sub-fields of program analysis.

The paper is structured around a simple taxonomy that classifies transformations based on how they change the code. The taxonomy is populated with example transformations drawn from the literature. Each transformation is described at a depth that facilitates broad understanding; detailed references are provided for deeper study of individual transformations. The taxonomy provides the reader with a framework for thinking about code-improving transformations. It also serves as an organizing principle for the paper.

1 INTRODUCTION

This paper presents an overview of compiler-based methods for improving the run-time behavior of programs — often mislabeled *code optimization.* These techniques have a long history in the literature. For example, Backus makes it quite clear that code quality was a major concern to the implementors of the first FORTRAN compilers [18]. Our focus is on traditional Algol-like languages; most of the techniques we discuss have been invented for use with languages like FORTRAN or C.

Both code "quality" and run-time "behavior" are ambiguous phrases. For any specific source-code procedure, the compiler can generate many translations that produce the same external results. Some of these translations will execute fewer instructions; some will require less running time; some will contain fewer bytes of code; some will require less memory space for data. A compiler "improves" the code to the extent that it reduces the resource requirements for executing that code—either time or space. Most of our attention will focus on transformations intended to decrease the running time of the program;[1] along the way we will chronicle some transformations whose sole purpose is to save run-time space.

It would be intellectually pleasing if the development of code improvement techniques had proceeded in an orderly fashion, focusing first on single expressions, then expanding to encompass ever larger regions until, finally, entire programs were being analyzed. This would provide a well ordered, historical organization to our survey. Unfortunately, that is not what occurred. Early commercial compilers used sophisticated analyses to look at entire procedures [19, 18]. Early academic papers attacked problems in smaller contexts [106]. Ershov talked about interprocedural transformations in the mid 1960's [97]; work on code generation for single basic blocks has continued into the 1990's [196, 163].

To an objective outside observer, this area often appears as a grab-bag collection of tricks. Myriad papers have been written that present a specific technique or method; fewer papers compare and contrast multiple techniques for achieving the same effect. Our goal for this paper is to provide insight into which problems can be attacked, how these problems have been addressed in the literature, and how the various methods relate to each other.

This paper examines a select subset of the many techniques that have appeared in the literature. We begin by proposing a simple taxonomy that organizes the transformations according to how they operate. We use that taxonomy to organize the presentation of transformations. We have selected example transformations to flesh out the taxonomy; they provide a fair coverage of work in each area. The techniques are presented

This work was supported by DARPA through Army Contract DABT63-95-C-0115.

[1]Ideally, the compiler should decrease the running time on all inputs to a program. A more realistic standard is to decrease the running time on most of the inputs and to increase it on few or none.

at a surface level; for more detailed information, we provide citations back to the original papers. After describing the transformations, we present a second taxonomy that organizes the transformations according to the way in which they improve the code. Both views are useful.

The remainder of this section explains our focus on uniprocessors, our emphasis on transformations rather than analysis, and the basic issues that determine the safety of a transformation. Section 2 presents our framework, organized around the ways in which a transformation can actually improve code quality. Section 3 explores transformations that achieve their benefits by reordering code rather than replacing it. It classifies among transformations based on the resulting static and dynamic frequency of execution for the computation that is moved. Section 4 describes transformations that replace or refine sections of the code being compiled. It classifies replacement strategies according to the amount of code used for the replacement. Section 5 explores the issues that arise when combining multiple transformations into a single operation. Section 6 examines an alternative taxonomy for transformations that can serve as a simple checklist for the compiler writer. Finally, Section 7 revisits the issue of safety and discusses some of the different approaches to safety taken by actual compilers.

1.1 Uniprocessor Compilation

This paper focuses on issues that arise in compiling for uniprocessor targets. It largely ignores parallelism, a subject that is generating its own large body of literature. In the arena of instruction-level parallelism, we discuss instruction scheduling. We do not cover transformations that expose multiprocessor parallelism of vector operations. These issues are well covered elsewhere [20, 12]. We intend to provide a framework for thinking about uniprocessor compilation, for understanding the effects of various transformations, and for evaluating whether or not a given transformation might help in your own compiler.

Uniprocessor compilation is important in many realms. Many interesting tasks are performed on single processors, even when those processors are part of a multiprocessor machine. Increasingly, embedded microprocessors are used as controllers in applications ranging from automotive braking systems through airplane avionics; in these real-time systems, performance matters.

Understanding uniprocessor compilation is also critical to compiling for multiprocessors. Compilers for multiprocessors must address all of the performance issues of uniprocessors; thus, the techniques described in this paper form a subset of the methods that should be applied for a multiprocessor. Indeed, the performance of individual processors in a multiprocessor computation often determines overall system performance.

Our focus on uniprocessor compilation leads us to an assumption about the context in which transformations are performed. We assume that transformations are applied to an intermediate representation, rather than the source code. Many of the transformations presented in Section 3 and 4 target inefficiencies in the translated code. Some of the transformations are equally applicable in a source-to-source transformation system [165]; others require require the elaboration of detail that occurs in translation to a lower-level intermediate representation.

1.2 Analysis Versus Transformation

Automatically improving code quality requires both detailed knowledge about the program's run-time behavior and techniques for using that knowledge to change the code. The process of deriving, at compile-time, knowledge about run-time behavior is called *static analysis*. The process of changing the code based on that knowledge is called *transformation* (or, in a common misnomer, *optimization*). Both analysis and transformation play a critical role in compiler-based code improvement. It is important, however, to distinguish between them.

Code improvement is the result of using analytically derived knowledge to improve the instruction sequence while retaining the meaning of the original code. At a high level, the process has four components: (*i*) discovering opportunities to apply a transformation, (*ii*) proving that the transformation can be applied safely at those sites, (*iii*) ensuring that its application is profitable, and (*iv*) actually rewriting the code. Adding a transformation to a compiler requires efficient ways of handling each of these problems. The role of analysis is to support transformation.

Static analysis has a rich history in the literature. The first static analyzer was probably the control-flow analysis phase of the FORTRAN I compiler built by Lois Haibt in 1956 [19, 18]. Vyssotsky built control-flow analysis and data-flow analysis into a FORTRAN II system for the IBM 7090 in 1961 [129]. More formal treatments of static analysis appeared in the late 1960's and 1970's. Hecht's 1977 book and Kennedy's 1981 survey paper describe the development and theory of data-flow analysis [129, 142]. Recent work has focused on static analysis of pointer-based values [58, 60, 85, 94, 157, 205, 230] and interprocedural data-flow analysis [74].

Compilers use other forms of static analysis to guide their transformations. Estimates of execution frequency for individual statements have long been used to guide decisions about program transformation. The FORTRAN I register allocator used a Monte Carlo technique to estimate frequencies for *basic blocks*; branch prediction remains a topic of enduring theoretical and practical interest. During the 1980's and early 1990's, *dependence analysis* assumed a critical importance. Data-dependence analysis was developed as a means of understanding when subscripted array references could and could not refer to the same value. This knowledge plays a crucial role in techniques for automatically discovering parallelism [171, 116]. Static single assignment form (SSA) has proven to be a powerful tool for encoding precise information about the relationship between control flow and data-flow [80, 33].

Analysis is a well understood field, in part, because it lends itself to mathematical formalisms. The literature of analysis is a succession of papers that present improved algorithms for well-defined problems and prove them correct and efficient. The literature of code transformation is less clear, in large part because the problem is not clearly defined. Because the goal of transformation is simply "to improve the code," individual techniques attack different effects with radically different approaches. Thus, to compare two techniques analytically, the author may need to reconcile two completely different approaches. To contrast them experimentally, the author must implement both techniques inside an optimizing compiler and measure their performance on realistic examples; this can take a prohibitive amount of effort. Thus, it is hard to find unifying threads in the literature; it is even harder to find guidance about what methods to include in a compiler.

1.3 Safety of Transformation

The single most important criterion that a compiler must meet is correctness—the code that the compiler produces should have the same "meaning" as the program presented for compilation. Each transformation implemented in the compiler must meet this standard. Generally, correctness is defined by the observable memory state after execution—that is, the values of visible variables after execution terminates. If the program terminates, all of its named variables that are still instantiated immediately prior to termination should have the same values under any translation scheme. In an optimizing compiler, we introduce a second criterion: the code produced by the compiler should not use extraneous amounts of either time or space. The goal of code-improving transformations is to reduce execution time and run-time memory requirements.

As a formal notion, correctness deserves discussion. For the transformations described in this paper, correctness is an issue of equivalence. Each transformation takes as input the code for some part of a program (often a single procedure). As output, it produces code for the same part of the program. If the two codes differ, we expect that the output is "better" by some performance metric. It must still reflect the meaning of the original source code, at least to the extent that the meaning is visible or observable.

To formalize this notion of equivalence, Plotkin defined the notion of *observational equivalence* [193].

> *For two expressions, M and N, we say that M and N are observationally equivalent if and only if, in any context C where both M and N are closed (that is, have no free variables), evaluating $C[M]$ and $C[N]$ either produces identical results or neither terminates.*

Thus, two expressions are observationally equivalent if their impact on the visible external environment is identical.

This notion provides the theoretical basis for code optimization. If the compiler can produce different translations with identical external behavior, then it is free to choose the faster or smaller translation. This statement, however, can be misleading. It suggests that the compiler can enumerate transformations and evaluate their efficiency. In reality, compilers work by generating an initial implementation for the source program and then refining it. They also work from a somewhat simpler notion of equivalence than Plotkin's: if, in their actual program context, the result of evaluating e' cannot be distinguished from the result of evaluating e, the compiler can substitute e' for e (assuming that e' is less expensive to evaluate).[2]

[2]By moving the context of Plotkin's definition outward, we can fold an arbitrary amount of local context into the expression.

Optimizing compilers routinely ignore the issue of divergence or non-termination (see § 7).

Unfortunately, optimizing compilers cannot guarantee a minimal running time. Many of the problems that arise in transforming code are NP-complete; others have polynomial complexity, but their best known solution is judged too slow for use in a compiler. Thus, in practice, compilers try to decrease the running time and space requirements of the program as far as possible, within the complexity constraints imposed by the user's expectation about compile times.

2 SOURCES OF IMPROVEMENT

In reality, the number of ways that a compiler can improve code is quite limited. While many techniques have been proposed in the literature, they achieve their impact in just a few distinct ways. Our goal is to organize the possibilities that have been presented in the literature. To classify a transformation, we use several measures. First, we consider its generality—will it improve code on all target machines or on some subset? Next, we place it in a taxonomy of uniprocessor optimizations—how does it achieve its improvement? Finally, we consider the scope of a transformation—over what range of instructions in the code can it be applied? This section explores these issues in more detail.

2.1 Generality

The most common distinction between transformations that one finds in the literature is the division of techniques into *machine independent* improvements and *machine dependent* improvements. This distinction is largely artificial. Almost all transformations can have unexpected negative consequences that arise from some constraint or feature in the architecture.

For example, the compiler may face a decision to either compute an expression's value or reuse a prior computation of the same value. In the absence of more detailed contextual information, reusing the value appears to be the better choice; not executing an instruction should be faster than executing it. However, reusing the value may lengthen its lifetime. Typically, this requires either the dedicated use of a register for a longer period, or the insertion of additional memory operations. Thus, the availability of spare registers, the cost of evaluating the expression, and the cost of memory operations can all determine whether it is faster to reuse the value or to recompute it.

In reality, the extent to which a transformation improves the code almost always depends on characteristics of the target machine. We prefer to think of machine independence as a conscious decision made by the compiler writer to simplify the transformation. We consider a transformation to be machine independent if the compiler writer decides to explicitly ignore target machine constraints like finite resources or the availability of specialized instructions. Similarly, we consider a transformation to be machine dependent if it explicitly accounts for constraints and features of the target machine.

This issue has been recognized since the earliest days of compilation. Backus reports that during the design of the FORTRAN I compiler

> *"I proposed that they should produce a program for a 704 with an unlimited number of index registers, and that later sections would analyze the frequency of execution of various parts of the program (by a Monte Carlo simulation of its execution) and then make index register assignments so as to minimize the transfers of items between the store and the index registers." [18, page 35].*

It is precisely this separation of concerns that has allowed compilers to progress in the ensuing years. Compiler writers ignore resource constraints during optimization and rely on the final stages of the compiler to make good decisions based on both the code being compiled and knowledge of the target architecture.

Most of the transformations that we discuss achieve their improvements with relatively simple assumptions about the underlying computer—like the assumption that preserving a value for reuse is cheaper than recomputing it. Some transformations, like instruction scheduling, are inherently machine dependent; even though the technique is important on a wide range of computers, any implementation for a specific machine relies on detailed knowledge of the target machine's behavior.

While this works mathematically, it misses the fundamental point: optimizations work because they capitalize on context. Felleisen proposed *conditional observational equivalence* to address this issue of context and equivalence under transformation [101].

2.2 Taxonomy

Many transformations have appeared in the literature. To organize our presentation, we have adopted a simple taxonomy based on how the transformations achieve their improvements. At the highest level, a transformation improves run-time behavior using one of two mechanisms: it moves a code sequence to another location in the program, or it replaces the code sequence with a "better" sequence. This fundamental distinction forms the top level of our classification scheme.

Code Motion Because the translation from source language into intermediate representation proceeds linearly through the source text, it usually respects the programmer's placement of operations. By analyzing the surrounding context, however, the compiler can often discover alternative placements for a computation that result in more efficient execution. For example, an expression evaluated inside a loop may use operands whose values don't change from iteration to iteration. We call such an expression *loop invariant.* Often, the evaluation of a loop invariant expression can be moved to a point before the loop. If the loop's body executes more than once, this should reduce the number of times the expression is evaluated, which usually speeds up overall execution time.

Code motion has both static and dynamic effects. Some transformations change the number of copies of an instruction sequence that appear in the executable, while others change the number of times the instruction sequence is executed. We refer to the first effect as static, while the second is dynamic. A transformation that increases the number of copies of an instruction sequence–its static frequency–may well decrease the number of times that it is executed–its dynamic frequency. The taxonomy distinguishes between these static and dynamic effects.

Code Replacement Typically, a compiler's initial translation produces code to handle the most general case of a programming language construct. In many case, the compiler can use contextual knowledge to tailor this general code to the specific context in which it will execute. The results can be dramatic. For example, Bernstein describes a routine in the PL.8 compiler whose performance was improved by ninety-eight percent when integer multiplies with a known constant operand were replaced with a series of shifts, additions, and subtractions [27]. Some replacements are done for their direct effect, like converting an integer multiply into a sequence of shifts, additions, and subtractions. Other replacements are done for their indirect effects; they improve the effectiveness of other transformations. For example, compilers may apply a replacement whose direct effect is to slow down the code, because the indirect improvements in other techniques far outweigh the immediate negative impact.

Some techniques have effects from both code motion and replacement. We discuss these techniques in both sections of the paper. The more detailed description occurs in the section that more directly fits the transformation's primary effect. Thus, for example, the code motion section describes instruction scheduling in detail; the mention in replacement is somewhat cursory. Similarly, the primary description of redundancy elimination occurs under replacement; a brief subsection in code motion describes the impact of some redundancy elimination methods on loop-invariant code.

2.3 Scope of transformation

The final axis to consider is the scope of optimization. The techniques described in this paper fall into one of five categories:

1. *Local* methods confine their attention to straight line sequences of code. These methods usually operate on *basic blocks* – maximal sequences of straight line code. Local methods are usually the simplest to analyze and understand. Bagwell published an early catalog of local optimizations [21].

   Inside a basic block, two important properties hold. First, statements are executed in some easily determined order. Second, if any statement executes, the entire block executes. If a run-time exception occurs, the execution of a block can be interrupted. These two properties let the compiler prove, with relatively simple analysis, facts that may be stronger than those provable on larger scopes. Thus, local methods sometimes make improvements that simply cannot be obtained on larger scopes.

2. *Superlocal* methods operate over *extended basic blocks* (EBBs). An EBB B is a set of blocks $\beta_1, \beta_2, \ldots \beta_n$ where β_1 has multiple predecessors and each β_i, $2 \leq i \leq n$ has β_{i-1} as its unique predecessor. A single block β_1 can be the head of multiple extended EBBs; these EBBs form a tree rooted at β_1.

   Inside an EBB, the compiler can use facts discovered in β_1 through β_{i-1} to improve the code in β_i. These additional facts can expose additional opportunities for transformation and improvement; larger contexts often lead to larger sets of opportunities. The trick is to optimize EBBs without a major cost increase. To achieve this goal, EBB-based algorithms often capitalize on the tree structure that relates the individual EBBs rooted at β_i.

3. A *regional* method operates over scopes larger than a single extended basic block but smaller than a full procedure. The region may be defined by some source-code control structure, like a loop nest [219]; alternatively, it may be a subset of some graphical representation of the code, like a dominator-based method [197, 160]. These methods differ from superlocal methods because they can handle points where different control-flow paths merge. They differ from global methods in that they consider only subranges of the procedure. This focus on a limited region can be either a strength or a limitation.

   Loop-based techniques are quite effective at exposing parallelism and improving data locality [20]. These techniques appeared in the first FORTRAN compiler; they are equally at home in a modern parallelizing compiler. Strength reduction and loop optimizations achieve their effects within individual loop nests. Dominator-based methods exist for problems ranging from instruction scheduling [218] through redundancy elimination [33]. These methods have proven to be both fast and powerful, resulting in continued interest in dominator-based transformations.

4. *Global* or *intraprocedural* methods examine an entire procedure. The motivation for global methods is simple: decisions that are locally optimal can have bad consequences in some larger context. The procedure emerged as a natural boundary for analysis and transformation. Procedures are abstractions that encapsulate and insulate run-time environments. At the same time, they serve as boundaries for separate compilation in many systems.

   Global methods almost always rely on global analysis. Data-flow analysis evolved to meet this challenge. Thus, global techniques usually involve some kind of intraprocedural (or procedure-wide) analysis to gather facts, followed by the application of those derived facts to specific transformation decisions. Global methods discover opportunities for improvement that local methods cannot.

5. *Whole program* or *interprocedural* methods consider the entire program as their scope. Just as moving from local to global increased the scope of analysis and exposed new opportunities, so moving from single procedures to the entire program can expose new opportunities. In looking at whole programs, the compiler encounters complications and limitations from name-scoping rules and parameter binding that do not exist inside a single procedure.

   We classify any transformation that involves more than one procedure as an interprocedural transformation. In some case, this entails analyzing the entire program with interprocedural data-flow analysis [77]; in other cases the compiler may examine just a subset of the source code [215, 17]. Transformations based on information derived from other procedures can create complex compilation dependences between procedures; analysis may be necessary to determine which procedures to recompile after a change to one of them [39].

It is commonly assumed that optimizing over larger scopes results in better code. There is little published evidence that either confirms or contradicts this notion. For example, this suggests that inline substitution, which creates larger single procedures that can be subjected to global optimization, should be profitable in most cases. The experimental studies are contradictory [84, 201, 76]; they show different results for C compilers than for FORTRAN compilers (see § 4.2.2).

This observation probably reflects several different effects. Local and superlocal methods work on constrained problems that are simpler than the general problems seen when optimizing entire procedures or entire programs. In the smaller scopes, the compiler can often prove stronger facts than in a larger scope; analysis becomes harder in larger scopes because the control-flow interactions and name-space manipulations become more complex.[3] Larger scopes expose more opportunities for optimization; unfortunately, they must rely on weaker analysis.

As the scope of a transformation increases, so does its potential to have a widespread impact on resource allocation. For example, many transformations increase the demand for registers. If the transformation creates too much register pressure, the spill code introduced during allocation can negate the original improvement (see § 4.3.1). Thus, moving to a larger scope of optimization is an act of probabilistic faith in the strength of the new method and in the ability of the rest of the compiler to handle the resulting code. It usually pays off. Sometimes, however, smaller scopes of optimization win.

2.4 Separation of Concerns

In any large software system, fundamental design decisions have a direct impact on construction time, ease of debugging, maintainability, and run-time performance of the final product. In an optimizing compiler, these issues surface in the choice of a structure for the optimizer.

- Structuring the optimizer as a series of passes, each performing one transformation, simplifies debugging and understanding. It does, however, incur costs from additional traversals of the intermediate representation and from reading and writing the intermediate representation at each pass.
- Combining multiple transformations together into a single pass can produce faster compilation times. It can complicate implementation, debugging, and maintenance. It may also increase the compiler's overall code size by duplicating simple functions in several passes.

Conceptually, a pass-structured optimizer is clean. In principle, each task can be implemented once. For example, the IBM PL.8 compiler relied on its dead-code eliminator to perform most code deletion; the other transformations simply inserted new instructions and rewrote reference to make the new code live and the old code dead [70, 16]. However, this separation has costs beyond the obvious ones. For some transformations, techniques that perform them concurrently can produce solutions that cannot be reached from any combination of the separate techniques [65, 67]. (See § 5.) In the sections that follow, we present the techniques as standalone ideas and point out when one algorithm addresses multiple concerns, as with Wegman and Zadeck's *sparse conditional constant propagation* algorithm [228].

3 Code Motion

In many situations, the compiler has the freedom to rearrange the order of computation without changing the program's observable memory state. The compiler can move an instruction; it can change the iteration order of a loop; it can interleave execution of different procedures. Taken collectively, transformations that capitalize on this ability to reorder computations are termed *code motion* techniques. We will classify these techniques according to the static and dynamic frequencies that result from the transformation. The first two categories are degenerate cases, but important.

1. *zero executions* – If the compiler can prove that the observable memory state of the program is unaffected by executing a particular operation, it can delete the operation. A well-known example of this kind of transformation is *dead code elimination.* Since the operations are completely eliminated, the transformation results in zero static instances as well as zero dynamic instances.
2. *one execution* – If the compiler can determine the result of a computation at compile time, it can eliminate any run-time evaluations and directly fold the value into instructions that subsequently reference it.[4] Compile-time evaluation is considered faster than run-time evaluation, on the assumption that the executable is invoked more often than the code is compiled. This transformation results in a single dynamic instance (at compile time), and zero static instances of the operation.

[3]For example, global data-flow techniques commonly assume that all paths through the control-flow graph correspond to executable paths at run-time. This may or may not prove to be true. In a local method, it is true; if any instruction executes, either they all execute or some exception is raised (and post-exception performance is rarely scrutinized).

Interprocedural analysis methods must deal with changes in the name space that occur at procedure calls. Values become inaccessible; values are renamed; values cease to exist. These effects are missing in most global analysis problems.

[4]This may take the form of an "immediate" operation or an immediate load followed by a conventional reference.

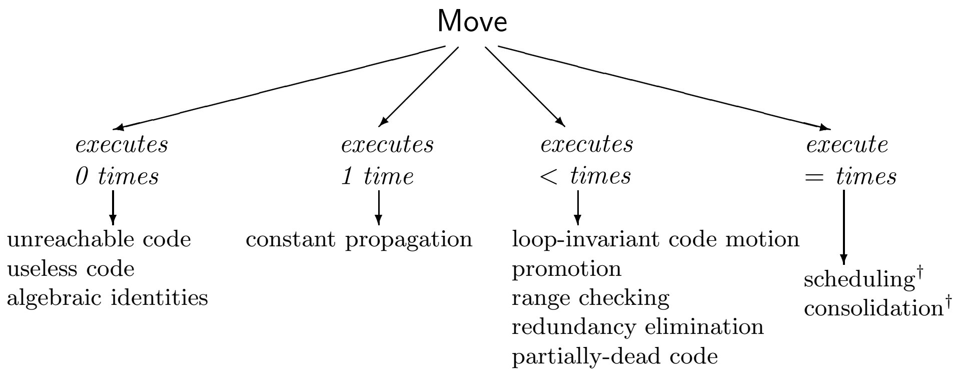


Figure 1: Hierarchy of Move Transformations

3. *fewer executions* – If the compiler can move a computation to a point in the code that executes less frequently, it can reduce the total number of instructions executed. An example of this situation arises when an expression occurs inside a loop, but its value does not change across the iterations of the loop. The expression, called a *loop invariant expression*, can often be moved outside the loop, where it is assumed to execute less frequently. These transformations attempt to reduce the number of dynamic instances of the operation; they may, in fact, increase the number of static instances.

4. *same number of executions* – Sometimes, the compiler can improve the code by relocating operations without reducing their execution frequency. These transformations fall into two general subcategories: replication methods that can create multiple copies of a code fragment to allow better tailoring to a specific execution context, and consolidation methods that eliminate multiple copies of a code fragment to shrink the resulting code. Neither technique tries to decrease the dynamic frequency of the code fragment. Replication increases the static frequency, while consolidation directly decreases the static frequency.

The remainder of this section presents transformations that achieve their primary effects through code motion. Figure 1 shows the transformations, classified by their impact on the execution frequency of the moved code. Two transformations appear in both the code motion and code replacement sections; they are marked with a raised dagger (†) in Figure 1 and Figure 11.

3.1 Executing Zero Times

Sometimes, a compiler encounters code that has no externally visible effect. If the compiler created two executable versions of the procedure, one that included all the code and one that included only the code that produced externally visible effects, a user would be unable to differentiate between them, except by the time and space required to execute them. These two executables are, in the sense of Plotkin, observationally equivalent [193].

When the compiler can determine that an instruction has no externally visible effect, it should remove the instruction. This can have two distinct effects: (1) if the instruction was in a location where it executed, the resulting code should execute fewer instructions; and (2) the executable code should be smaller, even if the instruction would not have executed. Both outcomes are profitable. We classify transformations that accomplish this as a degenerate case of code motion. These methods delete instructions—in effect, moving them to a place where they cannot execute.

3.1.1 Eliminating Useless Computations

Any evaluation whose removal cannot be discerned externally (ignoring running time and space), is considered *dead*. Dead code arises from several sources; perhaps the most common source is code that results of other transformations. Closer examination reveals that dead code falls

```
x ← 17                  x ← a + b
if (x = 0)              y ← b + c
     y ← 0              z ← c + d
else                    x ← d + e
     y ← z/x
```

Unreachable code *Useless code*

Figure 2: Categories of Dead Code

into two general categories:

unreachable code If no valid control-flow path reaches an instruction, removing it is safe. We call such an instruction *unreachable*. Removing unreachable code does not directly speed up execution since, by definition, the instruction will never execute. On the left side of Figure 2, the expression controlling the `if` can never be true, so the assignment under the `if` is unreachable.

Removing unreachable code has two benefits. First, it shrinks the code, which leads to faster compilation and a smaller executable. It can also change instruction cache behavior. Second, it can improve the precision of other analyses, indirectly improving the results of other optimizations [228].

useless code Given a program p and its variant p' created by removing statement s, if p and p' produce the same observable memory state, then s is *useless* and can be safely removed from the program.[5] On the right side of Figure 2, the first assignment is useless because x is redefined before its value is used.

Provided s is reachable, removing s should speed up execution. Eliminating useless code directly decreases code size. It can shrink the live ranges of some values by eliminating their last uses.

The literature refers to both unreachable code and useless code as dead code. The distinction between them, however, is important.

Algorithms Many algorithms have been proposed for discovering and removing useless code. The strongest form is conceptually simple. It requires two passes:

1. The first pass marks all instructions that are useful. It starts with a set of critical instructions, defined as any externally visible writes and any control-flow instructions. Each critical instruction is marked as useful.

   At each useful instruction, it traces backward from the operands along use-definition chains to the instructions that define their values. These instructions are marked as useful, and their arguments traced. Repeating this inductively, all useful instructions are marked.

2. The second pass walks the code in a sequential order and removes any unmarked instruction.

This method uses a fairly strong notion of "useless". Conceptually, it is reminiscent of classic garbage collection algorithms, like the Schorr-Waite marking algorithm [211].

Kennedy first published this algorithm for removing useless code; he claims that Jack Schwartz and John Cocke inspired the idea [141]. Several weaker techniques have appeared in the literature. In general, they are based on computing some notion of *liveness* and removing any instruction that produces a non-live result [61].

The algorithms for removing useless code do not, in general, remove unreachable code. Many systems have performed some kind of unreachable code elimination; few of those techniques have been recorded in the literature.[6] Cytron *et al.* present an algorithm for dead code elimination that removes both unreachable

[5]This is sloppier than Plotkin's notion, because it is always expanded to include context.

[6]For example, the PFC system ran a special pass after constant propagation and dead code elimination. It used a symbolic testing package developed for dependence analysis to check each loop for a zero trip count and each `if` statement for a constant-valued controlling expression [138].

code and useless code [80]. They mention an unpublished algorithm by Paige that also removes unreachable code.

Another technique for removing unreachable code has been embedded into constant propagation (see § 3.2.1). First Wegbreit, then Wegman and Zadeck presented algorithms that discover some control-flow predicates that cannot be satisfied and prune those paths from the control-flow graph [225, 227, 228].

Scope Fundamentally, the issue of reachability arises from inter-block control flow. Thus, the algorithms that detect unreachable code either are global in their scope, like the algorithm due to Cytron *et al.*, or rely on relatively local information to make regional decisions, like the methods of Wegbreit or Wegman and Zadeck. Useless code, on the other hand, can sometimes be detected locally; the example in Figure 2 is a purely local effect. Global information about live variables is necessary, however, to capture all but the simplest cases; thus, global algorithms like Kennedy's are widely used, and little attention has been paid to local or regional methods.

3.1.2 Algebraic Identities

The algebraic laws that define arithmetic introduce some subtle forms of useless code. The compiler can rely on algebraic identities within integer computations to remove useless code; with computations involving floating-point numbers, some care is required. Integer arithmetic constitutes a large portion of most computations. Most of the overhead introduced in translation to implement programming language abstractions uses integer arithmetic.

Many binary operators have algebraic identity elements. For example, $\forall x, x + 0 = x$, $\forall y, y - 0 = y$ and $\forall z, z \times 1 = z$. Recognizing uses of these identities can lead to better code. The idea is simple; if the compiler can prove that one operand is the identity element for the operator, then it performs no noticeable arithmetic.

In practice, the identities arise from two sources.

1. The compiler's front end generates fully general code where it is not needed. For example, it might generate the full address polynomial for an array reference. This generates a term of the form $\langle \textit{upper bound} \rangle - \langle \textit{lower bound} \rangle + 1$. Later in compilation, if it discovers that $\langle \textit{lower bound} \rangle$ has the value one, the computation simplifies to $\langle \textit{upper bound} \rangle - 1 + 1$. Local constant folding simplifies the constant expression to zero. Using the additive identity, this simplifies to $\langle \textit{upper bound} \rangle$.

2. Other optimizations, like constant folding, redundancy elimination, and strength reduction, substitute known constant values forward. The substitution is done without regard for the surrounding context. Subsequent simplification yields an expression containing an algebraic identity.

Several cautions are important. The implementation of an operator may give a semantics to an operation that appears to be an identity operation. For example, $x \leftarrow x + 0.0e0$ may force normalization of floating-point value x. Similarly, $y \leftarrow x \times 1.0e0$ has several possible side effects. It may set condition codes. More importantly, it copies the value of x into y. This creates a separate value with a separate lifetime, changing the register allocation problem. Replacing this with $y \leftarrow x$ may preserve neither effect. (The register allocator may coalesce away the copy, but it cannot coalesce away the multiply.)

These issues notwithstanding, simplifying algebraic identities almost always leads to better code. An instruction that implements and algebraic identity can be eliminated or replaced with a copy if and only if it does not set a condition code that is subsequently used. However, since most opportunities arise from compiler-introduced "overhead", the condition code is not often an issue. If a copy is required, it can often be removed by the register allocator.

Hanson presents a simple local algorithm for discovering algebraic identities and transforming the code to remove them [126]. His work is aimed at improving the code generated in a single pass compiler; it discovers a number of algebraic identities. The paper suggests that the technique is quite effective in its intended environment—a single pass compiler. His technique has several similarities to Balke's algorithm for local value numbering (see § 4.1.1).

Often, algebraic identities are handled during redundancy elimination. Hash-based value numbering algorithms are easily extended to discover values related by simple algebraic identities [71, 33]. Click extended the partitioning algorithm of Alpern *et al.* to discover congruences based on algebraic identities [65].

```
$x \leftarrow 17$
  ...
$y \leftarrow 23$
$z \leftarrow x + y$
```

Local example

```
$y \leftarrow 23$
if (...) then
  $x \leftarrow 17$
else
  $w \leftarrow 5$
  $x \leftarrow 12 + w$
$z \leftarrow x + y$
```

Global example

```
if (...) then
  $x \leftarrow 17$
  $y \leftarrow 23$
else
  $x \leftarrow 23$
  $y \leftarrow 17$
$z \leftarrow x + y$
```

A harder issue

Figure 3: Constant propagation examples

Scope Algebraic identities are a local issue. They arise on an operator-by-operator basis. Algorithms for recognizing and eliminating them require only local context; adding more context does not allow discovery of additional identities. These techniques have been embedded into regional and global algorithms for other problems, like redundancy elimination and constant propagation. In general, this happens to improve the results of the other transformation and to avoid constructing a separate pass to eliminate algebraic identities.

3.2 Executing One Time

The second degenerate case of code motion replaces an instruction with its result. If the compiler can determine the values of all the operands to an instruction, it may be able to interpret the instruction at compile time and replace instances of the instruction with the result that it would produce. The most common form of this transformation is called constant propagation.

3.2.1 Constant Propagation Analysis often reveals that the arguments to an operation are known constants. Once the operands' values are known, the operation can be evaluated at compile time and its result can be used directly in the compiled code. (In practice, the evaluation is replaced by an immediate load or a load from a memory location designated to hold the constant value.) The transformation is both simple and well known; in fact, Floyd mentioned it as an improvement to his 1961 scheme for code generation [106]. The real difficulty lies in discovering the values of as many operands as possible.

Consider the examples shown in Figure 3. The left code fragment shows a simple problem arising in a basic block. Because the block executes in a determined order, the compiler can easily prove that z gets the value 40. The center example shows one complication that arises in larger scopes; the compiler must discover that, along both the `then` path and the `else` path, x receives the value 17. It must recognize that this occurs on all paths reaching the assignment to z before it can prove that z gets the value 40. The final example shows a more difficult problem—one beyond the reach of constant propagation. Along either path, z receives the value 40. However, the value of x is inconsistent between the two paths, as is the value of y. Thus, constant propagation will conclude that neither x nor y have a known value at the assignment to z—a reasonable and correct conclusion, even though the sum $x + y$ always has the value 40.

As a data-flow analysis problem, constant propagation has a long history in the literature. Analysis cannot discover all the constant-valued expressions in a program; some will depend on inputs, others require uncomputable knowledge about which control-flow paths can be taken at execution time. Fortunately, discovering some subset of the constant values is worthwhile; each such expression eliminates instructions.

Kildall described an iterative scheme for discovering a set of constant values in a single procedure [146]. He introduced a lattice to describe the set of known constants at each point in the program, gave a set of equations to relate the lattice element at a given point to those of its predecessors in the control-flow graph, and presented an iterative algorithm that discovers a fixed point of this set of simultaneous equations. The fixed-point solution of these equations concisely represents the set of values known to be constant at each point in the program. Using this knowledge, the compiler can replace some expression evaluations with references.

Others have improved on Kildall's basic scheme. First Reif and Lewis, and then Wegman and Zadeck, showed that starting from an initial lattice value of $\top$ rather than $\perp$ can increase the set of discovered constants [199, 228].[7] Wegbreit presented a technique that used knowledge about constants to determine that some branches could not be taken; this can, in turn, expose additional constants [225]. Wegman and Zadeck combined Wegbreit's method of discovering non-executable paths with the optimistic initialization of Reif and Lewis to produce their *sparse conditional constant* algorithm [228].

Constants can also be propagated interprocedurally. In 1985, three papers described different techniques for interprocedural constant propagation. Wegman and Zadeck proposed inline substitution (see § 4.2.2) during analysis to expose constants propagated across calls [227]. Burke and Cytron proposed solving an interprocedural data-flow problem over the program's call graph using interval analysis [38]. Torczon *et al.* gave an iterative algorithm for solving a similar problem; their abstraction of a procedure's effects into a "jump function" lets the algorithm propagate constants returned by a procedure [46]. Grove and Torczon present an experimental evaluation of several techniques for implementing jump functions [121]. A surprising result of their work is the importance of interprocedural MAY MODIFY sets to intraprocedural constant propagation. Metzger and Stroud present experimental results obtained using their implementation in the CONVEX compiler [178]. Their algorithm is based on Torczon's work.

Implementation details While the transformation is simple, a number of subtleties arise. Most of them arise when constant propagation has the opportunity to rewrite an instruction because it has discovered some constant value.

1. When a constant value is the source of a copy operation, the compiler can either leave the copy intact or convert it into a load immediate instruction. On most architectures, these operands have similar costs. The copy is preferable, since register allocation may coalesce it away; the load immediate cannot be removed in the same way.
2. When one argument to an integer multiply is a known constant, and the other is unsigned or positive, the compiler replace the multiply with a sequence of shifts, additions, and subtractions (a weak form of strength reduction, see § 4.1.2). Often, the sequence is less expensive than the integer multiply instruction. Note, however, that this converts a commutative, associative operator, multiply, into a series of shifts and subtractions that do not commute. This can impede other transformations, like reassociation (see § 4.2.1).
3. If the intermediate representation includes a representation of address modes, constant propagation can convert some instructions into their immediate forms — for example, an add into an add immediate.
4. To get maximal benefits from constant values, the intermediate representation should expose them directly. This suggests the need for an immediate load in each data type supported by the hardware. Their presence in the intermediate language exposes them to both optimization and better spilling in the register allocator [35].

Scope Constant propagation makes sense in many different scopes. McKeeman included constant folding in his early peephole optimizer [173] (see § 4.1.3). Balke's local value-numbering algorithm (see § 4.1.1) performs constant folding inside a single block; as that method extends to larger domains [33], so does its constant folding. Extending the transformation to entire procedures requires global analysis; constant folding provided the motivation for Kildall's original work on the lattice-theoretic formulation of data-flow analysis [146]. Interprocedural constant propagation has been explored in several systems and found to be profitable [178, 121].

3.3 Executing Fewer Times

Several transformations attempt to move a computation to a point in the program where it is expected to execute fewer times than it would in its original location. The compiler cannot necessarily determine the number of times that a particular statement will execute. Thus, it may approximate or estimate execution frequencies. For example, compilers generally assume that the body of a loop executes more often than the code that surrounds the loop, even though the loop may not, in some particular case, execute at all. This assumption has focused much of the work on code motion toward the problem of moving code out of loops.

[7]Wegman and Zadeck introduced the term "optimistic" to describe this effect [227].

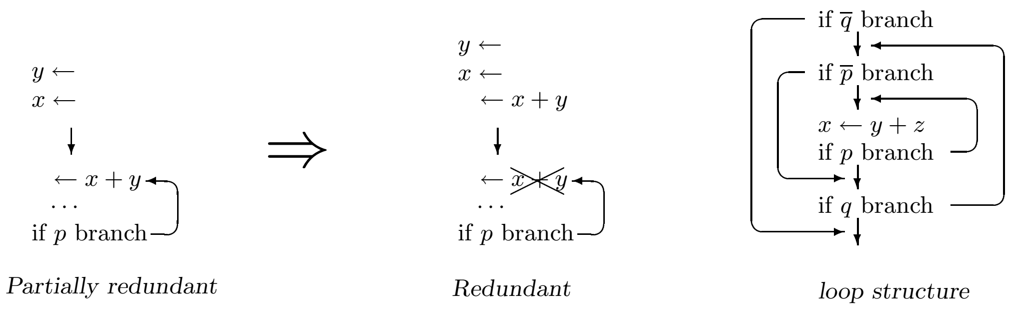


Figure 4: Partial redundancy

3.3.1 Loop-invariant code motion An important class of code motion algorithms attack the problem of discovering and moving loop-invariant operations. An operation is *loop-invariant* if it produces the same result in each iteration of the loop. If the compiler can prove that an operation is invariant in the innermost loop that contains it, the compiler can safely move it out of the loop. To avoid lengthening paths that approach but do not enter the loop, it should be placed in a block that executes if and only if the loop is entered. This block is often called a *landing pad*. Before performing code motion, the compiler can insert an empty block between the initial test for the loop and the loop's body to act as a landing pad.

Classic approach The classic algorithm for loop-invariant code motion works from use-definition chains [4]. Starting in an inner loop, the compiler examines each instruction. Using the use-definition chains, it checks whether or not all the definitions that reach the operands lie outside the current loop. If no operand uses a definition in the current loop, then the compiler concludes that the result of the operation cannot change inside the loop and the operation is moved to the loop's landing pad. With an appropriate data structure for the use-definition chains, the information requires no direct update to reflect the move; thus if instruction i is moved, a subsequent instruction that references i will discover that the result of i is outside the loop.

Partial redundancy elimination Partial redundancy elimination (see § 4.1.1) also performs some motion of code out of loops. Any loop-invariant expression is also partially redundant, as shown in Figure 4. On the left, the loop-invariant expression $x+y$ is partially redundant since it is available from one predecessor (along the back edge of the loop), but not the other. Inserting an evaluation of $x + y$ before the loop allows it to be eliminated from the loop body, as shown in the center column.

Thus, partial redundancy elimination moves some loop-invariant code. Because it computes the insertion points globally, it can move an evaluation out of several loops at once. Because it operates under a tight stricture against lengthening executable paths, it cannot move expressions out of a conditional inside a loop.[8] The same restriction can stop it from moving an expression out of multiple loops. If any of the loops have a "no-trip" branch around them, the branch acts as a barrier to code motion by partial redundancy elimination; moving the expression out beyond the conditional branch would lengthen the "no-trip" path around the loop. The right column of the figure shows this situation. The expression $y + z$ cannot move past the branch on $\overline{p}$ because it would lengthen that path.

Moving control structures Cytron, Lowry, and Zadeck proposed an improvement to the classic notion of loop-invariant code motion [81]. It begins by renaming values to provide unique names for definitions;[9] next, it moves both control flow and computation into the landing pad. They present two versions that differ with respect to the problem of "down safety." Their "STRICT" algorithm only moves computations that will always execute; the "NONSTRICT" algorithm relaxes that constraint and moves computations that will probably result in shorter executions. By moving control-flow and the code it contains, their algorithm can move more code out of loops than the classical approach. This is a specialized form of replication (see § 4.2.2).

[8]The expression is evaluated on one side and not the other; moving it out of the loop lengthens the path that did not previously evaluate it.
[9]The renaming algorithm is a pre-cursor to static single assignment form [80].

Other issues An operation may be invariant in multiple loops. Of course, the compiler designer would like to move that operation to its final location in a single step. The classic approach "bubbles" it outward by proving it to be invariant in a succession of nested loops. Partial redundancy elimination computes the outermost feasible location and directly places the instruction there.

Most of the algorithms for moving code out of loops ignore the fact that moving invariant code from inside a conditional to outside the enclosing loop can lengthen some paths [148]. Instead, they move an invariant operation to the landing pad even if the doing so lengthens some paths through the loop. This is a simple form of speculative execution. Algorithms based on partial redundancy elimination avoid this problem. Respecting the lengths of conditionally executed paths avoids some cases where code motion could hurt performance; it also avoids cases where the code motion would actually improve performance. Unfortunately, distinguishing between the two cases on the basis of static information is difficult. Fischer and LeBlanc suggest that execution profile information be used to make this decision. [103, p. 640].

Comparing the Techniques If we limit our consideration of partial redundancy elimination to its code motion effects, then the methods for moving loop-invariant code form a simple hierarchy, from the classic method to partial redundancy elimination to Cytron, Lowry, and Zadeck's technique. We know of no comparative data that shows how much improvement is attained for the additional complexity. Of course, partial redundancy elimination actually achieves multiple objectives. A compiler that uses it for redundancy elimination may stand to gain little by adding another pass that moves loop-invariant code.

Scope Loop-invariant code motion is inherently a regional problem. The classic algorithms have focused on single loops or self-contained nests of loops. The algorithms based on partial redundancy use global analysis to discover opportunities and to place the moved operations; these algorithms perform a single global analysis pass to obtain regional effects in each loop nest.

3.3.2 Promotion When the compiler cannot tell that it is safe to keep a value in a register, it must generate the loads and stores necessary to use the value from its actual location in memory. This situation can arise in several ways.

1. Array references can have subscript expressions too complex for analysis; this leaves the compiler unable to determine if two references to the same array actually refer to the same element.

2. Pointer-based references can be ambiguous if they involve pointers passed across procedure calls, or stored into non-local memory locations. If the compiler cannot understand the possible values taken on by the pointer, it must assume that other references may access the same location.

3. Call-by-reference parameter binding can create multiple names for a single memory location. Without interprocedural analysis, the compiler cannot determine that globals and parameters are distinct.

Static analysis can disambiguate some, but not all, of these references. For pointer and array references, the analysis usually occurs after the initial intermediate representation has been built.[10] Thus, the compiler's front end translates each potentially ambiguous reference in the general form, using loads before uses and stores after definitions.

Carr and Lu have proposed transformations that use the results of static analysis to improve formerly ambiguous references [45, 49, 51, 167]. These transformations attack related problems. The analysis required to support them, as well as the mechanisms necessary to transform the code, are quite different.

Scalar Replacement Carr's transformation, called *scalar replacement,* uses the results of dependence analysis to discover consistent and unambiguous uses of array elements inside a loop nest. It creates an explicit scalar temporary variable to hold the array element, and inserts memory operations outside the loop to load and store the value, as appropriate. When combined with loop transformations like *unroll-and-jam,* scalar replacement can have quite dramatic effects. Figure 5 shows a simple loop nest

[10]With aliasing introduced by parameter binding, the necessary interprocedural analysis can be performed before translation, avoiding the need for an explicit transformation to improve the code [77]. The local analysis requires for pointers and array references is usually complex enough that it must be performed after the initial translation.

```
do i = 1, n
  do j = 1, n
    a(i) = a(i) + b(j)
  enddo
enddo
```

Original loop nest

```
do i = 1, n
  t = a(i)
  do j = 1, n
    t = t + b(j)
  enddo
  a(i) = t
enddo
```

After scalar replacement

```
do i = 1, n, 2
  t0 = a(i+0)
  t1 = a(i+1)
  do j = 1, n
    t3 = b(j)
    t0 = t0 + t3
    t1 = t1 + t3
  enddo
  a(i+0) = t0
  a(i+1) = t1
enddo
```

After unroll-and-jam

Figure 5: Scalar replacement and Unroll-and-jam

transformed by Carr's techniques. The central column shows the original code rewritten by scalar replacement; in this form, most compilers manage to keep `t` in a register. The right column shows the result of applying unroll-and-jam to reduce the ratio of memory operations to arithmetic operations.

Register Promotion Lu's transformation, called *register promotion,* uses the results of interprocedural points-to analysis to disambiguate pointer-based memory references. It then "promotes" unambiguous values into newly-created register-based temporaries. To achieve this, it performs a simple, loop-by-loop analysis that pinpoints both the references and locations for the necessary loads and stores.

In both cases, the speedup comes from moving loads and stores out of loops. Thus, we classify these transformations as a form of code motion.

Scope Both scalar replacement and register promotion focus on loop nests within the program. They use regional analysis to obtain regional effects.

3.3.3 Range Check Optimization

Some compilers have insisted on proving that every reference to a dimensioned data object falls within its declared bounds. Pascal compilers tried to ensure the safety of all references, including array elements, discriminated unions, and pointer-based variables. Their task was complicated by the strong desire to limit Pascal compilers to a single pass for efficiency [102]. Many of the same problems arose in PL/I, where checking was optional, but often used [215]. The IBM PL.8 compiler required that every subscript expression be checked [16]. The PL.8 compiler tried to transform away the overhead of checking references. Generally, such checking falls into two cases.

1. For some references, the compiler can prove, at compile-time, that the subscript expression always evaluates to a legal value. In that case, the reference is safe and can be performed directly.
2. For other references, the compiler inserts code to check, at run-time, that the subscript expression falls within the object's declared bounds. In practice, this run-time checking can expose many perplexing errors cause by out-of-bounds references. It can also consume substantial amounts of run-time.

Several authors have examined the problem of decreasing the overhead of providing ubiquitous range checking. Taken together, these techniques can significantly lower the added overhead of ensuring safe access to subscripted variables.

Cousot and Cousot presented one of the earliest frameworks for using static analysis to avoid run-time range and reference checks [79]; Their scheme derived compile-time knowledge that would allow the compiler to avoid generating run-time checks. Harrison's work in the IBM experimental compiler system addressed similar concerns [128].

Fischer and LeBlanc describe several mechanisms to improve the accuracy and efficiency of range and reference checking in a Pascal compiler that they developed at Wisconsin [102]. They used knowledge about the iteration ranges of `for` loops to eliminate many checks, introduced tags to check pointer-based references, and added locks and keys to handle discriminated unions. They point out the limitations on analysis imposed by working in a single-pass compiler. Welsh extends their work on optimizing range checks for subscripted variables [229]; his work also addresses arithmetic exceptions in the checking code.

Markstein *et al.* focused on the problem of subscript range checking. They encoded the range checking operation as a "trap" instruction in the compiler's intermediate language. This avoided insertion of conditional control flow, and made the resulting code more directly amenable to other optimization. They showed how to extend strength reduction, redundancy elimination, and code motion to move traps out of loops or to eliminate them. Their approach was quite effective. Without optimization, range checking added fifteen percent or more to execution time; with optimization, it feel to two percent [170].

Gupta's work differs from Markstein *et al.* in that he proposes standalone techniques—they do not rely on the presence of other optimizations [122]. (Although, the results might be improved by the presence of other optimizations.) He recasts the range checking problem into two distinct pieces: local elimination and global elimination. He attacks the local problem with a method that determines when one test covers another, either because they are identical or because one subsumes the other. He formulates the global problem as a data-flow framework that tracks which checks are "available" at each point in the program; working by analogy to redundancy elimination, his method can discover many checks that are unneeded because the condition must already have been tested.

Asuru proposed an improvement to the work of both Gupta and Markstein. His technique improved the results for `repeat-until` loops and allowed elimination of additional checks [15]. Chin and Goh noticed that Asuru's formulation was unsafe; it reported errors when none occurred and it detected other errors after the out-of-bounds reference had already executed [59]. They propose remedies for both problems.

Scope Range checking is applied in local, regional, and global contexts. All the authors explored local issues. Harrison's analysis uses global analysis to make local improvements. Markstein's work used both regional techniques like strength reduction and broader techniques like global redundancy elimination. Gupta recast the problem as a global data-flow framework and used both global analysis and global transformations. The work from Asuru and Chin and Goh includes both local and global techniques.

3.3.4 Redundancy Elimination

The primary goal of redundancy elimination is to discover cases when the executable code will repeat a calculation and to replace the redundant computation with a reference to the previously produced result. Thus, we think of redundancy elimination as removing the code that evaluates an expression and rewriting instructions that subsequently use the value. The techniques based on partial redundancy elimination (PRE) also detect situations where inserting an evaluation along some path will make a later evaluation redundant [179] (see § 4.1.1). Thus, PRE has the effect of moving computations backward along paths in the control-flow graph. In particular, PRE can discover certain kinds of loop-invariant code, insert a duplicate computation before the loop, and remove the now-redundant computation inside the loop. This achieves, as a side effect, the same result as loop-invariant code motion.

Because PRE carefully avoids lengthening any path, it will not move code out of a loop if it lies on a conditionally-executed path inside the loop. Thus, it achieves less motion than aggressive techniques like Cytron, Lowry, and Zadeck's code motion algorithm [81] (see § 3.3.1). However, its conservative behavior limits possible down-side effects.

Scope PRE relies on global data-flow analysis to discover opportunities and to determine code placement.

3.3.5 Partially Dead Code Elimination

A further refinement of these ideas is the elimination of *partially dead code.* An expression is partially dead if some, but not all, of the paths that contain it do not subsequently reference it. Figure 6 shows a simple example. In the left column, labeled "Before," the assignment to x is live on the path into the `if`, but not into the `else` (assuming that x is not live after the `if–then–else`). On the right, in the column labeled "After," the assignment has been moved to a point where it is not partially dead—it is live along all paths. This notion is similar to partial redundancy, as suggested by its name (see § 4.1.1). A partially dead evaluation cannot be removed, but it can sometimes be moved to a point where the result is always live – that is, where the expression is *very busy.*

Several methods for eliminating partially dead evaluations have appeared in the literature. Feigen *et al.* propose a transformation that they call *revival* [100]. It moves a partially dead assignment to v to the point where v is used. It may also need to move some additional code with the assignment to preserve

```
x ← y * z
if (···)
    w ← x * z
else
    w ← 0
x ← ···
```

Before

```
if (···)
    x ← y * z
    w ← x * z
else
    w ← 0
x ← ···
```

After

Figure 6: Partially dead code

correctness. Knoop *et al.* present an optimization framework based on the ideas used in partial redundancy elimination. They solve a series of data-flow equations to discover partially dead code and to compute optimal placements for the code [150]. Steffen later published a method that eliminates all partially dead code, but introduces non-determinism as a side-effect [217]. Since it may require backtracking at execution time, it seems less practical than the other methods. Bodik and Gupta propose a method that uses program slicing technology to construct predicates for partially dead statements [28]. After transformation, the partially dead statement will execute only if execution then proceeds along a path where the result is live. They claim that the resulting technique eliminates occurrences of partially dead code that none of the other methods can remove.

Other transformations eliminate some partially dead code as a side effect; in particular, forms of forward substitution [61] and instruction scheduling that move computations into conditionally-controlled regions can move a partially-dead evaluation to a point where it is live along all paths.

Scope The definition of "partially dead" involves consideration of all control-flow paths through a procedure. Thus, the various techniques for partially-dead code elimination rely on global analysis and the transformations are either regional or global in nature.

3.4 Executing the Same Number of Times

Sometimes, the compiler can improve the running time of compiled code by moving instructions to new locations, even though they execute the same number of times. We will discuss two examples. Instruction scheduling is the purest example; in scheduling, we rearrange the order of execution to hide machine latencies. The second example is consolidation; it replaces multiple copies of an instruction with a single, appropriately-placed, copy. It fits in both this subsection and § 4.1. Since its primary effect is to reduce code size, we discuss it as a code replacement technique rather than as a form of code motion. However, from the perspective of any single execution path through the changed section of the code, consolidation simply moves the instruction to another spot on the same path.

3.4.1 Instruction Scheduling On many computers, the specific order in which instructions execute has a major impact on the overall time required to execute them. Some processors can issue multiple instructions in each cycle; to achieve maximal performance, the compiler must arrange the instructions so that each issue slot is filled. Some processors allow multiple instructions to execute concurrently, in pipelined fashion, on a single functional unit; to achieve correct execution, the compiler must ensure that an instruction completes before its successors try to use its result. Most processors allow the compiler to schedule instructions into the "delay slots" of a branch to hide branch latency. The number of delay slots grows as the product of functional units and branch latency. Memory accesses are even more complex. The pipelined access hardware allows multiple accesses to proceed concurrently. The presence of various layers of cache memory means that load latency is not only unpredictable, but also history sensitive. The scheduler must make simplifying assumptions about load latencies.

Today's machines are quite sensitive to scheduling. Further hardware developments may make performance less sensitive to scheduling. In particular, the increasing popularity of out-of-order execution and dynamic register renaming may produce machines that gracefully discover good dynamic schedules from

```
$cycle = 0$
$ready\text{-}list$ = leaf nodes in the precedence graph, DG
$inflight\text{-}list$ = empty list
while ( $ready\text{-}list \cup inflight\text{-}list$ not empty )
    for $op$ = each node in $ready\text{-}list$ in descending priority order
        if (an issue slot and functional unit exit for $op$ to start at $cycle$)
            remove $op$ from $ready\text{-}list$ and add to $inflight\text{-}list$
            add $op$ to schedule at time $cycle$
        endif
    endfor
    $cycle = cycle + 1$
    for $op$ = each node in $inflight\text{-}list$
        if ($op$ finishes at time $cycle$)
            remove $op$ from $inflight\text{-}list$
            check nodes waiting for $op$ in DG and add them to $ready\text{-}list$
                if all operands are available
        endif
    endfor
endwhile
```

Figure 7: List Scheduling Algorithm

code that contains a nearby static schedule. This kind of architectural development will certainly change the details of compiler-based instruction scheduling—the goal of scheduling will be to ensure that the appropriate instructions are within the processor's look-ahead window for out-of-order execution. It seems unlikely, however, that hardware techniques will completely eliminate the need for compiler-based instruction scheduling.

List scheduling List scheduling is the traditional algorithm used to schedule single basic blocks. The algorithm generates a schedule that satisfies the precedence constraints imposed by the flow of data and tries to minimize overall schedule length. The algorithm is easily extended to EBBs[11]; it also serves as the basis for many of the more complex regional techniques like trace scheduling and software pipelining. The list scheduling algorithm has three steps.

1. Build a *precedence graph* (DG) that encodes the constraints imposed by the flow of data from one operation to the next. A leaf in the graph is an unconstrained instruction, that is, one that can be issued at the start of the block. A root in the graph is constrained by the need for data from its predecessors; thus, it cannot be issued until all its predecessors complete.

2. Assigns a priority to each node in the precedence graph. Different heuristics for priority assignments to nodes have been tried; the most common approach uses the node's depth — the largest latency-weighted path length from the node to a root — as its priority.

3. Use an iterative algorithm to select and schedule operations from a queue ordered by the node's priorities. Figure 7 shows the details. *Ready-list* is a priority queue that contains only nodes whose operands are all available at *cycle*. *Inflight-list* contains all nodes issued before *cycle* that did not complete before *cycle*. The basic operations are simple. Slots are filled in from the *ready-list* in priority order. Nodes from the *inflight-list* are added to the *ready-list* at the end of the cycle in which their last operand becomes available.

In this form, the algorithm performs *forward list scheduling*. The *backward list scheduling* algorithm is derived by reversing the direction of edges in the precedence graph DG and redefining the notion of "ready." The backward algorithm moves a node from the *inflight-list* to the *ready-list* when issuing the node in the current

[11]On EBBs, the implementor will want to let the scheduler move instructions downward (along control flow paths) in the EBB.

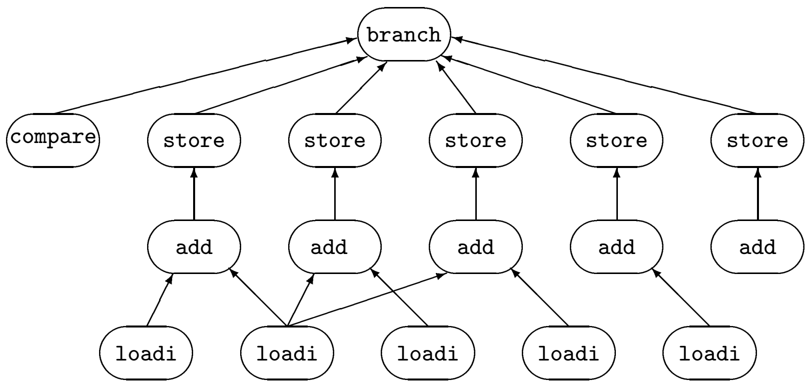


Figure 8: Backward List Scheduling Example

cycle would allow it to complete before any of its successors in the reverse DG are issued. Some blocks are best scheduled with the backward algorithm; Figure 8 shows such a block taken from the program go in the SPEC 95 benchmark suite. Of course, a similar block can be constructed where the forward algorithm does better than the backward algorithm. In practice, many compilers run both a forward pass and a backward pass and use the best result.

Several authors have published improvements to the base algorithm. An oft cited paper due to Gibbons and Muchnick [115] presents an algorithm that ...

Trace scheduling In trace scheduling, the compiler selects the most frequently executed path through a region (typically a loop), and schedules it (using list scheduling or one of its variants) [104]. It proceeds to schedule additional paths in decreasing order of execution frequency; for each path, it creates a schedule relative to the already constructed schedules for higher priority paths. The original trace scheduling work relied on data obtained from actual executions to prioritize traces [104]; Click presented a variant that used static information to establish priorities [66].

Kernel scheduling Kernel scheduling is another approach that tries to schedule over regions larger than an EBB. A kernel scheduler typically operates over a single loop nest, often focusing exclusively on the inner loop—relying on the folk theorem that the execution time of inner loops dominates total execution time. The goal of kernel scheduling is to create as efficient a loop kernel as possible–that is, it should have minimal length and it should keep as many of the available functional units as possible busy.

Perfect pipelining works by unrolling the loop and scheduling it until a steady-state pattern is found [5]. The steady-state pattern becomes the kernel of the loop; the algorithm generates a prolog to set up for the steady state and an epilog to unravel it on completion.

Software pipelining works by selecting a kernel length and performing list scheduling with a wraparound (or modulo) schedule [153]. Lam's algorithm computes a lower bound on feasible kernel length and tries to schedule at that length. If the list scheduler fails, it increases the kernel length and tries again. Prolog and epilog loops are required to fill the "pipeline" initially and to drain it on completion.

The effect of kernel scheduling is to improve performance by executing different iterations of the loop concurrently. This allows the compiler to fill "holes" in the schedule with operations from other iterations; by working with several iterations, the scheduler can fill the instruction issue slots in the loop's kernel while satisfying the precedence relationships imposed by the flow of data.

Allocation-sensitive scheduling Instruction scheduling interacts directly with register allocation (see § 4.3.1). When the allocator must spill a value, it inserts instructions, perturbing the schedule. When the scheduler moves an instruction, it changes the lifetime of the value defined by the instruction, and, possibly, of its operands.[12]

[12]If the operand is the last use of a value, then moving the instruction changes that value's lifetime. Moving the instruction past the definition of a value, or the last use of a value can change the possibilities for allocation.

Several authors have looked at methods for making the instruction scheduler sensitive to allocation. In general, these techniques either "under-allocate" to provide the scheduler with some room to maneuver, or they limit the range over which an instruction can move to prevent creation of excess demand for registers.

Goodman and Hsu built a system where the scheduler changed heuristics based on the local register pressure. In regions of low pressure, it used a more aggressive scheduler; when pressure exceeded a threshold value, it shifted to a scheduler that tried to minimize demand for registers [118].

Bradlee *et al.* built a system where the allocator used information about scheduling to make decisions about register use in each block [29]. The first phase computes a schedule length for each of several register limits. Next, a global register allocator uses this information to set a register limit for each block–in essence, the number of registers set aside for demand from sources other than the scheduler. Finally, the code is scheduled and local register allocation performed.

Brasier *et al.* take a more complex approach [30]. Their system first tries scheduling followed by allocation; if that produces spill code, it tries allocation followed by scheduling. If this latter approach succeeds without spilling, they back off slightly by constructing a graph that incorporates the constraints of both allocation passes, allocate that graph and schedule the resultant code.

Norris and Pollock created a graph-coloring register allocator that uses information about potential schedules to limit the impact of allocation on scheduling [186]. Their system adds edges to the allocator's interference graph that reflect the possible code reorderings allowed by the dependence graph used for scheduling. The allocator tries to color the constrained graph that results; if needed, it can remove some of the dependence-induced edges. This makes explicit the decision to accept a more constrained schedule to avoid introducing register spills.

Pinter followed a similar strategy [192]. Her system combined information from the dependence graph and the interference graph to build a graph that represents possible parallelism. The result, which she called a "parallel interference graph" is used to allocate registers without introducing dependences that would inhibit scheduling.

Motwani *et al.* attempt to combine the problems by using a form of α-β pruning [181]. Their algorithm computes a priority for each value as a function of its depth-based scheduling priority, $\mathcal{S}$, and its "register rank" $\mathcal{R}$–a priority that tries to minimize register pressure rather than schedule length. The priority is computed as $\alpha \times \mathcal{S} + \beta \times calR$. Their system then performs list scheduling, using this new priority, followed by local register allocation. Values for α and β are derived empirically, by running a large body of code through the scheduler and iterating over possible values subject to the usual constraints $0 \leq \alpha \leq 1$, $0 \leq \beta \leq 1$, and $\alpha + \beta = 1$.

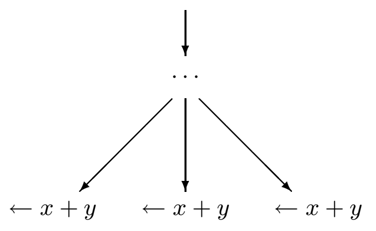
Scope Methods for instruction scheduling operate in many different scopes. Local and superlocal schedulers are quite common; list scheduling does well in these contexts. Kernel scheduling techniques like perfect pipelining and software pipelining consider an entire loop-nest as their scope; they are regional algorithms. Some global scheduling algorithms have been published–see, for example, Click's algorithm for placing instructions after his aggressive optimistic value numbering [66].

3.4.2 Consolidation

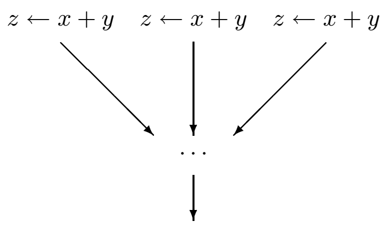
If several paths contain copies of a single computation, and those paths have a common prefix or suffix, the compiler may be able to replace the multiple copies with a single copy placed in the common subpath. If the common subpath precedes all the instances of the computation, and the operands of the computation are available at the end of the common subpath, then the compiler can move the computation to that point. The left side of Figure 9 depicts this situation. Assuming that neither x nor y is redefined, the compiler can move the evaluation of $x + y$ to the point immediately before the paths diverge. The right hand side shows the equivalent situation that can arise at a control-flow merge point. Again, assuming no redefinitions of x and y occur, the compiler can move the evaluation of $x + y$ to the merge point.

These transformations have been described in the literature in several forms. The left hand example, where code is lifted in the control-flow graph, is usually called *hoisting*. The right hand example has been called by several names, including *sinking* and *cross jumping*.

Hoisting Hoisting is a simple code motion technique that reduces code size rather than instructions executed. It reduces the number of static occurrences of the instruction rather than the dynamic execution count. The idea is quite simple. If (i) an expression is computed in several blocks, (ii) there exists a block that is



Opportunity for Hoisting



Opportunity for Sinking

Figure 9: Hoisting and Sinking

common ancestor, and (*iii*) the expression would yield the same value in the common ancestor, then the evaluation can be moved to the common ancestor. Two complications arise.

1. The compiler must discover the appropriate common ancestor. If a common ancestor exists, it must be a block that dominates all of the occurrences of the operation being hoisted. The operation should be placed at the end of the immediate predominator, if it will produce the same result. The immediate predominator is chosen because it lengthens live ranges less than other choices.

2. The compiler should avoid lengthening any paths through the code. To ensure that it does not lengthen any path, the compiler should only hoist an operation if it is used on every path leaving the new location. An expression is *very busy* at point p if and only if, along each path leaving p, it is used before it is killed. Thus, if a computation is very busy at the end of block b and b dominates every use that it reaches, the compiler can hoist the computation to the end of b.

A simple but effective algorithm for hoisting is shown in Figure 10. It relies on the computation of very busy expressions to identify candidates. It uses information about dominators to determine where to insert and where to remove evaluations.

Hoisting reduces code space. In applications like embedded systems, code space can be a critical issue. The hoisting algorithm shown in Figure 10 carefully respects the length of each path; it only hoists an operation that occurs on all paths. The set of opportunities might be increased by ignoring this path length constraint and hoisting operations that (1) are live on more than one path and (2) have operands that are available sat the insertion point.

Sinking Sinking is conceptually similar to hoisting. It eliminates instructions by moving them downward in the control flow graph past merge points. In principle, the compiler could perform data-flow analysis to determine that the expression is computed along all paths that reach the merge point, and that its operands are still available at the merge point. The best-known example of sinking is "cross-jumping".

To perform cross-jumping, the compiler follows a simple algorithm. At every label, the compiler looks back to branches that can target the label. If the same instruction appears before every branch that reaches the label, the instruction is moved across the branches and inserted at the label. In practice, cross jumping can be quite effective. It was implemented, as described above, in the Bliss-11 compiler [236]. If applied at link-time, it can discover repeated instruction sequences in the epilogue code for different procedures; in this application, it is sometimes called "tail merging". The CEDAR environment built at Xerox PARC in the early

```
1. calculate $\text{VeryBusy}(b)$, $\forall$ block $b$
2. $\forall\ e \in \text{VeryBusy}(b)$
   a. $\forall$ block $b_i$ that evaluates $e$,
      in a postorder traversal of the dominator tree
          if $b = idom(b_i)$, replace $e$ in $b_i$ with reference
   b. insert $e$ at end of $b$
```

Figure 10: A simple hoisting algorithm

1980's did this; Zellweger described this optimization in the context of her work on debugging optimized code [237].

Curiously, the literature describes hoisting and sinking quite differently. Hoisting is described as a high-level transformation, performed early in the optimization process and relying on global data-flow analysis to discover opportunities and prove safety. Sinking is discussed in terms of cross jumping, a technique applied late in compilation—after register allocation, for example. This is implicit in the description of the algorithm; it must discover that two machine instructions are textually identical before it can merge them by sinking them. The low-level approach ensures that cross-jumping does not lengthen any live-range's lifetime; at worst, an instruction is moved across a branch that neither defines nor kills a register. Undoubtedly, the low-level approach limits the applicability of cross-jumping. It makes the results very sensitive to the details of instruction ordering and the specific choices made in register allocation. More opportunities might be found by reformulating cross-jumping at a higher level and applying it earlier in compilation. We have not seen work that tried this approach.

Scope Hoisting and sinking are both regional transformations that operate over a limited region in the control-flow graph. In the low-level formulation typically used to describe cross-jumping, information gathering is performed regionally. In the higher-level formulation typically used to describe hoisting, the compiler performs global data-flow analysis to discover candidates. Both transformations can be applied interprocedurally, probably at link-time, to tailor procedure call and return sequences.

4 CODE REPLACEMENT

To translate a source-language program into target machine code, the compiler must create, for each construct in the source program, an equivalent sequence in the compiled code. While many many different target-code sequences are possible, the compiler usually has a very small set of translations (often just one) that it considers.[13] Since this selection is made in the front end, the compiler lacks global information; thus, the code sequences are often more general and less efficient than would be possible with more information. The techniques described in this section rewrite portions of a program to improve their run-time behavior, usually by taking into account more contextual information than is commonly available in the front end.

These techniques fall into three subcategories, based on the expected relationship between the original code and its replacement.

1. *replace with fewer or faster operations* – If the compiler can discover a faster, but equivalent, way to implement a sequence of operations, it should rewrite the code. Often, contextual knowledge allows the compiler to simplify operations. For example, if the compiler knows that x must have the value 2 when the statement $z \leftarrow x * 17$ executes, it can rewrite the statement as $z \leftarrow 34$ and avoid performing the multiply at run time.

2. *replace with the same operations* – Some transformations replace a sequence of operations with a reordered version of the original sequence. The new ordering might execute more quickly because it hides hardware latencies, improves memory system behavior, or improves contention for some resource. The new ordering might expose additional opportunities for optimization.

3. *replace with more or slower operations* – Sometimes, the compiler must replace a code fragment with an equivalent but slower sequence of operations to ensure correct execution. This situation often arises from the finite resources of the target machine; to simplify translation, the compiler assumes "enough" resources and, late in compilation, inserts additional code to map the "virtual" resource pool of translation onto the actual resources available in hardware.

Figure 11 shows this pictorially.

[13]Sometimes, the translation is quite careful. For example, Martin Richards' BCPL compiler selected an implementation for case statements by analyzing the set of case labels. It then chose between a linear search, a binary search, and a jump table based on the size of the set and its density [200].

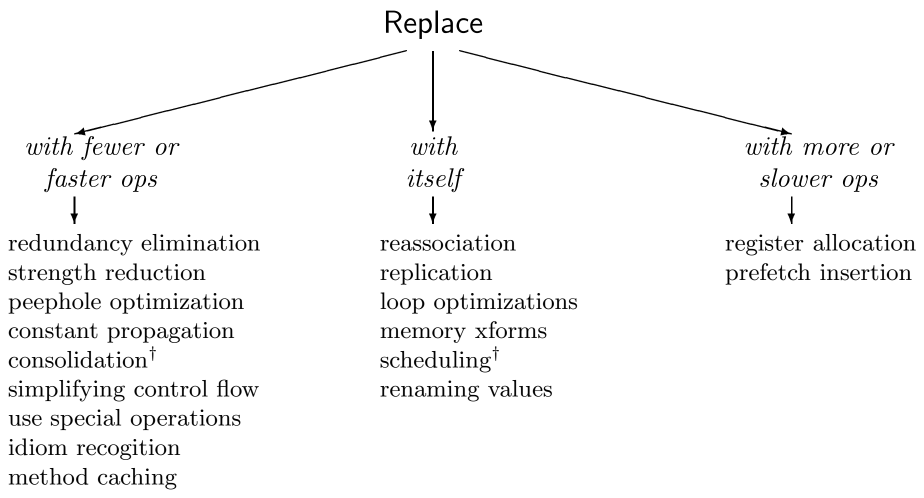


Figure 11: Hierarchy of Replace Transformations

4.1 Replacing with Fewer or Faster Operations

The obvious way to improve performance through code replacement is to rewrite a sequence of instructions with an equivalent sequence that is either shorter or faster. Many techniques achieve this goal.

4.1.1 Redundancy Elimination Discovering and eliminating redundancies is a powerful way of speeding up program execution. The basic idea is simple. Consider a path through the procedure. If it contains two evaluations of the expression x, and none of the variables used in x is redefined between the evaluations, then the second evaluation is redundant. Because the second evaluation of x must produce the same value, the compiler can replace it with a reference to the value produced by the first. The second evaluation is *redundant.* Of course, replacing the evaluation of x with a reference may require preserving the value of the earlier evaluation of x for a longer period than otherwise necessary. This often requires introduction of a compiler-generated temporary location (a temporary name).

Defining redundancy in this way works in the local case because a basic block contains only one path. In larger scopes, the definition must account for all the paths that can reach the second evaluation. Thus, in a larger scope, an evaluation of x at point p is redundant if and only if (*i*) each path from the entry to p contains an evaluation of x, and (*ii*) none of the constituent variables of x are redefined between the evaluation and p. If these conditions hold, we say that x is *available* at p. In effect, this definition gives the compiler a small theorem to prove for each instance of each expression. If the theorem can be proved, the evaluation is redundant and the compiler can eliminate it. If the theorem is false, the evaluation is necessary.

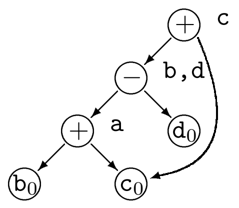
Many techniques for discovering and eliminating redundancy have appeared in the literature. We begin with local techniques, and progress to global ones. We will discuss loop-invariant code motion, which Cocke classified as a form of redundancy elimination [68], in Section 3.

Early work Floyd recognized the potential for improving expression evaluation by eliminating *common subexpressions*—expressions that can be replaced with a reference to some prior evaluation [106]. His 1961 paper presented an algorithm for generating good code for expressions. At the end, he mentions four possible improvements to his algorithm; one is detecting and eliminating redundancies. In 1965, Gear extended Floyd's technique to handle multiple expressions; he suggested that the only significant source of cross-expression redundancies would be address calculations [113]. As an example, he showed a stencil-like computation from the Gauss-Seidel algorithm.[14] Both Busam and England [40] and Cocke and Schwartz [68]

[14]A modern interpretation of Gear's observation might be that most common subexpressions arise from calculations introduced by the compiler to translate high-level language abstractions into assembly code. Features that can cause the compiler to introduce these overhead calculations include arrays, structures, objects, contexts, and procedure calls.

```
a ← b + c
b ← a - d
c ← b + c
d ← a - d
```

basic block



its dag

Figure 12: A basic block and its dag

discuss redundancy elimination.

Published algorithms for redundancy elimination fall into three major categories: hash-based techniques, data-flow techniques, and the partitioning method. The family of hash-based methods includes local, regional, and global techniques; the data-flow methods and the partitioning method have only been presented as global methods.

Hash-based techniques Perhaps the simplest mechanism for discovering and eliminating redundancies within a basic block is to construct a *directed acyclic graph* (DAG) for the block. A DAG is simply a tree with sharing—that is, each value is represented by exactly one node. A node may have multiple parents. Figure 12 shows a small basic block and its corresponding DAG. The DAG clearly shows that `b` and `d` have the same value. Aho, Sethi, and Ullman give a concise algorithm for building a DAG and regenerating the intermediate code from it [4, § 9.8]. Their 1970 paper is one of the earliest references to using DAGs in expression optimization [3].

The alternative to building a DAG is to use one of a family of hash-based techniques called "value-numbering". The original value numbering technique, invented by Balke in the late 1960's and described by Cocke and Schwartz [71], uses a linear scan of the basic block to construct a new name space where each distinct value is given a unique name. Since the constructed names are usually represented as small integers, they are referred to as "value-numbers". In these algorithms, an expression is redundant if and only if an expression with the same value number has already been evaluated.

The basic local algorithm, shown in Figure 13, is quite simple.[15] For each instruction in the block, it looks up the value numbers of the operands ($VN[y]$ and $VN[z]$ in the example). Next, it checks the hash table for a prior entry in the form $\langle VN[y]\text{op}VN[z]\rangle$; it the expression has already been entered, then the recomputation is replaced with a reference. Otherwise, a new value number is assigned to both the expression and the name on the left-hand side of the assignment. Extensions to perform constant folding and to account for commutativity and algebraic identities, like $x \cong x + 0$, $y \cong y \times 1$, and $z \cong \max(z, z)$, are straightforward.

```
for each instruction $a$ of the form "$x \leftarrow y$ op $z$" in block $B$
    $expr \leftarrow \langle VN[y]$ op $VN[z]\rangle$
    Simplify $expr$ if possible
    if $expr$ is found in the hash table with value number $v$
        $VN[x] \leftarrow v$
        Replace right-hand side of $a$ with $v$
    else
        $VN[x] \leftarrow$ next available value number
        Add $expr$ to the hash table with value number $VN[x]$
```

Figure 13: Local value numbering algorithm

[15]To keep the example simple, we have assumed an SSA-like name space. Without unique names, the algorithm must keep an extra table to map value numbers back into names; some replacements cannot be performed because a value has been killed [33].

The local algorithm can be adapted to operate over extended basic blocks and larger regions. The key to efficient operation on extended basic blocks is to capitalize on the tree structure of a set of extended blocks that are related because they share a common header. Simpson *et al.* show that using SSA and a scoped hash table [4] can both simplify the implementation of value numbering for extended blocks and increase the number of redundancies eliminated [33]. For larger regions, the algorithm must be extended to handle merge points in the control-flow graph; Morgan observed that this can be done cleanly during the construction of the SSA-form of the procedure [180]. Simpson *et al.* describe such an algorithm in more detail [33]. In general, transforming the code into SSA-form exposes more opportunities for redundancy elimination.

To further extend the scope of optimization, Simpson developed a hash-based, global, optimistic method for value numbering called SCC/VDCM [78]. It uses Tarjan's strongly-connected component finder to step through the SSA-graph for a procedure, applying the same basic hashing step as the local algorithm. When it encounters a cycle in the SSA-graph, it iterates over the cycle using two hash tables, one to record optimistic assumptions about values and another to record proven facts. Once value numbers have been assigned to all expressions, it uses a variant of *lazy code motion* [148] to transform the actual code.[16]

Cai and Paige have shown a technique for value numbering that avoids the use of hash-tables [42, 43]. It uses multiset discrimination to achieve the same results while avoiding the worst case behavior of hashing.

Data-flow techniques The second family of techniques uses data-flow analysis to discover redundancies. The mechanism for transforming the code to eliminate the redundancies varies.

Cocke appears to have presented the first paper that describes such a technique [68]. It used interval analysis to compute the set of expressions that are *available* on entry to each basic block—the set of *available expressions*. If an expression is available at a point where it is evaluated, the evaluation can be replaced with a reference. To accomplish the replacement, the compiler can insert a copy after each of the evaluations into a new name designated for this expression and replace the redundant computation with a reference to the new name. Cocke's method relies on lexical or "formal" identity; it does not take advantage of commutativity or algebraic identities. Nonetheless, this became the classic global method for solving the problem and one of the most commonly used examples for explaining data-flow analysis.

To improve on available expressions, Morel and Renvoise formulated "partial redundancy elimination" (PRE) [179]. An expression is partially redundant at point p if it is available on some, but not all, paths leading to p. Partial redundancy elimination works by (*i*) identifying partial redundancies, (*ii*) discovering those cases where a computation can be inserted to make a partially redundant expression fully redundant, (*iii*) inserting the requisite code, and (*iv*) replacing redundant expressions with references to earlier computations. The entire process is formulated as several sets of data-flow equations; when solved, these produce a set of insertions and deletions that can be done in a simple pass over the code. PRE is quite careful about code motion; it never lengthens a path through the code. Several authors have refined and improved the original formulation; Drechsler and Städel's work has a particularly practical bent [91]

Lazy code motion (LCM) is an improvement on PRE developed by Knoop, Rüthing, and Steffen [148]. It avoids some unnecessary code motion that PRE performs. It retains PRE's path-length property. It is the most refined of the descendants of PRE. Again, Drechsler and Städel present a slight reformulation that includes some practical improvements [92].

Partitioning Methods The third family of methods rely on a variant of Hopcroft's partitioning algorithm to prove expressions congruent. The congruence-finding phase is followed by a separate replacement phase. This application for partitioning was introduced by Alpern, Wegman, and Zadeck [13]. The congruence phase is optimistic; it finds some equivalences that neither the classic hash-based algorithms nor the data-flow methods can find. Alpern *et al.*'s formulation cannot capitalize on algebraic identities, or constant folding. Similarly, it cannot prove that two expressions with different textual operators are congruent, like $x + x$ and $2 \cdot x$. Alpern *et al.* suggest a replacement strategy based on dominators; Simpson showed that using PRE to perform post-partitioning replacement produces better results [33].

Click extended Alpern *et al.*'s partitioning framework [65]. His version combines the partitioning framework with Wegman and Zadeck's *sparse conditional constant propagation* technique; at the same time, it handles most of the algebraic identities found in the hash-based techniques.

[16] Once the scope grows large enough to handle cycles or loops, an off-line replacement technique is needed.

Comparisons Each of the techniques has its strengths and its weaknesses. There are several axes along which we can compare them.

Lexical vs. value identity The data-flow techniques, AVAIL, PRE, and LCM all operate on lexical identity. Briggs and Cooper showed some improvement from encoding value information directly into the name space before analysis [32]. In contrast, value numbering and the DAG construction build up a notion of value identity, discovering redundancy based on partial knowledge about the values of variables and expressions.

Code motion Both PRE and LCM perform some code motion in the process of eliminating partially redundant expressions. The other techniques do not deliberately move code.

Algebraic identities The hash-based techniques easily handle algebraic identities, while the data-flow techniques do not. The original partitioning algorithm, by Alpern *et al.*, did not handle algebraic identities; Click's extensions cure most of this problem.

The most powerful techniques from each category are: SCC/VDCM from hash-based value numbering, LCM from the data-flow algorithms, and Click's method from the partitioning techniques. A comparison of these three methods is worthwhile.

1. SCC/VDCM uses a variant of LCM to perform code motion. Thus, it captures the advantages of LCM.
2. Click's algorithm handles most of the identities that can be handled in SCC/VDCM. It does not include code motion; instead, he relied on a later scheduling phase to accomplish that task.
3. Both SCC/VDCM and Click's algorithm are optimistic; LCM is pessimistic.
4. Click's algorithm performs the combined constant propagation and unreachable code elimination of Wegman and Zadeck's Sparse Conditional Constant Propagation algorithm [227, 228] (see § 3.2.1). SCC/VDCM does the constant propagation but not the unreachable code elimination. LCM does none.
5. LCM is the fastest of these techniques. SCC/VDCM is next, followed by Click's method [78].

Scope DAG construction is a purely local technique. Versions of the hash-based value numbering algorithms operate locally, regionally, and globally. The data-flow techniques and the partitioning technique are inherently global.

4.1.2 Strength Reduction Operator strength reduction is a transformation that a compiler uses to replace costly (strong) instructions with cheaper (weaker) ones. For example, "$2 \times x$" is stronger than the equivalent expression "$x + x$." This weak form of strength reduction is widely used. Bernstein presents an algorithm for replacing integer multiply operations that have a known constant argument with a sequence of add, subtract, and multiply operations [27]. His work was performed in the context of the IBM PL.8 compiler. Fleischman encountered the same issues in trying to support fuzzy logic predicates [105]. Granlund implemented a similar technique in the Gnu C compiler [120].

A more powerful form of strength reduction replaces an iterated series of strong computations with an iterated series of weaker computations. The classic example replaces an integer multiply between a loop index variable and a constant with an add by the constant. This case arises routinely in loop-based array address calculations. Many other operations can be reduced in this manner. Allen, Cocke, and Kennedy provide a detailed catalog of such reductions [9].

Opportunities for strength reduction arise routinely from details that the compiler inserts as it converts from a source-level representation to a machine-level representation. To see this, consider the simple FORTRAN code fragment shown in Figure 14. The left column shows source code; the central column shows a low-level intermediate code version of the same loop. Notice the three instruction sequence that begins at the label L. The compiler inserted this code (with its multiply) as the expansion of `a(3,i)`. The right column shows the result of strength reduction. A second induction variable, $\mathtt{t}_3$, has been introduced to hold the running sum that is equivalent to `i` $\times$ `50`. The goal of a strength reduction algorithm is to automate this transformation.

Algorithms for strength reduction fall into three general categories.

```
dimension a(50,50)
...
sum = 0.0
do i = 1, 50
      sum = sum + a(3,i)
end
```

Source code

```
      sum ← 0.0
      i ← 1
      if (i > 50) goto E
L:    t1 ← i × 50
      t1 ← t1 + 3
      t2 ← iload ⟨@a+t1 ⟩
      sum ← sum + t2
      i ← i + 1
      if (i ≤ 50) goto L
E:    ...
```

Intermediate code

```
      sum ← 0.0
      i ← 1
      t3 ← 50
      if (i > 50) goto E
L:    t1 ← t3 + 3
      t2 ← iload ⟨@a+t1⟩
      sum ← sum + t2
      i ← i + 1
      t3 ← t3 + 50
      if (i ≤ 50) goto L
E:    ...
```

After strength reduction

Figure 14: Example code

1. A family of techniques have grown up around early work by Allen [7] and Cocke and Schwartz [71]. This includes work by Kennedy [140], Cocke and Kennedy [69], and Allen, Cocke, and Kennedy [9]. These algorithms examine the code's structure to discover loop nests and work from that knowledge to discover loop-invariant values (called *region constants*), induction variables, and instructions that can be reduced. Cocke and Markstein showed a transformation to strength reduce certain division and modulo operations [70]; their interest appears to come from calculations introduced when transforming code to improve its cache behavior. Markstein, Markstein, and Zadeck describe a sophisticated algorithm that combines strength reduction with reassociation [169].

2. Several methods have appeared that extend Morel and Renvoise's work on partial redundancy elimination to incorporate reduction of strength [87, 135, 133, 88, 149]. This work capitalizes on the code placement techniques developed for partial redundancy elimination to perform strength reduction. They incorporate all the strengths of work in data-flow based optimization, particularly with regard to the placement of inserted code. The strength of the data-flow techniques is that they require none of the control-flow analysis seen in the loop-based algorithms. However, this forces them to use a much simpler notion of region constant – only a simple literal constant can be classified as a region constant. Thus, they miss some opportunities. For example, they cannot detect a loop-invariant but non-constant value, like the induction variable of an outer loop.

3. A number of authors have looked at generalizations of strength reduction that work on set operations as well as classical arithmetic. Early first looked at generalizing strength reduction to operations on sets [93]; he called the transformation "iterator inversion." Fong picks up on that idea and generalized the process of finding induction variables [108] and applying strength reduction to set formers in SETL [107]. Paige and Schwartz present a generalized treatment of Early's ideas that they call "formal differentiation" [188].

Scope The weak form of strength reduction that replaces an individual computation with a short series of faster instructions is a local transformation. It may require global analysis to determine that the operands have appropriate properties, but the transformation simply replaces a single instruction with a short, faster sequence. The stronger form of strength reduction is inherently non-local, since it attacks computations performed repeatedly in a loop nest. The family of algorithms that follow from the work of Allen and Cocke are regional, although they may use global analysis to discover facts necessary to support the transformations. The data-flow techniques that build on partial redundancy elimination are global methods. They use global data-flow analysis to discover opportunities and to place instructions.

4.1.3 Peephole Optimization

Peephole optimization is a simple style of transformation that emerged as a response to the inefficiencies that can appear in the final code generated by a compiler. A classic peephole optimizer makes a pass over the code being compiled. At each point, it examines a small number of consecutive instructions, called its "peephole" or "window," and looks for sequences of instructions that can be

loadi offset(x) $\Rightarrow$ $\mathtt{r}_i$	loadi offset(x) $\Rightarrow$ $\mathtt{r}_i$
add $\mathtt{r}_{sp}$, $\mathtt{r}_i$ $\Rightarrow$ $\mathtt{r}_k$	add $\mathtt{r}_{sp}$, $\mathtt{r}_i$ $\Rightarrow$ $\mathtt{r}_k$
store $\mathtt{r}_j$ $\Rightarrow$ $\mathtt{r}_k$	store $\mathtt{r}_j$ $\Rightarrow$ $\mathtt{r}_k$
loadi offset(x) $\Rightarrow$ $\mathtt{r}_n$	move $\mathtt{r}_j$ $\Rightarrow$ $\mathtt{r}_m$
add $\mathtt{r}_{sp}$, $\mathtt{r}_n$ $\Rightarrow$ $\mathtt{r}_k$	add $\mathtt{r}_z$, $\mathtt{r}_m$ $\Rightarrow$ $\mathtt{r}_x$
load $\mathtt{r}_k$ $\Rightarrow$ $\mathtt{r}_m$	
add $\mathtt{r}_z$, $\mathtt{r}_m$ $\Rightarrow$ $\mathtt{r}_x$	
Original code	*After optimization*

Figure 15: Peephole optimization

improved. Early peephole optimizers worked from a small collection of patterns. McKeeman appears to have published the first description of a peephole optimizer in 1965 [173]. He described several patterns that his system would recognize and improve, including local constant folding, and a store followed by a load from the same location. Figure 15 shows this classic example. Notice that the peephole optimizer has inserted a copy from $\mathtt{r}_j$ into $\mathtt{r}_m$. While the use in the `add` instruction could have been rewritten to use $\mathtt{r}_m$ directly, the peephole optimizer cannot tell if this is the last use of $\mathtt{r}_m$. To be safe, it must copy the value rather than rewriting the reference.

Subsequent advances have automated the derivation of patterns [83, 145] and improved the pattern matching technology [238]. Together, these have increased the number of patterns that can be recognized and replaced.

The pattern matching techniques used to automate peephole optimization also lend themselves to improving local instruction selection. Fraser and Davidson describe a system that uses dynamic programming to search for better combinations of instructions [82]. They combine this expensive search with a table construction scheme to achieve efficiency; in effect, they "memoize" the dynamic programming engine.

Most peephole optimizers described in the literature consider physically adjacent instructions. In such a pass, the window consists of k instructions; it moves linearly through the code. This approach lends itself to relatively straightforward implementations; for example, the linear nature of the code might permit use of sub-linear time pattern matching methods like that of Knuth, Morris, and Pratt [151].

The alternative is to consider logically adjacent instructions—that is, instructions that are connected by use-definition chains or their equivalent. This requires global analysis to construct the use-definition chains. The window no longer contains uniquely ordered instructions; control-flow creates the situation where an instruction can have many immediate predecessors. In effect, each instruction is the root of a small dag that can be physically spread across the code. This complicates the pattern matching process and changes the set of patterns that actually match.

Logical and physical adjacency discover different kinds of improvement. Physical adjacency can eliminate inefficiencies in the local code sequence and replace a common sequence of instructions with a shorter, more powerful one. For example, the optimizer might easily replace an end-of-loop sequence with a single instruction with semantics like "decrement and branch if non-zero". Logical adjacency only examines instructions that are involved in computing a value. This may improve the handling of the value across distances; for example, this kind of pattern matching might do a better job of capitalizing on the use of address modes in memory optimizations or finding algebraic identities that are obscured by distance in the code.

Scope Peephole optimization based on physical adjacency is a local technique; in fact, it might be described as sub-local, since the arbitrarily limit on window size can prevent it from considering a full basic block. Using logical adjacency creates a global technique and exposes an entirely different set of combinations to improvement.

4.1.4 Constant Propagation

While the primary impact of constant propagation is to move the evaluation of an expression from run-time to compile-time (see § 3.2.1), it also has subtle replacement effects. If the compiler knows the value of one argument to an operation, it can often select a different, and more efficient, instruction to implement the operation.

In its simple form, the compiler might replace an `add` instruction with an `add immediate` when it knows

```
                                        a:  if n = 0 then go to b;
g(n,s) =  if n = 0 then s                   s := n × s
          else g(n − 1, n × s)              n := n − 1
                                            go to a;
                                        b:  ···

      Tail recursive                          Iterative
```

Figure 16: McCarthy's example

one operand. If the original code loaded the known value from memory, this eliminates the memory operation. If the original code kept the known value in a register, the immediate instruction may shorten the enregistered value's lifetime. This, in turn, reduces local demand for registers and may reduce spilling (see § 4.3.1).

In more complex examples, the compiler may discover values that determine the flow of control. Knowledge about loop bounds or the predicates on conditional branches can radically change the compiler's ability to improve code whose execution they determine. This kind of replacement can improve the precision of static analysis and it can enable other transformations, both replacement and code motion.

As a practical matter, the constant propagation phase provides a natural setting in which to perform weak strength reduction, such as converting an integer multiply into a sequence of shift, add, and subtract operations (see § 4.1.2).

Scope The code replacement effects of constant propagation are purely local. Constant propagation itself often relies on global analysis; the global effects are described in the Code Motion section of this paper (see § 3.2.1).

4.1.5 Consolidation

Consolidation be considered a form of replacement (see § 3.4.2). In consolidation, a small region of the program is rewritten to improve its code space efficiency. The primary effect is to move several identical copies of an instruction to a single location where one copy performs the same function. Alternatively, this can be viewed as inserting a copy that makes the others redundant.

4.1.6 Simplifying Control Flow

Many techniques try to simplify control flow. We classify several of them as replication, since they achieve their effect by duplicating portions of the original program. In this section we present two techniques that rewrite control flow operations to make them faster and simpler.

Eliminating Tail Recursion A particularly effective form of control-flow simplification can be used to convert certain kinds of recursion into iteration. If, one each path leaving a function, the final action is a self-recursive call, the compiler can transform the recursion into a simple branch back to the top of the function's body. This transformation, called "tail-recursion elimination," is quite old; Figure 16 is an example that McCarthy used to demonstrate the technique in 1963.

As an implementation matter, tail-recursion elimination replaces one abstraction for control with another, based on a detailed understanding of the mechanisms used to implement the procedure abstraction. Invoking the recursive call to return a value has a large overhead—creating and allocating an activation record, saving and restoring registers, and transferring control in and out of the function. Converting the call to a branch requires the introduction of a couple of explicit assignments. It halves the transfers of control and avoids the expense of creating an activation record. This technique has been used in many systems [216].

Some functions have a tail-recursive call near the end of each exiting path, but perform arithmetic on the result of the call. The compiler may be able to convert these calls to tail-recursions by introducing an additional parameter to accumulate the answer. The example in Figure 17 demonstrates this. The function on the left, *sum*, ends with a call to itself, but the call is an argument to the addition. On the right, the function has been transformed to accumulate its answer into n; this version is susceptible to tail-recursion elimination.

Branch Elimination After transformation, the code may contain branches that transfer control directly to other branches. This situation can arise when the optimizer either eliminates or moves the rest of the code from within a block. An entire chain of branches can be replaced with a single branch to the final target.

```
sum(l) =  if list = null then 0
          else head(l) + sum(tail(l))

          Not tail recursion
```

```
sum'(l, n) =  if l = null then
                  n ← 0
              else sum'(tail(l),head(l) + n)

              Transformed code
```

Figure 17: Accumulating parameters to create tail recursion

This reduces code size by eliminating instructions. It speeds execution by executing fewer instructions. A natural place to perform this important transformation is in the assembler rather than the compiler.[17]

4.1.7 Idiom Recognition

Several authors, working in different contexts, have suggested that the compiler recognize idioms that programmers use to encode particularly important computations. This is a practical response to the limitations of good compilers; some constructs are beyond the capabilities of a compiler to analyze and transform. If the idiom can be recognized efficiently, then the compiler can use a pre-optimized code fragment to implement it.

Perlis and Rugaber suggested that "idioms" play an important role in the use of programming languages, particularly those with reasonably powerful notation [190, 189]. Their paper catalogs common idioms used in APL programs. They suggest that language designers and implementors recognize the power of idioms and actively teach them to programmers. They discuss the computational importance of idioms and suggest that implementations should recognize idioms and provide particularly efficient translations for them.

Chatterjee's work on recognizing vector reductions can be considered an advanced form of idiom recognition. It recognizes certain cyclic patterns in the dependence graph and generates efficient vector code for them ... **

4.1.8 Method Caching

Most of the techniques mentioned in this paper use knowledge derived by analyzing the program to improve the sequence of instructions that implement some source-code construct. The "method caching" technique employed by Deutsch and Schiffman in the Smalltalk-80 system takes another approach. It relies on call-site specific type locality in Smalltalk programs to simplify the process of dispatching a message to some object.

Smalltalk-80 is an object-oriented programming system; early implementations of Smalltalk ran on custom-built hardware. Deutsch and Schiffman built one of the first Smalltalk-80 systems to run on a common commercial processor (the MC68020); it applied a handful of carefully-engineered techniques to improve performance.

In Smalltalk-80, the code that is invoked when an object receives a message is called the "method" for that message. To invoke a message, the Smalltalk system must determine the Object Class of the receiver and look up the message in the table of messages and methods associated with that Class. If the message is not found, the search climbs through the Class hierarchy until it either finds the appropriate inherited method or it fails by exhausting the possibilities in the hierarchy.[18] The overhead entailed in method lookup is a critical component in the performance of Smalltalk programs.

To improve method lookup, Deutsch and Schiffman introduced a single-element cache at each message invocation (or call site). The cache stored two values: the type of the receiver at the last message invocation from that site, and the code address of the method that was invoked. The message dispatch process was changed in the following simple way:

1. The first step compares the type of the current receiver against the stored type value from the previous invocation.

2. If the types match, the stored code pointer is invoked directly.

[17]Efficiency is quite important in this transformation. We recall one vendor who added this "feature" to their assembler. Unfortunately, they used a $O(n^2)$ algorithm to perform the transformation. Because the assembly files for production-sized programs can be quite large, this caused a shocking increase in compile times. A rapid "bug fix" replaced the transformation with another version that ran in roughly linear time; it restored rapid compilation!

[18]In principle, this is how the search operates. Techniques exist to make the process more efficient. Method caching will likely improve the behavior of even the best techniques.

3. If the types do not match, the standard message lookup procedure is invoked. After it has found the appropriate method, and before the method is actually invoked, the cached values are updated to reflect the new type and the corresponding method.

In their system, this simple scheme produced dramatic results, precisely because it capitalized on a simple property of the code. The programs in their study displayed significant temporal type locality, when measured individual call sites. By capitalizing on this simple but subtle property, Deutsch and Schiffman were able to avoid repeated, redundant lookups. The resulting code executes fewer operations, even though it includes more instructions.

This technique also makes an instructive point about static analysis. It does not require that all the executions of a given call site have the same receiver type—a property that might be determined with high-precision static analysis. Instead, it requires that successive calls sometimes have the same type—a property that is unlikely to be determined at compile-time. The authors recognized a dynamic property of the code that could lead to improvement, and taught the compiler to generate code that adaptively capitalized on it.

4.2 Replacing with the Same Operations

Some transformations literally replace a sequence of operations with a new copy of the same operations. The compiler might create multiple copies of a single basic block when the context that the block inherits along different paths differs significantly. By "cloning" the block, the compiler can create a place in the code where further tailoring can occur—through other transformations like redundancy elimination or strength reduction. This section includes transformations that literally replicate a sequence of instructions and others that replace a sequence with a different expression of the same computation.

In general, these transformations are applied for their secondary effects—the new opportunities that they create for other transformations. Some can directly improve the code; the direct effects of others can be negative. In applying these techniques, the compiler writer anticipates that the improvement from secondary effects outweighs any negative direct effects.

4.2.1 Algebraic Reassociation

Computer programs, even those thought of as "non-numerical," contain substantial amounts of arithmetic. Most non-trivial programs read and write memory locations; the address arithmetic required to locate objects in memory is almost unavoidable. This kind of arithmetic forms a large part of the "overhead" introduced by translation. Reducing these calculations can improve performance. In many cases, the compiler can use basic algebraic properties to speed up expression evaluation.

In considering these transformations, the compiler-designer must understand the idiosyncrasies of the arithmetic systems actually implemented on the underlying hardware. For example, the finite representations used in floating-point arithmetic provide a rough approximation of the real numbers that is neither truly associative nor distributive. Fortunately, address arithmetic and most other "overhead" calculations are performed using finite integers, where the hardware implementations preserve associativity and distributivity.[19]

The compiler can use commutativity, associativity, and distributivity to reorder expressions—a process sometimes called "reassociation". By itself, reassociation accomplishes little, if any, improvement. It may, however, make other transformations more effective. In particular, it can expose additional opportunities for redundancy elimination, loop-invariant code motion, and constant folding. It can also expose situations where two or more induction variables can be replaced with a single one (see § 4.2.3).

To see this, consider the simple expression shown in Figure 18. Three translations into three address code are shown. To see how this can change optimization, consider the case where x is 5 and z is 12, so that $x + y + z$ reduces to $17 + y$. The first and third versions of the three-address code obscure the opportunity; only the middle version has an addition with two constant operands. Using algebraic properties to reorder the computation can eliminate one of the two instructions by exposing the addition of 5 and 12 to constant folding.

The idea of exploiting associativity and distributivity to rearrange expressions is well known [109, 4]. Floyd mentioned it in 1961 as one of four areas where his algorithm could be improved [106]. Gear observed

[19]As long as the intermediate results stay within the finite range of the integer representation, algebraic manipulations do not change the results. In some cases, it may be desirable to use unsigned integers. This increases the range of addresses that can be represented. It also makes certain optimizations, like converting an integer multiply by a known constant value, simpler.

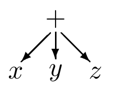
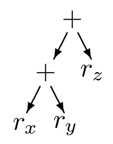
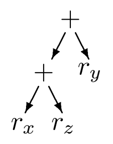
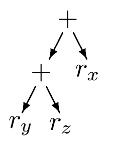
	Source code	*Low-level, three-address code*		
Code	$x + y + z$	$r_1 \leftarrow r_x + r_y$ $r_2 \leftarrow r_1 + r_z$	$r_1 \leftarrow r_x + r_z$ $r_2 \leftarrow r_1 + r_y$	$r_1 \leftarrow r_y + r_z$ $r_2 \leftarrow r_1 + r_x$
Tree				

Figure 18: Options for ordering simple expressions

that reordering expressions could expose a different set of redundancies and, in particular, that constant arguments should be grouped together to improve opportunities for constant folding [113]. Balke's classic local value numbering algorithm is easily extended to account for commutativity [71, 142]. Breuer described a technique for factoring expressions to improve detection of common subexpressions [31]. In the 1970 Symposium on Compiler Construction, papers by Bagwell, by Cocke, by Frailey, and by Aho, Sethi, and Ullman, all mention commutativity as an issue [21, 68, 109, 3]. Early attempts to capitalize on these properties concentrated on simplifying individual expressions. We know of two approaches to reassociation developed within IBM; both had the goal of exposing loop-invariant expressions. The FORTRAN H compiler used a front-end discipline for generating an array address expression as a sum of products and associating the sum to expose the loop-invariant parts [166, 209]. Medlock and Lowry mention in passing a late-stage transformation that tries to reorder expressions to expose more opportunities for constant-folding. Cocke and Markstein mention performing reassociation in the PL.8 optimizer, rather than in the front end [70].

In a chapter for an unpublished book by Allen, Rosen, and Zadeck, Markstein *et al.* describe a sophisticated algorithm for strength reduction that includes a form of reassociation [169]. Their algorithm attacks the problem on a loop-by-loop basis, working from inner to outer loops. In each loop, they perform some forward propagation and sort subexpressions into loop-variant and loop-invariant parts, hoisting the invariant parts. Their approach appears to be a development of earlier work within IBM. Other work by O'Brien *et al.* and Santhanam briefly describes what appear to be further developments of the Cocke and Markstein approach [187, 207].

Briggs and Cooper published a technique for using code shape to reorder expression operands based on global information [32]. Their algorithm uses SSA to assign a "rank" to each operand of an expression; the result's value is the maximum of its operands. The rank of an operand corresponds closely to the loop nesting level at which it was defined. For a binary operator, it designates the loop nesting level at which the expression is invariant. To reorder the expression, operands of a commutative operator are sorted into ascending rank order.

Scope While algebraic reassociation might work at any level, the algorithms discussed above work as regional or global techniques. The Cocke-Markstein method, and its descendants, focus on loop nests. The Briggs-Cooper approach uses global information to control the final order of expressions.

4.2.2 Replication Often, a compiler can use contextual knowledge to tailor code to the specific context in which it will execute, rather than including the general code that works in an arbitrary context. In fact, most of the techniques described in Section 4 operate by tailoring code fragments to their surrounding context. The primary impediments to this kind of code tailoring are (1) lack of specific compile-time knowledge about the values of variables and (2) the presence of multiple contexts in which the code must execute. Often, the compiler can ameliorate both situations by replicating code and adjusting the control flow structure to eliminate merge points. This idea has a long history in the literature. For example, Ershov proposed reducing procedure call overhead in this way in 1966 [97]. Various schemes for tailoring procedure linkages were described in the Allen-Cocke catalog [8]. The notion of replicating basic blocks appears to have been invented independently by at least three people, Urschler [223], Wegbreit [225], and Wegman [226].

Replication creates additional copies of the code, each of which executes in fewer contexts than the original code. It does not reduce the number of executions of the copied code; but, each copy executes in a more constrained, more defined context. Replication creates new points in the program where particular analytical facts can be true. After transformation, the code contains points that did not previously exist; at these points, new facts must be derived. By creating the right points, the compiler can derive strong sets of facts that lead to new opportunities for optimization. Essentially, the shape of the original code is chosen by the programmer for concise expression; the shape of the transformed code is chosen by the compiler to expose new facts to static analysis.

The results of replication before optimization can be dramatic; it can improve the results of a given optimizer by integer factors. However, it can also increase program size. Because compilers use algorithms with non-linear asymptotic complexities, this can lead to longer compile times.[20] Thus, the compiler must balance code space growth against possible improvements in performance.

Inline substitution Procedure integration, or inline substitution, is a form of replication that replaces a procedure call with the code that forms the body of the called procedure. Formal parameter names are replaced with references to the corresponding actual parameters and the name space of the called procedure is transformed to avoid conflicts. Inline substitution has several interesting effects.

1. It eliminates the code required to preserve and restore the caller's environment, as well as the code required to create and destroy a new environment for the called procedure. This saves instructions required by the system's linkage convention.

2. It creates a unique copy of the procedure body that can be tailored to the specific calling environment. This can create a place in the program where some statically derivable fact is true. For example, if two call sites invoke procedure x, and each passes it a different constant, static analysis will conclude that it receives no constant. Inlining separates the target of those calls and exposes both constants.

3. It exposes a larger context to classical intraprocedural optimization. Of particular interest is folding constant-valued parameters into the inlined procedure body [22].

Implementing inline substitution is relatively straightforward, as long as the intermediate representation can express all of the situations that arise. As Hecht pointed out, inlining can create code that has no expression in the original source language [129]. Hall describes several such situations [124, 76].

While implementing inline substitution is relatively simple, discovering when it is profitable can be complex. Many studies on the profitability of inline substitution have been published [210, 129, 201, 84, 177, 76, 57]. Unfortunately, profitability depends on the language, the heuristic used to guide inlining, and the specific optimizations performed by the compiler. For example, Davidson and Holler had good results with C compilers that performed little global optimization. In contrast, Hall's study of five global optimizing FORTRAN compilers had markedly mixed results; for a given FORTRAN program, the profitability of inline substitution varied widely from compiler to compiler.

Inline substitution plays a large role in optimization strategies for object-oriented languages [55]. To eliminate the overhead of procedure calls on every message dispatch, many authors have advocated inlining the method, or procedure body. If contextual information is available in the caller, this may allow the compiler to eliminate much of the conditional logic that selects the appropriate method. If not, it still presents the compiler with a larger scope for optimization; if message bodies are small, these effects can be quite powerful.

Procedure Cloning As an alternative to inline substitution, the compiler might produce multiple executable versions of a single procedure, each tailored to a different calling environment. Matching call sites with appropriate implementations of the called procedure should achieve some of the improvements of inline substitution without the full effects of merging the procedures. This idea has appeared in several guises in the literature; it has been implemented as a code-improving transformation in several research systems and in the Convex Applications Compiler.

[20]A classic example of this was found in the marketing materials for Perkin-Elmer's FORTRAN VIIz "Universal Optimizing Compiler", circa 1982. The compiler performed extensive inline substitution to improve the results of global optimization; they claimed to achieve improvements as large as a factor of four. The marketing script for the product, however, explicitly addresses the issue of compile-time speed with the following sentence: "We know of customers who have left compilations running all weekend, because they regarded the speed benefits as worthwhile."

Cloning is a conservative alternative to inline substitution. To understand its attraction, we must consider the problems that can arise with inline substitution.

1. Inlining can produce exponential growth in the size of the transformed code. While exponential growth is rarely reported, substantial increases in code size do occur. In their study of inlining with FORTRAN programs, Cooper, Hall, and Torczon showed increases in average procedure size by factors of two to eighteen[76]. Since the compiler traverses the code multiple times, this can lengthen compile time [76]. In a cloning algorithm, it is easy to limit overall growth.

2. Inlining merges two procedures. Whenever either procedure is changed, both must be recompiled. Thus, the compiler must remember all inlining decisions and use them as a partial basis for deciding what to compile in response to an editing change in the source code [39]. A careful implementation of cloning can avoid some recompilations by keeping around a non-specialized implementation of each cloned routine. Then, if an edit changes the environment inherited by a specialized clone in a way that invalidates the specialization, the compiler can simply re-link the call to reference the non-specialized implementation. Of course, the compiler might decide that it is profitable to create a new clone when recompilation analysis shows that conditions have changed.

3. Inlining creates code that compilers may not optimize well. The code created by automatic inlining can be quite different from that written by humans. Procedures are longer. Their name spaces are larger. The flow of control is not broken up by procedure calls that, for example, force the register allocator to spill many values. Hall's data shows that FORTRAN compilers are not as effective on this machine generated code. Because cloning simply creates specialized copies, the clones should have properties closer to those of the original code.

Cloning may ameliorate the negative aspects of inline substitution while creating opportunities for interprocedural specialization. For this reason, it can be considered a conservative alternative to inlining.

Clearly, cloning can be used to improve the quality of information available about the calling environment. For a forward interprocedural data-flow problem, the compiler can clone any procedure where the meet of multiple paths eliminates some facts from the set. For example, in interprocedural constant propagation, any procedure that receives different constant values from two different call sites might be cloned to create instances where those constants are exposed. The Convex Application Compiler does this; Metzger and Stroud report significant increases in the number of constants discovered and folded when forward cloning is used [178]. Cooper, Hall, and Kennedy give a general algorithm for cloning based on the values of forward data-flow sets [75].

The underlying idea—splitting nodes in the call graph to sharpen analysis—has appeared in several other contexts, including partial evaluation [37, 206], dynamic compilation for APL [134] and Smalltalk [86]. The SELF compiler used compile-time cloning to avoid losing static information at points where control flow merged; it would also clone methods dynamically based on run-time type information [56].

Block Cloning Just as procedure cloning can expose more interprocedural facts to the compiler's scrutiny, so, too, can block cloning increase the set of facts known inside a procedure. Wegman made this point in his thesis [226]. He presented an algorithm for "node distinction" to improve global analysis and optimization. He also discussed transformations where cloning might help and techniques to limit code growth from cloning.

In a single procedure, replication can produce improvement from several sources.

1. Eliminating multiple predecessors allows adjacent blocks to be merged. Creating larger basic blocks decreases the ratio of branches to useful instructions. It creates larger domains for strong, block-level optimizations. It provides the scheduler with more possibilities to fill idle cycles.

2. Eliminating points where control-flow paths merge can increase the effectiveness of data-flow analysis. At a merge, the compiler must assume that either path can be taken. If x has different known values on each path, replicating the block where the paths merge allows the compiler to create a block where each fact holds true. If knowing x's value allows additional constant folding on either path, that path is shortened.

The mechanics of replication are simple and highly dependent on the specific details of the compiler's intermediate representation. The difficult part of block replication lies in deciding which blocks to replicate and how many times to replicate them.

Mueller and Whalley presented results from using block cloning to eliminate conditional branches [184]. Their most aggressive technique eliminated five to seven percent of the instructions executed, while increasing code size by about fifty percent. Their measurements suggest that instruction caches can compensate for the increased code size; overall, they observed a minuscule decrease in the cache miss rate for the transformed programs.

Scope As this subsection has shown, replication can occur at any scope. Inline substitution and procedure cloning are interprocedural transformations; to perform them, the compiler must have access to several procedures at once. Block cloning is a regional transformation; a compiler performs it to modify the control-flow structure of the code in a way that may improve other optimizations. Little work has been done on local replication; the idea made little sense until the advent of machines that issue multiple instructions at each cycle. On some multiple-issue machines, particularly those with partitioned register sets, local replication may be an attractive alternative to certain inter-partition transfers.

4.2.3 Loop Transformations for Managing Memory

Loop optimizations have a long history in the literature [8]. Early loop transformations removed loop overhead; for example, unrolling a loop decreases the number of increments, tests, and branches required in its execution. Because some loop transformations change the relative ordering of iterations, they have proven to be powerful tools for automatic parallelization and memory hierarchy management. Bacon, Graham, and Sharp discuss loop transformations for parallelization [20]. Since our focus is uniprocessor compilation, we will describe loop transformations that target improving memory hierarchy performance; i.e., rearranging memory accesses such that as many as possible are satisfied by registers or the cache, thus reducing their effective cost. In the taxonomy, we can think of these transformations as replacements.

From the earliest days of compilation, compiler management of memory has been an issue. Initially, the concern was using the available memory effectively. Since memory was scarce, part of the compiler's job was to ensure that values were not retained beyond their usable lifetime (see § 6). More recently, the rise in relative memory latencies, coupled with a strong industrial focus on dense linear algebra codes, has produced a spate of papers that attack the problem of improving the behavior of array computations on machines with cache memories. In general, these techniques work by manipulating the iteration spaces of multiply-nested FORTRAN-style DO-loops.

We present loop optimizations to improve the memory hierarchy in the following categories and order, motivated roughly by the order of application in a compiler targeting a uniprocessor with a cache hierarchy and registers: enabling transformations (induction variable optimizations, unswitching, and embedding & extraction), individual transformations to improve cache use and effectiveness (loop permutation, fusion, distribution, strip mining, tiling, array padding, prefetching, and skewing), combined transformation strategies, and loop transformations to improve register usage (unroll and unroll-and-jam).

Induction (or index) variable optimizations From the earliest FORTRAN compiler, loop induction variables have received careful treatment. (Loop induction variables are defined in Section 4.1.2.) Compilers assume that code inside a loop executes more often than code outside the loop; the overhead arithmetic associated with maintaining induction variables falls largely inside the loop. Loop optimizations and strength reduction can produce code that includes many related induction variables inside a single loop; if the compiler can rewrite the loop to reduce the number of induction variables, it can remove the overhead operations required to keep their values current. In addition, dependence analysis tests reuse and independence of array accesses subscripted by induction variables. Two scalar transformations on induction variables are particularly important.

1. The compiler should recognize related induction variables. For induction variables that always have the same value, a carefully chosen global redundancy elimination pass, like SCC/VDCM [78], can discover and eliminate the duplication (see § 4.1.1). Reassociation may improve the results by exposing similarities (see § 4.2.1).

2. The compiler should recognize when it has eliminated all uses of an induction variable. If all uses

are gone, useless code elimination should remove the induction variable and its overhead. Sometimes, however, optimization creates a loop where the sole remaining use of an induction variable is in a control-flow test. In this case, the compiler should attempt to perform *linear function test replacement.* Here, the compiler attempts to replace the use of an induction variable in a control-flow test with an equivalent test on some other induction variable whose value is a linear function of the original induction variable. Allen, Cocke, and Kennedy give an algorithm for linear function test replacement [9].

Information about interprocedural uses of the induction variable may be important. Programmers often pass induction variables to functions that do not modify them. For example, the function may generate debugging output that contains the loop induction variable to improve its readability. Not knowing how the induction variable is treated inside the called procedure can inhibit the compiler's ability to eliminate it.

Loop unswitching When a loop contains invariant control flow, such as an `if-then-else` controlled by a loop-invariant expression, the compiler can often move the control-flow operations outside the loop. The result is code that determines which path will be taken, followed by a copy of the loop for each control-flow choice. Each independent copy of the loop is optimized to exclude the untaken paths. This structure eliminates some branches from inside the loop and should increase the basic block size within the loop, as illustrated by the following example.

```
do i = 1 to 100                              if (flag) then
   c(i) = a(i) + b(i)          ⇒                do i = 1 to 100
   if (flag) then           unswitch            c(i) = a(i) + b(i)
      d(i) = 0                                  d(i) = 0
   end                                          end
                                             else
                                                do i = 1 to 100
                                                c(i) = a(i) + b(i)
                                                end
```

The principal benefits of unswitching are executing fewer instructions and, in particular, executing fewer branches. The cost is increased code size due to replication. However, a careful implementation should be able to ensure that no execution path grows longer as a result of unswitching.

Does loop unswitching enable other optimizations by providing a large block of straight-line code? If yes, should we mention this benefit?

Loop extraction & embedding These two transformations move a loop's control flow across a call site. Extraction pulls the iterative control flow from the called procedure into the calling procedure. Embedding pushes ithe iterative control flow from the caller into the callee [125]. These transformations can be used to bring together in a single procedure to enable loop transformations a set of loops that was originally spread over two or more procedures. The effect is to convert an interprocedural analysis and optimization problem into an intraprocedural problem.

```
foo(x,y,z)              foo(x,y,z)                foo(x,y,z)
  ...                     ...                       ...
  do i = ...              do i = ...                call bar(...)
    do j = ...              do j = ...
      call bar(...)           do k =...             bar(a,b,c)
      end                       call bar(...)         do i = ...
    end                         end                     do j = ...
                              end                         do k = ...
bar(a,b,c)                  end                             ...
  do k = ...                                                end
    ...                   bar(a,b,c)                      end
    end                     ...                         end

  Original                Extracted loop            Embedded loops
```

Hall *et al.* used extraction and embedding as part of a coherent code generation strategy for parallel machines. Their algorithm uses the transformations to gather loop headers in a single procedure where intraprocedural loop fusion (§ 4.2.3) and loop interchange (§ 4.2.3) can be applied to them. They suggest viewing extraction as a form of partial inlining and embedding as its dual. interchange use

Loop fusion Fusion, or jamming, takes two or more loops that run over equivalent iteration spaces and combines them into a single loop. Fusion is safe when it does not change the values used or defined by any of the statements in either loop. The formal definitions of safety and profitability for fusion are typically formulated in terms of data-dependence [97, 6, 1, 221, 224]. The following example shows a trivial fusion of two loops.

```
do i = 1 to n                               do i = 1 to n
   c(i) = a(i) + b(i)          =>              c(i) = a(i) + b(i)
   end                                         d(i) = a(i) * e(i)
                              fuse             end
do j = 1 to n
   d(j) = a(j) * e(j)
   end
```

The obvious and immediate benefit is a reduction in loop overhead. A more significant benefit occurs when fusion converts inter-loop reuse into intra-loop reuse. In the example, both loops read the value of `a(i)`. For sufficiently large values of `n`, `a` will be flushed from the cache between its use in the first loop and its use in the second loop. In the fused loop, the references occur much closer together in time, which increases the likelihood that the second reference will use a value already loaded in the cache. While the example is simple, the effect occurs in practice and has a notable impact on performance [144, 176].

In general, fusion can expose opportunities for capturing reuse in the memory system—either in cache or in a register. By reducing memory latency and, in some cases, eliminating loads and stores, the combined loop may execute faster. Selecting an optimal strategy for fusion in a loop-nest is NP-hard [144]; when fusion and interchange are combined, the problem becomes NP-complete [44]. interchange use

Fusion is also a form of code motion, since it consolidates control-flow operations and interleaves the execution of statements in the loop bodies. We place it here, in code replacement, for proximity to the other loop optimizations.

Loop distribution Distribution (or fission) is the opposite of fusion; in the above fusion example, distribution takes the two statements on the right in a single loop, and puts them into two distinct loops. More generally, loop distribution takes a loop, replicates its control one or more times, and places each statement from the original loop in one of the new loops. It is safe, if all the statements forming a cycle in the dependence graph end up in the same loop. Loop distribution was introduced by Muraoka and is a powerful tool for automatic vectorization [185, 11, 222, 143]. For uniprocessors, loop distribution is typically used to enable other loop transformations [137, 176, 175]. It is also used to limit the amount of data accessed within a single loop to less than the cache size [90]. In our taxonomy, this replacement introduces additional operations.

Loop permutation Loop permutation takes two or more perfectly nested loops and reorders the loops. A set of loops is perfectly nested if each loop is completely contained within the loops that surround it, and all statements unrelated to the loop control flow are contained within innermost loop. When only two perfectly nested loops are involved, the tranformation is often referred to as *loop interchange*.

Loop permutation has a dramatic effect on the order in which data is accessed and thus on the performance of the memory hierarchy. It is safe only if no data dependences are reversed [10]. In the uniprocessor environment, the primary impact of loop permutation is to order operations to improve their spatial and/or temporal locality [174, 155]. The goal of loop permutation should be to get as much reuse in the innermost loop as possible since the memory hierarchy is most likely to be able to exploit reuse when it is close together in time. The following simple example shows interchanging two loops when there are no dependences.

```
do i = 1 to 50                                   do j = 1 to 100
   do j = 1 to 100                  ⇒               do i = 1 to 50
      a(i,j) = b(i,j) * c(i,j)  interchange            a(i,j) = b(i,j) * c(i,j)
      end                                           end
   end                                           end
```

In the above example, Fortran's column-major storage of arrays results in consecutive elements of `a, b,` and `c` sharing cache lines in most architectures because cache line sizes are greater than a single element. The transformed loop nest thus iterates and uses elements for all the arrays on the same cache line in consecutive iterations of the inner `i` loop. This ordering makes these accesses likely to hit in cache (and registers). The original version instead has 100 iterations of the `j` loop between accesses to the same cache line, which makes cache and register reuse harder to attain.

Loop interchange also exposes parallelism [10, 233] and can improve page behavior [1]. A compiler for the TI Advanced Scientific Computer was among the first to implement loop interchange [72]. For pipelined operations, loop permutation can separate a definition and a use in time, allowing functional unit latency to be covered with other independent operations. interchange use

Strip mining Loop strip mining transforms the iteration space of a single loop into two loops: the new outer loop is called the *iterator* and steps between strips of the new inner loop [164]. It does not change the order of any of the computations in the loop. We show an example below.

```
do j = 1 to 100                                do j = 1 to 100
   do i = 1 to 50                 ⇒               do ii = 1 to 50, 9
      a(i,j) = b(i,j) * c(i)  strip mining           do i = ii to min(ii+8,50)
      end                                               a(i,j) = b(i,j) * c(i)
   end                                                  end
                                                     end
                                                  end
```

Strip mining is always legal, and clearly increases the loop overhead. With respect to our taxonomy, it is a replacement with more operations. For vector machines, it is used to match the number of parallelism iterations to the length of the vector registers [11]. Below, we show the combination of strip mining and loop permutation to perform loop tiling.

Loop tiling When dense matrix computations are too large to fit in cache, tiling which combines strip mining and loop interchange can reduce capacity misses.[21] Tiling creates a loop structure that reduces the total volume of data between reuses by moving reuses closer together in the iteration space (*in time*). Consider the following classic example of tiling matrix multiply. interchange use

```
do i = 1, n                                      do kk = 1, n, tk
   do k = 1, n                    ⇒                 do jj = 1, n, tj
      do j= 1, n            tiling matrix              do i = 1, n
         z(j,i) = z(j,i) +     multiply                   do k = kk, min(kk+tk-1,n)
                  x(k,i)* y(j,k)                             do j = jj, min(jj+tj-1,n)
         end                                                    z(j,i) = z(j,i) +
      end                                                                x(k,i) * y(j,k)
   end                                                          end
                                                             end
                                                          end
                                                       end
                                                    end
```

[21]Tiling is also called *blocking* in the literature, but we use tiling for clarity since blocking has many additional meanings.

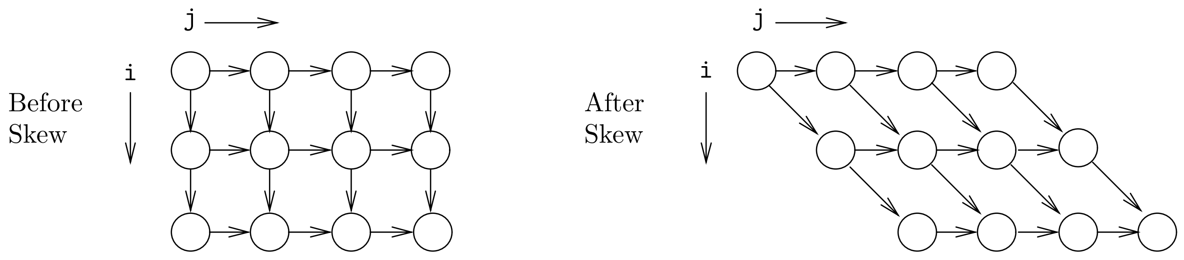


Figure 19: Effect of Loop Skew on Dependences and Iteration Space

Tiling only benefits loop nests with temporal reuse. In this example, the target reference for tiling is `y(j,k)`, which has loop-invariant temporal locality on the `i`-loop, i.e., it reuses the same location. Each iteration of the `i`-loop also accesses one row each of `x` and `z`. In the original nest, the `i`-loop accesses 2*`n` + $\texttt{n}^2$ elements per iteration. Between each reuse of an element of `y` there are accesses to `n` distinct elements of `z` on the `j`-loop, `n` elements of `x` on the `k`-loop, and $\texttt{n}^2$ `-` 1 elements of `y`. If the cache is not large enough to hold this many elements, then the reusable `y` data will be knocked out of the cache, and the program will have to repeat a costly memory access.

In the tiled version, the inner two loops are strip mined and then interchanged outside the `i`-loop. One iteration of the `i`-loop now accesses only `tk + tj + tk*tj` elements. Between reuse of an element of `y`, the `j` and `k`-loops access `tk` distinct elements of `x`, `tj` elements of `z`, and `tk * tj` elements of `y`. Given a nest where the loops are ordered based on the amount of reuse from innermost to outermost, the algorithms to create a tiled nest [234, 232, 112, 132, 50] pick a reference with loop-invariant temporal locality on an outerloop, strip mine all innerloops, and interchange the iterators outside the target loop in the same relative order. interchange use Of course, the interchanges must be legal.

The tiling algorithm must carefully choose the tile sizes [154, 98, 73, 202, 203, 204] because caches are a fixed size and have limited set-associativity of typically 1 or 2 (which means a given element may reside in only 1 or 2 locations). Lam et al. show that performance for a given problem size can vary wildly with tile size and develop an algorithm to select the largest square tiles without self interference in the cache [154]. Self interference occurs when different elements from the same array map to the same location. Several researchers have also proposed copying all the data in the tiled iterations into a contiguous piece of memory to avoid any conflicts in the cache mapping [154, 220, 98]. Coleman and McKinley use the Euclidian algorithm to select rectangular tiles that use more of the cache than square tiles and present an algorithm that is fast enough perform at execution time when the problem size and cache organization are always known. Rivera and Tseng extend and combine tile size selection with padding (see data transformations) [203, 204]. They achieve more consistent improvements than previous work, and see improvements on 3D stencil codes for which other techniques fail.

Loop Skewing Loop skewing is a transformation that changes the shape of the iteration space which, combined with loop interchange, exposes parallelism across a wavefront [156, 235, 132]. Lamport first published interchange uses the *hyperplane* method (widely known as the *wavefront* method), but used simple tests, which limited its applicability. Wolfe developed dependence tests and code generation for profitable loop skewing as a distinct transformation, and showed how to combine it with existing loop interchange algorithms to implement the wavefront method. We introduce it here because several researchers use skewing in the unimodular or related frameworks to improve memory hierarchy performance [231, 162, 64, 136].

Loop skewing is applied to a pair of perfectly nested loops that both carry dependences, even after loop interchange. Loop skewing adjusts the iteration space of these loops by shifting the work per iteration, changing the shape of the iteration space from a rectangle to a parallelogram. Figure 19 illustrates dependences in an iteration space before and after skewing. Skewing changes dependence distances for the inner loop so that all dependences are carried on the outer loop after loop interchange. The inner loop can then be safely parallelized. Skewing alone disrupts spatial locality, but combining it with loop permutation, tiling, or data copying can improve uniprocessor locality.

Data transformations So far in this section, we have assumed that the order in which array elements are stored is fixed by the language definition, which is true for Fortran and C, which store arrays in column-major and row-major order respectively. To improve data locality for uniprocessors and shared-memory multiprocessors, compiler researchers have proposed lifting this restriction, and languages such as Java do not require any particular storage model.[22] For the fixed storage models, the compiler emits fast code to translate a given array element into an address. Given an arbitrary storage map, the translation problem is harder and the resulting address calculation may be much slower. For this reason, researchers have mostly restricted themselves to dense storage maps where a single, arbitrary array dimension or diagonal is selected for contiguous storage and calculating the addresses is fast [64, 136].

Rivera and Tseng eliminate group interference in the cache from one or more arrays by simply changing the base storage offset [202] to remove mapping conflicts (which are described in the tiling section above.) Ding and Kennedy use *data regrouping* to combine two or more arrays into one to increase spatial locality [89, 90]. Consider the following.

```
do j = 1 to 100                                      do i = 1 to 100
   do i = 1 to 100                   =>                 do j = 1 to 100
      a(i,j) = b(index(i),j)                               a(i,j) = BC(1,index(i),j)
               * c(index(i),j)   data regrouping                    * BC(2,index(i),j)
      end                                                  end
   end                                                  end
```

In the code on the right, neither `b` nor `c` have spatial locality because of the index expression in the column index. In the regrouped code, `b = BC(1,index(i),j)` and `c = BC(2,index(i),j)`, and thus have spatial locality with each other on the inner loop. This transformation yields significant runtime improvements and can improve dense matrix locality as well.

Combining loop transformations To avoid exhaustively exploring the loop transformation space, a number of researchers have developed frameworks for deriving combinations of loop transformations to improve locality or parallelism. One advantage of these approaches is that they enable the compiler to evaluate different compound loop transformations without actually transforming the code.

Unimodular loop transformations combine loop permutation, reversal, skewing, and strip mining [132, 23, 231]. Wolf and Lam show how to use the dependence vector space for a loop nest to describe potential temporal and spatial reuse. These vectors form a matrix in which loop transformations become unimodular transformations on the matrix, and are legal when the vectors in the resulting matrix are lexicographically positive. Wolfe and Lam derive the final code generation phase directly from the transformed matrix. They also show how to intersect the potential reuse space and a localized reuse space obtained through transformations on the matrix to determine if a particular compound loop transformation yields good locality. Worst case, this algorithm is exponential in the number of loop nests, but a heuristic avoids this complexity in many cases.

Several other frameworks model the same or similar loop transformations [208, 162], but are more generally applicable than unimodular transformations. Cierniak and Li [64] extended the singular framework [162] to include data transformations (described above) and show further improvements in data locality, but did not include a practical algorithm for exploring the exponential search space. Kandemir et al. use a similar framework, but they include more data transformations and, more importantly, a practical algorithm for multiple loop nests with conflicting locality constraints [136].

Kelly and Pugh's reordering framework adds loop distribution, index set splitting, and statement reordering to loop interchange, skewing, and tiling [198, 137]. They show how to enumerate all the legal combinations of the transformations and generate correct code, but do not suggest how to choose among the options. interchange use

Ding and Kennedy use data regrouping combined with loop fusion, distribution, and interchange [89, 90]. Their experiments used Kelly and Pugh's reordering system to generate correct code and significantly interchange use

[22]Compiling for distributed-memory machines requires such a flexible storage model, but that issue is outside our scope discussion.

improve cache miss rates.

McKinley et al. compute temporal and spatial reuse of cache lines using the dependence graph to generate loop organizations with good locality [176]. Their cost model drives the application of compound transformations consisting of loop permutation, loop fusion, loop distribution, and loop reversal. This approach differs from the above because the cost model drives the exploration of the search space, whereas the other models generate potential transformations and then compare them.

Prefetching **Would "mitigate/eliminate the effects of" be an improvement over tolerates?**

Prefetching tolerates memory latency by transferring data from memory into the cache before the program actually accesses it. Compiler prefetching uses dependence analysis to find affine array accesses amenable to prefetching. Consider the following example. Assume Fortran column-major layout, that a cache block holds 2 elements of arrays `a` and `b`, that neither `a` nor `b` are in the cache, and for simplicity, that `a(1,1)` and `b(1,1)` are aligned at the beginning of the cache block.

```
do j = 1 to 100                                   do j = 1 to 100
   do i = 1 to 50                  ⇒                 do i = 1 to 50, 2
      a(i,j) = b(i,j) + k       prefetch                prefetch(a(i+2,j))
      end                                               prefetch(b(i+2,j))
   end                                                  a(i,j) = b(i,j) + k
                                                        a(i+1,j) = b(i+1,j) + k
                                                     end
                                                  end
```

In the original version, every other access to `a` and `b` miss in the first level cache. In the prefetch version, the next block of `a` and `b` are fetched into the cache in parallel with the computation of the loop nest. Unrolling the loop nest by two enables the compiler to insert only one prefetch per block. It also increases the amount of computation in the nest, which helps to hide the latency of the prefetch. Of course, the memory system must have sufficient bandwidth, and support two or more outstanding memory requests at a time. The compiler algorithms for inserting prefetches also generate prefetches for the first pieces of data in a pre-header. Additionally, they determine how many lines ahead to prefetch based both on the amount of computation and on the latency from memory or higher levels of the cache hierarchy into the first level cache.

Porterfield introduced compiler prefetching in his thesis [195, 47]. Gornish et al. simultaneously developed a prefetching algorithm for multiprocessors and uniprocessors [119]. However, Mowry et al. developed the algorithm [182] that is actually used in production compilers that generate prefetches. Researchers have also developed prefetching algorithms for pointer chasing codes [168, 41], but they require deeper pointer variable analysis.

In our hierarchy, this replacement generates more instructions, with the goal that the resulting code will execute more quickly because it is more specialized to the architecture.

Loop unrolling Unrolling is the simplest loop optimization. It replicates the body of a loop one or more times, adjusts the induction variables so that each copy computes a different iteration, and modifies the increment and test code to decrease the number of trips through the unrolled loop. The following example shows a trivial loop unrolled by a factor of two.

```
do i = 1 to 100              unroll     do i = 1 to 100 by 2
   c(i) = a(i) + b(i)          ⇒           c(i) = a(i) + b(i)
   end                       by two        j    = i + 1
                                           c(j) = a(j) + b(j)
                                           end
```

Provided the end conditions are handled correctly, unrolling is always safe.

The prime benefit of unrolling is a reduction in loop overhead. In the example, when compared to the transformed loop, the original loop requires twice as many conditional branches to execute. Of course, code size has increased. If this destroys instruction cache locality, the reduction in loop overhead is unlikely to compensate for the lost cache locality.

Get clarification on the following paragraph.

Unrolling can also be used to eliminate copy instructions. If a loop contains one or more cycles of copy instructions, unrolling by the least common multiple of the cycle lengths and renaming values appropriately can eliminate the copies [139]. This is particularly useful in conjunction with other transformations, like the combination of scalar replacement (see § 3.3.2) and unroll-and-jam.

Unroll-and-jam Unroll-and-jam is a particularly effective combination of unrolling the outer loop and fusing the resulting copies of the inner loop. This transformation was known in the early 1970's [8]. It found aggressive application as part of Carr's work in the early 1990's. He used unroll-and-jam, along with scalar replacement, to convert memory bound loops into loops where computation and memory references are balanced [45, 49, 51]. Carr extended unroll-and-jam to handle triangular loops and trapezoidal loops. Figure 5 shows a simple example of the complete sequence of scalar replacement followed by unroll-and-jam.

Streaming [Meadows et al.] What is streaming?

Keith - Do you have a reference for this?

4.2.4 Scheduling Instruction scheduling reorders a sequence of instructions to better match the execution order of the instructions to the resources and latencies of the processor on which they will run. The primary effect of scheduling is to rearrange instructions, so we view it as a form of code motion (see § 3.4.1). However, more aggressive forms, like software pipelining, can significantly change the code.

In particular, software pipelining replicates portions of the loop both before and after the central loop body. This creates a prolog loop that "fills" the pipeline before reaching the "steady-state" behavior of the loop kernel, and an epilog loop that "empties" the pipeline of any computations that remain after the kernel finishes. Within the kernel, the loop may include multiple copies of an operation—each operating on a different iteration of the loop. This type of kernel scheduling performs replication; thus, we must view it not only as a form of code motion, but also as a form of replacement.

4.2.5 Renaming Values Experience suggests that the compiler should not be bound to the control-flow shape selected by the programmer. Many transformations reshape the control-flow graph and reposition individual operations within the control-flow graph by analyzing where, in the particular context of the program being compiled, the operations can legally execute.

A natural corollary to consider is rewriting the name space to expose more opportunities for improvement to the compiler. Code motion frees the compiler from the programmer's positioning of operations. Renaming frees the compiler from the name space created by the programmer's choice of names and the discipline, often simplistic, used by the front-end's translation scheme.

Several transformations rewrite the name space in which the computation is expressed. Changing the name space can expose new opportunities for improvement; similarly, it can hide opportunities. While work in this area has been less extensive and, certainly, less well-known, it is clear that renaming can have a strong impact on the results of compilation.

- Scalar replacement (see § 3.3.2) rewrites array references, which the compiler treats as inherently ambiguous, into scalar temporary names, which the compiler treats as precise. Register promotion has a similar effect. These transformations attack limited, but important, problems.
- Building SSA form rewrites the name space to encode information about definitions and uses directly into the name space. It creates opportunities for other transformations by ensuring that values are never killed; it can increase both redundancies (see § 4.1.1) and register pressure (see § 4.3.1).
- Value numbering succeeds by constructing an artificial name space based on small theorems about run-time values. The difference in effectiveness between techniques based on value identity and lexical identity is partially attributable to this difference in name space. It is one element that contributed to the improvements seen by Briggs and Cooper in their work on reassociation (see § 4.2.1).
- Register allocation also manipulates the name space of the program. However, allocation is performed

to map the compiler-produced code onto the hardware. Because it occurs late in compilation, it is not viewed as creating opportunities for other transformations (see § 4.3.1).

Each transformation addresses a different part of the renaming problem. Conversion to SSA form is widely recognized as mechanism for renaming that improves other values.

Scope Scalar replacement and register promotion are essentially regional techniques. Conversion to SSA form is a global transformation that requires global analysis to support it. Both value numbering and register allocation are performed with techniques that operate at each possible scope.

4.3 Replacing with Slower Operations

Sometimes, the compiler must replace an instruction sequence with one that will execute more slowly. In general, this results from imposing realistic constraints of the target machine onto code that has been transformed without consideration for the finite resources available for execution. The best example of this is register allocation—the process of mapping the value space of the computation onto the hardware register set of the machine.

The dominant paradigm for global register allocation is coloring from *virtual registers*. In this approach, the compiler optimistically behaves as if it has an unlimited set of registers available to hold values. Transformations ignore the possible limitations imposed by a finite register set and exclude from their consideration issues like the cost of memory operations required to preserve a value across regions where demand for registers is high. In this scheme, the allocator must insert the memory operations necessary to map the virtual register set onto the target machine's physical register set. When demand for registers exceeds supply, the allocator must insert STOREs and LOADs to *spill* values to memory. These memory operations result in slower, but correct, code.

4.3.1 Register Allocation and Assignment

As compiled code executes, it loads values from memory into registers, creates new values, and stores some values from registers into memory. During code generation, it must decide, for each instruction, which values should reside in registers. This introduces two problems. The compiler must allocate registers to values – that is, decide which values will occupy a register. It must also assign registers – that is, choose a specific physical register for each enregistered value. This distinction may seem artificial; the difference is important to understanding the strengths and weaknesses of the various approaches to managing registers. In the literature, the distinction between these problems is almost always lost; both are lumped together under the aegis of "register allocation".

Register allocation is a complex and intricate problem. Fast, optimal algorithms exist for restricted forms of the local problem [18, 213]. Realistic versions of the problem, including register pairs, clean and dirty registers, and multiple register partitions quickly become NP-complete, even in the local case [117, p. 204]. If the scope is expanded to include significant control flow, the problem becomes NP-complete [212].

Paradigms Many different approaches to allocation have been tried. We can group the algorithms by the underlying model that they use to represent the problem.

Distances One of the oldest schemes for register allocation makes its decisions based on the distance to next use for each value. The idea, due to Sheldon Best in 1955, is quite simple [18]. Whenever a value must be spilled, choose the value referenced farthest in the future. (This is recognizable as Belady's MIN algorithm for optimal page replacement from 1965 [24].) Backus recalls that Best had strong arguments for the algorithm's optimality on single basic blocks. It is optimal as long as spills have the same cost. If some values require stores and others do not, then the arguments for optimality fall apart. Harrison described an extension of these ideas to larger scopes [127]. Hanson and Fraser used Best's scheme as the basis for their allocator in *lcc.* [110]. Liberatore *et al.* show a simple improvement to Best's algorithm that achieves a bounded distance from optimal local allocation [163].

Usage Counts Freiburghouse proposed a simple local allocation scheme based on usage counts [111]. He approached allocation as a problem to be solved in conjunction with redundancy elimination (see § 4.1.1). He modified his redundancy elimination technique to collect, for each value, a count of the number of times it is used in the block. Allocation decisions are based on this information. When a value must be spilled, the value with the fewest uses is chosen.

Freiburghouse compares his technique against other "replacement" criteria—Belady's MIN algorithm (Best's algorithm), least recently used replacement (LRU), and least recently loaded replacement (LRL). His data shows that the usage count technique is competitive. It inserts fewer loads than either LRU and LRL. It approaches the quality of MIN while, arguably, requiring somewhat less work at compile time.

Graph Coloring Lavrov first suggested that compilers solve storage allocation problems by using an analogy to graph coloring [159]. The idea was used for memory allocation in the ALPHA system [97]. The first complete register allocator based on the coloring paradigm was Chaitin's allocator in the PL.8 compiler [54, 52, 53]. The underlying idea is quite simple; the compiler constructs a graph that represents conflicts, or interferences, between values. Two values *interfere* if they are simultaneously live at some point in the program; obviously, two interfering values cannot share the same register. In the graph, the nodes represent values. Two nodes are connected by an edge if they interfere.

A k-coloring of a graph is an assignment of k colors (or integers) to the nodes of the graph such that no two nodes connected by an edge receive the same color (integer). For an arbitrary graph $\mathcal{G}$, discovering the smallest k such that $\mathcal{G}$ is k-colorable is NP-complete. Coloring allocators work by trying to discover a k-coloring for a specific k, usually choosing the number of hardware registers as the value for k. Two schools of coloring emerged in the early 1980's.

Coloring from registers: Chaitin's allocator operated inside IBM's PL.8 compiler. The allocator operated on code that was optimized for a machine with an unlimited number of virtual registers. The implicit assumption in the compiler is that any value that can reside in a register is compiled to reside in a virtual register. The problem of allocation, then, becomes mapping virtual registers onto physical registers and deciding, if necessary, which virtual registers to store in memory, or "spill", at each point in the code.

Chaitin-style allocators (1) build an interference graph, (2) use the graph to eliminate unnecessary copy instructions (often called coalescing or subsumption), and (3) attempt to construct, heuristically, a k-coloring for the graph, for $k =$ the number of hardware registers. If it fails to find a k-coloring, the allocator spills some values and tries again. This process is guaranteed to succeed eventually; in practice, it rarely requires more than three or four coloring passes [36]. Machine idiosyncrasies such as requirements for register pairs and alignment restrictions can be represented easily in the graph [34]. Many improvements have been discovered for Chaitin-style allocators, including better spilling techniques [26, 35, 25], stronger coalescing [114], and the use of clique separators to reduce space requirements [123].

Coloring from memory: Chow's allocator worked in a compiler which assumed that all values reside in memory [61, 62]. In this environment, allocation becomes an optimization that promotes values into registers; it improves run-time speed but is not necessary for correctness. The allocator assigns a priority to each node, builds a graph to represent interferences, and then assigns colors to nodes in priority order. The interference graph exists to arbitrate color assignment; it ensures the legality of each decision. The priority function is relatively simple; it estimates the savings in run-time realized by assigning the value to a register and divides the savings by the number of allocation units (basic blocks) in which the value is live [63].[23] When a value cannot be colored, the allocator splits the value into smaller regions and tries to color them.

Larus and Hilfinger built a Chow-style allocator in the SPUR LISP compiler. It used Chow's priority-based coloring but operated from a register-to-register intermediate representation [158].

Bin Packing Just as coloring allocators attack allocation and assignment by analogy to an NP-complete problem, another family of allocators work by analogy to bin packing. The first of these allocators was built for the BLISS-11 compiler [236]. The approach was carried forward into the PQCC project at Carnegie Mellon [161] and into a series of compilers for the DEC VAX computers [14]. Reportedly, the bin-packing allocators work quite well.

Hierarchical Decomposition Koblenz and Callahan describe a hierarchical global allocator that partitions the code into regions based on a decomposition of the control-flow graph [48]. It performs allocation and assignment within each region to produce a summary, and uses the summary as input to the allocation and assignment process in the surrounding region. Koblenz and Callahan introduce a preferencing mechanism to attack the problem of copies introduced at the boundaries between regions.

[23]This heuristic is strikingly similar to the area-based spill metrics proposed by Bernstein *et al.* [26].

The paper describes a Chaitin-style coloring mechanism for handling allocation and assignment within each region. The mechanism, however, allows the use of different allocators in different regions in a natural way. The hierarchical structure allows unrelated regions to be handled in parallel; on a multiprocessor machine like the Tera, this is attractive. The greatest potential problems appears to be the effectiveness of the preferencing scheme.

Scope A second criterion for grouping allocation algorithms is their scope. The characteristics of the problem change in different scopes; accordingly, the algorithms for local and global allocation use quite different metrics and achieve strikingly different results.

Local allocation Outstanding allocation algorithms exist for single basic blocks. Best's algorithm does an excellent job [19, 18]; in fact, many authors have argued that it produces optimal local allocations. Kennedy showed that the presence of multiple spill costs prevents optimal local allocation; nonetheless, MIN does an excellent job on single basic blocks. Methods by Horwitz *et al.* [131], Freiburghouse [111], Sethi and Ullman [213], and Aho and Johnson [2] all provide good allocations on a single basic block. Proebsting and Fischer present a variation on Sethi and Ullman's method that accounts for delayed loads and stores [196]. The coloring allocators have a particularly easy time assigning registers on single blocks because the interference graphs are constrained to interval graphs, whose minimal coloring can be found in linear time. Unfortunately, allocation is harder. If they must spill, the spill choice heuristics by both Chaitin and Chow are inferior to the best local methods when applied to a single block.

Regional allocation Several successful allocators have been built that approach the problem from a regional perspective. The allocator in the IBM FORTRAN H compiler used loop nests as the fundamental unit of allocation [209]. The Tera compiler relies on Koblenz and Callahan's hierarchical method; this lets the allocator use the parallelism available on the target machine. It also allows the compiler to use specialized allocators for regions that have unique needs, like software-pipelined loops. Knobe and Metzler tried a approach in the Compass compiler [147]. More recently, Hendren proposed a technique that focuses on individual loop nests and . . . . ⋆⋆

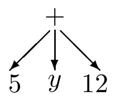
Global allocation The advantages of global register allocation over either local or regional allocation are quite simple. When allocation is performed over smaller scopes, code must be inserted at the interfaces between allocation regions to reconcile any differences between the regions. Both the allocation and the assignment can vary between adjacent regions; different allocations can require insertion of stores and loads; different assignments can require insertion of copies. A global allocator neatly sidesteps this problem. Global allocators, however, by their nature, sometimes generate unsatisfactory spills; for example, on a long block, Chaitin's method essentially devolves to use counts. While a strong local allocator might alleviate the specific problem, the problems introduced at block boundaries can overwhelm the improvement obtained with better local allocation.

In practice, most global allocators operate from the coloring paradigm [54]. The coloring allocators are well understood and produce reasonable results. The DEC compilers work from a bin-packing paradigm. The Tera compiler and the Compass compiler, with their hierarchical approaches, combine regional and global allocation. The Tera compiler uses coloring to allocate the higher levels in its hierarchy; it adds a preferencing mechanism that attempts to avoid copies at boundaries between the regions.

4.3.2 Prefetch Insertion To hide memory latency, several authors have suggested the use of "advisory" prefetch instructions [195, 183]. The idea is both simple and elegant. The compiler can tell the hardware, in advance, to fetch the cache line containing a given address from memory. If the cache line arrives before the actual reference executes, the system avoids losing time due to a cache miss. For this idea to work, the hardware must support advisory prefetch and the compiler must (1) estimate which references will cause cache misses and (2) insert prefetch instructions early enough to avoid latency.

Of course, prefetching a cache line may displace other locations from the cache. If the displaced item is referenced before the prefetched item, the net effect may slow down execution. In multiprocessor systems, cache coherence issues may further complicate the issue. ⋆⋆

Porterfield ?, Mowry et al.?, McIntosh?, others ?



	$t_1 \leftarrow 5 + y$	$t_1 \leftarrow y + 12$	$t_1 \leftarrow 5 + 12$	$t_1 \leftarrow 17 + y$
	$t_2 \leftarrow t_1 + 12$	$t_2 \leftarrow 5 + t_1$	$t_2 \leftarrow t_1 + y$	
The expression	*Leftmost*	*Rightmost*	*Reassociated*	*Folded*

Figure 20: Creating Opportunities

Scope Porterfield and Mowry both focused on loop nests. Their work is based upon data-dependence analysis. McIntosh looked explicitly at reuse that occurs across loop nests; he formulated a global framework for discovering such reuse and capitalizing on it.

5 Combining Transformations

Through most of this paper, and much of the literature, transformations are described as if they did not interact. This is done to simplify the design, study, and implementation of transformations; this "separation of concerns" goes back to the original FORTRAN compiler [19, 18]. However, transformations do interact and the interactions have a direct impact on the resulting code. Pollock was one of the first to characterize these interactions [194].

The interactions between two transformations, t_1 and t_2, can be characterized as follows:

- *t_1 and t_2 are independent* – Sometimes, two transformations are truly independent. For example, hoisting (see § 3.4.2) and useless code elimination (see § 3.1.1) do not interact—the order of application does not change the result. Hoisting a useless expression does not make it useful. Removing a useless expression prevents it from being hoisted, but the hoisted instruction would also have been removed. In either order, the result is the same.

- *t_1 enhances t_2* – Often, one transformation exposes opportunities for another transformation. Consider the impact that reassociation (see § 4.2.1) can have on constant propagation (see § 3.2.1), as shown in Figure 20. The expression, shown in the left column, has two natural translations corresponding to a leftmost treewalk and a rightmost treewalk (the second and third columns). In neither translation will constant propagation be able to fold together the two constants, because the sum $5 + 12$ is not computed. Reassociation can commute the addition to produce the code shown in the fourth column, which lets constant propagation fold the addition to produce the improved code in the fifth column.

- *t_1 inhibits t_2* – Sometimes, a transformation destroys opportunities for another transformation. Consider the expression $7 * y$. If the compiler knows that y is an unsigned integer, it can use a weak form of operator strength reduction (see § 4.1.2) to convert the expression into a shift and a subtraction $((y \gg 3) - y)$ [27]. On many machines, this is faster than an integer multiply. However, multiplication is commutative and subtraction is not, so this transformation removes opportunities for reassociation[24] (see § 4.2.1).

- *t_1 and t_2 are closely coupled* – Some transformations are directly and closely linked. Decisions made in t_1 change the set of possibilities for t_2 and *vice versa.* For example, instruction scheduling and register allocation exhibit this behavior. When the scheduler moves an instruction earlier or later in the code, it can change the lifetime of values created or used by the instruction. Similarly, when the allocator spills a value to memory, it inserts loads and stores that perturb the schedule.

The implementor of an optimizing compiler should understand these interactions. Unless two transformations are independent, the way in which they interact should influence the order in which they are applied. For example, reassociation should precede redundancy elimination, constant propagation, and conversion of

[24]The reassociation might group together other constants from the surrounding context, producing better code. Alternatively, it might break up the expression into a form where the multiply cannot be reduced to simpler operations. Without a great deal of context, predicting the tradeoff is impossible. In practice, converting multiplies to simpler operations inhibits reassociation enough to make a measurable difference [32].

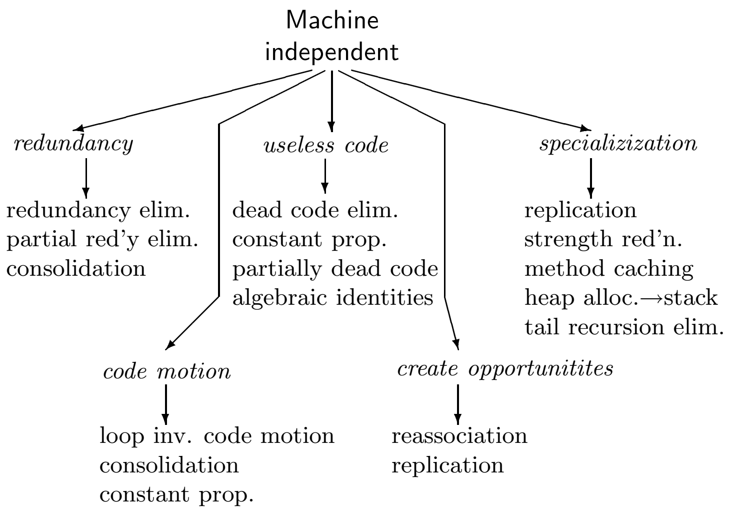


Figure 21: Machine-independent transformations

unsigned integer multiplies into simpler operations. Given a complete set of optimizations, a perfect ordering may not exist. For that reason, many optimizing compilers repeat some transformations more than once.

For some closely coupled transformations, no order of application yields the best solution. Solving the combined problem can lead to solutions that cannot be achieved by repeated application of the individual transformations, in any order. The problem with scheduling and allocation has long been recognized (see § 3.4.1). The same problem arises with constant propagation and dead code elimination. First Wegbreit [225], and then Wegman and Zadeck [227, 228] posed algorithms that solve the combined problem. These methods can prune paths that become non-executable due to constant values in the tests that control them. Click formalized this notion and presented an approach based on data-flow analysis that identifies when this situation exists and leads to an algorithmic solution for the combined problem [65, 67].

6 An Alternate Taxonomy

The taxonomy presented in Section 2.2 is intended to organize the space of transformations. It categorizes transformations by the way that they actually modify the code to improve it. Alternative taxonomies are possible; one that we have found to be particularly useful categorizes transformations according to the impact that they have on the compiled code. It deserves particular attention because it creates an informal checklist for the compiler writer. The framework also points out which techniques address the same problem; compiler writers probably gain more by attacking multiple problems than by attacking the same problem in multiple ways.

An effective compiler need not implement all these transformations. For example, the IBM FORTRAN H compiler, as enhanced by Scarborough and Kolsky, implemented only a few transformations [209]. Despite the paucity of its transformation repertoire, it generated code that, in many cases, could not be improved by hand. The transformations that it did implement were well chosen to attack the problems that mattered. The transformations that it did implement were applied thoroughly and consistently. Similar remarks apply to other compilers that have reputations as great optimizing compilers—notably the BLISS-32 compiler [236], the PL.8 compiler [16, 70] and the Deutsch/Schiffman Smalltalk-80 system [86].

The effects-based taxonomy divides the space of transformations by the impact that each one has on the compiled code and how it achieves that improvement. It assumes that there are a limited number of high-level effects that can actually improve the speed of compiled code. In the realm of machine independent transformations, we have identified five basic effects. Similarly, we classify machine dependent transformations into three basic effects. Figure 21 shows the taxonomy of machine-independent transformations, while Figure 22 shows the machine-dependent part of this taxonomy.

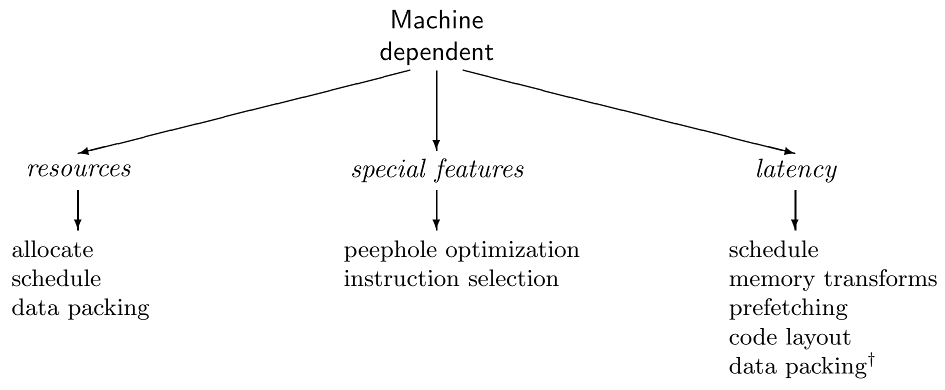


Figure 22: Machine-dependent transformations

Machine-independent transformations

1. *Replace a redundant computation with a reference.* If the compiler can replace an evaluation of expression x at point p with a reference to some previous evaluation of x that must always have the same value, the evaluation of x is redundant. If the evaluation takes more time than the reference, this transformation is profitable.

   The primary example is redundancy elimination (see § 4.1.1).

2. *Move an evaluation to a less frequently executed place.* If the compiler can move an evaluation of expression x to another point in the program where it will yield the same value and execute fewer times, the executable will run faster.

   Examples include loop-invariant code motion (see § 3.3.1) and partial redundancy elimination (see § 4.1.1).

3. *Specialize some general purpose code.* Compilers often produce code to handle the most general case of a programming language construct. If the compiler can prove that the full generality is not needed, it can often produce faster running code.

   Examples include constant propagation (see § 3.2.1), simplifying control flow (see § 4.1.6), strength reduction (see § 4.1.2), and peephole optimization (see § 4.1.3).

4. *Find useless code and eliminate it.* If the compiler can prove that evaluating expression x has no effect on the final state of the program—that is, it has no impact on correctness—then x is *useless.* Eliminating useless code can decrease execution time.

   Examples include dead code elimination (see § 3.1.1) and using algebraic identities (see § 3.1.2).

5. *Transform the code to expose other opportunities.* Some transformations reshape the code in a way that exposes new opportunities for other techniques. By themselves, these techniques do little or nothing to improve the running time of the code. In some cases, the direct effect of the transformation produces slower code. However, when followed by the right set of transformations, the resulting improvements can be dramatic.

   Examples include replication (see § 4.2.2), algebraic reassociation (see § 4.2.1), and some loop optimizations (see § 4.2.3).

Machine-dependent transformations

1. *Hide machine latencies.* Modern computer systems are complex. Functional units, branch units, and memory accesses typically have distinct latencies. To achieve a reasonably large fraction of peak speed, the compiler must arrange the code in a manner that hides, or manages, much of the latency.

Examples include instruction scheduling (see § 3.4.1), some loop optimizations (see § 4.2.3), and techniques used to influence the behavior of the memory hierarchy (see § 4.2.3).

2. *Manage limited machine resources.* The fact that computers have finite resources, such as registers, instruction issue slots, cache memories, and memory bandwidth, creates a class of problems that deal with allocation and assignment. Demand for these resources can be changed by the ordering of operations. Following Backus' 1956 advice, these problems are usually ignored until late in compilation, when they are attacked with heuristic techniques.

   The primary example is register allocation and assignment (see § 4.3.1).

3. *Select the most effective target machine instructions.* This takes two forms. The compiler can often replace a costly operation with a cheaper one. Alternatively, it can sometimes combine two or more operations into a single, more powerful operation. Both take detailed knowledge of the instruction set and the cost of various operations.

   This generally occurs as part of instruction selection, as idiom-recognition (see § 4.1.7) or as peephole optimization (see § 4.1.3)

A compiler writer may find this taxonomy particularly useful because it provides some insight into selecting transformations to include in an optimizer.

The base level of optimization should include a global transformation from each of these categories. For example, to "replace a redundant computation with a reference," the implementor might choose either partial redundancy elimination or SCC/VDCM. Because both fall into the same category of improvement, the implementor should expect significant overlap in the cases that they catch; thus, implementing a second technique would be wasteful, unless the optimizer already provides sufficient coverage of the other seven areas. The implementor might improve either form of redundancy elimination with algebraic reassociation (from "transform the code to expose other opportunities"). With partial redundancy elimination, consideration should be given to a renaming phase that encodes value information into the name space, as well [32].

Some transformations fit cleanly into this effects-based taxonomy, but not into the the Move/Replace taxonomy discussed in Sections 2.2, 3, and 4. For example, consider the following transformations.

Data packing — This transformation, explored by several authors in the late 1970's and early 1980's [1, 99, 221], rearranges the relative location in memory of data items. Reordering data items can have two distinct effects. First, placing items together that are referenced together in the code can improve cache behavior by increasing spatial locality. Second, if the compiler can determine that two data items are never simultaneously live, it can give them the same address and reduce the program's requirement for data-memory.[25]

Code layout — In virtual memory systems, the cost of a page fault is large relative to the costs that the compiler can directly manage. To improve working set behavior, some systems have resorted to a link-time transformation that reorders the code segments. Profile data on blocks and transfers of control can be used to place together code that executes together. This can shrink the working set by moving infrequently-executed code to infrequently referenced pages and decrease page faults by making better use of smaller working sets. Pettis and Hansen describe one such system [191]; others have been implemented since their paper appeared.

Both of these transformations manipulate the location of program objects in the run-time address space to achieve indirect savings in memory-operation latency. Since neither changes the code, except for the mapping of assembly-level labels to run-time addresses, neither fits into the instruction-oriented taxonomy presented in Section 2.2.

[25]Early work on data packing focused entirely on fitting the necessary data into the available memory, since locality was not an issue. For example, Ershov and his colleagues on the Alpha project used graph coloring techniques to determine when it was safe for two values to occupy the same memory location [95, 97, 96]. This application for coloring was first proposed by Lavrov in 1961 [159]; his paper also provides the intellectual underpinnings for graph-coloring register allocators.

7 Practical Treatment of Safety Issues

As discussed in Section 1.3, the primary task of any compiler is to produce an output program that has the same meaning as the input program. The goal of an optimizing compiler is to produce a particularly efficient implementation, measured in terms of time, space, or both. It must, however, preserve meaning.

In application, however, this rule is sometimes stretched to its limits. Many compilers have taken the position that they preserve meaning on programs that are well-behaved, in the sense that they execute to a "normal" completion. Programs that do not terminate, programs that produce run-time exceptions, and programs that use uninitialized variables are all occasionally treated in a cavalier fashion by an optimizing compiler.

- Code motion can move an erroneous computation earlier in the execution, so that the partial state left behind by an error differs from that left behind by the pre-transformation code.
- An uninitialized variable presents a particularly interesting quandary; Click points out that it may be convenient for the compiler to assume different values at different references to a single uninitialized variable [65].
- An obviously non-terminating loop may unexpectedly turn its immediate successors in the control-flow graph into unreachable code. If the loop relies on an exception handler to ensure its exit, it may fool all but the most careful analyzer. Spillman, for example, went to great trouble to model PL/I's ON-CONDITION facility, including the default handlers [215].

The FORTRAN standards have taken this approach one step further. They lay out a specific set of conventions that define a valid Fortran program. Any program that violates one of these strictures is deemed to be "non-standard conforming". On such a program, the compiler may produce any behavior that it desires. Since some of the strictures on the language are either subtle or obscure, this can lead to counter-intuitive behavior and programs that are difficult to debug. For example, the 1966 standard requires that the names of all global variables and all formal parameters to a procedure be distinct. Any program that does not meet this requirement is actually not a valid FORTRAN program. While the reason for the rule has been lost, the effect is clear; the compiler could replace the entire program with a simple return operation and claim a huge increase in efficiency.

Some authors have taken a rather cavalier approach to safety. For example, Holub wrote in the first edition of his book

> *You, as a compiler writer, must decide if it's worth the risk of doing this kind of optimization. It's difficult for the compiler to distinguish between the safe and dangerous cases, here. For example, many C compilers perform risky optimizations because the compiler writer has assumed that a C programmer can understand the problems and take steps to remedy them at the source code level. It's better to provide the maximum optimization, even if its dangerous, than to be conservative at the cost of less efficient code. A Pascal programmer may not have the same level of sophistication as a C programmer, however, so the better choice in this situation might be to avoid the risky optimization entirely or to require a special command-line switch to enable the optimization* [130, p. 677, 1st edition].

Undoubtedly, compilers have taken a similar attitude. This attitude should be discouraged; the compiler must preserve the meaning of the input program.[26] Holub softened his remarks considerably in the book's second edition.

On the other hand, safety is given a high priority in many systems. The work on range checking exists to lower the cost of ensuring that out-of-bounds references to subscripted variables are detected and handled appropriately (see § 3.3.3). A major application of type analysis is ensuring type-safe execution without the need for run-time checks. ¿From the perspective of safety, code-improving transformations exist to bring the overhead costs of ensuring safety below some threshold at which the user notices them (and turns them off).

[26]Without this restriction, the task of improving code becomes much easier. In fact, translation itself is unnecessary once meaning can be compromised.

8 SUMMARY

Many code transformations have been discussed in the literature, either individually, or in catalog-style papers [8, 20, 214]. This paper has taken a slightly different approach, by laying out a taxonomy for classifying transformations, and then populating the taxonomy with transformations taken from the literature. Our intent has been to provide the reader with a framework for thinking about code-improving transformations; the paper tries to describe each transformation at a depth that facilitates broad understanding. Detailed references are provided for more detailed study of individual transformations.

Acknowledgments

We are indebted to many of our colleagues in the compiler construction community for their insights, their references, and their recollections. In particular, Ken Kennedy and Peter Markstein have helped us track down the origin of ideas and techniques would otherwise have been lost. Devika Subramanian's comments on earlier versions of the paper have improved its organization, consistency, and presentation. Tim Harvey, Bob Morgan, L. Taylor Simpson, Phil Schielke, and F. Kenneth Zadeck assisted us by patiently answering questions, bringing work to our attention, and discussing important issues. Rajiv Gupta performed bibliographic wonders to track down some of the papers referenced in the bibliography. The folks in the Massively Scalar Compiler Project at Rice and at Tera have contributed to this effort with time, patience, and suggestions. This work has been supported by DARPA through US Army Contract DABT63-95-C-0115.

References

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, University of Illinois, Urbana-Champaign, 1978.

[2] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, Mar. 1976.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. A formal approach to code optimization. *SIGPLAN Notices*, 5(7):86–100, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compiler: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[5] A. Aiken and A. Nicolau. Optimal loop parallelization. *SIGPLAN Notices*, 23(7):308–317, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[6] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, 1972.

[7] F. E. Allen. Program optimization. *Annual Review in Automatic Programming*, pages 239–308, 1969.

[8] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, 1972.

[9] F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[10] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[11] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[12] R. Allen and K. Kennedy. *Advanced Compilation for Vector and Parallel Computers.* Morgan Kaufmann Publishers, Inc., *to appear*.

[13] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, Jan. 1988.

[14] P. Anklam, D. Cutler, R. Heinen, Jr., and M. D. MacLaren. *Engineering a Compiler: VAX-11 Code Generation and Optimization.* Digital Press, Bedford, Massachusetts, 1982.

[15] J. M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1992.

[16] M. A. Auslander and M. E. Hopkins. An overview of the PL.8 compiler. *SIGPLAN Notices*, 17(6):22–31, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[17] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. *SIGPLAN Notices*, 32(6):134–145, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[18] J. Backus. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[19] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, Feb. 1957.

[20] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformatiosn for high-performance computing. *ACM Computing Surveys*, *to appear.*

[21] J. T. Bagwell, Jr. Local optimizations. *SIGPLAN Notices*, 5(7):52–66, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[22] J. E. Ball. Predicting the effects of optimization on a procedure body. *SIGPLAN Notices*, 14(8):214–220, Aug. 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[23] U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.

[24] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, pages 78–101, 1966.

[25] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Notices*, 32(6):287–295, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[26] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[27] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[28] R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. *SIGPLAN Notices*, 32(7):159–170, July 1997.

[29] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 122–131, Santa Clara, California, 1991.

[30] T. S. Brasier, P. H. Sweany, S. J. Beaty, and S. Carr. CRAIG: a practical framework for combining instruction scheduling and register assignment. In *Parallel Architectures and Compilation Techniques (PACT '95)*, 1995.

[31] M. A. Breuer. Generation of optimal code for expressions via factorization. *Communications of the ACM*, pages 333–340, June 1969.

[32] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[33] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software–Practice and Experience*, 00(00), June 1997. Also available as a Technical Report From Center for Research on Parallel Computation, Rice University, number 95517-S.

[34] P. Briggs, K. D. Cooper, and L. Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems*, 1(1):3–13, Mar. 1992.

[35] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[36] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[37] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21, 1984.

[38] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *SIGPLAN Notices*, 21(7):162–175, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[39] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[40] V. A. Busam and D. E. Englund. Optimization of expressions in fortran. *Communications of the ACM*, pages 666–674, Dec. 1969.

[41] B. Cahoon and K. S. McKinley. Tolerating latency by prefetching Java objects. In *Workshop on Hardware Support for Objects and Microarchitectures for Java*, Austin, TX, Oct. 1999.

[42] J. Cai and R. Paige. "Look Ma, no hashing, and no arrays neither". In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, Orlando, Florida, Jan. 1991.

[43] J. Cai and R. Paige. Using multiset discriminiation to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1&2):189–228, 1995.

[44] D. Callahan. *A global approach to the detection of parallelism.* PhD thesis, Rice University, Houston, Texas, USA, May 1987.

[45] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[46] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *SIGPLAN Notices*, 21(7):152–161, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[47] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, Apr. 1991.

[48] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[49] S. Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, Department of Computer Science, Sept. 1992.

[50] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, Nov. 1992.

[51] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control-flow. *Software–Practice and Experience*, pages 51–77, 1994.

[52] G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[53] G. J. Chaitin. Register allocation and spilling via graph coloring. United States Patent 4,571,678, Feb. 1986.

[54] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, Jan. 1981.

[55] C. Chambers. Efficient implementation of object-oriented programming languages. Tutorial given at SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997.

[56] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *SIGPLAN Notices*, 24(7):146–160, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[57] P. P. Chang, S. A. Mahlke, W. y. Chen, and W.-M. W. Hwu. Profile-guided automatic inline expansion for c programs. *Software–Practice and Experience*, pages 349–369, May 1992.

[58] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[59] W.-N. Chin and E.-K. Goh. A reexamination of "optimization of array subscript range checks". *ACM Transactions on Programming Languages and Systems*, 17(2):217–227, Mar. 1995.

[60] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, Jan. 1993.

[61] F. Chow. *A Portable Machine-Independent Global Optimizer – Design and Measurements.* PhD thesis, Stanford University, Dec. 1983.

[62] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[63] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

[64] M. Cierniak and W. Li. Unifying data and control transformations for distriubed shared-memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 205–217, San Diego, CA, June 1995.

[65] C. Click. *Combining Analyses, Combining Optimizations.* PhD thesis, Rice University, May 1995.

[66] C. Click. Global code motion/global value numbering. *SIGPLAN Notices*, 30(6):246–257, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[67] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2), Mar. 1995.

[68] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[69] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), Nov. 1977.

[70] J. Cocke and P. Markstein. Measurement of program improvement algorithms. In *Proceedings of Information Processing 80.* North Holland Publishing Company, 1980.

[71] J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[72] W. L. Cohagan. Vector optimization for the acs. In *Advances in Neural Information Processing Systems*, pages 169–174, Princeton, NJ, Mar. 1973.

[73] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 279–290, La Jolla, CA, June 1995.

[74] K. D. Cooper, M. Hall, K. Kennedy, and L. Torczon. Interprocedural analysis and optimization. *Communications on Pure and Applied Mathematics*, pages 947–1003, 1995.

[75] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–118, Apr. 1993.

[76] K. D. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software – Practice and Experience*, 21(6):581–601, June 1991.

[77] K. D. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, Oct. 1986.

[78] K. D. Cooper and L. T. Simpson. Optimistic global value numbering. *In preparation.*

[79] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977.

[80] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[81] R. Cytron, A. Lowry, and F. K. Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, Florida, Jan. 1986.

[82] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, Oct. 1984.

[83] J. W. Davidson and C. W. Fraser. Automatic inference and fast interpretation of peephole optimization rules. *Software—Practice and Experience*, 17(11):801–812, Nov. 1987.

[84] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software – Practice and Experience*, 18(8):775–790, Aug. 1988.

[85] A. Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[86] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, Jan. 1984.

[87] D. M. Dhamdhere. On algorithms for operator strength reduction. *Communications of the ACM*, pages 311–312, May 1979.

[88] D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction. *International Journal of Computer Mathematics*, pages 1–14, 1989.

[89] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse.* PhD thesis, Dept. of Computer Science, Rice University, Jan. 2000.

[90] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of the Twelth Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, Aug. 1999.

[91] K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, Oct. 1988.

[92] K.-H. Drechsler and M. P. Stadel. A variation of Knoop, Rüthing, and Steffen's "lazy code motion". *SIGPLAN Notices*, pages 29–38, May 1993.

[93] J. Early. High level iterators and a method of automatically designing data structure representation. Technical Report ERL-M416, Computer Science Division, University of California, Berkeley, Feb. 1974.

[94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[95] A. P. Ershov. Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs. *Doklady Akademii Nauk S.S.S.R.*, 142(4), 1962. English translation in *Soviet Mathematics* 3:163–165, 1962.

[96] A. P. Ershov, L. L. Zmiyevskaya, R. D. Mishkovitch, and L. K. Trokhan. Economy and allocation of memory in the Alpha translator. In Ershov, editor, *The Alpha Automatic Programming System*, pages 161–196. Academic Press, 1971.

[97] A. P. Ershov[27] Alpha – an automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, Jan. 1966.

[98] K. Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, Sept. 1993.

[99] J. Fabri. Automatic storage optimization. *SIGPLAN Notices*, 14(8):83–91, Aug. 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction*.

[100] L. Feigen, D. Klappholz, R. Cassazza, and X. Xue. The revival transformation. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 421–434, Portland, Oregon, Jan. 1994.

[101] M. Felleisen. private communication. Discussion of observational equivalence and conditional observational equivalence, July 1994.

[102] C. N. Fischer and R. J. LeBlanc. Efficient implementation and optimization of run-time checking in Pascal. *SIGPLAN Notices*, 12(3):19–24, Mar. 1977. In *Proceedings of an ACM Conference on Language Design for Reliable Software*.

[103] C. N. Fischer and J. Richard J. LeBlanc. *Crafting a Compiler with C*. Benjamin/Cummings, 1991.

[104] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. *SIGPLAN Notices*, 19(6):37–47, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*.

[105] R. Fleischman. Supporing fuzzy logic selection predicates on a high throughput database system. Master's thesis, Massachusetts Institute of Technology, Feb. 1992.

[106] R. Floyd. An algorithm for coding efficient arithmetic expressions. *Communications of the ACM*, pages 42–51, Jan. 1961.

[107] A. C. Fong. Automatic improvement of programs in very high level languages. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 21–28, San Antonio, Texas, Jan. 1979.

[108] A. C. Fong and J. D. Ullman. Induction variables in very high level languages. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 104–112, Atlanta, Georgia, Jan. 1976.

[109] D. J. Frailey. Expression optimization using unary complement operators. *SIGPLAN Notices*, 5(7):67–85, July 1970. *Proceedings of a Symposium on Compiler Optimization*.

[110] C. W. Fraser and D. R. Hanson. Simple register spilling in a retargetable compiler. *Software – Practice and Experience*, 22(1):85–99, Jan. 1992.

[111] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, Nov. 1974.

[112] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.

[113] C. W. Gear. High speed compilation of efficient object code. *Communications of the ACM*, pages 483–488, Aug. 1965.

[27] The modern spelling is *Ershov*. Older variations include *Yershov* and *Eršov*.

[114] L. George and A. W. Appel. Iterated register coalescing. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 208–218, St. Petersburg Beach, Florida, Jan. 1996.

[115] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, 21(7):11–16, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[116] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. *SIGPLAN Notices*, 26(6):15–29, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[117] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[118] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, pages 442–452, July 1988.

[119] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 354–368, Amsterdam, The Netherlands, June 1990.

[120] T. Granlund. Private communication with p. briggs. Discussion of his work in building the routine `synth_mult` for the Gnu C Compiler, Sept. 1995.

[121] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. *SIGPLAN Notices*, 28(6):90–99, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[122] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, March–December 1993.

[123] R. Gupta, M. L. Soffa, and T. Steele. Register allocation via clique separators. *SIGPLAN Notices*, 25(7):264–274, July 1989. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[124] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, Apr. 1991.

[125] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, pages 424–434, Nov. 1991.

[126] D. R. Hanson. Simple code optimizations. *Software – Practice and Experience*, 13:745–763, 1983.

[127] W. Harrison. A class of register allocation algorithms. Technical report, IBM Thomas J. Watson Research Center, 1975.

[128] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, SE-3(3):243–250, May 1977.

[129] M. S. Hecht. *Flow Analysis of Computer Programs.* American Elsevier, North Holland, 1977.

[130] A. I. Holub. *Compiler Design in C.* Prentice Hall, 1990.

[131] L. Horwitz, R. Karp, R. Miller, and S. Winograd. Index register allocation. *Journal of the ACM*, pages 43–61, Jan. 1966.

[132] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, Jan. 1988.

[133] J. Issac and D. M. Dhamdhere. A composite algorithm for strength reduction and code movement. *International Journal of Computer and Information Sciences*, pages 243–273, 1980.

[134] R. Johnston. The dynamic incremental compiler of APL\3000. In *Proceedings of the APL '79 Conference*, pages 82–87. ACM, June 1979.

[135] S. Joshi and D. M. Dhamdhere. A composite algorithm for strength reduction. *International Journal of Computer Mathematics*, pages 21–44 (part 1) & 111–126 (part 2), 1982. Part 1 appeared in 11(1), Part 2 appeared in 11(2).

[136] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matix-based approach to the global locality optimization problem. In *The 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, Oct. 1998.

[137] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, Univ. of Maryland, College Park, Apr. 1993.

[138] K. Kennedy. private communication. Discussion of the origins of the def-use based dead code elimination algorithm and its application in several systems.

[139] K. Kennedy. *Global Flow Analysis and Register Allocation for Simple Code Structures*. PhD thesis, Courant Institute, New York University, Oct. 1971.

[140] K. Kennedy. Reduction in strength using hashed temporaries. SETL Newsletter 102, Courant Institute of Mathematical Sciences, New York University, Mar. 1973.

[141] K. Kennedy. Use-definition chains with applications. *Computer Languages*, 3:163–179, 1978.

[142] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnik and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[143] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, pages 407–416, New York, NY, Nov. 1990.

[144] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Bannerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of Languages and Compilers for Parallel Computers '93*, pages 301–321, Portland, Oregon, USA, Aug. 1993.

[145] R. R. Kessler. Peep – an architectural description driven peephole optimizer. *SIGPLAN Notices*, 19(6):106–110, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[146] G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, Oct. 1973.

[147] K. Knobe and A. Meltzer. Control tree based register allocation. Submitted to TOPLAS for publication, Nov. 1990.

[148] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[149] J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, Mar. 1993.

[150] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. *SIGPLAN Notices*, 29(6):147–158, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[151] D. E. Knuth, J. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, pages 323–350, 1977.

[152] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. Wiley, 1978.

[153] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[154] M. S. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, Apr. 1991.

[155] M. S. Lam and M. E. Wolf. Compilation techniques to achieve parallelism and locality. In *Proceedings of the DARPA Software Technology Conference*, pages 150–158, Apr. 1992.

[156] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, Feb. 1974.

[157] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[158] J. R. Larus and P. N. Hilfinger. Register allocation in the SPUR Lisp compiler. *SIGPLAN Notices*, 21(7):255–263, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[159] S. S. Lavrov. Store economy in closed operator schemes. *Journal of Computational Mathematics and Mathematical Physics*, 1(4):687–701, 1961. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3:810-828, 1962.

[160] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[161] B. W. Leverett, R. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production-quality compiler-compiler project. *IEEE Computer*, pages 38–49, Aug. 1980.

[162] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 1994.

[163] V. Liberatore, M. Farach, and U. Kremer. Hardness and algorithms for local register allocation. Technical Report LCSR-TR332, Rutgers University, June 1997.

[164] D. Loveman. Program improvement by source-to-source transformations. *Journal of the ACM*, 17(2):121–145, Jan. 1977.

[165] D. B. Loveman. Program improvement by source to source transformation. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 140–152, Atlanta, Georgia, Jan. 1976.

[166] E. Lowry and C. Medlock. Object code optimization. *Communications of the ACM*, pages 13–22, Jan. 1969.

[167] J. Lu and K. Cooper. Register promotion in c programs. *SIGPLAN Notices*, 32(6):308–319, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[168] C. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, Oct. 1996.

[169] P. W. Markstein, V. Markstein, and F. K. Zadeck. Reassociation and strength reduction. In *Optimization in Compilers.* ACM Press, *to appear.*

[170] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. *SIGPLAN Notices*, 17(6):114–119, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[171] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. *SIGPLAN Notices*, 26(6):1–14, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[172] N. McIntosh. *Compiler Support for Software Prefetching.* PhD thesis, Rice University, May 1998.

[173] W. McKeeman. Peephole optimization. *CACM*, 8(7), July 1965.

[174] K. S. McKinley. *Automatic and Interactive Parallelization.* PhD thesis, Rice University, May 1992.

[175] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, Aug. 1998.

[176] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[177] W. mei W. Hwu and P. P. Chang. Inline function expansion for compiling C programs. *SIGPLAN Notices*, 24(7):246–257, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[178] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, 2(1–4):213–232, March–December 1993.

[179] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb. 1979.

[180] R. Morgan. Private communication. Conversation about performing extended basic block optimizations during the renaming phase of the minimal SSA construction, Apr. 1994.

[181] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report 698, Courant Institute of Mathematical Sciences, New York University, July 1995.

[182] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, Oct. 1992.

[183] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Notices*, 27(9):62–75, Sept. 1992. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[184] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. *SIGPLAN Notices*, 27(7):322–330, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[185] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Feb. 1971. Report No. 71-424.

[186] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. *SIGPLAN Notices*, 29(6):266–277, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[187] K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski. Advanced compiler technology for the RISC System/6000 architecture. In *IBM RISC System/6000 Technology*. IBM Corporation, Armonk, New York, 1990.

[188] R. Paige and J. T. Schwartz. Reduction in strength of high level operations. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 58–71, Los Angeles, California, Jan. 1977.

[189] A. J. Perlis. Programming with idioms in APL. In *APL 79 Conference Proceedings*, pages 232–235. ACM, June 1979.

[190] A. J. Perlis and S. Rugaber. The APL idiom list. Technical Report 87, Yale University, Apr. 1977.

[191] K. Pettis and R. C. Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[192] S. S. Pinter. Register allocation with instruction scheduling: A new approach. *SIGPLAN Notices*, 28(6):248–257, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[193] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[194] L. L. Pollock. *An Approach to Incremental Compilation of Optimized Code.* PhD thesis, University of Pittsburgh, 1986.

[195] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, Department of Computer Science, May 1989.

[196] T. A. Proebsting and C. N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. *SIGPLAN Notices*, 26(6):256–267, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[197] R. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138. Spartan Books, NY, USA, Dec. 1959.

[198] W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 341–351, Cologne, Germany, June 1991.

[199] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 104–118, Los Angeles, California, Jan. 1977.

[200] M. Richards. The portability of the BCPL compiler. *Software–Practice and Experience*, 1(2):135–146, April–June 1971.

[201] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3), Aug. 1989.

[202] G. Rivera and C. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, June 1998.

[203] G. Rivera and C. Tseng. A comparison of compiler tiling algorithms. In *International Conference on Compiler Construction*, Amsterdam, The Netherlands, Mar. 1999.

[204] G. Rivera and C. Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of Supercomputing '00*, Dallas, TX, Nov. 2000.

[205] E. Ruf. Context-insensitive alias analysis reconsidered. *SIGPLAN Notices*, 30(6):13–22, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[206] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Proceedings of the PEPM '91 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, June 1991.

[207] V. Santhanam. Register reassociation in PA-RISC compilers. *Hewlett-Packard Journal*, pages 33–38, June 1992.

[208] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations (technical summary). In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, San Francisco, CA, June 1992.

[209] R. G. Scarborough and H. G. Kolsky. Improved optimization of FORTRAN object programs. *IBM Journal of Research and Development*, pages 660–676, Nov. 1980.

[210] R. Scheiffler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9), Sept. 1977.

[211] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, Aug. 1967.

[212] R. Sethi. Complete register allocation problems. *SIAM Journal of Computing*, 4(3):226–248, 1975.

[213] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(7):715–728, July 1970.

[214] R. L. Sites and D. R. Perkins. Universal p-code definition, version 0.2. Technical Report TR 78-CS-C29, Dept. of Applied Physics and Information Sciences, University of California, San Diego, Jan. 1979.

[215] T. C. Spillman. Exposing side-effects in a PL/1 optimizing compiler. In *Information Processing 71*, pages 376–381. North-Holland, Amsterdam, 1971.

[216] G. L. Steele, Jr. RABBIT: A compiler for SCHEME (A dialect of LISP) – A study in compiler optimization based on viewing LAMBDA as RENAME and PROCEDURE CALL as GOTO using the techniques of macro definition of control and environment structures, source-to-source transformation, procedure integration, and tail recursion. Technical report, Massachusetts Institute of Technology, May 1978.

[217] B. Steffen. Property oriented expansion. In *Proceedings of the International Static Analysis Symposium (SAS '96)*. Springer-Verlag, Sept. 1996.

[218] P. H. Sweany and S. J. Beaty. Dominator-path scheduling—a global scheduling method. *SIGMICRO Newsletter*, 23(12):260–263, Dec. 1992. In *Proceedings of the 25th Annual International Symposium on Microarchitecture.*

[219] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[220] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, pages 410–419, Portland, OR, Nov. 1993.

[221] K. O. Thabit. *Cache Management by the Compiler.* PhD thesis, Rice University, May 1982.

[222] R. A. Towle. *Control and Data Dependence for Program Transformation.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Mar. 1976.

[223] G. Urschler. Complete redundant expression elimination in flow diagrams. Technical Report RC4965, IBM Research, T.J. Watson Research Center, Aug. 1974.

[224] J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, pages 272–282, Salt Lake City, UT, Jan. 1984.

[225] B. Wegbreit. Property extraction in well-founded sets. *IEEE Transactions on Software Engineering*, 1(3), Sept. 1975.

[226] M. N. Wegman. *General and Efficient Methods for Global Code Improvement.* PhD thesis, University of California, Berkeley, CA, 1981.

[227] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, New Orleans, Louisiana, Jan. 1985.

[228] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

[229] J. Welsh. Economic range checks in Pascal. *Software–Practice and Experience*, 8:85–97, 1978.

[230] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. *SIGPLAN Notices*, 30(6):1–12, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[231] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[232] M. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, volume I, pages 75–81, Monterey, CA, Mar. 1989.

[233] M. J. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 536–543, St. Charles, IL, Aug. 1986.

[234] M. J. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, Dec. 1987. Extended version in a KAI Dec, 1987 TR.

[235] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, Aug. 1987.

[236] W. Wulf, R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. Programming Language Series. American Elsevier Publishing Company, 1975.

[237] P. Zellweger. An interactive high-level debugger for control-flow optimized programs. *SIGPLAN Notices*, pages 159–171, Aug. 1983. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, March 20–23, 1983.

[238] ZZZ. Unknown paper. Some reference is left undone. Need to track it down and fill it in!

GLOSSARY

basic block A basic block is a maximal length segment of straight-line code [19]. The key property of a basic block is that if any instruction in the block executes, every instruction executes (barring termination from an abnormal event like an arithmetic exception).

code shape This term, introduced in Wulf *et al.*'s book about the Bliss-11 compiler [236] is used to describe the cumulative effect of compile-time decisions about how to represent a given computation. Code shape issues include the specific details of the name space, the particular code sequences generated for source-level constructs like loops, case statements, and structure address computations.

control-flow graph To represent the flow of control within a single procedure, compilers often construct a control-flow graph (CFG). The nodes in this graph represent basic blocks; the edges represent possible transfers of control between blocks [19, 4].

data-flow analysis Data-flow analysis is compile-time reasoning about the run-time flow of values. This static analysis technique is often used to support optimization—proving the safety of a transformation, reasoning about its profitability, or simply locating opportunities for transforming the code.

Data-flow analysis works by posing the problem to be solved as a set of simultaneous equations over some domain of facts about the code. Many algorithms exist for solving such equations [142, 4]. Example problems include *available expressions, live variables,* and *constant propagation.*

dead code (See also *unreachable code* and *useless code*).

dependence analysis This term usually refers to techniques that analyze the possible values of array subscript expressions. The notion was introduced by Kuck [152]. Dependence analysis is an essential tool for understanding when two references to the same array may refer to the same value. Used in this way, it is a critical enabling technology for transformations that manage the memory hierarchy, that change the relative ordering of loop iterations, and that increase the amount of exposed parallelism in a program. See Maydan *et al.* [171] and Goff *et al.* [116] for a detailed introduction.

dominator In a control-flow graph, a node p *dominates* node q if and only if every path from the graph's unique entry node to q passes through p. In this case, p is q's dominator [197, 160]. Many nodes can dominate q; the closest such node is termed q's immediate dominator. By definition, p dominates itself.

Dominator-based methods generally operate on a tree that encodes the dominance relationship—if a dominates b, then a is b's ancestor in the tree.

dynamic This term is generally used to describe any analysis that incorporates information derived by executing the code being analyzed or transformed. For example, execution profile information is dynamic information. Dynamic (run-time) compilation has been used in many systems; examples include many APL compilers [134] and the Deutsch-Schiffman SMALLTALK system [86].

extended basic block (EBB) An extended basic block is a single basic block b, along with any block f that has a unique predecessor in the extended basic block. Thus, a single extended block consists of several basic blocks. The set of blocks related by the fact that they begin with the same initial block form a tree. Many local algorithms are easily extended to operate efficiently over extended basic blocks.

global An adjective used interchangeably with "intraprocedural."

graph coloring The coloring problem, stated simply, is to assign colors to the nodes of a graph so that no two adjacent nodes receive the same color. (Two nodes are adjacent *iff* an edge connects them.) The problem of determining if an arbitrary graph can be colored with k colors is NP-complete. The minimal k for which a coloring exists is often called the graph's *chromatic number.* Finding the chromatic number for an arbitrary graph is also NP-complete. These problems can be solved efficiently from some restricted classes of graphs.

instruction-level parallelism Many processors contain multiple independent functional units. To the extent that these units can operate concurrently, they are said to exhibit instruction-level parallelism. This term is usually used to contrast with the processor-level parallelism that occurs in multiprocessor systems.

intermediate language, or intermediate representation Multi-pass compilers typically use an internal notation to represent the code as it is translated and transformed. This lets the compiler avoid repeatedly re-parsing the source code. We call this internal notation an "intermediate language" or an "intermediate representation."

Intermediate languages vary in their level of abstraction; they range from abstract syntax trees that are quite similar to the source code through register transfer languages that are significantly lower in level than the instructions on the target machine. Intermediate representations vary in their complexity; some are structural forms that use restricted classes of graphs; others are simple linear forms that resemble program text. Many compilers use hybrid combinations. A common choice is to use a control-flow graph to represent inter-block transfers of control and a simple linear code to represent the contents of each block.

interprocedural An adjective meaning "across procedures." It is usually used to refer to an analysis or optimization that considers the whole program as its scope, although it can properly apply to a technique that looks at two or more procedures within a program. For example, inline substitution can be performed while only looking at the two procedures involved in the call site that is transformed.

intraprocedural An adjective meaning "within a single procedure". It is used interchangeably with "global".

liveness See "live variables."

live variables A value v is *live* at point p in a procedure if there exists a path from p to a use of v. Live variables is a classic global data-flow analysis problem that computes, for each block, the set of variables that are live on exit from the block.

$$\text{LiveIn}(b) = \text{Used}(b) \quad \cup_{s \in succ(b)} \quad (\text{LiveIn}(b) - \text{Killed}(b)) \tag{1}$$

local An adjective used to mean "within a single basic block" (see "basic block").

optimistic analysis Analysis algorithms are sometimes termed "optimistic". The term was introduced by Wegman and Zadeck [227, p. 293] to describe a property of the Reif-Lewis constant propagation algorithm [199]. Subsequently, the term was applied to many different techniques; Click formalized the notions of optimistic and pessimistic in his thesis [65].

Click defines optimistic analysis as one that constructs a set of facts top-down. It starts with the set of all possible facts as its initial condition and systematically applies a set of monotonic rules to determine which facts in the set are supported by the rules. The process must reach a fixed point; when it does, all remaining facts are true and no larger set of true facts exists for the rules.

pessimistic analysis Pessimistic analysis is the opposite of optimistic analysis (*see above*). This notion was formalized by Click in his thesis [65]. A pessimistic analysis begins with the empty set and repeatedly applies a set of monotonic rules to determine which facts should be added to the set, until it reaches a fixed point. Given a set of rules, the optimistic analysis will always produce a set of facts that is larger than or equal to the set produced by the pessimistic analysis.

post-dominator In a control-flow graph, a point p *post-dominates* a node q if and only if every path from q to an exit passes through p. By definition, p post-dominates itself. (See also *dominator.*)

reachability An instruction is reachable if there exists a valid execution path from the procedure's entry to the instruction. If it is not reachable, an instruction cannot execute and may be safely removed from the code emitted by the compiler.

This term should be used carefully; it is easy to confuse the notion of reachability, a simple graph-theoretic property, with the notion of a definition reaching some use, a more complex property that requires knowledge about both the path through the graph and the instructions along that path (see "reaching definitions").

reaching definitions The classic data-flow analysis problem used to build use-definition chains. For each use of a value in the code, the compiler calculates the set of definitions of the value that can reach the use.

$$\text{Reaches}(b) = \cup_{p \in pred(b)} \; (\text{Defined}(b) \; \cup \; (\text{Reaches}(b) - \text{Killed}(b)) \tag{2}$$

regional An adjective used to describe a technique that operates over a scope larger than a single block and smaller than the entire procedure.

register pressure While this term is rarely used in print, it arises in almost any verbal discussion of register allocation. It refers to the demand for registers at a specific point in the program.

static This term is generally used to describe any analysis that is performed at compile time. The code being analyzed is not actually executed. This can lead to some imprecision; for example, global data-flow analysis usually assumes that all paths through the procedure might execute, even though some may never execute. The opposite of static analysis is dynamic analysis.

static single assignment form Static single assignment form (SSA) is a particular intermediate representation [80]. The critical properties of SSA form are that (1) each name is defined exactly once, and (2) each use of a value refers to exactly one name. To transform a program into SSA form, the compiler renames variables to achieve the first property and then inserts special functions, called ϕ-functions to ensure the second property.

Two distinct ways of representing SSA form occur in practice. Some authors and compilers build a graph where each instruction or statement is a node and the edges represent the flow of values. Others simply adapt their existing intermediate representation to SSA form by adding a representation for the ϕ-function. In a low-level linear intermediate language, this usually looks like a n-ary ϕ instruction inserted directly into the code.

superlocal method An analysis or transformation method that operates over extended basic blocks. Often, these methods process together all of the extended basic blocks that share a common root.

uniprocessor A computer system that contains a single processor is said to be a "uniprocessor." The term is used in contrast with a "multiprocessor"—a computer system that contains more than one processor. Since many microprocessors contain multiple independent functional units, the term "processor" is usually taken to refer to all the units running from a single instruction stream.

unreachable code (See also *dead code*).

use-definition chains Many optimization have been formulated to operate over *use-definition chains*. The idea is simple; the compiler computes *reaching definitions* and then builds a data structure that links each use of a value with the definitions that can reach that use. In practice, the chains for a specific use might be a simple list of instructions. (Definition-use chains are also interesting; they can be built in the same fashion.)

useless code (See also *dead code*).

very busy expressions An expression is *very busy* at point p if, along every path leaving p, it is used before it is killed. To compute the set of expressions that are very busy on exit from block b, the compiler solves the following set of data-flow equations

$$\text{VeryBusy}(b) = \cap_{s \in succ(b)} \; \{\text{Used}(s) \cup (\text{VeryBusy}(s) - \text{Killed}(s))\} \tag{3}$$