Serializing Parallel Programs by Removing Redundant Computation

Michael D. Ernst

August 31, 1992
Revised August 21, 1994

Abstract

Programs often exhibit more parallelism than is actually available in the target architecture. This thesis introduces and evaluates three methods—*loop unrolling*, *loop common expression elimination*, and *loop differencing*—for automatically transforming a parallel algorithm into a less parallel one that takes advantage of only the parallelism available at run time. The resulting program performs less computation to produce its results; the running time is not just improved via second-order effects such as improving use of the memory hierarchy or reducing overhead (such optimizations can further improve performance). The asymptotic complexity is not usually reduced, but the constant factors can be lowered significantly, often by a factor of 4 or more. The basis for these methods is the detection of *loop common expressions*, or common subexpressions in different iterations of a parallel loop. The loop differencing method also permits computation of just the change in an expression from iteration to iteration.

We define the class of *generalized stencil computations*, in which loop common expressions can be easily found; each result combines w operands, so a naive implementation requires w operand evaluations and $w - 1$ combining operations per result. Unrolling and application of the two-phase common subexpression elimination algorithm, which we introduce and which significantly outperforms other common subexpression elimination algorithms, can reduce its cost to less than 2 operand evaluations and 3 combining operations per result. Loop common expression elimination decreases these costs to 1 and $\log w$, respectively; when combined with unrolling they drop to 1 operand evaluation and 4 combining operations per result. Loop differencing reduces the per-result costs to 2 operand evaluations and 2 combining operations. We discuss the tradeoffs among these techniques and when each should be applied.

We can achieve such speedups because, while the maximally parallel implementation of an algorithm achieves the greatest speedup on a parallel machine with sufficiently many processors, it may be inefficient when run on a machine with too few processors. Serial implementations, on the other hand, run faster on single-processor computers but often contain dependences which prevent parallelization. Our methods combine the efficiency of good serial algorithms with the ease of writing, reading, debugging, and detecting parallelism in high-level programs.

Our three methods are primarily applicable to MIMD and SIMD implementations of data-parallel languages when the data set size is larger than the number of processors (including uniprocessor implementations), but they can also improve the performance of parallel programs without serializing them. The methods may be applied as an optimization of a parallelizing compiler after a serial program's parallelism has been exposed, and they are also applicable to some purely serial programs which manipulate arrays or other structured data.

The techniques have been implemented, and preliminary timing results are reported. Real-world computations are used as examples throughout, and an appendix lists more potential applications.

This technical report is a revision (clarifying and expanding some sections) of the author's M.S. thesis [48], supervised by Charles Leiserson. This work was supported by a National Defense and Science Graduate Fellowship, by Defense Advanced Research Project Agency contract N00014-91-J-1698, and by Microsoft Corporation.

Contents

Chapter 1

Introduction and Motivation

Programmers would like to write a single program for efficient execution on parallel computers of different configurations and sizes, including the degenerate case of a single processor. This problem has received quite a bit of attention, but historically, the focus has been on parallelizing serial code. This report argues that the reverse—serializing parallel code—is both more natural and more effective. We show how to transform data-parallel programs (specifically, those which can be cast as generalized stencil computations—see page 8 for a definition) into programs that are partially parallel and partially serial. Such hybrid programs can take advantage of exactly the parallelism available at run time, resulting in running times that are competitive with implementations targeted for any specific number of processors.

Efficiently executing a single program on both parallel and serial computers is challenging because parallel and serial programs are written with different goals. Fast parallel programs permit processors to operate independently by eliminating dependences between computations on different processors. Efficient serial programs, on the other hand, have heavily optimized loops, and dependences often exist between iterations due to sharing of variables or results. In a parallel program, the critical path in any particular processor is made as short as possible without regard to whether a computation is also performed by another processor: while sharing results can pay, typically communication is much more expensive than computation. Serial implementations, on the other hand, aim to reduce the total amount of work done; communication through variables is cheap. As a result, when a serial program is naively run on a parallel machine, or a parallel program naively run on a serial computer, the performance is disappointing. We must transform the program in order to make it more amenable to fast execution by the target architecture.

The traditional approach toward execution of a single program on both parallel and serial architectures is to perform concurrentization or vectorization (collectively, *parallelization*) [143], transforming a serial program into one which can be sped up by being run on several processors or on a vector processor. *Dependence analysis* is the key to parallelization: each loop in the serial program is analyzed to determine whether its iterations may be run simultaneously. Data dependences prevent loop iterations from being run in parallel because of multiple uses of a variable. For instance, when a variable is set by one loop iteration and read by another, then reordering the loop iterations could change the program's result, so the loop cannot be parallelized.

While much progress has been made in parallelization, the field is far from mature. Most parallelizers just replace certain paradigms with a parallel version of the operation; since they operate by pattern-matching, a small change to the input program can affect its performance by orders of magnitude. Understanding the system's behavior requires detailed knowledge of the compiler, and the quest for good performance may force the user to write in a style easy for the compiler, but hard for people, to understand. Even the best dependence analysis is only approximate, erring on the conservative side for safety, and as a result parallelizers are often unable to take advantage of loops which have no real inter-iteration dependences. It is only fair to mention that the serial code programmers write in the quest for efficiency can be extremely complicated, with many artificial dependences added in order to permit reuse of variables and of results; the task of parallelization is inherently difficult. We avoid the difficulties of parallelization by transforming programs from parallel to serial form instead.

In the remainder of this chapter we first explain how serialization can speed up a program's execution, even on a parallel computer, by eliminating repeated computations. We present and give examples of three methods for doing so. Next we describe the problem domain and give a taxonomy of the types of repeated computation we can eliminate. We argue that serialization is an effective approach to the problem of running a single program on both serial and parallel computers. Finally, we outline the rest of the report.

1.1 Serializing parallel programs

This report takes the opposite approach from parallelization by starting with an explicitly parallel program and removing some of its parallelism. The resulting program can be run efficiently on either a serial computer or a parallel computer which does not have enough processors to exploit all of the parallelism inherent in the original problem.

Any parallel program can be run on a serial computer if the serial computer simulates each of the processors of a parallel machine; the simulated processors are called *virtual processors*. Similarly, any serial program can be run on a parallel machine by simply loading it onto one of the processors. Neither of these methods makes good use of the available resources, however. The execution time of the serial program naively run on a parallel computer is not decreased, even though additional processing power is available. The total work performed by the parallel program naively run on a serial computer is not decreased, even though the same computations may occur in different virtual processors being simulated by a particular physical processor.

The reason that a program can do less work when executed by a serial machine than by a parallel one is that when a value is computed on different processors of a parallel machine, there is no opportunity for elimination of redundant computation without incurring communication, which is even more expensive. When several virtual processors are simulated by a single physical processor, then redundant computations that were immune to elimination by virtue of being on different processors are suddenly being computed on a single physical processor; the virtual processors can share work at the cost of storing and retrieving a value—or even at no extra cost.

Virtual processors are simulated by a *virtual processor emulation loop* which executes in turn the instructions that would have been executed by each of the virtual processors. Therefore, in order to detect values used by more than one virtual processor, and to eliminate excess computations of those values, we only need to detect expressions computed during more than one execution of a loop body. When the program calls for multiple evaluations of an expression, then we can store the result after it is first computed; whenever it is needed thereafter, the result can be inexpensively retrieved from its storage location.

This method, applied to expressions for which only one value is considered at a time—for instance, in straight-line code or in a loop iteration considered independently of other iterations—is known as *common subexpression elimination*. No previously known common subexpression elimina-

```
for i = 1 to 500
  y[i] = f(i-1) * g(f(i+2))
```

Figure 1: A simple example of a loop common expression: f(j) is computed on both the $(j-2)$nd and $(j+1)$st loop iterations.

```
for i = 2 to 97
  newx[i] = (x[i-2] + x[i-1] + x[i] + x[i+1] + x[i+2]) / 5
```

Figure 2: Another loop common expression example. Each pair of loop iterations repeats 4 array references and 3 array element additions.

tion method works across loop boundaries, in which case a value is computed by lexically distinct expressions on different loop iterations. We provide methods for detecting and eliminating this important class of common subexpressions, which we call *loop common expressions*.

Figure 1 gives a simple example of a loop containing a loop common expression: `f(22)` is computed by both the 20th and 23rd iterations (those two appearances of the loop common expression are called its *instantiations*). No ordinary common subexpression elimination algorithm discovers this repeated computation: not only are the two expressions lexically distinct, but they also occur in different loop iterations. Another example with even more opportunity for optimization appears in figure 2. Not only can the computation of many summands (array references) be shared from one loop iteration to the next, but additions also appear multiple times: for example, `x[42] + x[43]` appears in the expressions for the 41st through 44th results, and the 92nd and 93rd results share `x[91] + x[92] + x[93] + x[94]`.

The examples throughout this report are written in pseudocode in the style of figures 1 and 2. Our implementation of the optimizations produces C [84], but the syntax and semantics of that language are somewhat obscure—particularly its `for` construct. For clarity, we have also renamed compiler-generated variables and occasionally performed simple restructuring. In all cases the result is true to the output generated by the compiler; no manual optimizations have been performed.

1.2 Three techniques for serialization

This report gives three techniques—*unrolling*, *loop common expression analysis*, and *loop differencing*—for optimizing data-parallel programs. These techniques eliminate repeated computation, but they also partially serialize the programs by adding data dependences. This is not a drawback, since we can exploit exactly the parallelism available at run time. The three methods are all program transformations that convert a straightforward data-parallel algorithm into a more complicated but more efficient serial one. This section briefly describes and demonstrates (on the examples of figures 1 and 2) the methods. Each of these optimization methods is discussed in greater detail in a chapter of its own.

The first method, unrolling, transforms loop common expressions into ordinary common subexpressions. In the unrolled loop, the texts of several iterations of the original loop lie side-by-side, so a common subexpression elimination algorithm can eliminate the repeated computation. Figure 3 shows the code of figure 1 after unrolling; the amortized number of function calls of `f` per result has been reduced from 2 to 1.6 (8 function calls for 5 results). In figure 4, the code of figure 2 has been unrolled, resulting in a reduction in the per-result number of array references from 4 to

```
for i = 1 to 496 step 5
  y[i]   = f(i-1) * g(f(i+2))
  y[i+1] = f(i)   * g(f(i+3))
  y[i+2] = f(i+1) * g(f(i+4))
  y[i+3] = f(i+2) * g(f(i+5))
  y[i+4] = f(i+3) * g(f(i+6))
```

Figure 3: The loop of figure 1 after unrolling to compute 5 results per loop iteration. Ordinary common subexpression algorithms can now arrange that `f(i+2)` and `f(i+3)` are each computed only once, reducing the number of calls to `f` from 2 per result to 8 per 5 results.

```
for i = 2 to 94 step 4
  newx[i]   = (x[i-2] + x[i-1] + x[i]   + x[i+1] + x[i+2]) / 5
  newx[i+1] = (x[i-1] + x[i]   + x[i+1] + x[i+2] + x[i+3]) / 5
  newx[i+2] = (x[i]   + x[i+1] + x[i+2] + x[i+3] + x[i+4]) / 5
  newx[i+3] = (x[i+1] + x[i+2] + x[i+3] + x[i+4] + x[i+5]) / 5
```

Figure 4: The loop of figure 2, unrolled to compute 4 results per iteration. Eliminating the maximal number of intra-loop common subexpressions cuts the per-result number of additions by half and array references, by more than half. The two-phase common subexpression elimination introduced in this report finds all possible common subexpressions, but other algorithms fail to do so.

1.75 and additions from 8 to 4. This method is always applicable, it uses simple, familiar building blocks, and the results can be quite good. However, we may have to unroll many times in order to expose common subexpressions, and some loop common expressions will always remain. Deciding how much to unroll can be tricky: if the latter example were unrolled to compute 6 results per loop iteration, it would require 3.5 additions per result, but if unrolled to compute 7 results, more than 3.7 additions per result would be required. This is surprising because usually more unrolling leads to better performance. Standard common subexpression elimination algorithms fail to find much of the repeated computation in figure 2. The *two-phase* algorithm, introduced in section 2.3.3.2 on page 25, does far better in practice, but the problem of finding optimal common subexpressions is NP-complete [5, 23].

The second method, loop common expression analysis, takes direct advantage of expressions that can be used by more than one loop iteration, or loop common expressions. Each iteration computes (and leaves in a temporary storage location such as a register) expressions that will be useful to subsequent loop iterations. In other words, iteration i arranges its computations so as to help iteration $i+1$, possibly resulting in slightly increased costs for iteration i, relative to ordering its computations in the greediest way. Any extra cost is more than offset by the fact that iteration $i-1$ has done the same thing, relieving iteration i of some work it would otherwise have to do. Unrolling can often reduce costs added by loop common expression analysis, further improving the overall gain. Figure 5 shows the code of figure 1 after elimination of multiple evaluations of loop common expressions; only 1 function call of `f` is required per result, though we have introduced a new temporary array and some extra operations to access it. Unrolling just twice eliminates the need for the array. Figure 6 shows that the number of additions and array references in figure 2 can be halved by unrolling to produce 2 results per iteration. Further unrolling to produce 4 results per iteration reduces the number of array references to 1 per result and the number of additions to 3 per result; it also eliminates the register-to-register move of figure 6. Like the method of

```
integer array t[0..3]
t[1] = f(0)
t[2] = f(1)
t[0] = f(2)
for i = 1 to 500
  s = f(i+2)
  y[i] = t[i mod 3] + g(s)
  t[i mod 3] = s
```

Figure 5: The method of loop common expression elimination applied to the code of figure 1. Only 1 application of `f` occurs per result computed, but accessing array `t` can be costly. Unrolling can eliminate the modulus and array indexing operations, though 3 temporary locations are still needed to hold old values of `f(i)`.

```
t2 = x[1] + x[2]
for i = 2 to 98 step 2
  t1 = t2
  t2 = x[i] + x[i+1]
  t1 = t1 + t2
  newx[i] = (x[i-2] + t1) / 5
  newx[i+1] = (t1 + x[i+3]) / 5
```

Figure 6: Loop common expression elimination applied to the code of figure 2. Each loop iteration computes two results while performing the same number of array references and additions as the loop of figure 2 (and one more division and register-to-register move). The reader is invited to determine how to reduce the per-result number of array references to just 1, without increasing the addition cost, by unrolling the loop to compute 4 results per iteration and taking advantage of additional loop common expressions.

unrolling and applying common subexpression elimination, loop common expression elimination is always applicable and is quite easy to implement. It usually outperforms simple unrolling in terms of operations performed, temporaries used, and code size.

The third method, loop differencing, computes a new result not from subexpressions of a previous result, but from the previous result itself, by adding it to the difference between the values computed by two loop iterations. This has some similarities with strength reduction optimizations. Figure 7 shows the result of applying loop differencing to the code of figure 2: array references are reduced to 2 per iteration and additions, to 4 per iteration. The loop differencing method is not always applicable—it cannot be used to speed the execution of the code of figure 1—but it often produces excellent speedups with no unrolling required at all. Its chief disadvantage is the use of the inverse of the original combining operator. Even if this inverse exists, if it is not exact, then overflow, underflow, or value drift may be a problem. The technique is numerically stable when the dynamic range of the numbers being operated on is not excessive—that is, their values are all approximately equal.[1] Despite its problems, this method is much better than the others for operations that combine many values into each result.

[1] Ordinarily, operating on two values that are nearly equal can significantly increase the relative error even while leaving the error's magnitude unchanged. Our restriction is that no value is nearly equal to and a sum containing that value, which happens only if the value is much larger than the other summands.

```
runningsum = x[0] + x[1] + x[2] + x[3]
for i = 2 to 98
  runningsum = runningsum + x[i+2]
  newx[i] = runningsum / 5
  runningsum = runningsum - x[i-2]
```

Figure 7: Loop differencing applied to the code of figure 2. Rather than computing each result from scratch, the difference between two adjacent results is added to one result to compute the next.

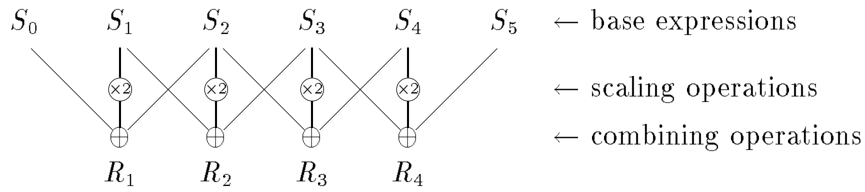


Figure 8: Relationship between the three components of the stencil $R_i = S_{i-1} + 2S_i + 2S_{i+1}$. The base expressions are the uses of S, the scaling operations are multiplications by 2, and the combining operations are additions.

1.3 The problem domain

Elimination of loop common expressions—those which appear in at least two loop iterations—often provides significant speedups, but unrolling, loop common expression analysis, and loop differencing are not applicable to every parallel program. This section describes the kinds of computation which produce loop common expressions and lists assumptions about the parallel program to be serialized. Appendix B presents a sampling of real-world problems to which our methods are applicable.

The most important stipulation is that we must be able to determine ahead of time where the repeated computation will occur: every result must be computed from the loop index in the same way. We cannot arrange to share computations from one loop iteration to the next if we don't even know ahead of time which computations will be performed.[2] For instance, a pointer jumping [35] loop contains no loop common expressions. Although many virtual processors—even many being simulated by the same physical processor—may follow a particular pointer, which ones do is data-dependent.

Each result of a *stencil computation* (or simply *stencil* [22]) depends on a small set of values with a particular structural relationship to one another. For instance, a result array is computed from one or more similarly-shaped source arrays; each result array element depends on source array elements at particular relative offsets from it. Figure 2 is a stencil computation. We extend the class of stencil computations to those using the loop index arbitrarily in the result expression, as in figure 1. A stencil is comprised of three types of computation, as illustrated in figure 8.

- The *base expression* of a stencil appears in all of its subexpressions; a *base value* is the value of a base expression. In figure 1, `f(·)` is the base expression, and in figure 2, `x[·]` is.

[2] Memoization [1, 109] may be profitable in such cases. A memoized function stores all the results it has computed, along with the arguments that produced those results. When the function is called, it first does a table lookup with its actual parameters and possibly just returns a previously-computed result; otherwise the result is computed in the ordinary way, saved for future reference, and returned. Memoization can eliminate redundant computation that no static analysis could, but its run-time costs are high.

- A *scaling operation* is any computation performed on some but not all of the base values; application of the function g in figure 1 is a scaling operation, and there is no scaling operation in figure 2.

- The stencil's *combining operation* produces a result from the scaled base values; this is multiplication in figure 1 and addition in figure 2.

The discussion is simplified by addressing loops which compute just one stencil; the generalization to multiple stencils per loop is straightforward. For pedagogical simplicity, most of examples are weighted-sum stencils in which the base expression is an array reference, the scaling operations are multiplications by fixed coefficients, and the combining operation is addition. A convenient shorthand for such stencils is a list of scaling coefficients; for instance, the stencil of figure 2 would be rendered $\langle 1, 1, 1, 1, 1 \rangle$. (The relative offset of the result is ignored, because it has no effect on any of the optimizations, so $\langle 1, 0, 0, 1 \rangle$ represents both $x_i = y_{i-1} + y_{i+2}$ and $x_i = y_{i+1} + y_{i+4}$.) The techniques of this report apply to the much larger class of generalized stencil computations, however, not just to weighted sums.

We make the following further assumptions about the computation being optimized.

explicitly parallel program The iterations of the input loops may be safely run in parallel. Previous stages of the compiler may have transformed parallel constructs into such loops, or may have marked some sequential loops as free of data dependences. We ignore loops containing data dependences that could change the value of a potential loop common expression between uses. These transformations may require introduction of temporaries; for instance, independent execution of the iterations of

```
doall i, 1 < i < 100
  a[i] = (a[i-1] + a[i+1]) / 2
```

requires the use of a temporary array [143]. We use loops because they are more familiar than parallel constructs (each of which has its own semantics) and to emphasize the applicability of our techniques to serial as well as parallel programs.

evenly distributed work The programming style described above—explicitly parallel, with many virtual processors performing exactly the same computations—is known as *data-parallel*. We add one more assumption common among data-parallel and scientific programs: each loop iteration completes in about the same amount of time. The assumption is satisfied if the amount of computation in the loop body is not heavily data-dependent and the processors of the physical machine are equally powerful and equally heavily loaded. Small irregularities in the cost per iteration, which are lost in the noise when enough iterations are considered together, present no problem if the target machine is MIMD (multiple-instruction, multiple-data); data-parallel programs targeted for such machines are often referred to as SPMD (same program, multiple data) [40, 129].

This assumption reduces scheduling to evenly distributing loop iterations among processors, which can be done at compile time even if the run-time number of iterations and processors is not yet determined. If such a scheduling policy could assign some processors much more work than others, then the more lightly-loaded processors will finish earlier and then sit idle while waiting for the others to finish; this could easily wipe out the optimization gains or even the speedups that accrue from parallelization. Supporting a `future` [59] command, arbitrary message-passing, or process migration greatly increases the complexity and overhead of a programming system.

efficient implementation Since we care about performance on machines with any number of processors from 1 to as many as the problem's inherent parallelism, we demand that the input program run fast on a machine with infinitely many processors. It is not difficult to write poor parallel programs that can be transformed to produce good serial implementations, but our goal is to permit a single program to run well on a serial machine, an infinitely parallel one, and anything in between.

While many algorithms are amenable to such transformations, not all are. For instance, a good smoothing operation in digital signal processing is to determine the median element of every window of width w in a vector of size v [69, p. 86; 108, p. 516]. On a computer with v processors, the best way to compute this is to use an algorithm with $O(w)$ worst-case running time on each processor [19]. On a serial computer, that would require $O(vw)$ time; it is better to use an order statistic tree [35, p. 281] which can be constructed in $O(w \log w)$ time and updated in $O(\log w)$ time, for a total cost of $O((v + w) \log w)$. It is not obvious how to transform the parallel algorithm into the sequential one (or vice-versa) except by pattern-matching.

Similarly, we cannot hope to convert bubble sort into AKS sort [7] or to convert a convolution into a Fast Fourier Transform (FFT; see [108] for references), an elementwise multiply, and another FFT. (Section 7.3 does show how loop common expression elimination enables bubble sort to be transformed into insertion sort. Section B.1 discusses tradeoffs between computing convolutions via stencils and FFTs—the former is sometimes preferable.) The optimizations presented in this paper result in a new implementation of an algorithm but not in an entirely new algorithm: the transformed program computes all the results that the original one did, though it performs fewer operations in order to do so. Typically this reduces the constant factors in the cost of execution rather than improving its asymptotic running time.

non-speculative computation When excess resources are available, processors that would otherwise be idle can perform *speculative* computations whose results might never be needed. This does not slow the computation down, and it may speed it up. On a machine without extra processors, on the other hand, speculative computation never improves performance and often degrades it. Since the serialized program computes all the results computed by the original implementation, a speculative program cannot be efficient when serialized.

high virtual processor ratio If there are few virtual processors per physical processor, then the virtual processor emulation loops are not run many times. Their execution doesn't consume a significant amount of the machine's resources, so it is not worthwhile to spend a lot of effort optimizing them. Additionally, when the virtual processor ratio is very low, there may be fewer optimization opportunities since little serialization is acceptable.

associative operations For some of our optimizations, the stencil combining operation must be associative, because a key part of those methods is reordering computations so that they are not all done left-to-right. The method of loop differencing also requires that the operator be commutative and have an inverse.

repeated computation Obviously, we cannot speed up a program by removing repeated computation unless some computations are repeated. The input program must contain a stencil

```
v[i] = 2 * w[i-3] + 2 * w[i-1] + 2 * w[i+1] + 2 * w[i+3]
```

Figure 9: A periodic stencil computation with base expression `2 * w[·]`; addition is the combining operation.

$$bode_i = \frac{14}{45}f(i-2) + \frac{64}{45}f(i-1) + \frac{24}{45}f(i) + \frac{64}{45}f(i+1) + \frac{14}{45}f(i+2)$$

Figure 10: An aperiodic stencil computing Bode's rule for numerical integration [108], which is exact for polynomials up to and including degree 5. The area under the curve $f(i)$ between $i - 1/2$ and $i + 1/2$ is better approximated by $bode_i$ than by $f(i)$, by Simpson's rule, or by the trapezoidal rule. The base values are applications of f and the scaling operations are $\langle \frac{14}{45}, \frac{64}{45}, \frac{24}{45}, \frac{64}{45}, \frac{14}{45} \rangle$.

computation in which a base expression appears at least twice. For instance, a loop with the body `r[i] = a[i] + b[i+1] + c[i+2]` would not be worth optimizing by eliminating redundant loop common expressions, because only the loop index additions could be sped up.

The optimizations discussed in this report are orthogonal to the second-order ones which some researchers call serialization of parallel computations. Those improvements, which typically depend on better use of the memory hierarchy or reduced overhead for simulating processors, can be applied to a program after our transformations in order to speed it up even more. Section 8.1 discusses those efforts.

1.4 Categorizing loop common expressions

This section describes periodic and aperiodic stencils, explicates the loop common expressions appearing in them, and shows how to transform complicated stencils into simpler ones that are easier to process.

Recall from in section 1.3 (page 8) that there are three types of loop common expression: base expressions, scaled expressions, and applications of combining operators. A stencil is called *periodic* if the operands of its combining operations form a pattern which repeats at least 3 times; that entire pattern is considered the base expression, and there is no scaling operation. Computations of periodic stencils can benefit from sharing computation of base values and applications of combining operations. *Aperiodic* stencils permit computations of base values and applications of scaling operations to be shared among loop iterations.

Figures 9 and 10 give examples of periodic and aperiodic stencil computations, respectively. The opportunities for avoiding repeated computation are as follows:

base values In figure 9, `2 * w[14]` appears in the expressions for the 11th, 13th, 15th, and 17th results. In figure 10, $f(3)$ is computed for 5 results.

scaled values In figure 10, $\frac{14}{45}f(91)$ is needed twice, for $bode_{89}$ and $bode_{93}$. Periodic stencils, such as that of figure 10, have no scaling operation.

combining operations In figure 9, `2 * w[51] + 2 * w[53]` appears in the expressions for `v[50]`, `v[52]`, and `v[54]`.

For each of the three methods for reducing redundant computation in stencils, we give concrete numbers for its performance on base values, on scaling operations, and on combining operations (when the scaling operations are identical), and describe how the method treats the cases differently. In many cases the treatment of base values is a special case of that for scaling operations and succumbs to the same methods.

How effectively a stencil's loop common expressions can be exploited depends on the stencil's base expression and and scaling and combining operations. Those elements can be chosen in multiple ways (selecting a larger base expression leads to fewer scaling and combining operations), and a stencil can sometimes be split into simpler stencils which can be handled individually. Here we give an overview of how these choices are made; section 5.1.2 on page 56 provides more details.

Elimination of loop common expressions can be simplified by splitting a stencil into pieces which are optimized separately; the optimized pieces are then recombined into a single computation. For instance, the aperiodic stencil $\langle 2,3,2,1,2,3,2\rangle$ can split into $\langle 2,0,2,0,2,0,2\rangle$ and $\langle 3,0,1,0,3\rangle$; the former is periodic, so it can be efficiently processed by the method of loop differencing. Since it is convenient for all non-zero scaling factors to be the same even in aperiodic stencils, the latter could be further split into $\langle 3,0,0,0,3\rangle$ and $\langle 1\rangle$. This divide-and-conquer approach greatly simplifies the code for processing stencils, because only simple patterns need be explicitly addressed. Scaling and combining operations are optimized using these simplified forms; base values are optimized after the recombining step, so it is easy to guarantee that the result is as good as it would have been, were the more complicated form directly processed—no common computation is hidden by appearing in two separately processed stencils.

Another way to split a stencil into simpler ones is to selecting base expressions which use the index expression i multiple times. For the computation

$$x_i = 2y_{i-2} + 3y_{i-1} + 2y_i + 2y_{i+1} + 3y_{i+2} + 2y_{i+3}\ ,$$

the obvious base element is a y value, the scaling operations are multiplications by 2 and 3, and the combining operation is addition; $\langle 2,3,2,2,3,2\rangle$ represents this view of the computation. Another decomposition of this computation uses base expression $2y_{i-1} + 3y_i + 2y_{i+1}$, the identity scaling operation, and a combining operation which adds two scaled values from three loop iterations apart; $\langle 1,0,0,1\rangle$ is the shorthand for this presentation of the computation, which abstracts away the fact that the base elements are themselves stencils which can be represented $\langle 2,3,2\rangle$. Any stencil which can be represented $\langle 1,0,0,1\rangle$ should be optimized in the same way. (As another example, $\langle 3,14,3,14,3,14,3,14\rangle$ is optimized exactly like $\langle 2,0,2,0,2,0,2\rangle$ of figure 9, except that the base expressions are different. Base elements which are stencils can be optimized by a recursive application of these techniques.

For a given stencil, the base expression should be chosen as small as possible such that the entire computation can be expressed in terms of (non-trivial) scaling and combining that expression. For instance, $\langle 1,2,1,2,2,4,1,2\rangle$ would be recast as $\langle 1,0,1,0,2,0,1\rangle$ with base expression $\langle 1,2\rangle$. After choosing the smallest possible base expression, if the resulting stencil can be reduced further, it should be. The only real problem with splitting stencils or using large base expressions is one of terminology: a stencil may have several different sets of base values, one for each recursive application of loop common expression elimination. The one in question should be clear from context; we will sometimes speak of $x_i = 3y_{i-1} + 3y_{i+2}$ as $\langle 3,0,0,3\rangle$ (in which case the base expression is y_i) and sometimes as $\langle 1,0,0,1\rangle$ (with base values three times as great).

1.5 Serialization is effective

In order to achieve efficient execution on all computers, with numbers of processors ranging from 1 to infinity, we can either maintain multiple versions of a program, each tuned for use on a specific number of processors, or we can maintain one efficient canonical version of the program and generate from it versions appropriate for any specific number of processors. Maintaining multiple programs, either explicitly or by way of conditional statements in a single source, is easily dismissed, because the versions must be independently written, debugged, and maintained. Thus, the only realistic possibilities are serialization of a parallel program and parallelization of a serial program. Sequentialization is a better strategy because it guarantees good parallel performance, because sequentialization is easier than parallelization, and because data-parallel programs tend to be clearer and simpler than their serial counterparts. (Parallelization has the advantage of being applicable to dusty-deck codes as well as to new ones, which accounts for the interest in it.)

When parallel performance is important, it is better to write data-parallel than sequential algorithms. When we start with an efficient parallel algorithm, we are guaranteed good performance on parallel hardware, time on which is usually much more valuable than time on a serial machine. If serialization fails, the results are not as dire as if parallelization fails, and each is certain to fail some of the time.

Sequentialization also has the advantage of being easier than parallelization. The task of a parallelizer is to remove data dependences, while a serializer may add them wherever convenient. It hard to find accurate approximations to data dependence and to remove them without changing the value computed. The difficulty is compounded by the fact that good serial algorithms tend to be complicated and hard for a parallelizer to manipulate because of data dependences that are not strictly necessary to achieve the correct result but which were introduced incidentally in the process of hand-optimizing the code. While progress has been made on parallelization, it has resisted the efforts of many talented researchers. Many parallelizers do little more than pattern-match against the input program, which makes them unreliable and their behavior hard to understand. Serialization, on the other hand, is an easier task which avoids these problems inherent in parallelization. This report gives three simple methods—unrolling, loop common expression elimination, and loop differencing—for serializing parallel programs by eliminating redundant computation. Two of them are optimal when it is possible to unroll sufficiently, and all three perform well even when the unrolling amount is limited. This stands in sharp contrast to the failures of parallelizers even on loops that, to humans, obviously have no data dependences [143, p. 96].

Another reason to prefer serialization to parallelization is that data-parallel programs tend to be much simpler than their serial counterparts, in large part due to the local view of the computation that the data-parallel model permits. The programmer can concentrate on the data and the computations, and spend less time manipulating control structures and thinking about program flow and dependences. Built-in data structures such as the vector or array, and uniform methods for manipulating them, make large-scale programming significantly easier and less error-prone. As a result, data-parallel programs are easier to write, read, maintain, and manipulate than serial programs solving the same problem.

To illustrate the comparative complexity of the two types of programming, consider two implementations of convolution with a two-dimensional binomial filter; this is an important preprocessing step in the Canny edge detector [25] and other vision applications. The kernel of the data-parallel version [27] is 4 lines of *Lisp [92, 135, 136], each containing 2 arithmetic operations. The kernel of the sequential version [85] is 24 lines of C [84], containing 174 arithmetic operations in all. The comment at the beginning of the procedure reads,

```
/* do the 2D convolution as two 1D convolutions */
/* this code is VERY hairy.  see wjr's */
/* for an example of what it's really doing */
```

The folk theorem that parallel programs are harder to write than serial ones may be true for some models of MIMD programming, but it is not the case for data-parallel programming.

While not every program can be efficiently written in this style, the data-parallel model has come to be widely accepted in the parallel processing and scientific computation communities, even for programming serial machines. There are data-parallel versions of the most popular serial languages—C (C* [110, 133], Dataparallel C [60]), Fortran (Fortran 90 [13], CM Fortran [134, 115], Fortran D [55], and others), and Lisp (CM Lisp [126], Paralation Lisp [113, 114], *Lisp [92, 135, 136])—as well as data-parallel languages designed from first principles (NESL [17], VCODE [18]) and sequential languages that were already partly data-parallel (APL [76] and its dialects [73, 77, 80, 142], SETL [41, 116], etc.). This list is far from exhaustive. Data parallelism is not a radical departure from existing programming practice—it is primarily a matter of using some new abstractions.

1.6 Outline

This section outlines the report and highlights its original contributions.

The introduction has explained the goal of permitting a single program to run efficiently on both parallel and serial computers. Our method is to eliminate recomputation of loop common expressions, which appear in two or more loop iterations; loop iterations correspond to virtual processors in a parallel program. We introduced three techniques for removing redundant computation (unrolling, loop common expression elimination, and loop differencing) and defined stencil computations and their constituent parts (base expressions, scaling operations, and combining operations). Next we argued that serialization of parallel programs can be even more effective and natural than parallelization of serial ones. This report recognizes the important opportunity for optimization represented by loop common expressions and introduces methods for optimizing multiple iterations of a loop while examining only one copy of the loop body's text. Even if performing extra work appears to increase the cost of a single loop iteration, it can decrease overall running time.

The next three chapters each address one of our techniques for eliminating redundant computation. The first is unrolling with common subexpression elimination; the unrolling step converts loop common expressions into ordinary common subexpressions, while the common subexpression elimination step prevents their reevaluation. We give formulas for the costs (extra operations and extra temporary variables) and benefits (operations eliminated) of unrolling. After evaluating a number of common subexpression elimination strategies, we introduce the two-phase common subexpression elimination algorithm, which separates the subproblems of determining which computations may be shared and deciding which ones will actually be computed, and give algorithms for implementing it. This method far outperforms traditional common subexpression analysis, which combines the two stages in a greedy or even arbitrary manner. Unrolling can sometimes degrade rather than improve performance; we show how to choose a good unrolling. We prove upper and lower bounds on the minimum number of operations required to compute unrolled stencil computations. Finally, we examine the use of unrolling and ordinary common subexpression elimination to uncover loop common expressions in the original, non-unrolled loop.

Chapter 3 presents a direct method for finding and eliminating loop common expressions; it hinges on discovering patterns in the structure of the computation performed by each iteration. We give simple methods for removing all recomputation of loop common expressions and prove bounds on the cost of computations after loop common expression elimination has been run. We also show how all array reference and index manipulation overhead can be eliminated by unrolling loops to scalarize arrays and how to adjust the sizes of temporary arrays when they cannot be scalarized.

The third method for eliminating redundant computation is loop differencing, which symbolically computes the difference between the results computed by consecutive loop iterations. Given the result computed by a particular iteration, subsequent ones can be computed more efficiently by subtracting the difference than by computing them from scratch (or even from subexpressions computed by the previous iteration). This optimization uses operators' inverses to undo some work, creating subexpressions that were never used in computing the previous result. The method results in very good code for evaluating periodic stencils. We also show how to generate extremely efficient code for certain aperiodic stencils, then discuss the method's primary problem, its potential numerical instability, and how to avoid it.

In chapter 5 we discuss implementation topics, including relatively minor algorithmic details glossed over in previous sections, the integration of our techniques with other compiler optimizations, design decisions in our implementation, and other methods for optimizing stencil computations. Timing results obtained by running the compiler output and comparing the different methods with one another and with the original code are presented in the next chapter. Chapter 7 deals with extensions to our methods, including scheduling of jobs onto processors, extensions to two-dimensional stencils, optimization of purely serial algorithms, and other topics.

Finally, we discuss previous research related to the serialization of parallel programs, which has focused on the reduction of system overhead, not on the computations being performed. Other work of interest includes iterator inversion, parallelization, and direct attacks on stencil computations.

The two appendices present auxiliary material. The first contains an outline of the proof that unrolling to compute $w + 1$ results per loop iteration, where w is a stencil's width, optimizes the number of combining operations required to compute the results. Appendix B lists numerous real applications to which our optimizations are applicable and discusses when stencil implementations are preferable to other implementations.

Chapter 2

Unrolling with Common Subexpression Elimination

Common subexpression elimination is the traditional method for removing redundant computation from computer programs, but it cannot find loop common expressions, because it considers only one representative loop iteration. Loop unrolling can transform loop common expressions into ordinary common subexpressions. Even after this transformation, most common subexpression elimination algorithms find only part of the redundant computation. We show how to augment these algorithms to enable them to perform well on computations with this structure. The resulting method is fairly straightforward and uses readily available technology.

This chapter first briefly reviews common subexpression elimination and loop unrolling, then gives two ways to couple these methods to prevent recomputation of loop common expressions. The first is to execute the unrolled and optimized code, which outperforms the original version, though it may have to be unrolled quite a bit to improve the results significantly. Furthermore, the traditional methods for common subexpression elimination do not find all its common subexpressions. We introduce a new two-phase method for common subexpression elimination which separates the conceptually distinct stages of identifying which expressions appear multiple times and deciding which of those redundant computations to eliminate. We show how to achieve theoretically optimal results by unrolling the proper number of times and using a good common subexpression elimination algorithm. Unrolling and common subexpression elimination are well-known, but their application to elimination of loop common expressions and the analyses of their efficacy are original. The second way to use unrolling and common subexpression elimination is to find, in the common subexpressions detected in an unrolled loop, a pattern of results which can be used as loop common expressions. These methods are ad hoc and computationally expensive; chapter 3 shows a more direct way to find loop common expressions.

2.1 Common subexpression elimination

The traditional method for removing redundant computation in computer programs is common subexpression elimination, which uses textual analysis to detect when an expression occurs multiple times in the source program. The computation is performed only once and the expression's value is saved; thereafter, when the value is needed, the stored value is retrieved, which is cheaper than reevaluating the expression. There are two classes of algorithm for common subexpression elimination: *partial redundancy elimination* and *value numbering*. Because each method has advantages

```
for i = 1 to 400
  sm[i] = .25 * r[i-1] + .5 * r[i] + .25 * r[i+1]
```

Figure 11: Smoothing a signal by convolving it with a small binomial filter.

and drawbacks, many compilers use both.

Partial redundancy elimination [42, 43, 82, 83, 101, 123, 139] identifies lexically identical expressions appearing anywhere in the program text. To prevent lexically identical but semantically different expressions from being considered equivalent, partial redundancy elimination is used in conjunction with an algorithm which identifies which expressions are generated, transmitted, and killed by each basic block. For instance, there are no common subexpressions in

```
x = a + b
...
b = c
...
y = a + b
```

and `a+b` must be recomputed because the intervening redefinition of `b` might change its value.

Value numbering [5, 6, 23, 30, 57, 58, 79, 122] assigns a unique identifier to each value computed (not to the identifier that names the value). When an operator is reapplied to a set of operands, the previous result can be used instead. For instance, in

```
x = a + b
...
c = b
...
y = a + c
```

value numbering would recognize that (in the absence of other assignments to `a`, `b`, or `c`) the expressions `a+b` and `a+c` stand for the same value. Value numbering is restricted to extended basic blocks (sequences of instructions with only one entry point), but a start toward extensions to whole-program optimization has been made [10, 111].

Neither method can find loop common expressions—whose values are multiply computed by a loop even though no particular iteration contains redundant work—because both methods consider only one loop iteration; partial redundancy elimination has the further weakess of only identifying lexically identical expressions. As an example of a loop common expression which they cannot detect, consider smoothing a digital signal by convolving it with a binomial filter, which is the discrete approximation to a Gaussian filter. The code in figure 11 implements this operation; the filter has been kept small for simplicity. The expression `.25 * r[22]` is computed at both the 21st and 23rd iterations; we would like to avoid repeating this multiplication.[3]

[3] We ignore for the time being that we can eliminate a multiplication by transforming the loop body into `sm[i] = .25 * (r[i-1] + 2 * r[i] + r[i+1])` (but see section 5.2.1 on page 60 for a discussion of this optimization). The multiplications stand for arbitrary operations which may not distribute over addition; for instance, this could have been `sm[i] = f(r[i-1]) + g(r[i]) + f(r[i+1])`.

```
for i = 1 to 397 step 4
  sm[i]   = .25 * r[i-1] + .5 * r[i]   + .25 * r[i+1]
  sm[i+1] = .25 * r[i]   + .5 * r[i+1] + .25 * r[i+2]
  sm[i+2] = .25 * r[i+1] + .5 * r[i+2] + .25 * r[i+3]
  sm[i+3] = .25 * r[i+2] + .5 * r[i+3] + .25 * r[i+4]
```

Figure 12: A 4-unrolled version of the code of figure 11.

2.2 Loop unrolling

The obvious way to permit common subexpression elimination algorithms to consider several loop iterations at once is by *unrolling*: producing a new loop each of whose iterations does the work of several iterations of the original loop. Figure 12 shows the result of unrolling the smoothing operation of figure 11; if the old loop was performed 400 times, the new loop only needs to be performed 100 times. More to the point, expressions computing the same value which used to be in different loops (and thus hidden from ordinary common subexpression elimination algorithms) now occur together in straight-line code.

The original loop iterations (the iterations of the non-unrolled loop) are called *logical iterations*, while the loop iterations of the unrolled loop are called *physical iterations*. In figure 12, each of the physical iterations contains 4 logical iterations. The number u of logical iterations per physical iteration is called the *loop unrolling amount* or just the *loop unrolling*; we say that the loop has been u-unrolled. The loop in figure 12 has been 4-unrolled, while that in figure 11 is 1-unrolled—that is, it isn't unrolled at all.

The benefits of loop unrolling go far beyond transformation of loop common expressions into ordinary common subexpressions. Loop unrolling plays an important role in all of the optimizations described in this paper. Loop unrolling is also commonly used to achieve second-order improvements by reducing loop iteration overhead, improving use of the memory, etc. Unrolling can also have negative second-order consequences such as overflowing the instruction cache; all of these second-order effects are ignored.

Unrolling and common subexpression analysis can be used in two ways to eliminate redundant computation of loop common expressions. The initial step of either method is unrolling, which transforms some loop common expressions into ordinary common subexpressions. The first method leaves the loop unrolled and takes advantage of the savings gained from common subexpression elimination. The second method infers a pattern of loop common expressions from the common subexpressions in the unrolled loop, then attempts to reroll the optimized unrolled loop into one which reuses values from iteration to iteration. We discuss these two methods in the following sections.

2.3 Common subexpression exposure

Our first method for using unrolling to reduce the recomputation of loop common expressions is to unroll the loop, changing loop common expressions into ordinary common subexpressions, and then to apply standard techniquest to avoid redundant computation. (We introduce a new method for common subexpression elimination because those appearing in the literature perform poorly in this problem domain.) Following the taxonomy of loop common expressions introduced in section 1.4, we discuss in turn the impact of this optimization on scaling operations, on base values, and on combining operations.

2.3.1 Scaling operations

Suppose that a stencil contains (only) 2 occurrences of a scaling expression. (This discussion ignores loop common expressions besides scaling expressions.) In figure 11, $d = 2$, and the scaling expression is multiplication by .25. If we ignore the `.5 * r[i]` term, then when the loop is 1-unrolled, as in figure 11, it requires two scaling operations per result and (by definition) no additional temporary locations. When the loop is 4-unrolled, as in figure 12, two extra temporary locations are required, to store the values of `.25 * r[i+1]` and `.25 * r[i+2]`, and this reduces the number of scaling operations to 6 for 4 results, or 1.5 per result.

If d loop iterations intervene between reapplications of the scaling operation and the unrolling amount is u, then the numbers of scaling operations required and saved per unrolled loop, and the total number of additional temporary variables required, are given by the following table.

Unrolling vs. width	Scaling operations Required	Scaling operations Saved	Extra temps
$u \leq d$	$2u$	0	0
$d \leq u \leq 2d$	$u + d$	$u - d$	$u - d$
$u \geq 2d$	$u + d$	$u - d$	d

We can justify the values in this table in the following way.

- If $u \leq d$, then the unrolling has not converted any loop common expressions into common subexpressions. If the first logical iteration scales base values i and $i + d$, then the uth logical iteration scales base values $i + u - 1$ and $i + d + u - 1$; since $i + u - 1 > i + d$, the base values operated upon by the u logical iterations are disjoint.

- If $u > d$, then $(i + u - 1) - (i + d) + 1 = u - d$ of the values computed by logical iterations 1 through $u - d$ can be reused by logical iterations $d + 1$ through u, if they are saved in temporary storage.

- If $u > 2d$, then some logical loop iterations are both the beneficiaries of work previously done and the benefactors of later logical iterations. No more than d storage locations need to be allocated because each value is last reused d logical iterations after it is first computed, and only one new scaled value is produced per logical iteration. Thus, the transformed code requires as many extra temporary variables as operations are saved, up to a maximum of d.

The per-result costs are graphed in figure 13 for $d = 2$ and $d = 5$. Compare the left side of the figure with the results given at the beginning of this section.

The savings for aperiodic stencils which contain more than 2 instances of a scaling expression are similar. If there are i instances of the scaling expression and the distances between neighboring instances are $d_1, d_2, \ldots, d_{i-1}$, then when the loop is u-unrolled, the original $u \cdot i$ scaling operations are reduced by

$$(u - d_1) + (u - d_2) + \cdots + (u - d_{i-1}) \ ,$$

where the parenthesized expressions, if less than zero, should be taken to be zero instead. The same expression gives the number of temporary variables required to remember old scaled values, except that each parenthesized expression is also upper-bounded by the appropriate d_j, so the maximum

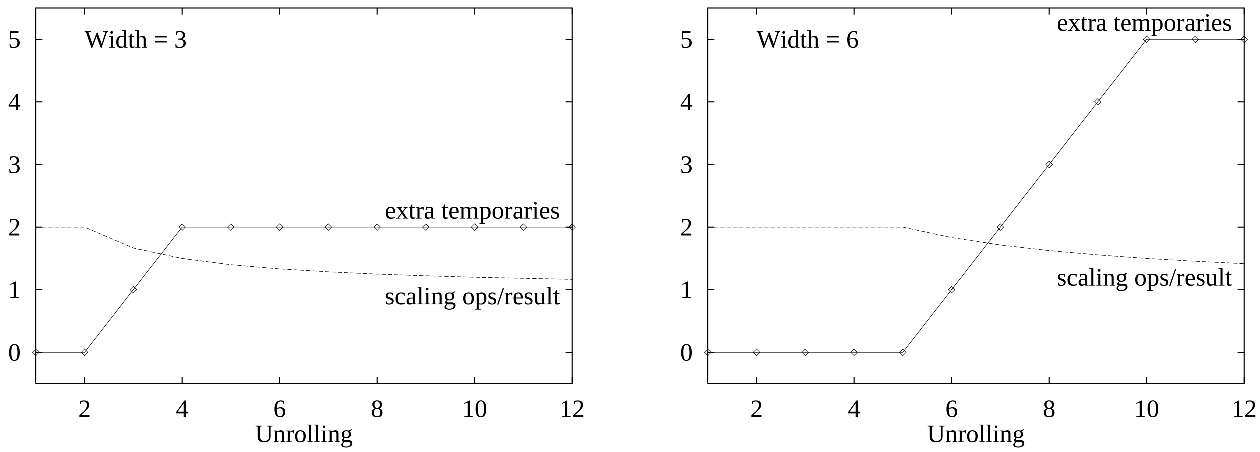


Figure 13: Plots of the number of scaling operations per result, and total additional temporary storage locations, required to avoid recomputation of scaling operations for the stencils $\langle a, 0, a\rangle$ (for which $d = 2$) and $\langle a, 0, 0, 0, 0, a\rangle$ (for which $d = 5$) at various unrollings. The width w is one greater than the index difference d.

is $\sum_{j=1}^{i-1} d_j = d$. Figure 14 shows the scaling savings gained by unrolling an aperiodic stencil of width 8 ($d = 7$) which contains 3 instances of a scaling expression. In general, we cannot save any stencil combining operations (in this example, the additions which produce the final results from the scaled values) when there is no pattern to the occurrences of the scaling expressions.

To analyze the efficacy of unrolling for an aperiodic stencil containing multiple occurrences of more than one scaling operation, we split the stencil into multiple ones, each containing only one scaling expression (stencil splitting was discussed in section 1.4 on page 12). For instance, $\langle a, 0, b, 0, a, b\rangle$ would be split into $\langle a, 0, 0, 0, a\rangle$ and $\langle b, 0, 0, b\rangle$. Each of the resulting stencils is analyzed independently to determine its the costs and savings, and these results are added up to get a total for the entire stencil.

A great deal of unrolling is required in order to approach the theoretical limit of one operation per logical iteration. While the unrolled loops permit some of the repeated computation to be avoided, they still contain unexploited loop common expressions. (For example, the 6-unrolled loop on the right side of figure 14 performs 13 scaling operations, but the lower bound is 1 scaling operation per result. If the scaling operation appears twice, There are $\min(d, u)$ unexploited loop common expressions per physical loop.) Simply unrolling and performing common subexpression analysis is not a real solution to the problem of removing loop common expressions.

2.3.2 Base values

Unrolling also results in a reduction of recomputations of base values. The formulas for the number of base value recomputations avoided are exactly the same as for scaling operations. In the usual case, every slot of a w-element stencil has a nonzero scaling operation; that is, w consecutive base elements, possibly scaled, appear in the expression for each result. The total cost to compute u results in a u-unrolled loop is $u + d$ base operations (plus the costs of scaling and combining).

The formula of section 2.3.1 says that up to d additional temporaries are required. We can reduce the number of extra temporaries to $\min(u, d)$ by computing several results simultaneously rather than remembering many base values. The idea is to localize all the references to a particular

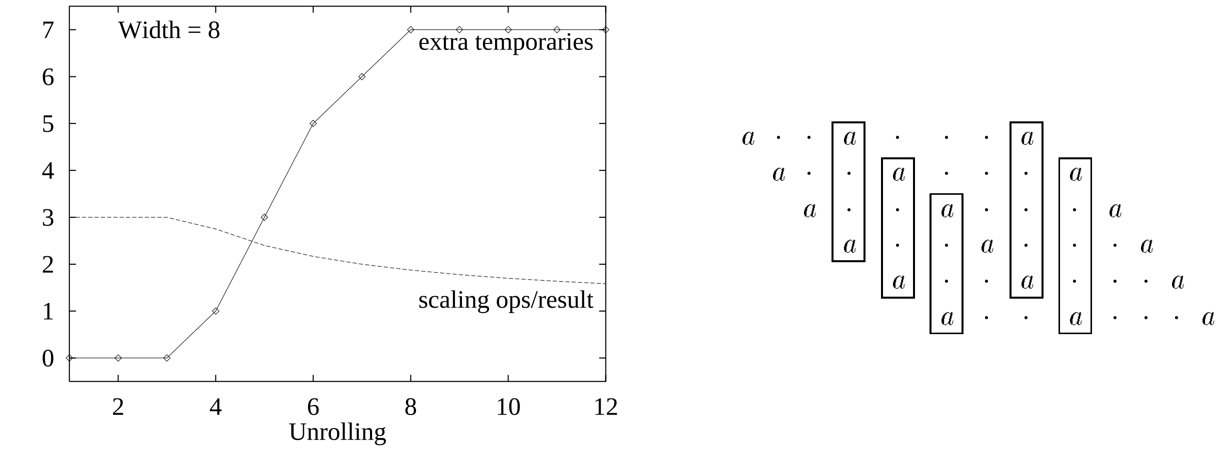


Figure 14: Scaling costs per iteration and temporaries required to remember scaled values for the stencil $\langle a, 0, 0, a, 0, 0, 0, a \rangle$, which has $d_1 = 3$ and $d_2 = 4$. A schematic of the 6-unrolled loop is also shown: each row corresponds to a different loop iteration, and the loop iterations are offset so that common computations appear in the same column. Boxes indicate opportunities for saved scaling operations: only one of the scaling operations in each column need be performed. If we u-unroll for $u > 7$, then some scaled values can be used thrice.

---

base element rather than all the computations for a particular result. For instance, the code of figure 2, when 2-unrolled and rescheduled, becomes that of figure 15, which requires only 2 extra temporary variables. (This number is a conservative estimate, because the behavior of the code scheduler or the structure of the computation many cause the rescheduled code to require even fewer extra temporaries. In figure 15, we have only really added one temporary variable because the unrescheduled code required analogs to both `nxi` and `xt`.)

At $u = d$, which may be a fairly high unrolling, the number of base element computations per result has only been reduced to 2. See figure 16 for a graph of the base value cost versus unrollings for stencils of various widths, when the rescheduling optimization is applied.

In general, to find the total operation cost for an aperiodic stencil, we consider the base values and each particular scaling operation separately and add those to the combining operation cost (which usually cannot be reduced by unrolling or elimination of loop common expressions). The total number of extra temporaries is computed similarly, except that in some cases scaling operation optimizations reduce the number of base expression temporaries required. For instance, suppose we are computing the code of figure 12 (on page 19) one result at a time and we are also maintaining the previous 2 scaled values (multiplications by .25). In other words, when we are about to compute `sm[`j`]`, we have stored `.25 * r[`$j-1$`]` and `.25 * r[`j`]`. We only need to have remembered 1 old base value (`r[`j`]`, which we will multiply by .5), not 2; there is no need to remember `r[`$j-1$`]`, which was first computed 2 iterations earlier. One base value temporary is saved for every redundant scaling operation at the trailing edge of the stencil (but usually several temporaries were used for storing scaled values). When the index increases with each iteration, this is the left side, where the smaller indices are used. (This optimization appears again in section 3.3.)

2.3.3 Combining operations

In this section we address the effects of unrolling and common subexpression elimination on reducing the computation of loop common combining operations in periodic stencils. The section is divided

```
for i = 2 to 96 step 2
  nxi = x[i-2]
  xt = x[i-1]
  nxi = nxi + xt
  nxi1 = xt
  xt = x[i]
  nxi = nxi + xt
  nxi1 = nxi1 + xt
  xt = x[i+1]
  nxi = nxi + xt
  nxi1 = nxi1 + xt
  xt = x[i+2]
  newx[i] = (nxi + xt) / 5
  newx[i+1] = (nxi1 + xt + x[i+3]) / 5
```

Figure 15: The code of figure 2, 2-unrolled and rescheduled to consolidate references to each source array element. Temporary variables `nxi` and `nxi1` serve as accumulators for the values of `newx[i]` and `newx[i+1]`, respectively, and `xt` holds a base value—an element of array `x`. The unrescheduled 2-unrolled code appears in figure 17.

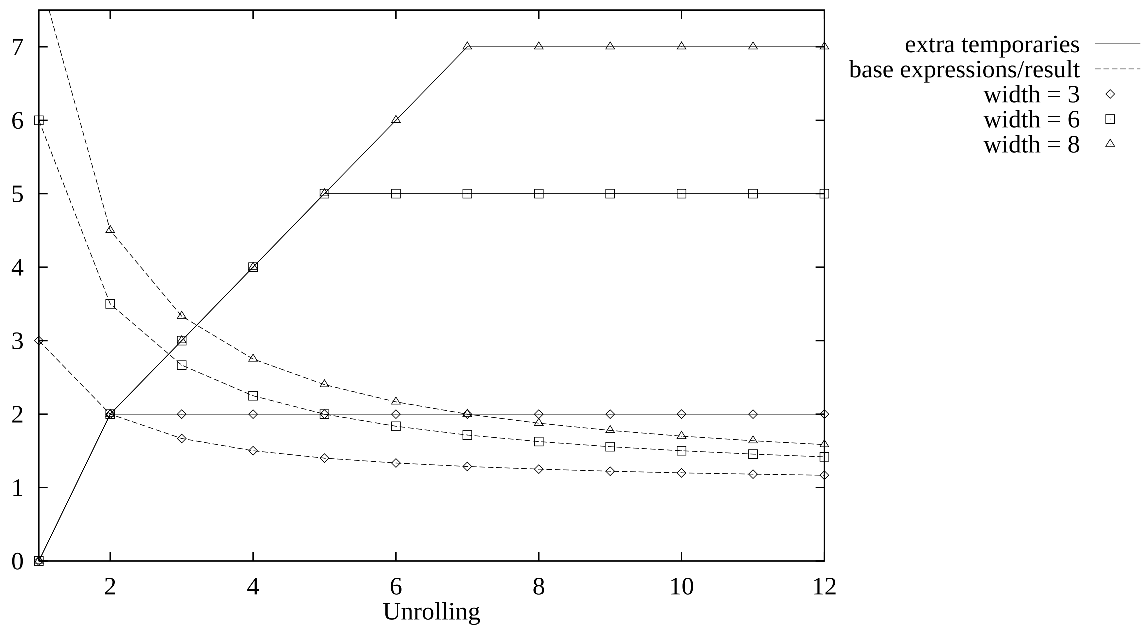


Figure 16: A graph similar to that of figures 13 and 14. The solid lines show how many extra temporaries are required in order to take full advantage of the base expressions exposed as common subexpressions by unrollings from 1 to 12. The number of base expressions evaluated per physical loop iteration is $u + d = u + w - 1$; the dashed lines show this cost prorated over the number of results computed.

```
for i = 2 to 96 step 2
  newx[i]   = (x[i-2] + x[i-1] + x[i] + x[i+1] + x[i+2]) / 5
  newx[i+1] =          (x[i-1] + x[i] + x[i+1] + x[i+2] + x[i+3]) / 5
```

Figure 17: The code of figure 2, 2-unrolled and formatted to emphasize that not only the base values (array references), but also the operations upon them can be shared between these two results. We can see that 3 additions of array elements are identical in the two expressions.

into three parts. The first notes that fixing the parse of the source expression (determining how operations are associated) has severe negative consequences. We propose either the use of an intermediate representation with n-ary operators or a reassociation component of the compiler to correct the problem. The second part demonstrates that choosing common subexpressions is hard; not only is it NP-complete [5, 23], but good heuristics are hard to come by even in the limited problem domain of unrolled stencil computations. We propose a new common subexpression elimination algorithm that performs well on unrolled stencils (and generally); it first determines which computations appear more than once and only then selects some to evaluate. The last part of this section argues that even choosing how much to unroll is a difficult problem; surprisingly, unrolling more may actually degrade performance, even in the absence of poor interactions with the memory hierarchy.

2.3.3.1 Fixed association obscures common subexpressions

In the computation of a periodic stencil, not only can computations be saved in computing the operands, but some of the combining operations can be saved as well. Consider the loop of figure 17; this is just the loop of figure 2, 2-unrolled and formatted to emphasize that 3 of the additions may be shared between the two sums. After computation of the base value array references (which we shall ignore for the remainder of this section), each iteration of this physical loop should only require 5 additions, not 8.

Unfortunately, few compilers would find any shared additions in this code, because most compilers perform common subexpression elimination on either three-address machine code or on a binary tree representing the computations to be performed. No matter how the two expressions of figure 17 are parsed (into left-associative, right-associative, or minimum-height binary trees), they have no addition nodes in common.

There are two obvious solutions to the problem of fixed association. We can permit the nodes of the intermediate representation to have arbitrary degree, so that all of the operands which can be shared are readily accessible; this adds slightly to the cost of operations on the intermediate form. Alternately, we can permit reassociation; for instance, we might transform `(a+b)+c` into `a+(b+c)` in the hope that `b+c` is used elsewhere in the program. This approach is computation-intensive and uncertain of success in the reassociator, but the rest of the compiler is unaffected. These two methods are discussed in more detail on page 57 in section 5.1.4; we assume from now on that this nontrivial problem has been taken care of.

After discovering which computations are repeated in an unrolled loop, we still must choose which common subexpressions to compute; we also must decide how far to unroll loops, if we have a choice.

```
a + b + c
    b + c + d
        c + d + e
            d + e + f
                e + f + g
                    f + g + h
                        g + h + i
```

Figure 18: Simultaneously choosing two common subexpressions to be evaluated can degrade performance, even if the subexpressions are disjoint and equally heavily used. If both `c+d` and `f+g` are computed, then at least 12 additions are required to compute these 7 results. The optimal result—11 additions—is achieved by computing one or the other of `c+d` and `f+g`, but not both.

2.3.3.2 Finding good common subexpressions

To effectively reduce redundant computation in a program, we must first determine which computations *may* be shared, then choose which ones actually *will* be shared. Most compilers leave the first step—determining which computations could be shared—to the vicissitudes of the parser and so might find no common subexpressions at all in the code of figure 18. In that figure, choosing both `c+d` and `f+g` as common subexpression yields poor results. Finding optimial common subexpressions is NP-complete [5, 23], but the potential savings can be large, and fast heuristics can perform well. Such heuristics take as input an expression forest (one tree per result) of n-ary trees, where the arity of associative operator nodes is arbitrary, and return a forest of binary trees which may share structure. We now compare several such heuristics and discuss the tradeoffs involved with using them.

multiple commonest The obvious approach to finding a good binary parse of an expression forest is to choose some of the common subexpressions occurring most frequently. Choosing multiple such subexpressions simultaneously is sub-optimal. Even if two choices appear equally good, selecting one may make the other into a poor choice, even if if no result contains both of the subexpressions. For instance, in figure 18, `c+d` and `f+g` play symmetric roles in the computation, and one must be selected as a common subexpression, but both should not be.

arbitrary commonest The common subexpressions to be evaluated should be selected one at a time, but making an arbitrary choice among those with highest multiplicity is also a bad idea. In the following code, first choosing `c+d` instead of (say) `b+c` results in a total cost of 7 instead of 6 additions.

```
a + b + c
    b + c + d
        c + d + e
            d + e + f
```

leftmost commonest Stencil computations can be ordered in a logical way from left to right. Choosing to evaluate the most commonly occurring subexpressions from one end or the other—but not both—results in a better strategy, than making an arbitrary choice. Figure 19 shows the result of applying this heuristic to a 9-unrolled 5-element periodic stencil. Two more operations are performed than need to be.

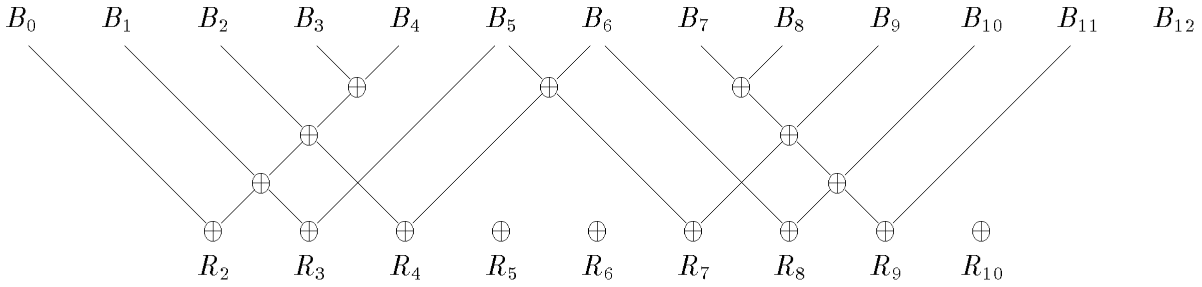


Figure 19: Schematic of a 9-unrolled 5-element periodic stencil, such as the window sum of figures 2 and 17. The B_is are base values and the R_is are results. The pattern of binary addition nodes shown is that chosen by most heuristics (such as leftmost-commonest and leftmost-oldest-commonest) for selecting among the possible execution orders, though those which include randomness or simultaneous choice do worse. Where lines do not connect addition nodes to addends, the node represents the combination of the obvious nodes (for instance, $R_5 = (B_3 + B_4) + (B_5 + B_6) + B_7$), but the associativity of those additions is incidental. The total cost is 21 additions for 9 results, or 2.33 additions per result.

leftmost commonest oldest The "leftmost commonest" heuristic has no history—after the leftmost among the most frequently occurring subexpressions has been chosen, the next round recomputes which subexpressions are most common. A variation on this rule which performs better in some circumstances is to, at the next step, only consider subexpressions which were also under consideration at the previous step—that is, those whose multiplicities at the last step were the same as that of the expression we actually chose. The scope of the search is expanded only when none of the old elements has multiplicity greater than 1. For figure 19, this variation happens to perform identically to the simpler "leftmost commonest" heuristic. Many other variations of the "leftmost commonest" were tested; their performance is uniformly disappointing, even on the restricted subproblem of unrolled stencil-based computations.

leftmost shallowest commonest Common subexpressions that can be incorporated into larger common subexpressions are more valuable than those whose value can be used just once. The "leftmost shallowest commonest" heuristic chooses, among the nodes with highest multiplicity, the one with the smallest expression depth—that is, the one closest to the leaves. There are more opportunities for a shallow node to contribute to future common subexpressions than for a deep one, so given that choosing them has the same immediate benefit, it pays to invest in the one that's more likely to pay off in the future. This argument is far from rigorous, of course: that is why we call the technique a heuristic (the problem is NP-complete). Figures 19 and 20 show the results of two representative algorithms on a 9-unrolled 5-element periodic stencil.

leftmost largest commonest restricting The final heuristic chooses the largest (in terms of number of summands), leftmost common subexpression, but immediately after doing so, restricts its attention to that expression's subexpressions until they are fully parsed. This rule performs optimally for unrolled stencil computations, as demonstrated in figure 21.

Determining which common subexpression elimination heuristics perform best is not easy: even ones which intuitively seem to be good bets perform poorly in some domains. We have, however, discovered which ones give acceptable performance. Two difficulties with many of these heuristics

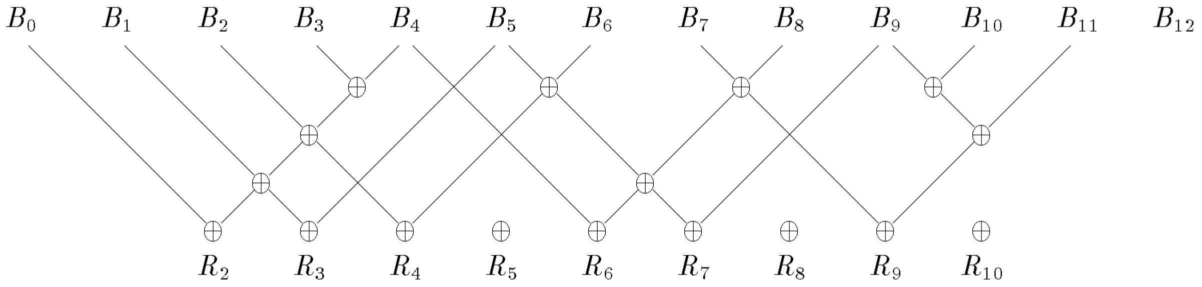


Figure 20: Schematic similar to that of figure 19, but with the leftmost-shallowest heuristic used to choose addition nodes. The total cost is 20 additions for 9 results, or 2.22 additions per result.

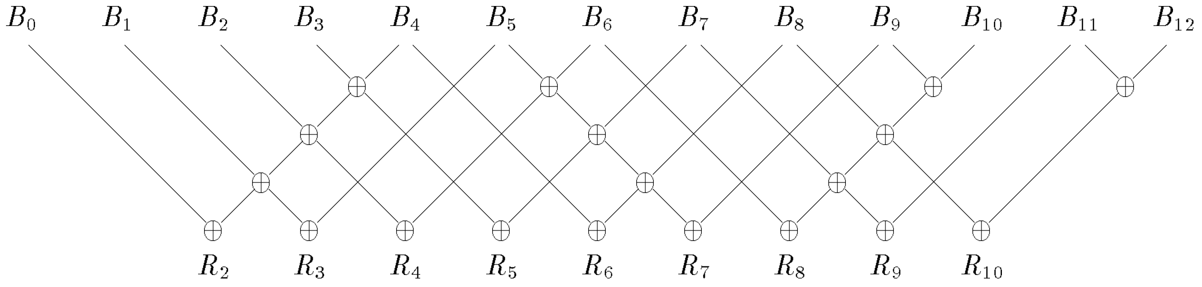


Figure 21: Schematic similar to that of figures 19 and 20 in which an optimal pattern of common subexpressions for the 9-unrolled loop is shown. The total cost is 19 additions for 9 results, or 2.11 additions per result. For a 6-unrolled loop, the cost is to 2 additions per result, as demonstrated by the 12 additions that produce the first 6 results $R_2, \ldots, R_7$.

are that they require an ordering of the arguments left-to-right and they are not local. Nevertheless, they are better than performing exhaustive search.

There is a good, cheap alternative to these complicated heuristics: split the logical loop iterations into groups of $w + 1$ results each (plus one group for the remainder), then optimize each group separately. This works because (as we shall shortly prove) the per-result combining operation cost is minimized at an unrolling of $w + 1$: we can get optimal performance at any particular unrolling by splitting the results into as many groups of size $w+1$ as possible, plus one more group containing the remaining results. Furthermore, at an unrolling of $w+1$, nearly every heuristic described above can find the optimum association, even if they do poorly at other unrollings.

2.3.3.3 Determining how much to unroll

The number of loop common expressions turned into ordinary common subexpressions increases with the amount that a loop is unrolled. Since unrolling by a greater amount has other benefits as well (for instance, the physical loop overhead is prorated over more logical iterations), it seems reasonable to conclude that a compiler should unroll as much as possible, subject to such constraints as the target machine's instruction cache size. Surprisingly, this is not the case: the incremental benefit of unrolling a loop one more time can be negative, and the cost per logical iteration can be higher for a loop that has been unrolled more, even when the other benefits of loop unrolling are factored in and even when memory hierarchy effects are ignored. This section explains why and substantiates the claim that $(w + 1)$-unrolling leads to optimal per-result combining operation

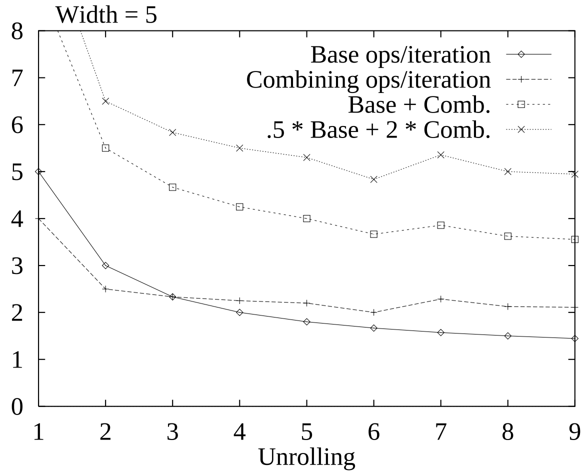


Figure 22: This graph shows the per-result costs, in terms of base expression evaluations and of combining operations, of a 5-element periodic stencil. The graph also shows the combined per-result cost of both base and combining operations, both when they are equally expensive and when the combining operation costs four times as much as evaluating the base expressions. This would be the case when, for instance, the base operations were array references and the combining operations were `min` or `max` macros; if the combining operations were function calls, the cost ratio would be even higher.

---

costs.

Figure 22 illustrates an example in which prorated costs are larger at greater unrollings. The per-result combining operation cost for a 5-element stencil increases by over 14% from a 6-unrolled to a 7-unrolled loop; the per-result combining operation cost for an 8-unrolled loop is 6% higher than for a 6-unrolled one. These effects are even more dramatic for wider stencils. The graph shows the results obtained by the optimal association; if we use an association from figure 19 or 20 instead, then the per-result cost is also greater at an unrolling of 9 than at 8 or 10. The graph also shows the combined cost of the base element evaluations and the combining operations, for two ratios of those individual costs. When the base operations are more expensive than the combining operations, then the reduction in the base expression cost washes out the cost of the combining operations and the total cost is monotonically decreasing until the unrolling is very large. (That plot is not shown.) At very high unrollings it is easier to find instances of negative incremental benefits for additional loop unrolling, but the percentage differences are less, and no one would unroll that much anyway. In each case the loop overhead contributes insignificantly to the per-result cost, so it has not been added in.

Computing any number of w-element sums requires a minimum of $3(w-1)/(w+1)$ operations per result; this minimum can (only) be met at unrollings which are multiples of $w+1$. Showing that this is an upper bound on the minimum is straightforward: we simply give a construction which meets the bound. Proving its optimality at that unrolling, which is also fairly easy, introduces methods that will be used in the subsequent proofs. For this section only, we number both the source and result elements starting at 1.

Theorem 1 *All the results of a $(w+1)$-unrolled w-element periodic stencil can be computed using $3(w-1)/(w+1)$ binary combining operations; furthermore, it is impossible to do better.*

Proof: Suppose that no operations have yet been performed. Regardless of how we associate its w summands, $w-1$ operations are required in order to compute R_1, the leftmost result. Similarly, it

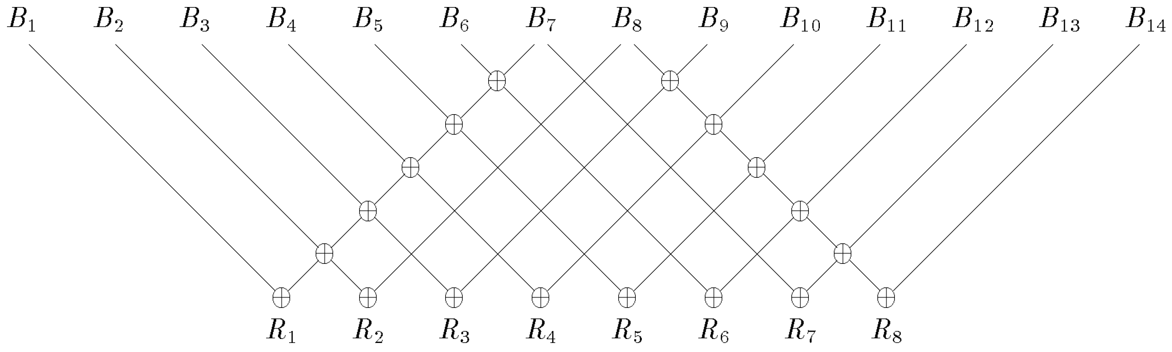


Figure 23: The optimal combining operation cost of 2.25 per result is achieved for the 7-element sum by 8-unrolling. Note that the subscripting conventions are different for this figure than for figures 19–21.

takes $w-1$ operations to produce R_{w+1}, the rightmost result, which shares no summands with R_1. The $w-1$ other results require at least 1 addition operation each—the one that actually produces the result. This proves that $3(w-1)$ operations is a lower bound for producing $w+1$ results and so, when considering $w+1$ results at a time, we cannot better the claimed lower bound of $3(w-1)/(w+1)$ operations per result.

However, we can meet the bound. If we fully right-associate the sum which computes R_1 and fully left-associate that for R_{w+1}, then for any R_i such that $2 \leq i \leq w$, two subexpressions are available that sum to R_i, namely $B_i + \cdots + B_w$ and $B_{w+1} + \cdots + B_{i+w-1}$. The former was computed as a subexpression of R_1 and the latter as a subexpression of R_{w+1}, as illustrated by figure 23. Thus, we can compute the $w-1$ summands $R_2, \ldots, R_w$ with only 1 additional operation each, for a total cost of $3(w-1)$ operations, or $3(w-1)/(w+1)$ operations per result. ■

Now that we have shown that the per-iteration cost of $3(w-1)/(w+1) = 3 - 6/(w+1)$ is an upper and lower bound for $(w+1)$-unrollings, it remains to be shown that this cost cannot be bettered at any other unrolling. For $u < w+1$ we show a stronger result, that it cannot even be equaled, except when $w = 2$. In that case we do not even consider the stencil to be periodic, since there are no combining operations to share.

Theorem 2 *When $u \leq w$ and $w \geq 3$, every association of binary combining operations in a u-unrolled w-element periodic stencil requires at least $3 - 4/w$ operations per result.*

Proof: Computing R_1 requires $w-1$ operations. Since R_1 and R_u share $w+1-u$ summands, they can share up to $w-u$ operations, if both sums are associated correctly. Therefore, R_u's additional cost can be as low as $u-1$ operations. Figure 24 shows this situation.

The remaining $u-2$ results require a minimum of one operation each, even if two expressions that sum to them have already been computed in the course of computing R_1 and R_u. The total cost for the u results is therefore lower-bounded by $(w-1)+(u-1)+(u-2) = w+2u-4$, and the per-element cost is $(w+2u-4)/u = 2 + (w-4)/u$, which decreases monotonically with u. Since we assumed that $u \leq w$, its minimum is $3 - 4/w$. ■

Proving that even at larger unrollings we cannot do better than $3(w-1)/(w+1)$ combining operations per result is more intricate; we defer that proof to appendix A and here show a weaker result.

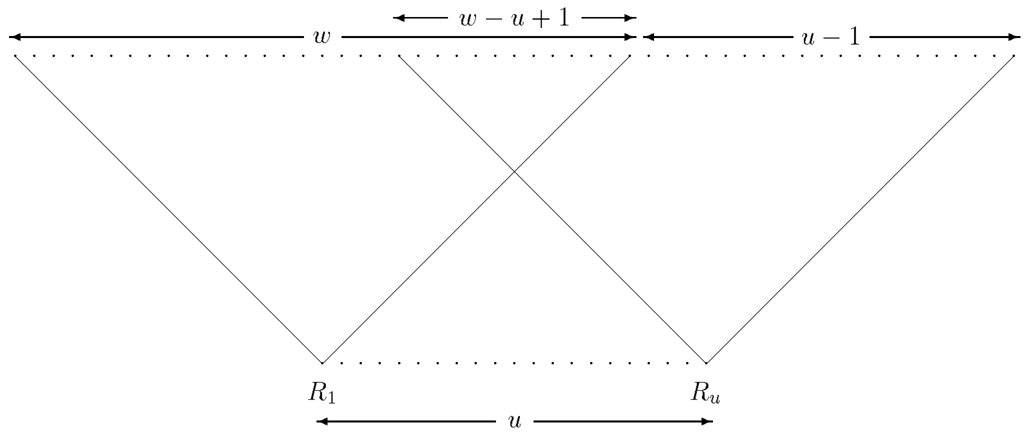


Figure 24: When $u < w$, R_1 and R_u share $w - u + 1$ summands. After R_1 and the base values have been computed, computing R_u can cost as little as $u - 1$ more combining operations.

Theorem 3 *Computing u w-element periodic stencils requires at least* $2(u - \lfloor (u-1)/w \rfloor) + w - 3$ *combining operations, which is no less than* $2(w-1)/w$ *operations per result.*

Proof: This theorem is a corollary of the proofs of the preceding two theorems.

Every wth result $(R_1, R_{w+1}, \ldots)$ requires $w-1$ operations; there are $1 + \lfloor (u-1)/w \rfloor$ such results. R_u requires an additional $(u-1) \bmod w$ operations, at best, and there are $u - \lfloor (u-1)/w \rfloor - 2$ additional results to be computed, each requiring a minimum of 1 more operation. The total operation cost is

$$\left(1 + \left\lfloor \frac{u-1}{w} \right\rfloor\right)(w-1) + ((u-1) \bmod w) + u - \left\lfloor \frac{u-1}{w} \right\rfloor - 2$$

Since $\lfloor (u-1)/w \rfloor w + ((u-1) \bmod w) = u - 1$, we can rewrite this as

$$2u + w - 2\left\lfloor \frac{u-1}{w} \right\rfloor - 3 = 2\left(u - \left\lfloor \frac{u-1}{w} \right\rfloor\right) + w - 3 \, .$$

The per-result claim of the theorem follows from the fact that $\lfloor (u-1)/w \rfloor / u < 1/w$. ■

2.4 Loop common expression exposure and rerolling

Unrolling loops and performing common subexpression elimination cannot reduce the number of loop common expressions in a physical loop, only prorate the cost over a greater number of logical loop iterations. As a result, the method often requires large amounts of unrolling in order to approach its best performance. Although we show in the next chapter a direct way to remember the values of loop common expressions from iteration to iteration, first we briefly investigate the use of unrolling and common subexpression elimination to do so. In particular, we introduce and evaluate the methods of massive unrolling and edge linking.

Finding reused base expressions and scaling operations is fairly easy, so this section focuses on detecting loop common combining operations. It is worthwhile to to try to find loop common expressions even if unrolling is an option, because exploiting loop common expressions is usually superior to unrolling, even though the theoretical lower bound on combining operations per result is lower for unrolling. The bound for unrolling is about $3 - 6/w$, but that value is only attainable at relatively high unrollings; at lower unrollings the number is $O(w)$. The bound for loop common

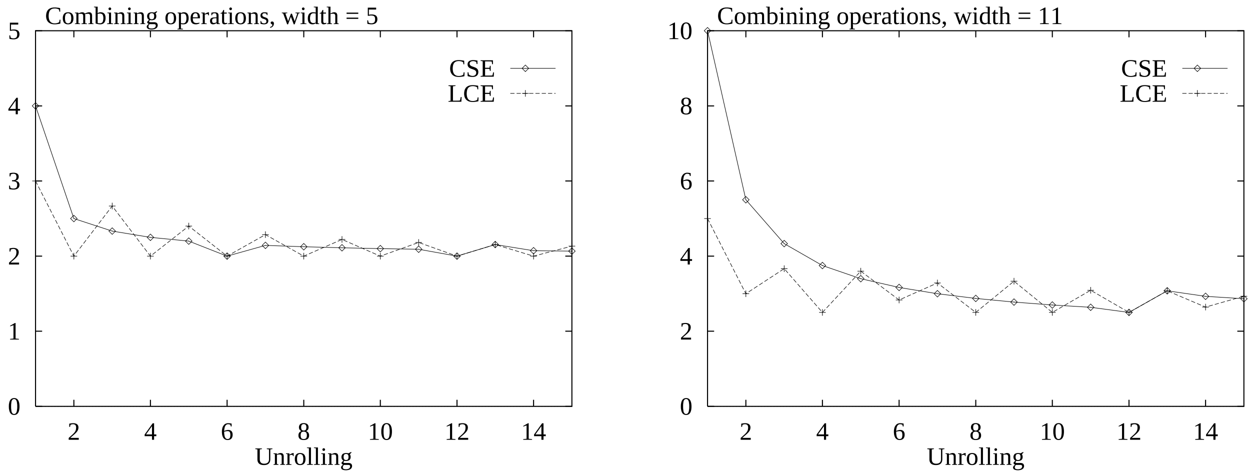


Figure 25: The tradeoffs between (optimal) common subexpression elimination and loop common expression elimination, when considering combining operations in periodic stencils. While unrolling and then performing common subexpression elimination can achieve better results, that requires wide stencils and quite a bit of unrolling. At low amounts of unrolling, loop common expression elimination is superior, though it is more sensitive to the exact amount of unrolling.

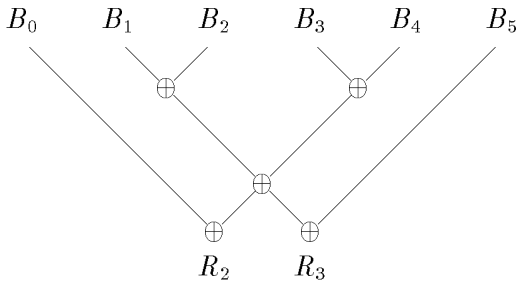


Figure 26: $B_3 + B_4$ is a loop common expression, since it can be used at this iteration and at the next one (when it plays the role played here by $B_1 + B_2$).

expression elimination is about 4, but even if no unrolling is done the method reduces combining operations per result to less than $2 \log_2 w$. Figure 25 graphs the tradeoffs between the two methods.

The key to finding loop common expressions is forcing the subexpressions computed by two physical iterations to line up. In figure 26, the next iteration of the physical loop computes R_4 and R_5. For that physical iteration, the value $B_3 + B_4$ can play the role played by $B_1 + B_2$ at this iteration. Since $B_3 + B_4$ need not be recomputed, only 4 additions, not 5, are required per physical loop iteration. Ordinary common subexpression elimination does not construct computations that can be shared between physical loops because it is greedy: it attempts to optimize the current computation (the current loop iteration, basic block, etc.) without regard for other users of its subexpressions. Common subexpression algorithms arrange that the expressions computed are used by as many results as possible, and this means choosing to compute expressions near the center of the unrolled loop, not near the edges. Figures 19 and 20 give evidence of this proclivity: at the left side of the physical loop iteration, the sums are primarily right-associative, and on the right side, they are mostly left-associative.

We present two methods for overcoming the local, one-iteration view of ordinary common subexpression elimination. The first, *massive unrolling*, unrolls sufficiently to dilute or nullify the edge effects caused by the fact that some computations near the edges of the loop are used less frequently than others near the center. The second method, *edge linking*, forces the edges to line up by either wrapping the computation into a ring or, whenever a sum is chosen on one edge, choosing the corresponding one on the other edge.

2.4.1 Massive unrolling

The massive unrolling method performs ordinary common subexpression elimination on heavily unrolled loops. Common subexpression elimination algorithms tend to associate computations away from the edges of the unrolled computation, but if the unrolling amount is sufficiently large, then the center is far from the edges and will be optimized in a non-greedy way even by an ordinary common subexpression elimination algorithm. After the algorithm has run, we look in the center of the expression forest for a pattern of loop common expressions among the nodes chosen. To put it another way, we snip out a section of the middle whose ragged edges mesh with one another.

It is possible that no pattern of common subexpressions that double as loop common expressions can be found. If they can, we presume that they optimize the combining operations well, because that was the criterion for choosing them as good common subexpressions. Register-to-register moves may also be required because of a need for the old and new values to be active simultaneously. For instance, in figure 26, $B_3 + B_4$ will eventually need to occupy the register which holds $B_1 + B_2$, but the former cannot be evaluated into that register because the two values need to be added. Moving $B_3 + B_4$ into the location where it is expected by the next loop iteration is an example of altruistic work done by one iteration that saves work for another iteration.

We can arrange that no base elements are recomputed; this requires $w - u$ register-to-register moves (or zero, if $u \geq w$). If the stencil is periodic, then not all of the base elements need to be saved (for instance, B_3 in the example above), and the *total* number of register-to-register moves is $w - u$. The number of scaling operations is reduced in a similar fashion; we leave the details as an exercise for the interested reader.

While this method can save work, it is an ineffective way to find loop common combining expressions. Here we examine a few of the scheme's failings.

First, the loop must be very heavily unrolled before a pattern emerges in its center, because edge effects (which result from the common subexpression algorithm's greediness at the edges of the unrolled loop) can affect the pattern of expressions which computes a result up to w elements distant from either edge. In order to find a pattern of size u, the loop must be unrolled to compute $3w + u$ results, which requires $4w + u$ source elements. (Another way to say this is that the u centermost results share no base expressions with the leftmost and rightmost results.)

The unrolling amount actually required depends on the common subexpression elimination algorithm chosen. For instance, when using the leftmost-oldest-commonest heuristic, a pattern of size 4 becomes apparent for a width-7 periodic stencil when u-unrolling for $u \geq 20$ (and also for $u = 17$, if the arbitrary associations that it does not fully specify happen to line up correctly, but not for $u = 18$ or 19). The pattern can first emerge for a width-9 stencil after 22 unrollings. The shallowest-commonest heuristic does better on the 7-element stencil but worse on the 9-element one. (These heuristics are defined on pages 25ff.)

Computing common subexpressions for such a large expression forest noticeably slows the compiler, but smaller ones are not guaranteed to produce an acceptable result. Even after finding the common subexpressions in the large unrolled loop, we still must find a pattern in them. This is a complicated problem, but the difficulty is assuaged by the fact that we only have to look in the center of the expression forest. (Chapter 3 gives a better loop common expressions algorithm which only requires one search of the base expressions.)

Finally, we have no control over the size of the repeated computation that we find—that is, the number of results it computes. If it is larger than the maximum tolerable unrolling, it is useless. This is a reason why the optimal common subexpression elimination algorithm may be undesirable: the result requires $(w+1)$-unrolling. We may need to decide how much we are willing to unroll before choosing the common subexpression elimination heuristic.

In the next section we give another way to find loop common expressions by using ordinary common subexpression elimination; the next chapter gives a more direct way for doing the same thing.

2.4.2 Edge linking

Rather than unroll enough to mitigate the edge effects by distance from the edges, we can attempt to eliminate edge effects altogether by guaranteeing that the edges mesh. The effect is of a circular rather than linear set of iterations.

There are several similar ways to achieve this effect; the two most straightforward are to reduce the source and result array indices modulo some value, or to arrange that whenever a computation near one edge is performed, its mate at the other edge is also chosen. Indices that are affected by the modulus reduction (we must keep track of which ones are) represent cross-iteration temporary values—that is, loop common expressions. The biggest problem with this technique is that it requires an unrolling of at least $u = w$ so that no two source array elements are mapped onto the same element in the circle; in fact, we must have $u > w$ because at $u = w$ all of the results are identical. If we unroll that much, we might as well just use the optimal association. We want to use loop common expressions to permit a small unrolling amount, but this technique is no help in doing so. We can try to extract a smaller pattern, but there is no guarantee that one can be found. This technique also suffers from the fact that some unrollings are inherently better than others, as graphed in figure 22.

On the other hand, if the unrolling amount is fixed beforehand (perhaps by other constraints), this is an effective way of determining what loop common expressions to use. It also illustrates the advantages of the non-optimal common subexpression elimination heuristics. For instance, when a width-5 periodic stencil is 8-unrolled, 17 operations are required. No loop common expressions exist if these are arranged as a block of width $6 = w + 1$ and another of width 2. Non-optimal heuristic are likely to arrange the computations so as to reveal at least one loop common expression, reducing the cost to 16 combining operations per 8 results. Furthermore, the pattern of computations is actually only 2 results wide and so we can see that such a large unrolling is not really necessary after all.

Finding loop common expressions by unrolling and doing ordinary common subexpression elimination can work, but the unrolling amount, either of the expression upon which common subexpression elimination is run or of the resulting loop, can be unmanageably large. Loop common expressions do not naturally fall out of common subexpressions in unrolled loops. The technique has the merit of applying familiar technology, but we will shortly see a method for finding loop common expressions which is nearly as easy to implement and is conceptually simpler to boot.

Chapter 3

Loop Common Expression Elimination

This chapter gives direct methods for eliminating redundant computation of loop common expressions (expressions that appear in more than one loop iteration). This represents an improvement over the unrolling and common subexpression elimination method of chapter 2, which could only reduce the number of redundant computations per logical loop iteration, not eliminate them entirely. Loop common expression elimination is simple to implement, runs efficiently in a compiler, and is always applicable. The method achieves its greatest incremental gains at small unrolling amounts; if large unrollings are permissible, other methods are superior (see, for example, section 2.4.1 on page 32). Even at small unrollings, loop common expression elimination is nearly as optimal common subexpression elimination at high unrollings. Application of loop common expression elimination requires the use of a modest number of extra temporary variables; fewer temporaries are required by this method than by unrolling.

Loop common expression elimination is similar to common subexpression elimination in that after an expression's first appearance, its value is placed in a temporary storage location and retrieved whenever it is needed thereafter. Since it is cheaper to retrieve the value than to recompute it, the resulting program is more efficient. Loop common expression elimination is a good method for improving the running time of a parallel program when run on a machine with limited parallelism, because different virtual processors which are emulated by the same physical processor can share results without incurring any communication cost. This method can also speed up parallel programs even when the virtual processor ratio is low, and it is applicable to some serial algorithms as well (see section 7.3).

Computing and storing a value that can be used by the next loop iteration incurs space and time costs. Extra space is needed to store values across loop iterations. The time cost comes from two sources. First, the value may have to be moved to the storage location. Second, the optimal way to compute the current loop iteration's result may not compute the loop common expression—it might associate operations differently, for example. While arranging that the loop common expression is available for the next loop iteration causes extra work to be done, such costs are typically more than offset by the work done by the previous loop iteration to help the current one.

As an example, consider again the 5-element periodic stencil, exemplified by the window sum of figure 2. It must be 6-unrolled in order to reduce the cost of combining operations to 2 per result by unrolling and common subexpression elimination alone; figure 27 shows this. We can do just as well

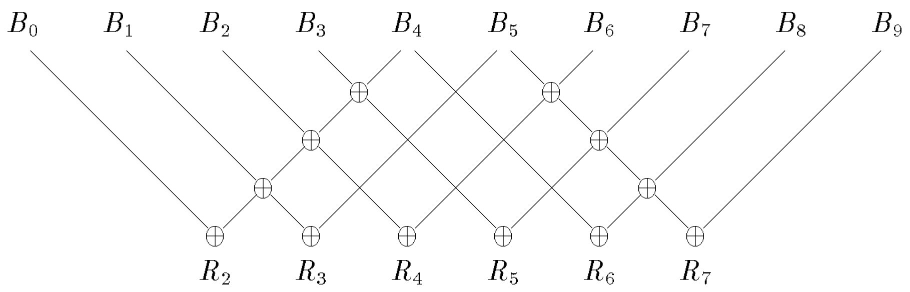


Figure 27: The optimal combining operation cost achievable via unrolling a 5-element periodic stencil (such as the window average of figure 2) is 2 per result, achieved by 6-unrolling.

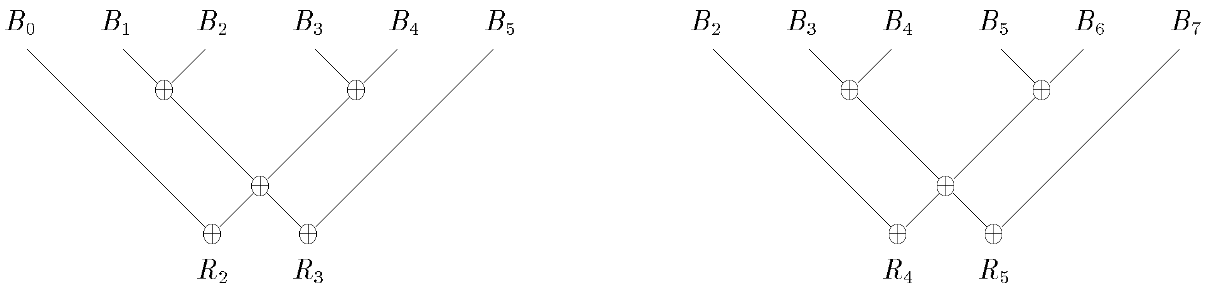


Figure 28: Two physical iterations of a 2-unrolled 5-element periodic stencil; each of the physical iterations contains 2 logical iterations and so computes 2 results. $B_3 + B_4$ is a loop common expression, since it can be used at both the physical iteration on the left and the one on the right. $B_5 + B_6$ is also a loop common expression which can be used at the next physical iteration. When loop common expressions are taken into account, the cost per result is 2 combining operations, just as in figure 27.

by 2-unrolling and using loop common expressions. Figure 28 shows the computation of 2 results; this appears to require 5 operations. However, on the next loop iteration we can reuse the value $B_3 + B_4$ and avoid doing the figure's leftmost addition. The number of combining operations, after a single startup operation, is just 4 per 2 results, or 2 per result.[4] The number of registers required to hold intermediate values is also decreased from 5 to 3. Figure 6 gives the code corresponding to figure 28.

We now describe simple methods for finding and eliminating loop common expressions in scaling operations, in base expressions, and in combining operations.

3.1 Finding patterns

Finding loop common expressions is difficult because the lexical expressions differ even though their instantiations in different loop iterations are identical. To get around this difficulty, we temporarily replace all (possibly shifted) instances of the index variable with a distinguished value. For instance, the loop bodies of figures 1 and 2 become, respectively,

```
y[i] = f(·) * g(f(·))
```

[4]Since $B_1 + B_2$ and $B_3 + B_4$ need to be live simultaneously, 1 extra register-to-register move per physical iteration may be needed; 4-unrolling eliminates it entirely.

```
for i = 2 to 398
  sm[i] = (1/16) * r[i-2] + (1/4) * r[i-1] + (3/8) * r[i] +
            (1/4) * r[i+1] + (1/16) * r[i-2]
```

Figure 29: Unoptimized code for convolution by the binomial filter of width 5, which is commonly used for smoothing digital signals. Array `r` contains the input signal, and the smoothed result is placed in array `sm`.

and

`newx[i] = (x[·] + x[·] + x[·] + x[·] + x[·]) / 5 .`

This representation makes it easier to find common expressions. Base expressions—in these cases, `f(i)` and `x[i]`—are then discovered by ordinary common subexpression elimination. Scaling operations are whatever is left after combining operations; ordinary common subexpression elimination can identify which ones are shared.

The key to the methods discussed in this chapter is finding patterns in the input stencil. We do not look for common expressions at particular distances; rather, we find common expressions at any distance and use as many of them as possible.

We now discuss loop common expression elimination applied to scaling operations, base expressions, and combining operations in turn. Optimization of these parts individually is conceptually simpler than considering the computation as a whole.

3.2 Scaling operations

Once a stencil has been separated into base expressions, scaling operations, and combining operations, dealing with the scaling operations is quite straightforward. As in section 2.3.1, we initially consider only aperiodic stencils with exactly two instances of a particular scaling operation. We defer the optimization of base expressions to section 3.3 on page 40.

The algorithm is as follows. Allocate enough temporary storage locations to hold the values of the scaling operation applied to the last d base values, where d is the number of loop iterations between uses of the loop common scaling expression. This guarantees that the value which is needed at the current iteration, which was computed d iterations previously, is still available. Treat the d storage locations like a first-in first-out queue, storing a new value into each location after using its old value. The only additional requirement is the addition of some startup code to fill the queue before entering the loop. This method works analogously when a scaling operation is used more than two times, as in the stencil $\langle a, 0, 0, a, 0, 0, 0, a \rangle$; a single queue still suffices.

In order to optimize several scaling operations at once, we create several queues and perform the optimization on them individually. Figures 29 and 30 show the result of applying the method to convolution by the binomial filter of width 5, whose stencil is $\langle \frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16} \rangle$.

Several criticisms can be made of the code of figure 30. A minor one is that we cannot fill in new array values until the old ones have been used (or safely moved elsewhere), which results in the use of 2 extra temporary variables `s1` and `s2`. These variables are not necessary in this particular example, because the code scheduler can reorder the computations to require no more temporaries than (the best scheduling of) the unoptimized version of this loop. An extra temporary and/or register-to-register move is sometimes required; it can be eliminated by expanding the size of the array by one element. (See section 3.3.1 on page 40 for a discussion of when and how to adjust the array size.)

```
t16[0] = (1/16) * r[0]
t16[1] = (1/16) * r[1]
t16[2] = (1/16) * r[2]
t16[3] = (1/16) * r[3]
t4[0] = (1/4) * r[1]
t4[1] = (1/4) * r[2]
i16 = 0
i4 = 0
for i = 2 to 398
  s1 = (1/4) * r[i+1]
  s2 = (1/16) * r[i+2]
  sm[i] = t16[i16] + t4[i4] + (3/8) * r[i] + s1 + s2
  t4[i4] = s1
  t16[i16] = s2
  i4 = (i4 + 1) mod 2
  i16 = (i16 + 1) mod 4
```

Figure 30: Elimination of loop common scaling expressions in the code of figure 29. The scaling operations $\frac{1}{16}$ and $\frac{1}{4}$ are used aperiodically with $d_{16} = 4$ and $d_4 = 2$. The arrays `t16` and `t4` act as queues, holding previous results of the scaling operations until they are needed.

A more significant complaint is that for each division saved, we have introduced an array reference, an addition, and a modulus reduction. The additions can be eliminated by performing the modulus reductions directly on the loop variable `i`. (When the modulus is a power of 2, as in this case, the modulus reduction can be replaced by a bitwise and operation.) Also recall that the scaling operation could be a function call or other expensive operation(s) and so its cost could swamp the array overhead. The cost of the array-accessing operations need not be a worry, as the next section show how to eliminate them.

3.2.1 Unrolling to scalarize arrays

This section presents a method, based on unrolling and ordinary compiler optimizations, for changing an array used as a queue into a set of scalar variables. An array is a convenient queue representation because the code for each loop iteration is identical and no unrolling is required. Nevertheless, temporary variables are preferable to an array because they can be kept in registers and carry no overhead for accessing or storing, and no index variables need be maintained. The disadvantage of using a set of scalar variables is that each temporary variable holds the value of a particular lexical expression, so different instantiations of a loop common expression may be placed in different variables.

The simplest way to maintain a queue with a set of registers is to arrange them in a chain and shift live values forward when the first element is no longer needed. In figure 30, such code would look like

```
i16 = 0
for i = 2 to 395 step 4              for i = 2 to 395 step 4
   ... = ... t16[i16] ...               ... = ... t16[0] ...
   t16[i16] = ...                       t16[0] = ...
   i16 = (i16 + 1) mod 4
   ... = ... t16[i16] ...               ... = ... t16[1] ...
   t16[i16] = ...                       t16[1] = ...
   i16 = (i16 + 1) mod 4        ==>
   ... = ... t16[i16] ...               ... = ... t16[2] ...
   t16[i16] = ...                       t16[2] = ...
   i16 = (i16 + 1) mod 4
   ... = ... t16[i16] ...               ... = ... t16[3] ...
   t16[i16] = ...                       t16[3] = ...
   i16 = (i16 + 1) mod 4
```

Figure 31: An example of scalarizing an array by unrolling the code of figure 30. Since the value of `i16` is 0 at the beginning of every physical loop iteration, constant folding can convert the loop on the left side to that on the right. Then the array elements, which are not used or set elsewhere, can be placed in registers or changed into temporary scalar variables.

```
... t16_0 ...
t16_0 = t16_1
t16_1 = t16_2
t16_2 = t16_3
t16_3 = a[i+2] / 16
... t16_3 ...
```

Each variable has a fixed meaning—for instance, `t16_1` contains the third most recently computed value; that value will be reused by the next iteration. The d temporary locations can be allocated in registers; the array implementation used d locations in main memory. There is no array reference overhead, but $d-1$ register-to-register moves are used per *physical* iteration. The register-to-register moves can be eliminated entirely by u-unrolling, where $u \geq d$, though then $(u \bmod d) + d$ temporary variables are required; this is d iff d divides u.

A more direct way to eliminate array referencing and register-to-register move operations is to unroll the code of figure 30 and apply standard optimizations. Figure 31 shows part of the unrolled code before and midway through optimization; we have concentrated on the 4-element array `t16`.

Scalarizing arrays is an underappreciated but highly valuable effect of loop unrolling. When the unrolling amount is a multiple of the array size, the array references become compile-time constants. The compiler can treat each array reference as a variable whose home location in the store is known, just like any other local variable.[5] The resulting code should be exactly the same as when scalar temporaries were used in the first place, whether or not the entire array fits in the machine's registers.

If the unrolling amount is not a multiple of the array size, the array can be scalarized by expanding the array until its size divides the unrolling amount. Regardless of how big this makes the array, constant folding and liveness analysis reduce the number of storage locations required to

[5] Determining that no other program commands read or write the storage location is made more difficult if the array is declared globally or the language permits arbitrary pointer manipulations.

$(u \bmod d) + d$, the same number as are required in the register case. There is never any real reason to use temporaries instead of arrays as the queue abstraction, since arrays can be conveniently resized and both methods usually result in the same machine code after optimization. Temporary scalars should be used, however, if the unrolling amount is constrained to be small and the queue size is just a little larger than that, in which case the overhead of temporary scalars is less than an array's would be.

3.3 Base values

Base values are maintained analogously to scaled values: the last several values are remembered so that each base expression need only be evaluated once. In this section we discuss why loop common expression elimination should be performed on base expressions last, why recursive application of loop common expression elimination is worthwhile, and when and how to change the sizes of the queues used for loop common values.

Base value optimization is performed after scaled values are optimized because the scaling optimizations may reduce the number of base values that need to be maintained. Once a scaled value is computed, the base value may not need to be remembered—only the scaled value need be stored. Thus, the space requirements for optimizing both base and scaled values can be little or no more than those for optimizing base values alone. For instance, in the binomial stencil $\langle \frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16} \rangle$, although each base expression appears in 5 logical iterations, it only appears in 3 iterations of the loop of figure 30, because each scaling operation is applied to each base element only once. We previously saw this optimization on page 22 of section 2.3.2 as a reduction in the number of temporary variables required to take full advantage of an unrolled loop. Here the same savings apply, namely 1 storage location for each loop common scaling operation at the left edge of the stencil.

Recall that the base values of a stencil may involve the loop iteration variable more than once. On page 12 we noted that when optimizing we should treat the stencil $\langle 2, 3, 2, 2, 3, 2 \rangle$ exactly like $\langle 1, 0, 0, 1 \rangle$ with base value $2y_{i-1} + 3y_i + 2y_{i+1}$. In this case, computations of the base values themselves contain loop common expressions: the base values at elements 22 and 24 are $2y_{21} + 3y_{22} + 2y_{23}$ and $2y_{23} + 3y_{24} + 2y_{25}$, respectively. When computing these values, we can share not only the y_i expressions, but also some of the multiplications: in other words, this base expression is itself a stencil! After a stencil computation has been split into its base expressions and scaling operations, it pays to perform loop common expression optimization on both the base expressions and the scaling operations. The base expressions of $\langle 2, 3, 2, 2, 3, 2 \rangle$ can be optimized. An example whose the scaling operations can be optimized is $h(g(f(i))) + g(f(i+1)) + f(i+2)$, in which the base expression is $f(\cdot)$ and the scaling operations are $\langle h(g(\cdot)), g(\cdot), \cdot \rangle$. A part of the pattern of scaling operations may yield a new base expression—in this case, the first two scaling operations can be considered a stencil with base expression $g(\cdot)$ and scaling operations $\langle h(\cdot), \cdot \rangle$.

3.3.1 Adjusting the sizes of temporary arrays

This section discusses tricks for adjusting the sizes of the queues used to maintain temporary values from iteration to iteration. Changing the array size to be compatible with the unrolling amount can permit the array to be scalarized, significantly reducing overhead (section 3.2.1). The technique is equally applicable to scalar variables as to an array representation: in the scalar case, the savings are primarily eliminated register-to-register moves. Naturally, all of these optimizations

```
t1 = y[0]
t2 = y[1]
t3 = y[2]
t4 = y[3]
for i = 4 to 92 step 4
  x[i] = t1 + t3 + t4 + y[i+4]
  t1 = y[i]
  x[i+1] = t2 + t4 + t1 + y[i+5]
  t2 = y[i+1]
  x[i+2] = t3 + t1 + t2 + y[i+6]
  t3 = y[i+2]
  x[i+3] = t4 + t2 + t3 + y[i+7]
```

Figure 32: Computation of stencil $\langle 1,0,1,1,0,0,0,0,1\rangle$ using a queue of length 4. (A queue of length 3 also suffices, but more complicated code results and a temporary variable must be used.) Two array references are required per result; to reduce that number to one, the queue must have length 8.

are only worthwhile if repeating the computation is more expensive than maintaining the queue; reducing the queue maintenance cost makes the method of loop common expression elimination more attractive and applicable to even more expressions. We start by discussing maintaining large arrays, then extend our techniques to deal with several arrays at once.

When an array is large (compared to either the maximum unrolling amount or the number of registers available), it can pay to reduce its size to cover only some of its uses. For instance, consider the stencil $\langle 1,0,1,1,0,0,0,0,1\rangle$. It could be worthwhile to forget the base value after the rightmost computation, but after computing it for the next time, to then remember it for 3 more loop iterations so that it can be reused twice. Figure 32 illustrates the result. In this case, to eliminate one base value computation requires a queue of length 1, to eliminate a second requires an additional 2 queue elements, and to eliminate a third requires 5 more than that.

The cost of queue maintenance is not linear in the number of storage locations. The cost is increased significantly if the number of registers is exceeded (because some values must be spilled and later reloaded from memory) or if the maximum permissible unrolling is exceeded (because then register-to-register moves or array references are unavoidable). We also saw that if the unrolling is more than, but not a multiple of, the queue size, then eliminating register-to-register moves or array references may require up to twice as many storage locations as the queue size.

Another way to reduce the number of storage locations and the unrolling required to eliminate queue operations is to use two smaller queues rather than one large queue. The stencil $\langle 1,0,1,0,0,0,1,1,1\rangle$, for instance, could have its storage cost reduced to 4 locations while computing the base value only twice by using 2 arrays, each of size 2.

If exterior constraints prevent us from 8-unrolling but permit us to 4-unroll, we can still eliminate all repeated computations. We would use 2 queues of size 4 and at each logical iteration move a value from the head of the first queue to the tail of the second one. The additional cost is 1 register-to-register move per logical loop iteration. We can even split arrays more than once, but the returns diminish quickly. If the original stencil had size $d+1$ and the original queue of size d is split into $\lceil d/u\rceil$ queues of size u (the unrolling amount), then the per-physical iteration cost is $u(\lceil d/u\rceil - 1)$ register-to-register moves. The $u = d/2$ case is illustrated in figure 33.

Sometimes it is desirable to make an array slightly larger or smaller. For instance, we may want

```
t1 = y[0]
t2 = y[1]
t3 = y[2]
t4 = y[3]
r1 = y[4]
r2 = y[5]
r3 = y[6]
r4 = y[7]
for i = 4 to 92 step 4
  nx = t1 + t3 + t4
  t1 = r1
  r1 = y[i+4]
  x[i] = nx + r1
  nx = t2 + t4 + t1
  t2 = r2
  r2 = y[i+5]
  x[i] = nx + r2
  nx = t3 + t1 + t2
  t3 = r3
  r3 = y[i+6]
  x[i] = nx + r3
  nx = t4 + t2 + t3
  t4 = r4
  r4 = y[i+7]
  x[i] = nx + r4
```

Figure 33: Computation of stencil $\langle 1,0,1,1,0,0,0,0,1\rangle$ using two queues of length 4 rather than one of length 8. No repeated computation occurs, but use of the smaller queues requires the extra register-to-register moves `t`i `=` `r`i.

to adjust the array size to be the same as that of another array (so that we can use a single index into both) or the same as the unrolling (so that it can be scalarized). Changing the array size may also reduce other costs—for instance, if the size is a power of two, then modulus operations can be performed with a single bitwise operation.

Increasing a uniform array's size is trivial: just add the extra elements, as mentioned in section 3.2.1 on page 39, and then some of its values are dead across the loop iteration boundary. Decreasing the array's size, on the other hand, makes a value unavailable. We can either recompute it when it is needed or remember it until then, which requires another queue and an operation to move the value from one queue to the other; in the simplest case, when we only need to decrease an array's size by 1 element, the entire additional cost is 1 register-to-register move per logical iteration. Significantly shrinking an array can be costly.

A final technique, for use as a last resort and only when the queue representation is an array, is to partially unroll and then convert a single array into two interleaved ones. For instance, most of the physical loops in this paper are unrolled by at least 2, so we could use separate arrays for odd and even values. This helps to reduce the modulus. For instance, if the modulus is 2, then the add-and-modulus operation can be expressed as a single arithmetic operation: if $i \in \{0,1\}$, then

$((i+1) \bmod 2) = 1 - i$. Further unrolling would be even cheaper, but might not be possible.

3.4 Combining operations

Periodic stencils offer an opportunity for loop common expression optimization of combining operations. For stencils of width w, combining operation costs decrease from $w-1$ to no more than $2\lfloor \log_2 w \rfloor$ per result, even without unrolling. For u-unrolled loops, the upper bound is less than $4+2(\log_2 w)/u$ combining operations per result. Meeting these bounds requires w storage locations. We assume that $u < w$, because when the unrolling is very large, we can do better with the techniques of section 2.3.3, which discover no loop common combining operations. We sometimes used large unrollings to show when unrolling with common subexpression elimination does well, but the constraint that $u < w$ is reasonable if w is large. We further assume that $u = 2^i$ for some integer i, because if u is not a power of 2, the storage requirements become excessive and the operation bound also increases somewhat.

We first discuss the combining operation costs, then the storage requirements, of the code produced by our algorithms; we also prove the bounds claimed.

3.4.1 Combining operation costs

The algorithm for non-unrolled loops is extremely simple: we compute each result by building a minimum-height binary tree, associating the largest binary subtrees leftward, left-associating the combining operations, and reusing computations wherever possible. This arrangement makes the number of distinct heights at which nodes are found (the number of *levels* in the tree) equal to the conventional definition of the tree's height (the length of the maximum-length path from the root to a node). If at least 2 subexpressions represent the combination of 2^i values, for any i, a queue is allocated for them, so they can be reused. Using a minimum-height tree minimizes the number of results that must be remembered and so the number of queues. Naturally, we must fill the queues before the first loop iteration.

Theorem 4 *By using a minimum-height binary tree and loop common expressions wherever possible, a 1-unrolled periodic stencil of width w can be computed using as many combining operations as the number of levels in the tree, which is bounded between* $\log_2 w$ *and* $2\lfloor \log_2 w \rfloor$.

Proof: We can establish both bounds on the number of levels in the tree by considering the binary representation of w. Summing 2^i values requires a tree with i levels of operation nodes. We can sum 2^j values, for any $j < i$, without using any additional levels (all such sums are loop common expressions when the 2^i-element sum has already been computed).

Summing w values requires $b - 1 + b_1 - 1$ operations, where the binary representation of w has b bits, b_1 of which are set to 1. If $w = 2^i$, the operation total is $i = \log_2 w$; otherwise $b = \lceil \log_2 w \rceil = \lfloor \log_2 w \rfloor + 1$ and $b_1 \leq b$, which gives us the upper bound.

The number of combining operations required per result is equal to the number of tree levels. Exactly one combining operation is performed per level because we presumed that, if there was more than one operation at a particular level the values were maintained as loop common expressions. ■

If the loop is u-unrolled, we can share even more operations without jeopardizing the use of loop common expressions. The following simple algorithm combines two previously-seen algorithms in a straightforward way.

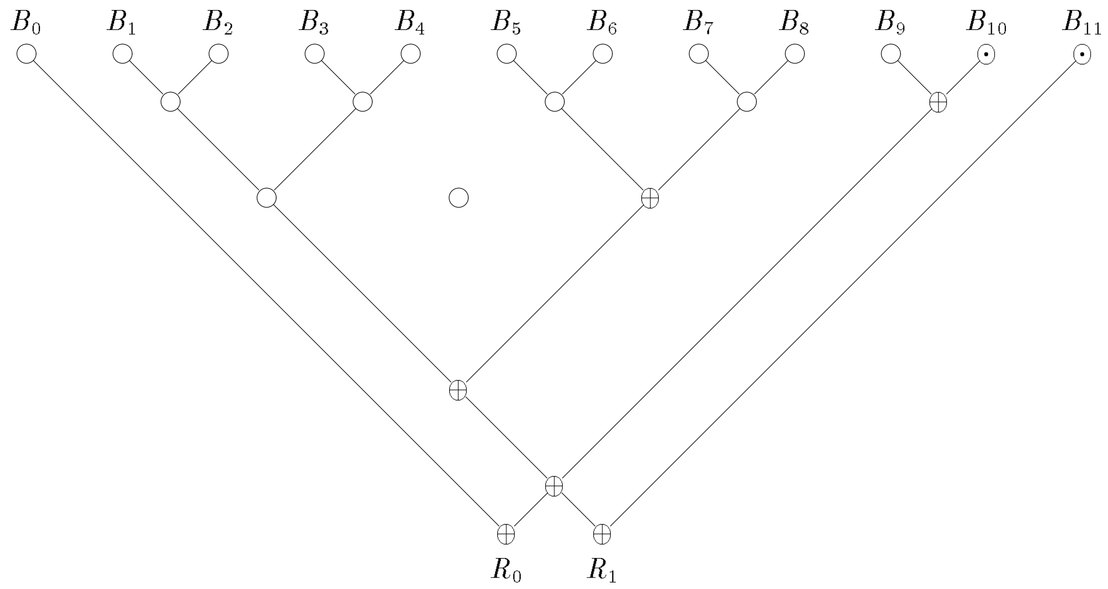


Figure 34: Optimizing combining operations by eliminating loop common expressions from a 2-unrolled periodic stencil of width 11. The dotted circles are base elements which must be computed at this (physical) loop iteration, and the circles containing plus signs represent combining operations which must be performed at this iteration. Empty circles are values that have been computed by a previous iteration. The total cost per 2 results is 2 base expression evaluations and 6 combining operations.

First, sum the centermost $c = w - u + 1$ operations, taking advantage of loop common expressions where possible. Call this value C. This sum (but no larger one) is a subexpression of every result, so each result is now a sum of u subexpressions. Associate those sums as advantageously as possible according to the methods of section 2.3.3.3; it is possible to take advantage of a few more loop common expressions at this step.

Figure 34 illustrates the method for $w = 11$ and $u = 2$, and figure 35 gives a more intricate example in which $w = 29$ and $u = 8$. (These numbers are not outrageously large; some vision applications average 64×64 blocks of pixels [112].) If u is not a power of 2, then the previously-computed values may not conveniently line up with the ones used at this iteration. Since we build a binary tree, most results can be used twice without going through the contortions of the algorithm for finding loop common expressions which was given in section 2.4.2 on page 33 and which has no guaranteed performance lower bound anyway.

Theorem 5 *The algorithm given above computes u w-element periodic stencil results using* $4u + 2\lfloor \log_2(w - u + 1) \rfloor - 2\log_2 u - 4$ *combining operations.*

Proof: Theorem 4 showed that we can compute C with $2\lfloor \log_2 c \rfloor$ operations by using loop common expressions and a minimum-height binary tree with $c = w - u + 1$ leaves. We require a few more operations than that in this case, however. Because the loop is u-unrolled, none of the values depending on the rightmost u base values have been computed yet. This means that computing the first $\log_2 u$ levels of the tree requires $u - 1$ operations; the other (up to) $2\lfloor \log_2 c \rfloor - \log_2 u$ levels require 1 operation each.

We have now reduced each result to the sum of u subexpressions. Theorem 2 on page 29 showed that we can compute u overlapped u-element sums with $3u - 4$ combining operations. In fact, if

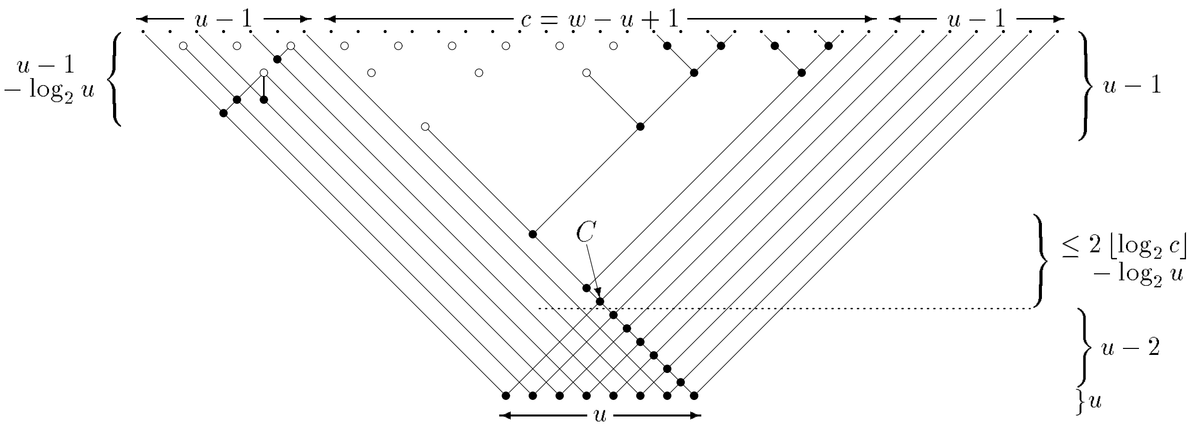


Figure 35: Loop common expression elimination applied to a periodic stencil with width $w = 29$ and unrolling $u = 8$. Dark nodes are computation done at this physical iteration; light nodes are loop common expressions that need not be recomputed. Combining operation costs are noted at the sides of the diagram.

we have associated C so that the largest complete binary tree is on the far right or far left, then $(\log_2 u) - 1$ of the combining operations are loop common expressions, which reduces this part of the cost to $3u - 3 - \log_2 u$ operations.

The total operation count for the u results, then, is

$$(u-1) + (2\lfloor \log_2 c \rfloor - \log_2 u) + (3u - 3 - \log_2 u) = 4u + 2\lfloor \log_2 c \rfloor - 2\log_2 u - 4$$

combining operations, which proves the theorem. It is easy to see that this is less than $4 + 2(\log_2 w)/u$ operations per result, which verifies the claim made at the beginning of this section. ■

We conjecture that by slightly modifying the algorithm and performing a more sophisticated analysis, the bound can be reduced by approximately 1 operation per result, making it even more competitive with common subexpression elimination, even at larger unrollings.

3.4.2 Space requirements

This method appears to require a large number of temporary storage locations, because we remember base values, plus values at $\lfloor \log_2 u \rfloor - 1$ tree levels. In fact, a *total* of w locations can store all the base values and results of combining operations that will be reused. (We might have required $w - 1$ storage locations to hold the base elements alone.) Furthermore, these w storage locations can be arranged as a single queue (or as several smaller ones, if desired).

The key to reducing the storage requirements to a single w-element queue is noticing that, because we associated the largest of the b_1 binary trees to the left, every value is the left argument of only one combining operation. Furthermore, after that use, the value is not used by subsequent physical loop iterations and so need no longer be remembered as a loop common expression. For instance, in figure 34, after $B_5 + B_6$ has been used to compute $(B_5 + B_6) + (B_7 + B_8)$, it will not be needed again, so we can use one storage location for both of these values. (In fact, this can be the storage location that originally contained B_5, which is not used again after the computation of $B_5 + B_6$.) Despite the replacement of some old queue values, the $(\log_2 u) - 1$ loop common expressions that will be used on the left side of the physical loop will be available.

Chapter 4

Loop Differencing

This chapter introduces the method of *loop differencing* for optimizing the combining operations of periodic stencils. The method is easy to implement, and the resulting code requires only 2 operations per result, no unrolling, and no extra temporaries (except any used to maintain base values). While loop differencing requires less unrolling and results in fewer operations per result (when width $w > 3$) than our other methods for exploiting loop common expressions, loop differencing is not without its drawbacks. Loop differencing requires that the stencil combining operator be associative and commutative and have an inverse. The method can be numerically unstable, and it is applicable only to periodic stencils and certain aperiodic ones with distributive scaling operations. Loop differencing has some similarities with iterator inversion [44, 52, 53, 104, 105]; see section 8.3 on page 75 for more about iterator inversion.

Adjacent logical iterations of a periodic stencil (one whose combining operations form a repeating pattern) can theoretically share all but one of each of their computations, but fixing a particular association of the operands prevents most pairs of iterations from sharing that many. For instance, consider the 7-element periodic stencil $\langle 1, 1, 1, 1, 1, 1, 1 \rangle$, which we represent as $x_i = y_i + y_{i+1} + y_{i+2} + y_{i+3} + y_{i+4} + y_{i+5} + y_{i+6}$. After computing x_{22}, we could compute x_{21} with only 1 operation if we had associated x_{22} as

$$x_{22} = (y_{22} + y_{23} + y_{24} + y_{25} + y_{26} + y_{27}) + y_{28} \ .$$

Similarly, we could compute x_{23} with only 1 operation if the previous sum had been

$$x_{22} = y_{22} + (y_{23} + y_{24} + y_{25} + y_{26} + y_{27} + y_{28}) \ .$$

However, we cannot start from x_{22} and compute both x_{21} and x_{23} with 1 operation apiece, because x_{22} was computed one way or the other (or neither); it could not have contained both $(y_{22} + y_{23} + y_{24} + y_{25} + y_{26} + y_{27})$ and $(y_{23} + y_{24} + y_{25} + y_{26} + y_{27} + y_{28})$ as subexpressions. This is the crux of the proofs in section 2.3.3.3 and appendix A that width-w periodic stencil computations require at least $3(w - 1)/(w + 1)$ combining operations per result.

When computing x_i, we would like to have computed, as a subexpression of x_{i-1}, the sum of the leftmost 6 (more generally, $w - 1$) elements needed by x_i, so the new result can be calculated with a single operation. If the previous iteration didn't compute the sum of those $w - 1$ elements, we can compute it ourselves, then pay 1 extra operation to get this iteration's total by adding in the element we don't have in common with the previous iteration. Computing the $w - 1$-element sum could take up to $w - 2$ operations, depending on how the previous iteration's operations

```
rs = y[0] + y[1] + y[2] + y[3] + y[4] + y[5]
for i = 0 to 94
  rs = rs + y[i+6]
  x[i] = rs
  rs = rs - y[i]
```

Figure 36: Loop differencing applied to the stencil $\langle 1,1,1,1,1,1,1\rangle$. Rather than computing each result from scratch, the difference between two adjacent results is added to one result to compute the next.

were associated. We will see how to compute it efficiently by working backward from the final result rather than forward from subexpressions that have already been computed. This technique computes the $w-1$-element sum with a single operation, leading to a total cost of 2 combining operations per result.

4.1 Inverting the combining operator

Even if we did not compute the sum of the $w-1$ shared summands as a subexpression of the previous result, we can find its value with a single operation by applying the following identity, which we have illustrated using the 7-element stencil introduced earlier in this chapter.

$$\begin{aligned} x_i &= y_i + (y_{i+1} + y_{i+2} + y_{i+3} + y_{i+4} + y_{i+5} + y_{i+6}) \\ x_i - y_i &= (y_{i+1} + y_{i+2} + y_{i+3} + y_{i+4} + y_{i+5} + y_{i+6}) \\ x_{i+1} &= (y_{i+1} + y_{i+2} + y_{i+3} + y_{i+4} + y_{i+5} + y_{i+6}) + y_{i+7} \\ x_{i+1} &= (x_i - y_i) + y_{i+7} \end{aligned}$$

This arithmetic identity holds regardless of how x_i was actually calculated, if the combining operator (here, $+$) is associative and commutative and has an inverse. Commutativity and the inverse were both required in order to get from the first line to the second. Applying an inverse can lead to a loss of precision; see section 4.3 for a discussion. Some combining operations—such as the minimum operator—have no inverse.

Programs embodying this idea compute a running sum into which new values are added and from which old ones are subtracted. Figure 7 on page 8 shows the code produced by loop differencing for a 5-element periodic stencil.

We have used a subexpression of the previous result, but one that was not computed in the course of producing that result. To create this subexpression, it proved cheaper to work backward from the final result than forward from its subexpressions that had already been computed.

This is interesting because we have used an inverse to go backward, something which is rarely worthwhile in straight-line code. For instance, to compute

```
a = X + Y
b = Y + Z
```

where X, Y, and Z are arbitrary expressions, it is always more efficient to use

```
t = Y
a = X + t
b = t + Z
```

than

```
t = X
a = t + Y
b = a - t + Z
```

and so ordinary common subexpression elimination algorithms do not even attempt this transformation. This use of the inverse is only useful when `a` has been computed at some other location in the program, but Y has not, and the extra cost of computing X (and doing a subtraction) is less than the cost of computing Y. Most compilers would try to store the value of Y while `a` was being computed, and if that was not possible, would give up on performing common subexpression elimination on the computation of `b`.

4.2 Differencing

A perhaps more intuitive way to determine how to obtain one result from another is by symbolically computing the difference between the two results. This method gives the loop differencing method its name, but in most cases we need not resort to it, for we can simply recognize patterns in the stencil's scaling operations. Computing these differences is more general than pattern-matching, however, and can be used when it fails.

As an example of the method, we can compute the difference between two iterations of the 7-element window sum as follows:

$$
\begin{array}{rrcl}
 & x_{i+1} & = & \phantom{y_i +{}} y_{i+1} + y_{i+2} + y_{i+3} + y_{i+4} + y_{i+5} + y_{i+6} + y_{i+7} \\
- & x_i & = & y_i + y_{i+1} + y_{i+2} + y_{i+3} + y_{i+4} + y_{i+5} + y_{i+6} \\
\hline
 & x_{i+1} - x_i & = & -y_i \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad + y_{i+7}
\end{array}
$$

We can perform just 2 operations to get from one result to the next.

In this case (and in the examples that follow) we found a good result by comparing a loop iteration with the immediately preceding one. In general the method requires examining multiple iterations. This can be expensive, though finding good base elements helps, since in most cases shifting by the width of a base element is most advantageous.

4.2.1 Aperiodic stencils

Loop differencing is applicable not only to periodic stencils, but also to some aperiodic ones, if the scaling operation is distributive over the combining operation. Distributivity permits base values to be broken into parts; the scaling operation is applied to each part and the resulting values are combined. Weighted window sums—the most common sort of stencil—satisfy this property; any linear function satisfying the equation $f(x + y) = f(x) + f(y)$ distributes over addition.

We now examine a few types of aperiodic stencil to which loop differencing is applicable. These examples are important enough, and common enough, that recognition of them should be built into a compiler, and symbolic loop iteration differencing would be performed only if none of the built-in patterns were recognized. Like any optimization, in most cases symbolic loop differencing is not applicable, so other optimizations such as loop common expression elimination should also be attempted.

The following diagrams show only scaling operations, not base or combining operations, for clarity.

4.2.1.1 Arithmetic sequences

Consider the stencil $\langle 5,4,3,2,1\rangle$, whose scaling operations (coefficients) form an arithmetic sequence. When we compare two iterations with one another, we get the following result:

$$\begin{array}{ccccccc} & & 5 & 4 & 3 & 2 & 1 \\ - & 5 & 4 & 3 & 2 & 1 & \\ \hline & -5 & 1 & 1 & 1 & 1 & 1 \end{array}$$

The difference looks no easier to compute than the results themselves. Although it includes only 1 non-unity scaling operation, it has 6 summands, which is more than the results do. All is not lost because the difference contains loop common expressions and so can be computed efficiently. In particular, the difference can be split into the stencils $\langle -5\rangle$ and $\langle 1,1,1,1,1\rangle$. The former requires 1 scaling operation, and loop differencing permits us to compute the latter with 2 combining operations per result.[6] Adding these 2 stencil sums to the previous result takes 2 more combining operations. Our total cost per result (after base values) to compute $\langle 5,4,3,2,1\rangle$ is not 4 scaling operations and 4 combining operations but 1 scaling operation and 4 combining operations. Obviously, this trick works for any arithmetic sequence, regardless of the first value or the (constant) difference between adjacent coefficients.

We can compute an ascending and descending arithmetic sequence in a similar way, except that the extra scaling and combining operations are not needed:

$$\begin{array}{ccccccccccc} & & 1 & 2 & 3 & 4 & 5 & 4 & 3 & 2 & 1 \\ - & 1 & 2 & 3 & 4 & 5 & 4 & 3 & 2 & 1 & \\ \hline & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

The stencil $\langle 1,2,3,4,5,5,4,3,2,1\rangle$, which has no distinguished middle element, is no more difficult to compute than this one. Either requires just 6 combining operations to compute a new result: 2 each to maintain the values of the stencils $\langle -1,-1,-1,-1,-1\rangle$ and $\langle 1,1,1,1,1\rangle$, and 2 more to add those values to the previous result. We can do even better by noticing that the best way to optimize the stencil $\langle -1,-1,-1,-1,-1,1,1,1,1,1\rangle$ is like $\langle -1,0,0,0,0,1\rangle$ with base expression $\langle 1,1,1,1,1\rangle$. This requires no more temporary storage locations than the previous method but saves an additional 2 combining operations per result.

We saved several combining and scaling operations by considering the difference between full results rather than splitting $\langle 1,2,3,4,5,4,3,2,1\rangle$ into two smaller stencils, each an arithmetic sequence, and optimizing them individually. More to the point, using loop differencing is significantly better than using loop common expression elimination to prevent reapplication of the scaling operations 2, 3, etc. A compiler should always check whether loop differencing is applicable before attempting loop common expression elimination.

A practical use for weighted window sums with coefficients in two arithmetic progressions is data windowing. Convolving with a square window results in “leakage” because the window turns on and off rapidly and so has substantial high-frequency components. The leakage can be reduced by using a window function which changes more gradually from zero to a maximum and back to zero. One recommended function is the N-element Parzen window (or the similar Bartlett window) [108],

[6] A stencil of this size can also be optimized to 2 combining operations per result by using unrolling (chapter 2) or loop common expression elimination (chapter 3).

which ramps linearly approximately from 0 to 1 and back down again:

$$w_j = 1 - \left| \frac{2j - (N-1)}{N+1} \right| , \qquad 0 \le j \le N-1 \; .$$

4.2.1.2 Geometric sequences

Stencils whose coefficients form a geometric sequence succumb to the same sort of analysis, except that instead of symbolically subtracting a previous result from the one we wish to compute, we perform elementwise division. For instance, consider the stencil $\langle a^5, a^4, a^3, a^2, a \rangle$:

$$\begin{array}{ccccccc} & & a^5 & a^4 & a^3 & a^2 & (a) \\ \div & (a^5) & a^4 & a^3 & a^2 & a & \\ \hline & & a & a & a & a & \end{array}$$

We have enclosed in parentheses the results which cannot be considered (since both dividing by zero and dividing zero are meaningless in this context). We can compute a new result with only 2 combining operations and 2 scaling operations:

$$\begin{array}{ccccccc} & a^5 & a^4 & a^3 & a^2 & a & \\ - & a^5 & & & & & \\ + & & & & & & 1 \\ \times a & & & & & & \\ \hline & & a^5 & a^4 & a^3 & a^2 & a \end{array}$$

In this figure, "$\times a$" represents a single scaling operation applied to the entire sum. This method works for any $a > 0$, including negative and fractional values.

If the stencil's last scaling coefficient is not a, then 1 more scaling operation per timestep is required. (This cost could be absorbed into the cost of computing the base values, but it must be paid at some point.)

If multiplication by $1/a$ is preferred to multiplication by a (for instance, the latter is inexact because $a = 1/3$), then the stencil $\langle a^5, a^4, a^3, a^2, a \rangle$ can be treated as $\langle b, b^2, b^3, b^4, b^5 \rangle$, where $b = 1/a$, and processed from right to left instead of from left to right.

We saw on page 50 that when a stencil's coefficients contain ascending and descending arithmetic sequences, similar methods to those used for simply ascending or descending sequences could be used to extract even more performance. Differencing can indicate when a stencil's coefficients form an ascending and descending geometric sequence, but there is no simple-to-compute value which can be added to or multiplied by a previous result in order to obtain a new one. The best method is to split the stencil into two simple geometric sequences and optimize them separately. Although it may seem that we are giving up the opportunity to avoid some recomputations (for instance, of multiplication by a^2 in $\langle a, a^2, a^3, a^4, a^3, a^2, a \rangle$), we save much more than we would have by optimizing those operations—in fact, no scaling by a^2 need be done explicitly!

Practical examples of stencils whose coefficients form a geometric sequence abound. Most physical systems suffer exponential decay as exemplified by the unit response of a resistor-capacitor circuit to an impulse of current or voltage; such circuits are frequently used to model other systems [117, 119]. The circuit's decay is mirrored exactly by such a stencil, except that subtraction of the last term (a^5 in our example) is an artifact of the limited window size and the desire for quick parallel evaluation. It is interesting that in this example, the serialized code is more natural and more like the system being modeled than the parallel implementation.

Convolution by the ascending-and-descending geometric stencil $h[n] = a^{|n|}$ is used to smooth digital signals. It also provides an excellent approximation to the blurring effect of an imperfect lens or out-of-focus imaging system [69, 119].

4.3 Numerical stability

The chief disadvantage of the loop differencing method is its potential numerical instability. In practice, instability is not a serious problem, because it occurs only when one base value is much larger than the sum of the others and when inexact floating-point operations are used. Loop differencing guarantees exact results when exact operations such as integer addition and logical operations are used.

Numerical instability has two sources. First, some mathematical methods are not guaranteed to converge; certain inputs produce an incorrect answer or no answer at all. A simple example is Newton's iterative method $r_{n+1} = r_n + f(r_n)/f'(r_n)$ for finding a root of an equation [45, p. 128], which does not converge for the function $f(x) = \sqrt[3]{x}$ when the initial estimate $r_0 = \pm 1$. The second, more interesting, type of numerical instability results when the computer implementation of a well-defined mathematical method produces incorrect results because of the computer's finite precision or approximations to ideal mathematical operations. As an example, consider calculating all the non-negative integer powers of the golden ratio $\phi = (\sqrt{5} - 1)/2$. Since $\phi^{n+1} = \phi^n - \phi^{n-1}$, the results can be obtained iteratively by subtraction, without any exponentiation. However, this method gives completely wrong answers by about $n = 16$ because of roundoff error and the admixture of the other solution, $(-\sqrt{5} - 1)/2$, to the recurrence [108, p. 27].

We have shown that, if the combining operation is associative and commutative and has an inverse, the code resulting from application of the loop differencing optimization produces the same results as did the original, unoptimized code. (This is the definition of a correct optimization.) Although reasonable computer implementations of such operations as floating-point addition and multiplication are commutative, they are not necessarily associative or distributive and may not have true inverses [87]. The IEEE standard [33, 70, 74, 75] guarantees commutativity, and a limited form of associativity (to within rounding error) is provided by the inclusion of denorms [32, 34], which ensure that $x - y = 0$ iff $x = y$ and that $(y - x) + x$ returns y even if $y - x$ underflows. As an example of associativity failure, consider adding the author's mass (67.7 kg) to the rest mass of a proton (1.67×10^{-27} kg), then subtracting out the author's mass again. We would hope to be left with the proton's mass, but in fact on most computers we get

$$\texttt{(1.67e-27 + 67.7) - 67.7} \quad \Longrightarrow \quad \texttt{0}\ .$$

This error is due to the computer's limited precision, not to spontaneous proton decay [96].

Errors can be introduced by roundoff error (for instance, `1/3` might equal .33333333, which, multiplied by 3, produces .99999999, not 1), by underflow (as illustrated above), or by overflow (which is similar). The order in which operations are performed can have an important effect on the final result.

In floating point additions and subtractions, roundoff error amounts to about half of the least significant bit (call this quantity ϵ) in the result. If the magnitudes of the operands and the results are approximately equal, this is not too bad, and the order of the operations does not matter. When nearly equal numbers are subtracted from one another, then the relative error can be increased even when the absolute error remains approximately the same [87, 108]; in the proton example above, 67.7 is a good approximation to $67.7 + 1.67 \times 10^{-27}$, but 0 is a bad approximation to 1.67×10^{-27}, even though both approximation err by the same amount, 1.67×10^{-27}.

Even if the roundoff error for a particular sum is small, it can accumulate from repeated additions. After n results have been computed, the cumulative error (or *drift*) may be as great as $2n\epsilon$ (suppose that all the subtractions were rounded down, while all the additions were rounded up). If the roundoff occurs up and down with equal probability, the expected drift is only $\sqrt{2n}\epsilon$, but properties of the data or the machine implementation may produce the worse result [108]. This problem can be solved by computing the sum from scratch (i.e., without inverses) every so often; this brings the error back down to zero and resynchronizes the results with what they ought to be. Another trick is to work in both directions (left and right) from wherever the result is known to be accurate; this reduces the number of operations between resynchronizations by half.

Chapter 5

Implementation Issues

The methods of this report have been implemented in a compiler optimization phase which was used to generate most of the examples in this report and the results presented in chapter 6. (The implementation ordinarily produces C, but it was modified to produce pseudocode instead for the examples of this report, in order to avoid burdening the user with C's baroque `for`-loop syntax.) The methods are simple, so we spend only a little time discussing our particular implementation and leave the details to the reader. In the second part of the chapter we point out how these optimizations interact with other compiler optimizations and present two alternative ways to compute stencils, one taking advantage of distributivity and the other using scans to perform combining operations.

5.1 Details of the implementation

Our improved common subexpression elimination algorithms, loop common expression elimination, and loop differencing have been implemented in a prototype compiler. The compiler, which consists of little besides those optimizations, peephole optimizations, and a back end which outputs C [84], proves the practicality of these optimizations. While we discuss the compiler's performance only on stencil kernels, it operates on entire programs. Adding an existing C front end [81] would have been easy, but it was more direct to use the compiler's intermediate representation as input. Construction of the compiler occupied only a few weeks of work.

The compiler is written in GNU Emacs Lisp [94, 91, 144] because of its excellent programming environment. The intermediate language is a medium-level list-based multiple-arity representation; converting to a low-level representation such as three-address code would have obscured the structure of the program and made the compiler's job harder, not easier.

At present the compiler only recognizes stencils whose scaling operations are multiplications and whose base expressions are array references; the combining operation can be arbitrary. Extending the compiler to recognize arbitrary scaling operations would be straightforward. We have not done so because the compiler is intended as a proof of concept, not a production-quality optimizer.

The input need not be in any stylized form (such as that required by the CM convolution compiler [22]). An initial simplification step exposes stencil computations by folding constants, consolidating expressions, distributing operations, ordering expression operands canonically, and performing other peephole optimizations. The stencil may contain some base elements which are not used at all, or some which are used without a scaling operation.

If a stencil computation is found, it and the containing loop are replaced by a higher-level construct, and optimizations are performed on this more tractable form. It explicitly lists the scaling operations, the base expression, the combining operation, the relative offset of the result (which we have ignored elsewhere in this report), and several types of information about the context in which the result is used. Contexts such as `f(·+2)` are used to represent computation on an unknown value, particularly for base expressions and scaling operations.

The compiler chooses base expressions that are as large as possible, following the rules given in section 1.4 on page 12. The optimizer first attempts to apply loop differencing (only pattern matching, not symbolic loop differencing, has been implemented to date). If loop differencing is not applicable, loop common expression elimination is tried. If either of these methods succeeds, the resulting code can either compute one result at a time (as most of our examples do) or reference each source array element only once (as does the code of figure 15 on page 23), depending on what resources are scarcest. If neither method succeeds, the stencil is computed naively.

5.1.1 Connections with other optimizations

It is important that standard compiler optimizations be linked with the ones presented in this paper; the synergy results in better code than either could produce alone. For instance, peephole optimizations and code reorganization can expose apparently unstructured computations as stencils, and further simplifications are convenient midway through our optimizations to simplify the code and to evaluate compile-time constants. Elimination of loop common expressions may enable a compiler's other optimizations to be performed more effectively (for instance, by removing false data dependences); see section 7.3 for an example.

Optimizations which improve the use of the memory hierarchy, such as loop jamming or scheduling loop iterations onto particular processors, often reduce the opportunity for optimizations based on loop common expressions but can still be successfully applied after loop common expression elimination.

5.1.2 Wide base elements

Most of the base elements in our examples contain only a single instance of the loop index variable, but we saw on page 12 that it is more profitable to consider some stencils to have wider base elements. For example, consider the periodic stencil $\langle 1,2,1,2,1,2,1,2\rangle$, whose pattern has period 2 and occurs 4 times. As for any width-8 stencil, eight base elements must be remembered if none are to be computed; when the base elements have period greater than 1, some of these base elements overlap. Each result only combines 4 of the queue elements, however; we effectively have two interleaved queues, each of size 4.

Figures 37 and 38 show code for this stencil before and after loop differencing optimization. We have departed from our practice of giving pseudocode to show actual C input and output. For clarity, some variables have been renamed, and some optimizations have been disabled in order to make the code simpler. Some such disabled optimizations are:

1. Temporaries have been allocated as a two-dimensional array; ordinarily they would be stored in a one-dimensional array (or, more commonly, in registers).

2. Modulus operations have been left in the program rather than being converted to bitwise operations; ordinarily `x % 4` would be transformed into `x & 3`.

3. Arithmetic operations performed modulo 2 have not been simplified; ordinarily `(x + 1) % 2` would be transformed into `1-x`.

```
int eight_elt()
{
  int i;
  for (i=minindex; i<=maxindex; i++)
    R[i+2] = S[i]   + 2 * S[i+1]
           + S[i+2] + 2 * S[i+3]
           + S[i+4] + 2 * S[i+5]
           + S[i+6] + 2 * S[i+7];
}
```

Figure 37: Original C code for the eight-element stencil $\langle 1, 2, 1, 2, 1, 2, 1, 2\rangle$. `S` is the source array, and `R` is the result array.

4. Unrolling has been disabled in order to keep the code small; unrolling would reduce loop overhead, scalarize arrays, and eliminate some arithmetic operations.

5.1.2.1 Partial patterns

We can conveniently process stencils such as $\langle 1, 2, 3, 1, 2, 3, 1, 2\rangle$ whose last base element is incomplete. We use the entire base expression ($\langle 1, 2, 3\rangle$ in this case) as our loop common expression, and the code generated is nearly the same as it would have been if the last base expression had been complete. Each base expression is first used before it is fully computed, but the full base expression (which will be used by one or more future loop iterations) is placed on the queue. This is a good example of altruistic computation, since the full value placed on the queue is not used by the current loop iteration. Figure 39 gives an example for the case when each base expression is computed all at once; the details are similar when the code should refer to each expression containing a particular loop reference (in figure 39, array references) only once.

5.1.3 Loop initialization

Since each loop iteration assumes that the previous iteration has computed some results and left them in temporary storage locations, we must add initialization code to set up these variables for the first few iterations.

The initialization code is trivial to compute: we simply run a modified copy of the main loop for $w - 1$ iterations. The modified copy fails to store results, or even to compute them, but every expression which is needed to compute a loop common expression is computed.

Figure 38 gives an example; although $w = 4$, the initialization loop runs 6 times because there are two interleaved queues to be initialized.

5.1.4 Reassociation

Section 2.3.3.1 showed that if a compiler is to discover common subexpressions effectively, the parse of the source program must not be fixed ahead of time. In most existing compilers, however, the common subexpression elimination phase operates on a binary tree representing one parse of the input program; only common subexpressions made explicit by that particular parse will be discovered. The result can be very poor code in certain important situations, such as unrolled loops.

```
int diff_eight_elt()
{
  int B[2][4] = { { 0, 0, 0, 0 },
                  { 0, 0, 0, 0 } };
  int RS[2] = { 0, 0 };
  int i;
  int thisshift = 1;
  int thisoffset = 0;
  for (i=minindex; i<=minindex+5; i++)
    {
      B[thisshift][thisoffset] = S[i] + 2 * S[i+1];
      RS[thisshift] += B[thisshift][thisoffset];
      thisshift = (thisshift + 1) % 2;
      if (thisshift == 0)
        thisoffset = (thisoffset + 1) % 4;
    };
  for (i=minindex+6; i<=maxindex+6; i++)
    {
      RS[thisshift] -= B[thisshift][thisoffset];
      B[thisshift][thisoffset] = S[i] + 2 * S[i+1];
      RS[thisshift] += B[thisshift][thisoffset];
      R[i-4] = RS[thisshift];
      thisshift = (thisshift + 1) % 2;
      if (thisshift == 0)
        thisoffset = (thisoffset + 1) % 4;
    };
}
```

Figure 38: Optimized C code produced by the implementation for the eight-element stencil $\langle 1, 2, 1, 2, 1, 2, 1, 2\rangle$ after loop differencing. `S` is the source array, and `R` is the result array. Temporary arrays `B` and `RS` hold the base values and the running sums. There are two running sums, one each for even- and odd-indexed results, and also two four-element queues of base values. Most examples, though also produced by the implementation, appear in a simpler pseudocode for clarity.

```
for i = ...
   newbase = S[i+6] + 2 * S[i+7]
   R[i] = ⟨first queue element⟩ + ⟨fourth queue element⟩ + newbase
   newbase = newbase + 3 * S[i+8]
   ⟨Insert newbase onto queue.⟩
```

Figure 39: Pseudocode for evaluation of the stencil $\langle 1, 2, 3, 1, 2, 3, 1, 2\rangle$, whose last base expression is incomplete. `S` is the source array, and `R` is the result array.

This section discusses the two general solutions to this problem: adding operators of arbitrary arity to the compiler's intermediate form or performing associative transformations on the input.

Actually, the problem is not always soluble. Reassociation might be illegal because the operators are not associative or because the value of the expression could be changed by the transformation. Some languages explicitly disallow reassociation in certain circumstances; for instance, a Fortran compiler may cause the processor to "evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated" [13, sec. 6.6.4]. Therefore, `a+b+c` could be evaluated as either `(a+b)+c` or `a+(b+c)`, but neither of those may be substituted for the other. In some cases, the situation is even worse than it appears. In the IBM XL Fortran compiler, "The [VAST] preprocessor rewrites the expression with additional parentheses to make the common expression apparent to the compiler" [72, ch. 4]. In fact, this compiler *requires* this preprocessing step because "the compiler can [only] recognize... duplicate expressions [when] they are either coded in parentheses or coded at the left end of the expression" [72, ch. 7]. The XL C compiler is similar [71].

Our implementation represents associative operators as nodes of unlimited degree [21, 98], and the children of commutative operators are unordered: the intermediate form reflects the semantics of the source program rather than those of the target machine. This makes sense because it simplifies implementation of our optimizations, which are machine-independent but source-code-dependent. A disadvantage of this method is that operations on variable-arity trees tend to be slightly more expensive than those on fixed-arity trees in other parts of the compiler. The internal representation of the list of a node's children is similar to the internal representation of a *supernode* or *cluster* [118] of binary nodes all representing the same operation, but the complexity is hidden behind an abstraction barrier instead of being exposed to the compiler writer.

Another approach is to use the standard binary tree intermediate representation, but to permit reassociation, such as transforming `(a+b)+c` into `a+(b+c)` in the hope that the latter form will expose more common subexpressions than the former. Reassociation can have other benefits as well. Transforming `scalar1 * (scalar2 * vector)` into `(scalar1 * scalar2) * vector` changes a vector multiplication into a scalar multiplication [97], and transforming `11+(11+x)` to `(11+11)+x` removes a run-time addition entirely, since addition of two constants can be done by the compiler. Often commutative as well as associative transformations are required in order to achieve these gains. Simply ordering the operands canonically can cause common subexpressions to be missed, and canonical ordering will not expose the common subexpression shared by `a+c` and `a+b+c`.

Reassociation is attractive because it can be easily integrated into existing compilers and applied only where desired. However, reassociation can be very expensive: for a particular ordering of n operands, there are $O((n-1)!)$ associations. (There are $n-1$ choices for the first pair, and then we are basically associating $n-1$ elements. Some of the $(n-1)!$ associations thus produced will be duplicates, but not enough to reduce the asymptotic bound.) For each of these associations, a common subexpression elimination algorithm must be run to determine how many common subexpressions that association exposes. The complexity can be reduced by using dynamic programming or computing only the overall change in code cost caused by a small change in the association, but the methods are still time-intensive. In addition, for any set of n operands, there are $n!$ ways to commute them before the associativity is even considered.

The PL.8 [15, 31] and Id [137] compilers perform reassociation of loop invariants to increase the effectiveness of loop-invariant code motion. Since these expressions are typically very small, the cost is small, but the speedups were not found to be dramatic either, for loop-invariant expressions [62].

5.2 Alternative implementations

This section describes two other ways to optimize stencils. When the scaling operation distributes over the combining operation, factoring out scaling operations reduces the number of scaling operations performed for aperiodic stencils to the same number that loop common expression elimination achieves, and it is slightly simpler to boot. Scans can be used to evaluate periodic stencils and, if hardware support is available, may be competitive with the methods we have presented.

5.2.1 Factoring scaling operations

When the scaling operations distribute over the combining operation, we can directly factor a stencil to reduce the number of operations performed. For instance, the 5-element binomial stencil $\langle \frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16} \rangle$ can also be viewed as $\frac{1}{16} \times \langle 1, x, 1 \rangle$, where x is $4 \times \langle 1, \frac{3}{2}, 1 \rangle$; this form contains only 3 scaling operations. (Loop common expression elimination structures this computation as $\langle \frac{1}{16}, 0, 0, 0, \frac{1}{16} \rangle + \langle \frac{1}{4}, 0, \frac{1}{4} \rangle + \langle \frac{3}{8} \rangle$ and so also requires just 3 scaling operations per result.)

The resulting code is slightly more straightforward than that produced by loop common expression elimination, and the method can be performed automatically. Furthermore, it uses 2 fewer temporaries in this example, since only base values, not scaled values, need to be remembered. Even if we used this method for optimizing the scaling operations, we still must use loop common expression elimination to optimize base expression evaluations, so the result is essentially the same as that produced by loop common expression elimination, except that some of the computations have been rearranged. This method is also not as general as loop common expression elimination, since it requires distributivity. Therefore, in our implementation we find it more convenient to use a single mechanism than to implement a general one and a specialized one as well, but in practice adding this method would be worthwhile.

When the coefficients form an arithmetic or geometric sequence, loop differencing outperforms this technique, as demonstrated in sections 4.2.1.1 and 4.2.1.2.

5.2.2 Scans

Scans (also known as reductions or parallel prefix computations) [93, p. 32] can compute periodic stencils via a method similar to loop differencing. After the base operations have been computed, a single $\oplus$-reduction (where $\oplus$ is the combining operation) sets the ith value of the result vector to the sum of the first i base elements. To find the value of a window of width w, we only need to subtract (that is, perform the combining operation's inverse upon the two values) two values which are separated by width w.

This implementation's cost is 1 scaling operation and 1 combining operation per result, plus 1 scan, which costs around $O(\log n)$, where n is the machine size. The differencing implementation required 1 scaling operation and 2 combining operations per result.

Computing stencils using scans reduces the danger of instability or inaccuracy due to round-off problems, but it increases the danger due to truncation and overflow. Roundoff errors are reduced because the depth of the operation tree for computing a particular value is shallower—since fewer operations were performed, the maximum cumulative drift is smaller. The worst case error is reduced to about $(2 \log n)\epsilon$, compared to $2n\epsilon$ for the standard running-sum implementation. (Section 4.3 gave methods for arbitrarily reducing the latter value, at the cost of some extra computation.)

While the roundoff problem is reduced, the problem of truncation is exacerbated, and it is difficult to predict which will be worse for a particular application. The error analysis predicting precision within $(w+1)\epsilon$ holds only if the combining operation and its inverse each cause a loss of no more than ϵ precision. This will not necessarily be the case if two approximately equal values are subtracted. Suppose that our computer has b bits of precision, we are computing a total of t window sums, and the base values are all about equal (say, approximately 1). The last few scan values are equal to about t, so they have the same first $\log t$ bits; their difference will only have $b - \log t$ bits of precision. If t is large (1,000,000 data elements is modest for today's supercomputers), then we have lost 20 bits of precision.

While the scan operation may have hardware support or be very efficiently coded at a low level as system software, that is not a sufficient justification for its use. On a machine of size n, a scan requires $O(\log n)$ time, much more than the running sum method. Intuitively, we do not need the scan's ability to communicate information all the way across the machine, and that feature slows down even nearby communications. Purely local communication patterns will be more efficient since they provide exactly what we need. Even if we use the scan operation, we still need to perform local communications to move the values that will be subtracted near one another. When the virtual processor ratio is high, scans are implemented using sequential algorithms on each processor anyway; we might as well do so directly ourselves and avoid the overhead performed by the combining tree.

Chapter 6

Timing results

This chapter compares the efficacy of our three methods for reducing recomputation of loop common expressions. We present timings of the code produced for both periodic and aperiodic stencils by unrolling, by loop common expression elimination, and by loop differencing. The performance depends on the actual stencil and the cost of its execution, but we can usually improve performance by at least a factor of four and sometimes by even more.

We ran our compiler on serial programs whose loops were marked as parallelizable but which were otherwise unannotated; this information could easily have been provided by a good dependence anlysis algorithm. The compiler produced serial C code. In some cases, to make testing easier, the transformed code was rewritten using C preprocessor macros [124], but no hand-optimization was performed beyond that done automatically by the compiler. The C program was compiled with a standard compiler [125, 130] with optimization flags on. We report timings from executing the object code on a Sun-4 SPARCstation 1+ [131].

We used a variety of base expressions, scaling operations, and combining operations in order to see the effect of varying the relative costs of those parts of the stencil. The base expressions were either an array reference or an evaluation of the polynomial $(x+1)(x+2)(x+3) = x^3+6x^2+11x+6$; in each case the value was coerced to an extended-precision floating-point type. The aperiodic stencils were binomial filters; their scaling operations were multiplications and their combining operation was addition. The periodic stencils used addition and minimum (each implemented as a macro and as a function) as their combining operations.

The times reported are seconds on a SPARCstation 1+. When the base expression is an array reference, the time is that required for the computation of 1,000,000 results; when the base expression is a polynomial evaluation, 100,000 results were computed.

We give results for aperiodic stencils (binomial filters) and periodic stencils (window averages) of widths 5, 7, 9, and 11, at unrolling amounts of 1, 2, 4, 6, 8, 10. There is nothing magic about odd widths or even unrollings; those are just the values we happened to choose.

These results are somewhat preliminary; we expect to give fuller results in a future version of this paper.

6.1 Aperiodic stencils

Figure 40 graphs the relative effectiveness of loop common expression elimination and common subexpression elimination on aperiodic stencils—in particular, binomial filters. (Loop differencing is not applicable to this stencil.) When the loop common expressions do not represent much

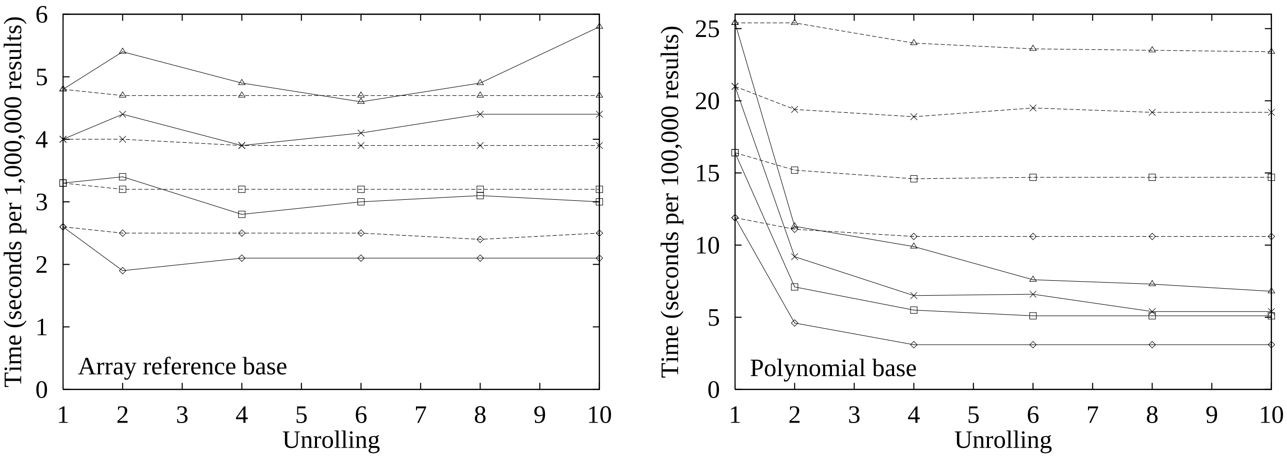
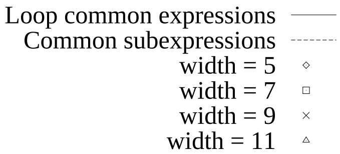


Figure 40: Timings for optimized aperiodic stencils with base expression an array reference (left graph) and a polynomial evaluation (right graph). The legend for these graphs appears at right. The line type indicates whether loop common expressions or just common subexpressions were eliminated, and the point type indicates the size of the binomial filter used.



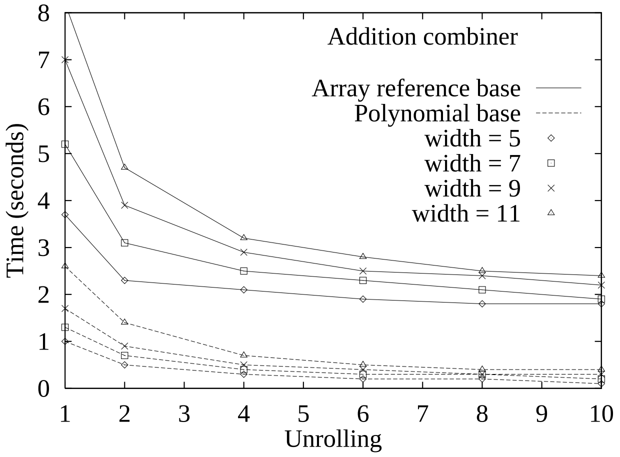
computation (an array reference, in the left graph), there is little to be gained from optimizing them. When they represent more work (a polynomial evaluation containing several multiplications, on the right side of the figure), the savings are significant even at low unrollings. Nothing is gained by unrolling more than $w-1$ times, where w is the stencil's width, and large unrollings can even decrease performance due to the allocation and manipulation of many extra temporary variables. Common subexpression elimination also gained little performance in this case.

The speedups shown in the graph for loop common expression elimination are a severe underestimate, because many of our techniques were not implemented in the prototype compiler or were turned off for this test to avoid even the appearance of fudged data. For instance, when the unrolling was too small for the entire base queue to be scalarized, then only part of it was scalarized; we could have split the queue into two or more smaller ones of size no greater than the unrolling and scalarized them individually. Similarly, when a queue for a scaling operation was larger than the unrolling, it was omitted entirely. No effort was made to optimize loop common expressions when the loop was 1-unrolled, though that would have saved more time.

6.2 Periodic stencils

We performed similar experiments for periodic stencils, but here there was an additional variable: the combining operation used. We only report the results for an addition macro (which is very inexpensive: a single machine instruction) and an addition function (whose function call overhead is fairly expensive), because they are representative of a spectrum of costs.

In this case common subexpression elimination has a larger effect, as indicated by figure 41, which shows speedups gained without any cross-iteration optimization at all. The graph for loop common expression elimination is similar, but drops off even faster; to avoid graph overload, we do not plot this. The time required for loop differencing is essentially independent of the unrolling and the stencil size, since it performs just two base element evaluations and two combining operations



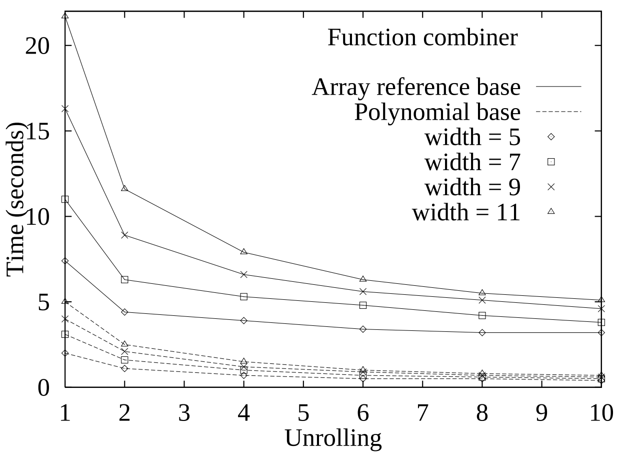


Figure 41: Speedups for a periodic stencil (a window average) achieved by unrolling and performing our common subexpression elimination algorithm. (Ordinary algorithms find no common subexpressions, so the performance is nearly independent of the unrolling amount.) The combining operation for the left graph is addition; the right graph uses a function (whose body happens to be a simple addition) as its combining operation. The legends for the two graphs are identical: the line style indicates whether the base expression was an array reference or a polynomial evaluation, and the point style indicates the size of the window of values being averaged.

per result. The following table gives timings at a few points for all three methods. The base expression is a polynomial evaluation; when it is an array reference, the relative numbers are similar (though their magnitudes are smaller), except that loop differencing is dramatically better when the combining operation is addition.

Addition combiner			
	$w = 7$ $u = 2$	$w = 9$ $u = 2$	$w = 11$ $u = 4$
CSE	7.1	8.1	5.4
LCE	5.0	3.4	5.6
Diff.	4.3	4.3	4.3

Function combiner			
	$w = 7$ $u = 2$	$w = 9$ $u = 2$	$w = 11$ $u = 4$
CSE	8.9	10.8	8.0
LCE	7.3	4.8	7.6
Diff.	5.5	5.5	5.5

Like the numbers for loop common expression elimination, those for loop differencing are an underestimate, because the current implementation never tries to optimize loop common expressions.

Chapter 7

Extensions

In this chapter we present several extensions to our methods for reducing the number of loop common expressions recomputed by a program. First we discuss scheduling of jobs onto processors, which can be done at compile time using standard techniques even if the problem size and machine size are not known at compile time. Next we show how to efficiently compute two-dimensional stencils using the methods we have developed for one-dimensional ones. We give an example of how loop common expressions can be optimized even in purely serial programs and, lastly, discuss some miscellaneous extensions.

7.1 Scheduling jobs onto processors

While our goal is to improve the performance of parallel programs on both parallel and serial machines, we have only discussed compilation for a single physical processor. A collection of serial programs does not make a parallel program, and in this section we briefly discuss additional concerns in a real system, primarily the allocation of data, virtual processors, and work among physical processors, and communication among physical processors.

Perhaps most importantly, the data (and virtual processors) must be assigned to physical processors. We rely on previous work, which has primarily been concerned with minimizing communication [2, 86]. Our optimizations are only applicable if a particular physical processor executes a collection of subsequent loop iterations, in order. (If the loop iterations are randomly distributed among the processors, then it may be the case that no physical processor is assigned internally redundant work, even though work is being repeated in the system.) This requirement is met by most data distribution algorithms. Another issue for a real compiler for a parallel machine is communication of values from processor to processor; not all of the data is local, though much of it may be if the virtual processor ratio is high. This issue has also been adequately discussed in the literature [2, 22, 60, 61, 63, 86, 141].

Although some previous scheduling work has required compile-time specification of the number of physical processors [37, 38], all that is really needed is a simple way to allocate work at run time. For instance, if each physical processor can determine the data set size n, the machine size m, and its own processor number i ($0 \leq i < n$), then it can compute at run time the lower and upper bounds on its virtual processor emulation loops, namely $\lfloor in/m \rfloor$ and $\lfloor (i+1)n/m \rfloor - 1$. These values could have been compile-time constants, but the information is likely not to be available then and the cost of computing them at run time is very slight. Furthermore, if they are compile-time constants, then the programs run on each processor are different, complicating storing and debugging. More complicated expressions involving run-time loads or processors of differing powers can be derived but are of limited use.

Although the machine and problem sizes need not be known at compile time, that is when the jobs are scheduled onto physical processors. This differs from the *work pile* model of parallel processing in which, when a processor becomes idle, it obtains more work from a centralized location [54, 59, 95, 99, 100]. If the tasks do not all complete in about the same amount of time, efficient execution in that model requires additional overhead in the form of communication and synchronization. For the stencil-based computations considered in this paper, instruction-level simulation of the Alewife parallel architecture [4, 28, 103] has shown a work pile scheme to be up to 250% as expensive as prescheduling [46, 47]. The non-prescheduled code also has fewer opportunities for further optimization.

7.2 Two-dimensional stencils

Two-dimensional stencils are found especially in relaxation problems, but also in numerous other applications. In this report we have considered only one-dimensional stencils, primarily because the exposition is considerably easier for them. Here we outline two approaches to optimizing two-dimensional stencils (directly or by doing multiple one-dimensional stencils) and discuss some implementation issues.

The loop common expressions of a two-dimensional stencil can be directly exposed. For instance, the coefficients corresponding to a formula for the Laplacian operator ∇^2 [12, 26, 67, 69], which is used to compute two-dimensional derivatives, are

$$\begin{array}{ccc} 1/20 & 1/5 & 1/20 \\ 1/5 & -1 & 1/5 \\ 1/20 & 1/5 & 1/20 \end{array} .$$

The result at a location is the negative of the location's data value, plus one-fifth of each of its horizontal and vertical neighbors, plus one-twentieth of its diagonal neighbors. A more popular, but worse-behaved, version of this is

$$\begin{array}{ccc} & 1/4 & \\ 1/4 & -1 & 1/4 \\ & 1/4 & \end{array} .$$

If either stencil is strip-mined (producing a *multistencil* [22]), then the common scaling operations can be relatively easily exposed, even those in different rows and columns. The major difficulty with this method—and the others for two-dimensional stencils—lies in choosing heuristics for splitting a stencil into smaller pieces in order to optimize the size of the base expressions and how often each is used.

A simpler method for computing a two-dimensional stencil is to combine the results of several one-dimensional stencils. If the two-dimensional convolution can be performed as two orthogonal one-dimensional convolutions, then the resulting program is trivial. Relatively few stencils can be so decomposed [66], but some important ones—including many smoothing operators—can.

Any two-dimensional stencil can be performed as a series of one-dimensional stencils. First a one-dimensional stencil computation is performed for each column of the original two-dimensional stencil; then these results are added up by performing a one-dimensional row stencil. (We can just as easily reverse the order of the directions.) For instance, to compute the nine-point Laplacian shown above for an array a, we would first compute, for each point, the stencils

$$e_{i,j} = a_{i,j-1} + 4a_{i,j} + a_{i,j+1}$$

and

$$c_{i,j} = 4a_{i,j-1} - 20a_{i,j} + 4a_{i,j+1} \ .$$

Then the final value can be computed by the stencil

$$\nabla^2_{i,j} = e_{i-1,j} + c_{i,j} + e_{i+1,j} \ .$$

When several one-dimensional stencils are combined to produce the two-dimensional convolution, then either the first pass can be done completely before starting the orthogonal one, or they can be interleaved via strip mining. The former is conceptually simpler, but in the presence of virtual memory the latter is probably better because the computation is more local and less extra storage is required; the latter may also expose more computations as loop common expressions. In any event, a single horizontal (vertical) pass over the data can compute several horizontal (vertical) stencils, even if they have different sizes. In the example above, e and c would be computed in a single loop.

7.3 Loop common expressions in serial algorithms

Loop common expression elimination can be as profitable for serial algorithms as for parallel ones. We have so far concentrated on parallel programs (though we have examined execution on a single processor) because they provide particularly fruitful ground for application of the method. Another advantage of parallel applications is their lack of data dependences, which simplifies the application of our methods: we don't have to worry about whether an expression's value will change from one iteration to the next, invalidating our stored copy. This problem prevents most compilers from even attempting such transformations.

Properly applied, however, loop common expression analysis can improve the performance of serial algorithms. Here we give, as an example, the transformation from bubble sort to insertion sort. (We will show two implementations of bubble sort; one cannot be transformed into insertion sort, and the other can.) Bubble sort is among the most trivial of algorithms: it sorts n values in $O(n^2)$ time. Insertion sort also requires $O(n^2)$ time, but it reduces the number of array references and stores by more than half. Here is an implementation of bubble sort:

```
for j = 2 to n
  for i = j-1 to 1 step -1
    if a[i+1] > a[i]
      then
        swap(a[i+1], a[i])
```

Ordinary common subexpression elimination and inlining of `swap` can transform this to

```
for j = 2 to n
  for i = j-1 to 1 step -1
    olda = a[i+1]
    newa = a[i]
    if olda > newa
      then
        a[i] = olda
        a[i+1] = newa
```

but no further. Even cross-iteration optimization seems likely to be foiled, because there are apparently no guarantees about the values of elements of array `a`: it can be assigned to in the body of the loop. If we maintain the temporaries `olda` and `newa` as loop common expressions, however, we can optimize away an array reference:

```
for j = 2 to n
  olda = a[j]
  for i = j-1 to 1 step -1
    newa = a[i]
    if olda > newa
      then
        a[i] = olda
        a[i+1] = newa
      else
        olda = newa
```

The inner loop now accesses each array element only once; at the beginning of each iteration `olda` = `a[i+1]`, but when the `then` clause is taken, it does not need to be set at all. The remainder of the transformation to insertion sort requires the insight that the inner `for` loop does not need to run all the way down to 1, but only until the test fails. Since the first $j-1$ elements are already in order, the jth element only needs to be inserted in its proper place. Recognizing this property probably requires human intervention, though it is possible that a theorem-prover could determine it. Let us suppose, then, that the original bubble sort implementation is as follows:

```
for j = 2 to n
  i = j-1
  while (i > 0) and (a[i] > a[i+1])
    swap(a[i+1],a[i])
    i = i-1
```

We can quickly get to

```
for j = 2 to n
  olda = a[j]
  newa = a[j-1]
  i = j-1
  while (i > 0) and (newa > olda)
    a[i] = olda
    a[i+1] = newa
    newa = a[i-1]
    i = i-1
```

Now it is not too difficult to notice that one of the array assignments undoes the other one. The code's final form is

```
for j = 2 to n
  olda = a[j]
  newa = a[j-1]
  i = j-1
  while (i > 0) and (newa > olda)
    a[i+1] = newa
    newa = a[i-1]
    i = i-1
  a[i+1] = olda
```

This version contains only 1 array load and 1 array store in its inner loop; this is even more efficient than the version of insertion sort in [35, p. 3].

The optimizations were mostly standard ones, but they could not be applied until the loop common expressions had been eliminated, making clear where the dependences actually were. What is noteworthy about this is not so much that bubble sort was converted to insertion sort, but that it was done by general optimizations rather than by pattern-matching. (It would still work if, for instance, the loops were reversed to run in the opposite directions.) We did have to start with a reasonable implementation of bubble sort. It is no surprise that we can write implementations (like the first one, on page 69) that our methods cannot transform all the way to insertion sort.

7.4 Other complications

In this section we discuss a few more of the finer points in a practical implementation of our methods. We show how to perform loop common expression optimizations even when the stencil computation being performed is not known at compile time, how to deal with inactive virtual processors, and how to optimize when a single value, not an array, is the result.

If a stencil's pattern of scaling operations, but not the scaling operations themselves, can be determined at compile time, then all of the methods of this report are still applicable without change. (For instance, occasionally the weights

$$\begin{array}{ccc} 1/12 & 1/6 & 1/12 \\ 1/6 & -1 & 1/6 \\ 1/12 & 1/6 & 1/12 \end{array}$$

are preferable to those on page 68 for the Laplacian operator [68, p. 191]. At compile time, even if we did not know which version we desired, we would know the pattern of loop common expressions.) In fact, the methods become even more attractive in this case, because optimizations of the scaling operations cannot be performed as effectively and so they will be relatively more expensive than when they are known at compile time. In fact, all of the optimizations of this report can be performed at run time, if desired. When a computation is applied to a great many data, this may be worthwhile.

We have so far assumed that every virtual processor computes a stencil and remembers the result, but in many applications not all virtual processors participate in every computation. There are two possibilities for inactive or masked-out processors: they can contribute a base element but not store a result, or they can not contribute a base element. (The former may be the case even if there is no valid data at that point—the contribution may be the combining operation's identity.) If no base element is contributed, then none of the w results that depend on it are valid, where w is the width of the stencil. Since the results on either side of this gap have no computation in common, we might as well stop the computation and start it from scratch, as at the beginning of a loop. If there are many such results, then the loop startup overhead consumes a relatively high proportion of resources. If, on the other hand, every virtual processor contributes a base element but some do not compute results, then it is rarely advantageous to change the pattern of computations. The combining operation which produces that result can be omitted (except in the case of loop differencing), but stopping the loop and starting it up again is usually much more expensive.

The examples in the rest of this report have produced entire arrays as results, but sometimes we want a single result, such as the sum of that array's elements. (In this case we would not actually produce the array at all.) For instance, when numerically integrating, we might not be interested in the improved approximations to the area under each interval of the curve, but only in the total area under the curve. Our methods are still applicable in this case, but other simple techniques are even more attractive. For example, if the scaling operation is distributive and all references to a particular base element are moved to one loop iteration, then all loop common base expressions are eliminated and only one scaling operation per base value is required.

Chapter 8

Perspective

While the methods of this report are original, some of them have been independently discovered in the past and used to hand-optimize inner loops [127]. Our techniques also share some similarities with previously published work. This chapter surveys related work in serializing parallel programs to reduce overhead, in iterator inversion (a form of strength reduction which is similar to loop differencing), in parallelizing serial programs, and in other attacks on stencil computations and loop common expression elimination. Finally, we recap our contributions.

8.1 Reducing overhead

Previous work on serializing parallel programs can be viewed as *overhead reduction*, because it makes no change to the program's computations but only reduces its overhead. The number and type of program computations remain fixed, but the operations may be reordered and/or some system operations may be removed. Such transformations reduce overhead for looping constructs, task management, and storage allocation. The methods of this report reduce overhead, but more importantly, they reduce the amount of computation required to generate the program's output. The false metric of MIPS (millions of instructions per second) count is not necessarily improved by elimination of loop common expressions, but the program completes faster, which is what the user really cares about.

The most direct way to reduce a loop's resource usage is to unroll the loop, prorating the fixed physical loop overhead over several logical iterations. This effect is only noticeable if the loop body cost is comparable to the loop overhead. Other benefits of loop unrolling include better use of the data and instruction caches and exposure of more instructions to optimizations and to the register allocator and instruction scheduler.

Loop fusion (also known as loop jamming) [8, 143] also reduces loop overhead. If the bounds on two adjacent loops are identical, and the transformation would not change the data dependence relation between the loop bodies, then they can be consolidated into a single loop. Not only is the loop overhead for the second loop removed, but data locality can be improved. Figure 42 displays an example of this transformation.

Most overhead reduction work for parallel programs focuses on increasing the grain size of computations [138]. When tasks are very small, a disproportionate amount of time is spent switching between tasks rather than performing the program's computations. Increasing the size of each thread decreases the overhead and so the program's running time. Since the data-parallel programming model has no concept of a task, grain size modification is not directly applicable to our

```
for i = L to U                 for i = L to U
   a[i] = ...          ==>        a[i] = ...
for i = L to U                    d[i] = ...  a[i] ...
   d[i] = ...  a[i] ...
```

Figure 42: Loop fusion or loop jamming. The two loops can be joined, reducing loop overhead, if the bounds on the two loops are identical and the fusion does not change the data dependence relation(s) between the two loop bodies. The values of the expressions L and U must not be modified by the loop bodies or between the loops. In this example loop fusion also improves data locality, since `a[i]` is accessed while its value is still in the cache or a register.

problem domain.

Loop throttling (also known as k-bounded loops) [36, 37, 38] is another method for partially sequentializing a parallel program. In a loop whose iterations are independent and so can all be run simultaneously, loop throttling limits the parallelism by permitting only a limited number of loop iterations to be active at once. So long as this leaves enough parallelism to keep the machine busy, it does not result in idle processors. Throttling lowers resource usage, because the presence of extra tasks adds to task management overhead, including time for task switching and space for storing swapped-out tasks. These methods are more applicable to our problem domain than increasing the grain size, but data-parallel programs rarely have either a task queue which consumes resources or run-time interlocks to delay computations whose values are not yet ready. In any event, the methods of this paper do at least as well as loop throttling, since the number of loop iterations active at once is exactly the number of processors actually available.

8.2 Vectorization

Vectorization, or optimization of computations for execution on vector processors, is an important and active area of research, in part because vector processors were long the machine of choice for computation-intensive programming. Compilation for vector processors can employ all the optimization techniques used for serial code, plus additional techniques aimed at two goals: ensuring that the machine's most valuable resource, the vector unit, is fully utilized, and managing the memory hierarchy to avoid loads and stores (and, where loads or stores are required, to avoid cache misses). Allen and Kennedy [9] present many examples of the field's technology, though they have no implementation of the techniques for demonstrating their effectiveness. Allen and Kennedy view the main problems vector register allocation as subdividing the vector operations into sections that fit the hardware of the target machine and transforming the program to improve locality of reference.

Because vector unit operations are much faster than the equivalent number of scalar operations, it is advantageous to transform loops into vector operations where possible. The first step is checking data dependence, which indicates whether the iterations really are independent and so can be processed by the vector unit. Loops must then be sectioned into pieces that fit in the vector registers. In some cases, loop indices can be adjusted in order to align vectors and permit them to be processed by the vector unit; when references overlap, the entire vector register must be reloaded. This is in contrast to the work described in this report, which makes a virtue of misalignment on different loop iterations in order to reduce the total work done by a loop.

The focus in vectorizaton is effective use of the vector registers, which are expensive to empty and fill. They may also be a scarce resource. Thus, much attention is paid to ensuring that results can be used as computed. This work is analogous to standard register allocation, which seeks to arrange that results can be used soon after being computed, though vector register allocation is complicated by the fact that not only the data, but also the particular subset of it being operated upon, must match. Another important optimization is data placement to avoid cache thrashing. Transformations such as loop interchange, loop reversal, loop splitting, and use of temporary registers aid in these goals.

All of these transformations achieve speedups by moving computations into more effective ALUs or by improving use of the memory hierarchy. They are quite different from loop common expression elimination, which changes the computations being performed in order to reduce the total work required. It is likely that each could be extended into the other's domain in order to complement one another and further improve performance.

8.3 Iterator inversion

The method of loop differencing has similarities to iterator inversion (also known as finite differencing) [44, 52, 53, 104, 105, 106], a form of strength reduction intended to transform high-level abstract code into efficient code. An expression's value is kept *available* by updating it when values it depends on change. Instead of recomputing the expression's value from scratch, the new value is determined from its old value; the code that does this is called the expression's derivative. Given a large collection of simple derivative rules and a chain rule for combining them [105], we can determine the derivatives for many expressions.

The iterator inversion work is targeted for the SETL language [41, 116], and the only modifications to a program value supported are adding an element to a set and removing an element from a set (derivatives are provided, or can be inferred, for a number of interesting operations on sets). In that limited problem domain, significant speedups are achieved, but most of the speedup results from the inefficiency of the input programs. (The canonical example is computation of the size of a set once per iteration, where each iteration also removes one element from the set.) The optimizations are performed only when they can be proved to asymptotically improve running time; constant factor speedups are not considered cost-effective in the compiler.

Loop differencing differs from iterator inversion in several important ways. Most importantly, it can improve efficient input programs. We have given methods for computing the exact costs and savings for each optimization, so that the compiler can tell which ones will improve a program's performance. Loop differencing's unique use of inverses also sets it apart from iterator inversion and other optimizations. Iterator inversion is theoretically more widely applicable than loop differencing since it can operate on any expression reused from loop to loop, but in practice it is more limited. Loop differencing does not even require the particular reused expression to be explicitly mentioned in the source program, much less be a set. (The summands of a periodic stencil may be thought of in that way if desired; scaling operations complicate the picture.)

8.4 Reversing parallelization

There is a sizable body of work on parallelizing sequential programs, both automatically and by hand; the ideal mechanism for sequentializing parallel programs would be to simply reverse those techniques. The prospects for this are poor because parallelization techniques are ad hoc, because removal of data dependences is very different from their addition, and because most parallelization work is vectorization rather than concurrentization.

While each parallelization method is internally consistent, they do not all fit into a common framework; even after some parallelizations had been turned into serialization techniques, the next one would not be any easier to reverse. In fact, most parallelizations simply recognize patterns and transform the input according to heuristics specified by the programmer. Casting parallel and serial algorithms into a more general form is an interesting and challenging research problem which would make parallelizations conceptually simpler and would also make reversing them less tedious.

While a parallelizing compiler attempts to remove dependences, a sequentializing compiler adds them. Parallelizing compilers devote much of their energy to discovering the "unnecessary" dependences that efficient single-threaded implementations usually add. Dependence analysis indicates which control and data dependences are not inherent in the computation but are added by the implementation (by reusing variables, for instance). It is much harder to decide where they can be most advantageously added, without affecting performance. Dependences can be arbitrarily added, so choosing the right ones can be difficult; when removing dependences, on the other hand, there are a finite number of possibilities. Another difficulty in dependence removal is the lack of hard-and-fast algorithms for telling when the job is done.

Our goal has been to make data-parallel programs execute more efficiently. Concurrentization, which results in code runnable on a multiprocessor, is the sort of parallelization that could be most advantageously reversed to address this goal, since its output is typically a data-parallel SPMD program. The literature on concurrentization continues to grow, but that body of work is still relatively small because historically most work on parallelizing dusty-deck code has been vectorization.

8.5 Stencil computations

The Connection Machine Convolution Compiler [22] addresses the same problem domain as this paper: its techniques are designed to optimize the performance of stencil computations on the CM-2 [132]. Most of its optimizations, such as strip mining, software pipelining, and loop unrolling, are well-known; the others are specific to, or mandated by, details of the architecture (floating point and vector unit timing, vector sizes, and so forth). Its schemes for register reuse in multistencil computations are similar to our cyclic reuse of temporary variables.

8.6 Parallel intermediate representations

Compilers of serial languages to serial machines may use a parallel intermediate representation, such as the program dependence graph [51], program dependence web [16, 24], MIT dataflow graph [14], or value dependence graph [140]. Such a representation exposes instruction-level parallelism, enables code motion, and can simplify analyses and transformations.

The potential drawback of a parallel representation is that it must be serialized before serial code is emitted [49, 50, 120, 121, 128]. A frequent strategy is to produce a control flow graph (CFG) from the parallel representation and to geneate code from the CFG using well-understood methods.

While this work can also be viewed as serializing parallel code, the intent and approach are completely different from those used in transforming explicitly parallel code into a serial form. In particular, it shares nothing with loop common expression elimination.

8.7 Contributions

The analysis and the techniques (with the exception of unrolling and traditional common subexpression elimination, which are well-known) of this report are original. In this section we highlight our new contributions, which primarily appear in the chapters devoted to our three optimization methods: unrolling with common subexpression elimination, loop common expression elimination, and loop differencing.

The primary contribution is the idea of loop common expressions, which can be optimized despite occurring in different loop iterations. Ordinary optimizations do not remove redundant computation (or do much of anything else) across loop boundaries. We showed that altruistically computing loop common expressions for the use of future iterations, which a greedy optimizer with a narrow (inter-loop-iteration) view would never do, improves overall performance. The small additional cost is more than washed out by the work done by the previous iteration to help the current one. We show how to optimize three types of loop common expression: base expressions, scaling operations, and combining operations.

In our discussion of unrolling with common subexpression elimination to reduce redundant computation, we observe two interesting phenomena. First, although the topic is considered mature, current common subexpression elimination methods do quite badly in many common and important applications. Their problem is that they work on a fixed parse of the input, and any obvious parsing method obscures most common subexpressions in unrolled stencil computations. We solve the problem by using a multiple-arity (rather than binary) intermediate representation and by separating the common subexpression elimination process into two stages. The first stage determines which expressions appear multiple times, taking advantage of associativity and commutativity, and the second stage chooses some of them to actually execute. Previous algorithms left the first stage to the vicissitudes of the parser or, at best, combined the two stages in a greedy way. Our method improves the performance of the resulting code by 4 times or more.

Our second observation is that unrolling can degrade performance by up to 33%, even if the resulting code does not exceed hardware limits such as instruction cache size or number of registers. This is surprising because the common wisdom is to unroll as much as possible subject to those constraints. We show how to select an unrolling thgat does not incur extra costs.

We prove that in the absence of loop common expression optimization or use of inverses, at least $3(w-1)/(w+1)$ combining operations per result are required to evaluate a stencil of width w, no matter how much the stencil computation loop is unrolled or what common subexpression elimination algorithm is used. When the unrolling $u \leq w$, the bound is about $2 + w/u$ We give an algorithm that meets these bounds and characterize performance at all unrollings.

Next we turn our attention to direct methods for optimizing loop common expressions. We give simple algorithms for removing all redundant base and scaling operation computations. These techniques are interesting in that they implement cross-iteration optimizations without performing unrolling or examining more than one copy of the loop body; they examine the structure of the computation, which is a more straightforward method. We proved that, even without unrolling, a stencil computation's combining operation costs can be reduced to $\log w$ per result, its scaling operation costs to 1 per result per distinct scaling operation, and its base expression costs to 1 per result. For u-unrolled loops, the combining operation costs drop to less than $4 + 2(\log_2 w)/u$ per result; the other costs are already minima and are not included.

We showed that unrolling can scalarize arrays, transforming them into collections of scalar variables. The latter representation is significantly more efficient because no array manipulation is required and all loads and stores are to locations known at compile-time or link-time. This optimization alone makes unrolling worthwhile in many cases. We also showed how to adjust the sizes of arrays when they cannot be scalarized, or in order to make them scalarizable.

Our third method, loop differencing, permits a loop iteration to share with the preceding loop iteration a unique type of common expression—one that was never computed. The essential insight is that undoing work can be faster than doing work, so such expressions can be computed more cheaply by working backward from previously-computed values than by working forward from (other) previously-computed values. Results can be computed with just 2 base expression evaluations (or fewer if loop common expression elimination is also performed) and 2 combining operations each.

Finally, we discuss some implementation details stemming from our experience with a prototype and present experimental verification that our methods do improve programs' performance.

Throughout we emphasize practical, rather than synthetic, appliations; many examples are taken from real programs. An appendix lists further real-world problems to which our techniques are applicable.

While our main thrust is improvement of parallel execution times, our methods are applicable to serial programs too.

Appendix A

Optimality of $(w+1)$-unrolling

This appendix outlines the method used to show that if loop common expressions are not taken advantage of and the combining operator's inverse is not used, then at least $3(w-1)/(w+1)$ operations per result are always necessary when evaluating a w-element stencil, regardless of its unrolling amount u.

We have already shown this result for $u \leq w+1$ in theorems 1 and 2, but for $u > w+1$ we only showed, in theorem 3, that we need at least $2(w-1)/w$ operations per element. We assume the reader is familiar with these proofs, which appear in section 2.3.3.3 on pages 28–30.

As the full proof is extremely tedious, we illustrate the method for only a few cases and let the dedicated reader finish the rest.

Theorem 6 *At least* $3(w-1)/(w+1)$ *operations per result are required to evaluate a* $(w+2)$*-unrolled* w*-element sum.*

Proof: This theorem is obvious for $w=2$ and true by inspection for $w=3$. For $w \geq 4$, we will prove that at least $3w$ operations are required to compute the $w+2$ results; this proves our claim since $3w/(w+2) > 3(w-1)/(w+1)$.

The proof is by contradiction. Suppose we can compute all $w+2$ results, $R_1, \ldots, R_{w+2}$, from base elements $B_1, \ldots, B_{2w+1}$, using just $3w-1$ combining operations.

Regardless of how they are computed, R_1 and R_{w+1}, having no summands in common, require $w-1$ operations each to produce. Results $R_2, \ldots, R_w$ require at least 1 additional operation each; this boosts the total to at least $3w-3$.

There are two possibilities for the number of operations performed to compute R_{w+2}, in addition to those already used for R_{w+1}: 1 or 2. If R_{w+2} required 3 additional operations, the grand total would be at least $3w$, contradicting our hypothesis.

The operation cost for R_{w+2} is lower-bounded by the depth s of B_{w+1} in the expression for R_{w+1}. Figure 43 graphically represents the computation of R_{w+1}; s is the number of addition nodes between B_{w+1} and R_{w+1}, inclusive (in this case, $s=4$). Computing R_{w+2} requires the summation of $B_{w+2}, \ldots, B_{2w+1}$, and there are at least $s+1$ operands to be combined: B_{2w+1}, which has not yet been operated upon at all, and s more in the range $B_{w+2}, \ldots, B_{2w}$.

We examine the two cases $s=1$ and $s=2$ in turn.

$s=1$ The final operation which produced R_{w+1} summed B_{w+1} with $(B_{w+2} + \cdots + B_{2w})$. To compute R_w, we must add a minimum of 3 terms: B_w, B_{w+1}, and $(B_{w+2} + \cdots + B_{2w})$. The first two terms, which are base values, have not yet been used in any sums useful to R_w, but the

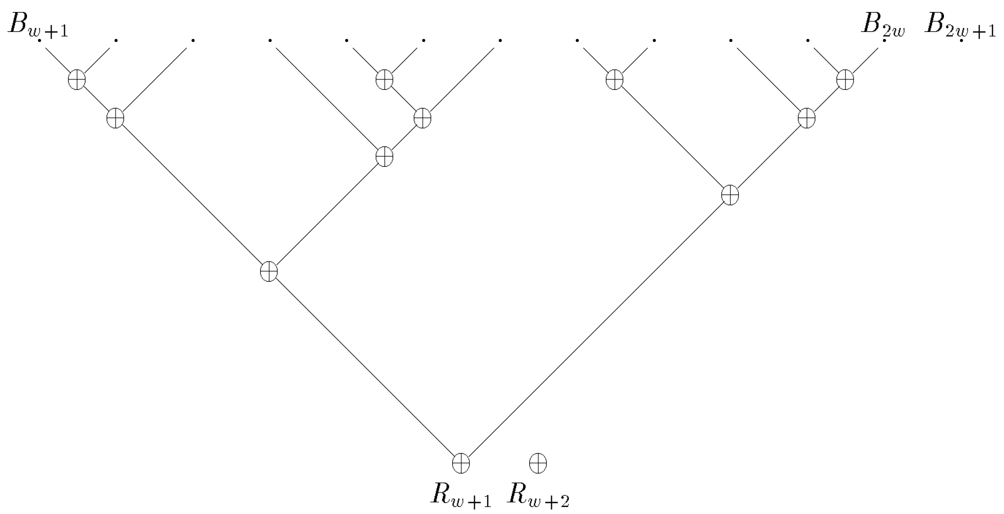


Figure 43: Computing R_{w+2} requires at least s operations, where s is the number of addition nodes directly between B_{w+1} and R_{w+1}, inclusive. In this figure $w = 12$ and $s = 4$.

third term may have been computed as a subterm of R_{w+1} and of R_{w+2}. If it was not, then computing R_w costs at least 3 operations, and we are over our operation limit, so we must assume that it was. Figure 44 shows the situation so far.

Computation of the next result, R_{w-1}, requires at least 2 operations. When all the operations performed so far are added to the minimum of 1 per result required for the $w-2$ results not yet considered, the total is at least $3w$, which contradicts our hypothesis.

$s = 2$ There are two additions between B_{w+1} and R_{w+1}. Figure 45 shows three of the $w-2$ possibilities for the last few elements added to create R_{w+2}.

If we are to use fewer than $3w$ operations, then each of $R_2, \ldots, R_w$ must require only 1 additional operation. We can arrange this for R_w by choosing the third association shown in figure 45. If we do so, R_{w-1} requires a minimum of 2 operations, so all $w+2$ sums require at least $3w$ operations, contradicting our hypothesis. ■

Theorem 7 *At least* $4(w-1)$ *operations are required to evaluate a* $(w+2)$*-unrolled* w*-element sum. This bound is tight.*

Proof: The proof, a minor extension of the proof of theorem 6, is left as an exercise in book-keeping for the reader. The idea is to show that if computing R_{w+2} is cheap, then computing R_w is expensive. ■

Theorem 8 *At least* $3(w-1)/(w+1)$ *operations per result are required to evaluate a* $(w+3)$*-unrolled* w*-element sum.*

This is a corollary of theorem 7, but since we gave no proof of theorem 7, we sketch one here which also shows some of the subtleties in extending this series of theorems.

Proof: For small w, the theorem can be shown on a case-by-case basis. When $w \geq 6$, we will show that at least $3w+3$ operations are required to evaluate a $(w+3)$-unrolled w-element sum.

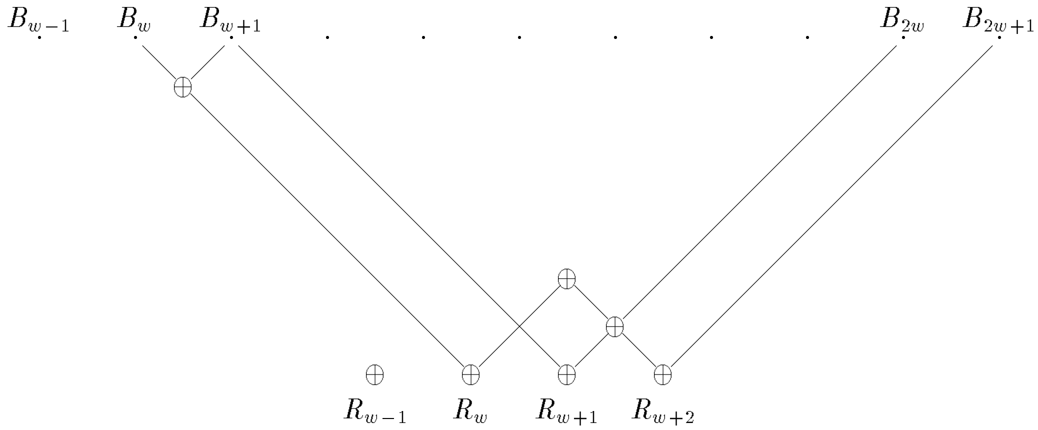


Figure 44: The $s = 1$ case in the proof of theorem 6. After R_{w+1} has been computed, if R_{w+2} requires only 1 additional operation, then R_w requires at least 2.

The proof is by contradiction; suppose the $u = w + 3$ results can be computed with $3w - 2$ operations.

We can think of ourselves as being granted $5 = 3 \cdot 2 - 1$ operations with which to compute R_{w+2} and R_{w+3}, so long as we do not increase the number of operations required for $R_1, \ldots, R_{w+1}$ above $3(w-1)$, which is optimal. If we use fewer than 5 operations for R_{w+2} and R_{w+3}, then we are permitted to increase the operation count for the first $w+1$ sums correspondingly while still using fewer than $3w + 3$ operations overall.

Let s_{w+2} and s_{w+3} be the incremental costs to compute R_{w+2} and R_{w+3}, respectively. We consider four cases for s_{w+2}.

$\boldsymbol{s_{w+2} = 1}$ R_{w+1}'s right subexpression was $B_{w+2} + \cdots + B_{2w}$, and we know from the proof of theorem 6 that $s_w \geq 2$. By inspection, $s_{w+3} \geq 3$, but if we are to meet the operation limit, then $s_{w+3} \leq 3$, so suppose $s_{w+3} = 3$. Now $s_w = 2$ (it cannot be 1, and if $s_w > 2$, we exceed our operation limit), but that forces $s_{w-1} \geq 2$, which is too many operations.

$\boldsymbol{s_{w+2} = 2}$ We consider three possibilities for s_{w+3}.

$\boldsymbol{s_{w+3} = 1}$ We must have chosen the first diagram in figure 45. Now computing each of $R_4, \ldots, R_w$ costs at least 2 operations, far outspending our budget.

$\boldsymbol{s_{w+3} = 2}$ The right child of R_{w+1}'s left child must have been either a base element or a sum whose left operand was a base element.

Since we have now used 4 of our 5 free operations, we require that $s_w \leq 2$. If R_w's left child is a sum of l base elements, then $R_{2+l}, \ldots, R_{w-1}$ require 2 operations each, and the total is too large.

$\boldsymbol{s_{w+3} = 3}$ $R_2, \ldots, R_w$ must require only one operation apiece. If $s_w = 1$, we must have chosen the third diagram in figure 45, but then $s_{w-1} \geq 2$.

$\boldsymbol{s_{w+2} = 3}$ If $s_{w+3} = 1$, then $s_w \geq 2$ and $s_{w-1} \geq 2$. On the other hand, if $s_{w+3} = 2$, we can have $s_w = 1$, but in that case $s_{w-1} \geq 3$. In either case we have used too many operations.

$\boldsymbol{s_{w+2} = 4}$ Every other result ($R_2, \ldots, R_w$, and R_{w+3}) must require only one operation, but we cannot have both $s_w = 1$ and $s_{w-1} = 1$. ■

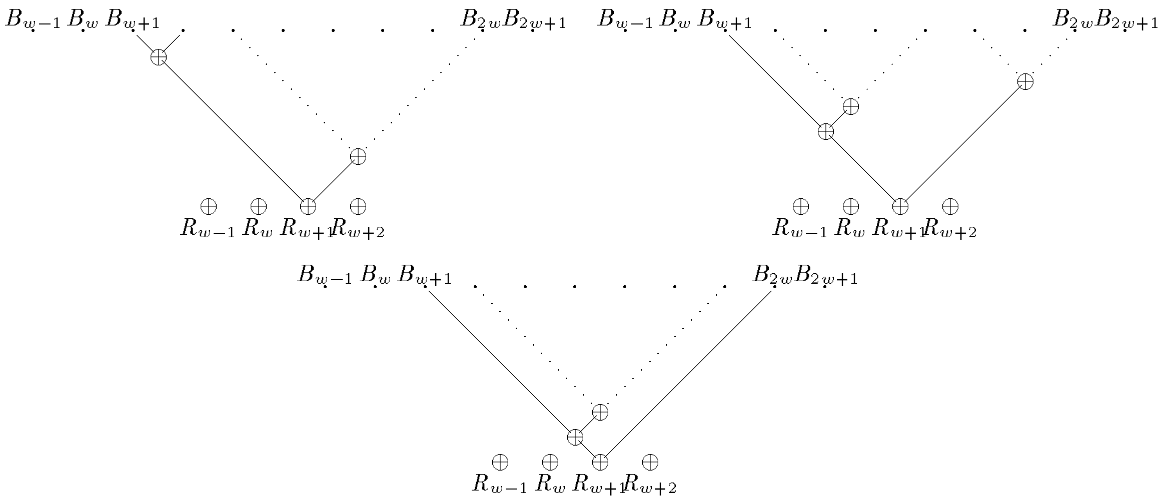


Figure 45: The $s = 2$ case in the proof of theorem 6. After R_{w+1} has been computed, if R_{w+2} requires 2 additional operations, then there are $w-2$ different possibilities for the immediate subexpressions of R_{w+1}. Three of those possibilities are shown.

The proof proceeds similarly for larger unrollings and is much simplified by strategic use of theorem 7. The number of extra operations is actually

$$\left\lfloor \frac{3(w-1)}{w+1} u \right\rfloor - 3w - 3 - 1 \ ; \quad \text{for} \quad u < \frac{4(w+1)}{3(w-1)} \ , \quad \text{this is} \quad 3(u-w-1) - 1 \ .$$

An added complication at high unrollings is that we must consider the possibility that s_i is big enough that some precomputed expressions can be ignored. For the small s_i that we have considered, we must use those subexpressions or fail to meet our operation goal.

We can partially simplify the situation by assuming that every adjacent pair of results R_i and R_{i+1} share at least one computation. If they did not, then we we might as well split the unrolled computation into two smaller ones of sizes i and $u-i$ and optimize them separately. By induction on u, their per-result operation costs are at least $3(w-1)/(w+1)$, and therefore so is that of the original computation.

Appendix B

Applications

This appendix lists several important and common types of stencil computation. For each of them a justification for not using some other implementation, and a list of practical applications, is provided.

B.1 Convolutions

A weighted sum stencil is another name for a convolution, which is an important problem in signal processing and other applications. All window sums are really convolutions [108]. Every linear shift-invariant or linear time-invariant system performs a convolution [69].

B.1.1 Why not use FFT?

Most implementations of convolution operate not in the original problem domain but in its Fourier transform. The convolution code operates by performing a Fast Fourier Transform (FFT; see [108] for references), a single element-by-element multiplication, and another FFT, for a running time of $\Theta(n \log n)$ on n data points. When the stencil's size is $O(n)$, the cost of a direct implementation can be $O(n^2)$ operations. While the FFT implementation is attractive (and sometimes superior to a stencil-based implementation), efficiency or correctness concerns can dictate that the computation be performed directly.

B.1.1.1 Efficiency

The cost of directly performing a convolution with a filter of width w is $O(wn)$; when $w < \log n$, this is less than $O(n \log n)$, the cost of the FFT implementation. Furthermore, the constant factors are lower in the stencil computation than in the FFT one, which performs multiplications of complex numbers, among other operations. If efficiency is a primary concern and n is large, then only a small constant number of operations per datum may be acceptable, but the cost of FFT grows super-linearly.

An algorithm which makes good use of the machine's resources can outperform an asymptotically faster algorithm. On a machine organized as a grid, a stencil implementation of convolution is natural [88], while the FFT implementation requires expensive operations such as communication with distant processors. Grids are an attractive and common parallel architecture because they are easy to build, scalable, and have been shown to be faster than other networks even for many computations which are not grid-based [3, 39].

Perhaps the most important practical case in which FFT is unacceptable is when real-time response is required. FFT is a batch algorithm which works on many data at once, but (even after our optimizations have been performed) the stencil implementation can produce outputs at the same rate as inputs are provided.

B.1.1.2 Correctness

If high-frequency components are present in either the stencil or the data with which it is convolved, then the Fourier representation is very large. For instance, a square wave's frequency-domain representation is infinite; convolution with any finite portion is inaccurate to some degree. (When the tails are very thin, as in the binomial and Gaussian distributions, it does not take a very large filter to accurately approximate the result of an infinitely large one for many problems.) Inaccuracy also results from adding together the results of several small applications of the FFT method in order to reduce its $O(\log n)$ overhead. Such a decomposition is very complex and its gains are slight [108, p. 431]. The stencil implementation, on the other hand, gives exact answers.

When convolutions model real systems, such as the human visual system, physical realizability is important: it is desirable that what the model does, could be the real system's mechanism as well. Therefore, the computation should be causal (depending on only inputs seen so far) and have finite support (depend only on a finite number of input values). The FFT method is acausal: data are processed in a batch, not incrementally. The Fourier transform of any finite stencil is infinite, so the requirement of finite support is also violated.

B.1.2 Applications

Convolutions are widely used for noise smoothing, linear edge enhancement, edge crispening, digital filtering, numerical relaxation, and other applications [89]. Some of these applications appear below; others are deferred to later sections.

Correlation is a measure of how closely two inputs are related [88, p. 287; 108, p. 433] and is computed by the formula

$$\mathrm{Corr}(g,h)_j = \sum_{k=0}^{N-1} g_{j+k} h_k \ .$$

Correlations of delayed signals can be computed to see how much the signals must be shifted (this is called the *lag*) to achieve the best correlation. While the FFT method can compute the correlation at all lags simultaneously, this information isn't very useful. For instance, in stereo matching, if the two pictures are shifted by more than a few pixels, then they probably aren't related at all.

Smoothing makes trends more evident in noisy data and removes glitches and other spurious anomalies. Smoothing is the first step of many signal processing applications, because some algorithms perform particularly well on smoothed surfaces [65]. Even when smoothing is not an explicit step, the desired convolution is sometimes first convolved with a smoothing operator and the resulting stencil, which simultaneously smooths and performs the original operation, is used instead. In fact, estimation formulae with large support (wide stencils) are typically equivalent to formulae of small support applied to smoothed images [68, p. 190; 64]. Averaging, weighted-sum, minimum or maximum, and median filters are all common in digital signal processing [108]; all but the median filter can be efficiently implemented as stencils.

Blurring or simulating the effect of an imperfect lens or an out-of-focus imaging system is done with the stencil $h[n] = a^{|n|}$ [119, p. 275; 69, p. 104]. Section 4.2.1.2 on page 51 showed how to process this stencil efficiently. This convolution cannot be performed by the FFT method because lenses are only linear-shift-invariant only for limited displacements and because aberrations vary with distance from optical axis [69, p. 105].

Polynomial multiplication is just convolution [78, p. 386; 90, p. 198]; so is integer multiplication [90, pp. 162, 174].

B.2 Vision and digital signal processing

Many vision algorithms iteratively produce new images from old ones by local operations [69, pp. 77–80]; for an extensive list of papers using the iterative approach, see [67, pp. 534–536]. One example is finding an image's skeleton by etching away the boundaries of an object; like many of the algorithms, this one may require some communication with neighboring processors [69, p. 81].

Basically all low- and medium-level vision algorithms, even non-iterative ones, are parallelizable [88, p. 272]. Parallel algorithms are particularly attractive for machine vision because they simulate the parallel operation of the human visual apparatus.

Most of the applications mentioned in the previous section could be classified as signal processing; here we mention some others.

Filtering to remove noise need not be done via a convolution. In fact, averaging (which is a stencil computation, but not a convolution) is effective at smearing details and reducing spatial resolution. Other good filtering combining operators are minimum, maximum, and median [69].

Edge detection often involves convolution with an edge detection matrix; examples of vertical and horizontal ones are

$$\begin{matrix} -1 & -c & -1 \\ 0 & 0 & 0 \\ 1 & c & 1 \end{matrix} \qquad \text{and} \qquad \begin{matrix} -1 & 0 & 1 \\ -c & 0 & c \\ -1 & 0 & 1 \end{matrix} .$$

Usually $c = 1$ or 2. When the magnitude of (the result of) the convolution is large, an edge has been detected [107].

Other more sophisticated edge detection methods (such as Canny's [25], mentioned in the introduction as an example of the relative complexity of parallel and serial code) also use stencils in the course of their computation.

Brightness estimation is another application for large-scale averaging. To estimate the brightness difference across an edge, a large area on each side of it must be averaged (so that local effects do not dominate the average). A larger averaging area reduces the effects of noise and makes weak edges easier to detect (but an excessively large area can include other edges by accident) [69].

Repetitive smoothing is required by some stereo matching algorithms that use very heavily smoothed images to find an initial match and successively less-smoothed images for finer matching, once an approximate match has been computed [69].

Reconstructing images from their projections is done via convolution [107]. This problem is also known as the inverse Radon or inverse Hough transform. There are also interesting ad hoc approaches to removing redundant computations for this problem [20].

Template matching can be solved by performing two-dimensional convolution [88].

Stereo matching is done by shifting an image by a small amount and checking correlation locally (using a small section of the image). Motion detection is similar but may have to deal with two-dimensional shifts and different shifts in different parts of an image [56].

B.3 Partial differential equations

Partial differential equations (PDEs) [108, pp. 636ff] are very common in scientific applications. Perhaps the most popular method for solving them is the finite element method (the finite differencing method is the same solution, recast in a different light).

An iterative relaxation method can be used to solve partial differential equations. The key is computing a partial derivative using old values at adjacent points and using fixed values where boundary conditions apply. (This is the Dirichlet space [29].) The discrete approximations of these partial derivative operators are also called *computational molecules* [66]; these are just stencils, so our methods are directly applicable.

We briefly discuss just one vision application, the variational approach to machine vision. It sets up a criterion function to determine the goodness of fit between an actual image and that predicted from one's solution. The Euler equations for these variational problems are typically coupled partial differential equations, often including second, fourth, or other higher even order.

Direct solutions are out of the question because the problems have hundreds of thousands of unknown parameters. Iterative finite element solutions of systems of partial differential equations like this one often use the multigrid method. However, multigrid does not work when the PDE is nonlinear (as in this case: the reflectance map usually depends nonlinearly on the gradient[7]), and multigrid becomes complicated when there are boundary conditions [65]. The traditional characteristic strip method is neither biologically likely nor efficient and robust [67].

Optical flow (computing a vector field showing how image brightness patterns appear to be moving) [68, 69] was the first problem solved using the variational approach. Another application is determining height and gradient from shading [65]. Hundreds of iterations may be required, even for images of moderate size, especially if the contrast is low, so it is a good candidate for optimization.

The fourth-order biharmonic operator is preferable to the second-order Laplacian for iteratively computing these partial derivatives [66]. The Laplacian is only marginally stable, while the biharmonic is numerically stable even in the presence of noise. The biharmonic is also more amenable to our optimizations because its stencil is larger. (Generally, large stencils are desirable, to ensure stability [108]. When stencils are large, then there is more opportunity for redundant computation to be eliminated, and the expense of the computation makes efficient execution even more important.) The Laplacian uses only 5 to 9 points, but the biharmonic uses 13 to 25 points. The obvious form

[7]The moon is a notable exception: the full moon is just as bright at the edges as at the center, which is why it looks more like a disc than a sphere.

for the two-dimensional stencil is the convolution of the 5-point Laplacian operator with itself [11]:

$$\begin{array}{ccccc} & & 1 & & \\ & 2 & -8 & 2 & \\ 1 & -8 & 20 & -8 & 1 \\ & 2 & -8 & 2 & \\ & & 1 & & \end{array},$$

Starting from the 9-point form of the Laplacian results in a better stencil,

$$\begin{array}{ccccc} 1 & 8 & 18 & 8 & 1 \\ 8 & -8 & -144 & -8 & 8 \\ 18 & -144 & 468 & -144 & 18 \\ 8 & -8 & -144 & -8 & 8 \\ 1 & 8 & 18 & 8 & 1 \end{array},$$

but this still isn't as good as one customized to work well for a 4th-order equation. One used in a program for interpolating digital terrain models from contours [66] was:

$$\begin{array}{ccccc} -1 & -1 & -4 & -1 & -1 \\ -1 & 8 & 18 & 8 & -1 \\ -4 & 18 & -76 & 18 & -4 \\ -1 & 8 & 18 & 8 & -1 \\ -1 & -1 & -4 & -1 & -1 \end{array}.$$

There are many other applications for the biharmonic operator, such as the stress function for the edges of a plate under tension [26].

B.4 Other applications

Our optimizations can benefit many other applications, from the direct implementation of Neville's algorithm for constructing an interpolating polynomial [108] to histogram equalization, which computes the average of the neighborhood around each point [107]. We have already mentioned others, such as numerical integration, and simulation of physical systems such as electrical circuits. Some of these only use a value twice, but that use is in the inner loop, where any gain is worthwhile.

Band-pass filters are used to convert an ordinary or suppressed-carrier AM signal into single-side-band (SSB) AM signal, which only occupies half as much bandwidth [119]. Because of the real-time constraint, if this is done digitally, it should be done in the time domain as a stencil convolution.

The methods used in the vision examples (differentiation and solution of systems of equations) are quite general and apply to a large class of problems even when no partial derivatives are involved.

Subsurface imaging, an important seismic application used for oil exploration, uses iterative methods which involve the stencil

$$\begin{array}{ccccc} & & -1 & & \\ & & 16 & & \\ -1 & 16 & -60 & 16 & -1 \\ & & 16 & & \\ & & -1 & & \end{array}$$

and a few other terms; this operator is fourth order in both space dimensions [102].

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, 1985.

[2] Santosh G. Abraham and David E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[3] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[4] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings, 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 1990.

[5] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Computer Science Series. Addison-Wesley, Reading, Massachusetts, 1986.

[7] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 1–9, 1983.

[8] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[9] Randy Allen and Ken Kennedy. Vector register allocation. Technical report, Rice University, Houston, Texas, April 1986. Revised March 1988.

[10] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 296–306, San Diego, California, January 1988.

[11] William F. Ames. *Numerical methods for partial differential equations.* Academic Press, second edition, 1977.

[12] R. S. Anderssen and P. Bloomfield. Numerical differentiation procedures for non-exact data. *Numer. Math.*, 22:157–182, 1974.

[13] ANSI. ANSI Fortran Draft S8, Version 111.

[14] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[15] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 22–31, Boston, Massachusetts, June 1982. Proceedings were also published as SIGPLAN Notices 17(6).

[16] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271. ACM Press, June 1990.

[17] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, Carnegie Mellon University, Pittsburgh, Pennsylvania, January 1992.

[18] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, College Park, Maryland, October 1990.

[19] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[20] Martin L. Brady and Whanki Yong. Parallel discrete approximation algorithms for the Radon transform. In *Proceedings of SPAA '92: The 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 91–99, San Diego, California, June 29–July 1, 1992.

[21] Melvin A. Breuer. Generation of optimal code for expressions via factorization. *Communications of the ACM*, 12(6):333–340, June 1969.

[22] Mark Bromley, Steven Heller, Tim McNerney, and Guy L. Steele Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 145–156, June 26–28, 1991.

[23] John Bruno and Ravi Sethi. Code generation for a one-register machine. *Journal of the ACM*, 23(3):502–510, July 1976.

[24] Philip L. Campbell, Ksheerabdhi Krishna, and Robert A. Ballance. Refining and defining the program dependence web. Technical Report CS93-6, University of New Mexico, Albuquerque, March 1993.

[25] John Francis Canny. Finding edges and lines in images. Technical Report 720, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, June 1983.

[26] Brice Carnahan, H. A. Luther, and James O. Wilkes. *Applied Numerical Methods.* John Wiley & Sons, New York, 1969.

[27] Todd Cass. `canny.lisp`, 1987. Connection Machine implementation of Canny's edge detector.

[28] David Chaiken, Beng-Hong Lim, and Dan Nussbaum. ASIM users manual. Alewife Systems Memo 13, MIT Laboratory for Computer Science, Cambridge, Massachusetts, August 1990. 9 pages; revised November 26, 1991.

[29] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, Massachusetts, 1992.

[30] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970.

[31] John Cocke and Peter Markstein. Measurement of program improvement algorithms. Computer Science 35193, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, February 7, 1980.

[32] W. J. Cody. Analysis of proposals for the floating-point standard. *IEEE Computer*, 14(3):63–68, March 1981.

[33] Jerome T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. *IEEE Computer*, 13(1):68–79, January 1980. Errata appear in *IEEE Computer*, 14(3):61, March 1981.

[34] Jerome T. Coonen. Underflow and the denormalized numbers. *IEEE Computer*, 14(3):75–87, March 1981.

[35] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, Cambridge, Massachusetts and New York, New York, 1990.

[36] David E. Culler. Managing parallelism and resources in scientific dataflow programs. Technical Report MIT-LCS-TR-446, MIT Laboratory for Computer Science, Cambridge, Massachusetts, March 1990.

[37] Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[38] Ron Cytron. Limited processor scheduling of doacross loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 226–234, August 1987.

[39] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[40] F. Darema-Rogers, V. A. Norton, and G. F. Pfister. Using a single-program-multiple-data computa-

tional model for parallel execution of scientific application. Technical Report RC 1152, IBM, Yorktown Heights, New York, November 12, 1986. Revised version.

[41] Robert B. K. Dewar. The SETL programming language. Manuscript, 1978.

[42] D. M. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Computer Languages*, 15(2):83–94, 1990.

[43] Karl-Heinz Drechsler and Manfred 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, October 1988.

[44] J. Earley. High level iterators and a method for automatically designing data structure representation. *Computer Languages*, 1(4):321–342, 1975.

[45] Charles Henry Edwards, Jr. and David E. Penney. *Elementary Differential Equations with Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[46] Michael D. Ernst. Sequentializing parallel grid programs. Manuscript, May 13, 1992.

[47] Michael D. Ernst. Serializing parallel programs (abstract). In Charles E. Leiserson, editor, *Proceedings of the 1992 MIT Student Workshop on VLSI and Parallel Systems*, pages 13–1–13–2, July 21, 1992.

[48] Michael D. Ernst. Serializing parallel programs by removing redundant computation. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1992.

[49] Jeanne Ferrante and Mary Mace. On linearizing parallel code. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 179–190, January 1985.

[50] Jeanne Ferrante, Mary Mace, and Barbara Simons. Generating sequential code from parallel code. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 582–592, June 1988.

[51] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[52] Amelia C. Fong. Inductively computable constructs in very high level languages. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pages 21–28, San Antonio, Texas, January 29–31, 1979.

[53] Amelia C. Fong and Jeffrey 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, January 19–21, 1976.

[54] G. C. Fox, A. Kolawa, and R. Williams. The implementation of a dynamic load balancer. Pages 114–121.

[55] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report COMP TR90-141, Rice University Department of Computer Science, Houston, Texas, December 1991. Revised February, 1991.

[56] W. Eric L. Grimson. Computing stereopsis using feature point contour matching. In A. Rosenfeld, editor, *Techniques for 3-D Machine Perception*, pages 75–111. Elsevier Science Publishers B.V. (North-Holland), 1986.

[57] Patrick A. V. Hall. Common subexpression identification in general algebraic systems. Technical Report UKSC 0060, IBM United Kingdom Scientific Centre, Peterlee, County Durham, England, November 1974.

[58] Patrick A. V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20:244–257, May 1976.

[59] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[60] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* Scientific and Engineering Computation. MIT Press, Cambridge, Massachusetts, 1991.

[61] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[62] James E. Hicks. Personal communication, 1992.

[63] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. Technical Report CRPC-TR91162, Rice University Center for Research on Parallel Computation, Houston, Texas, April 1991. Revised August, 1991.

[64] Berthold K. P. Horn. Hill shading and the reflectance map. *Proceedings of the IEEE*, 69(1):14–47, January 1981.

[65] Berthold K. P. Horn. Height and gradient from shading. *International Journal of Computer Vision*, 5(1):37–75, 1990.

[66] Berthold K. P. Horn. Personal communication, 1992.

[67] Berthold K. P. Horn and Michael J. Brooks, editors. *Shape From Shading*. Artificial Intelligence. MIT Press, Cambridge, Massachusetts, 1989.

[68] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[69] Berthold Klaus Paul Horn. *Robot Vision*. MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1986.

[70] David Hough. Applications of the proposed IEEE 754 standard for floating-point arithmetic. *IEEE Computer*, 14(3):70–74, March 1981.

[71] IBM. *XL C User's Guide*.

[72] IBM. *XL FORTRAN Compiler/6000 Version 2.2 User's Guide*.

[73] IBM. *APL2 Programming: Language Reference*, August 1984. Order number SH20-9227-0.

[74] Institute of Electrical and Electronics Engineers. IEEE standard for binary floating-point arithmetic. 345 East 47th Street, New York, NY 10017, August 12, 1985. IEEE Standard 754-1985.

[75] Institute of Electrical and Electronics Engineers Computer Society. A proposed standard for binary floating-point arithmetic: Draft 8.0 of IEEE Task P754. *IEEE Computer*, 14(3):51–62, March 1981.

[76] K. E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[77] K. E. Iverson. A dictionary of apl. *APL Quote Quad*, 18(1):5–40, September 1987.

[78] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Massachusetts, 1992.

[79] Mattias Jarke. Common subexpression isolation in multiple query optimization. In Won Kim, Davis S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, Topics in Information Systems, pages 191–205. Springer-Verlag, Berlin, 1985.

[80] M. A. Jenkins, J. I. Glasgow, and C. McCrosky. Programming styles in Nial. *IEEE Transactions on Software Engineering*, January 1986.

[81] Kirk Johnson. *Using the LALR parser generator*, September 19, 1991. Documentation version 0.9.

[82] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization: Part I. *International Journal of Computer Mathematics*, 11:21–41, 1982.

[83] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization: Part II. *International Journal of Computer Mathematics*, 11:111–126, 1982.

[84] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[85] Greg Klanderman. Canny edge detector. `smooth.c`, May 18, 1990.

[86] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[87] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1981.

[88] Vipin Kumar, P. S. Gopalakrishnan, and Laveen N. Kanal, editors. *Parallel Algorithms for Machine Intelligence and Vision*. Symbolic Computation—Artificial Intelligence Series. Springer-Verlag, New York, 1990.

[89] H. T. Kung and S. W. Wong. A systolic array chip for the convolution operator in image processing. VLSI Document V046, Carnegie-Mellon University Computer Science Department, Pittsburgh, Pennsylvania, February 1980.

[90] S. Lakshimivarahan and Sudarshan K. Dhall. *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems.* Supercomputing and Parallel Processing. McGraw Hill, New York, 1990.

[91] Daniel LaLiberte. *Edebug User Manual: A Source Level Debugger for GNU Emacs Lisp*, March 1992. Edition 1.2.

[92] C. Lasser. *The Essential *Lisp Manual.* Thinking Machines Corporation, Cambridge, Massachusetts, July 1986.

[93] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

[94] Bil Lewis, Dan LaLiberte, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Cambridge, Massachusetts, December 1990. Version 1.03.

[95] John D. C. Little, Kattta G. Murty, Dura W. Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989, November–December 1963.

[96] J. M. LoSecco, Frederick Reines, and Daniel Sinclair. The search for proton decay. *Scientific American*, 252:54ff, June 1985.

[97] Larry Meadows. Personal communication, 1992.

[98] Randall Mercer. The CONVEX FORTRAN 5.0 compiler. In Lana P. Kartashev and Steven I. Kartashev, editors, *Third International Conference on Supercomputing*, volume II, pages 164–175, Boston, Massachusetts, May 1988.

[99] Joseph Mohan. A study in parallel computation—the traveling salesman problem. Technical Report CMU-CS-82-136, Carnegie-Mellon University Department of Computer Science, August 18, 1982.

[100] Joseph Mohan. Experience with two parallel programs solving the traveling salesman problem. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 191–193, 1983.

[101] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[102] Jacek Myczkowski and Guy L. Steele Jr. Seismic modeling at 14 gigaflops on the Connection Machine. In *Proceedings, Supercomputing '91*, pages 316–326, Albuquerque, New Mexico, November 18-22, 1991.

[103] Dan Nussbaum. ASIM reference manual. Alewife Systems Memo 28, MIT Laboratory for Computer Science, Cambridge, Massachusetts, January 1991. 17 pages; revised November 26, 1991.

[104] Bob Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 58–71, Los Angeles, California, January 17–19, 1977.

[105] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. Technical Report LCSR-TR-8, Rutgers University Laboratory for Computer Science Research, New Brunswick, New Jersey, August 1980. Revised December, 1981.

[106] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[107] Theodosios Pavlidis. *Algorithms for Graphics and Image Processing.* Computer Science Press, Rockville, Maryland, 1982.

[108] William H. Press, Brian P. Flannery, Saul A. Teukolksy, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, England, 1988.

[109] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages*, pages 315–328, 1989.

[110] J. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL87-5, Thinking Machines Corporation, Cambridge, Massachusetts, April 1987.

[111] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming

Languages, pages 12–27, San Diego, California, January 1988.

[112] Bill Ross. Personal communication, 1992.

[113] Gary Sabot. *Paralation Lisp Reference Manual*, May 1988.

[114] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, Cambridge, Massachusetts, 1988.

[115] Gary W. Sabot. Optimized CM Fortran compiler for the Connection Machine computer. In *Proceedings of Hawaii International Conference on System Sciences 25*, pages 161–172. IEEE Computer Society, 1992.

[116] J. T. Schwartz. On programming: An interim report on the SETL project, Installments I and II. Technical report, Courant Institute of Mathematical Sciences, New York University, New York, New York, 1974.

[117] Stephen D. Senturia and Bruce D. Wedlock. *Electronic Circuits and Applications.* John Wiley & Sons, New York, New York, 1975.

[118] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[119] William McC. Siebert. *Circuits, Signals, and Systems.* MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, Cambridge, Massachusetts and New York, New York, 1986.

[120] Barbara Simons, David Alpern, and Jeanne Ferrante. A foundation for sequentializing parallel code — extended abstract. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 350–359, 1990.

[121] Barbara Simons and Jeanne Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR 03.465, IBM, Santa Teresa Laboratory, San Jose, California, February 1993.

[122] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra database interface. *Communications of the ACM*, 18(10):568–579, October 1975.

[123] Arthur Sorkin. Some comments on “A solution to a problem with Morel and Renvoise's ‘Global optimization by suppression of partial redundancies’”. *ACM Transactions on Programming Languages and Systems*, 11(4):666–668, October 1989.

[124] Richard M. Stallman. *The C Preprocessor.* Free Software Foundation, Cambridge, Massachusetts, first edition, April 1989.

[125] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Cambridge, Massachusetts, February 1992. For GNU CC version 2.0.

[126] Guy L. Steele Jr. CM-Lisp. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, 1986.

[127] Guy L. Steele Jr. Personal communication, 1992.

[128] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmond, WA, August 1993.

[129] J. M. Stone, F. Darema-Rogers, A. Norton, and G. F. Pfister. The VM/EPEX FORTRAN preprocessor reference. Technical report, IBM, Yorktown Heights, New York.

[130] Sun Microsystems. *C Programmer's Guide*, 1989. Part number 800-3844-10.

[131] Barbara Tansy. *SPARCstation1 Sun System User's Guide.* Sun Microsystems, 1989.

[132] Thinking Machines Corporation. Connection Machine Model CM-2 technical summary. Technical Report HA87-4, Cambridge, Massachusetts, April 1987.

[133] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide*, 1990. Version 6.0 Beta.

[134] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran User's Guide*, preliminary edition, October 1991. Thinking Machines confidential.

[135] Thinking Machines Corporation, Cambridge, Massachusetts. *Getting Started in *Lisp*, June 1991. Version 6.1. First printing.

[136] Thinking Machines Corporation, Cambridge, Massachusetts. **Lisp Dictionary*, October 1991. Version 6.1. Revised printing.

[137] Kenneth R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report MIT-LCS-TR-370, MIT Laboratory for Computer Science, Cambridge, Massachusetts, August 1986.

[138] Kenneth R. Traub. Sequential implementation of lenient programming languages. Technical Report LCS-TR-417, MIT Laboratory for Computer Science, Cambridge, Massachusetts, October 1988.

[139] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3):191–213, 1973.

[140] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. Technical Report MSR-TR-94-03, Microsoft Research, Redmond, WA, April 13, 1994.

[141] Skef Wholey. Automatic data mapping for distributed-memory parallel computers. Technical Report CMU-CS-91-121, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991.

[142] Robert G. Willhoft. Parallel expression in the APL2 language. *IBM Systems Journal*, 30(4):498–512, 1991.

[143] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press and Pitman, Cambridge, Massachusetts and London, England, 1989.

[144] Jamie Zawinski and Hallvard Furuseth. Compilation of Lisp code into byte code. `bytecomp.el`, March 9, 1992. Version 2.05.