PREDICTING THE EFFECTS OF OPTIMIZATION ON PARALLEL PROGRAMS

by

James R. Larus

Computer Sciences Technical Report #953

August 1990

Predicting the Effects of Optimization on Parallel Programs

James R. Larus
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706, USA
608-262-9519

August 6, 1990

Abstract

Generally, the only way to test and experiment with a new compiler optimization or execution strategy for parallel programs is to implement the idea. Unfortunately, the large cost of modifying a compiler and runtime system is a strong deterrent to experimentation and thorough evaluation of new optimizations and their interactions with computer architectures. This paper describes an alternative technique for evaluating the benefits of a change to a parallel compiler. In this approach, a simple program, driven by detailed program traces, simulates the effect of a new optimization or execution strategy.

The paper contains measurements from applying a wide variety of optimizations to six C programs. The results identify the optimizations that are effective for each program. More generally, they indicate that programs contain more parallelism than is currently exploited, non-numeric programs have less parallelism and are more difficult to optimize than numeric programs, and that optimizations are far more effective in combination than individually.

Contents

List of Tables

1 Introduction

Parallel computers require compilers capable of detecting and exploiting the parallelism inherent in programs. These compilers are necessary both for compiling languages designed for parallel programming as well as for existing languages. Typically, a compiler analyzes a program to find control and data dependences that constrain parallel execution, then transforms the program to remove or alleviate these constraints, and finally produces code suited to execute on a parallel computer. *Optimizations* are transformations that reduce a program's execution cost. Conventional optimization eliminates redundant computations and replaces operations with less expensive ones. Parallel optimization also restructures and reorders a computation so it better conforms to a parallel computer. An *execution strategy* identifies program fragments that can run concurrently and provides a framework in which they execute. Optimization enhance execution by enlarging the concurrent region and removing execution constraints.

Designing new optimizations and execution strategies is an empirical process [12, 17]. Typically, a new idea is implemented in a compiler and evaluated on a few examples. These experiments are expensive since a parallelizing compiler is a large, complex program that requires considerable time and expertise to modify and extend [1, 18]. The high cost of implementing and testing a new idea discourages experimentation with new optimizations, approaches to parallel execution, and computer architectures. This problem is particularly acute for architectural studies, which frequently are based on traces from existing systems that do not attempt to account for the interaction of optimization and hardware changes.

This paper presents a simple, effective technique for evaluating compiler optimizations and parallel execution strategies and for studying program characteristics. The pp system traces, in greater detail than previous work, an execution of a sequential C program. The other part of the system, a simulator (opp) reads the trace and uses details of the program's structure and memory reference pattern to simulate the program's execution on a parallel computer. By ignoring or rearranging events in a trace, opp models the effect of optimization. In turn, by modifying opp (a 3,000 line program), a compiler writer or computer architect can test new optimizations or execution models.

The primary benefit of this technique is that a new idea can be quickly evaluated on many programs without a large effort. The simulator is easily modified because of its small size and simplicity. Because tracing and sim-

ulation are mechanical processes, an enhancement can be tested on a wide range of programs in a short time. opp's measurements identify programs for which an optimization is ineffective and explain why it failed. It is able to quantify the performance of small portions of a program and hence identify bottlenecks.

This paper demonstrates the utility of simulating optimizations by presenting measurements from applying optimizations and execution strategies to six programs. Two programs are traditional, array-manipulating numeric programs. Three are non-numeric programs, and the other is a hybrid.

The next section briefly discusses related work (Section 2), the following section outlines a different view of compiler optimization (Section 3), and Section 4 describes the workings of opp. The final section illustrates opp's operation by examining several program optimizations (Section 5.1) and execution strategies (Section 5.2).

2 Related Work

Trace-driven simulation is widely used to study parallel hardware but rarely used to study software. Computer architecture studies frequently depend on a simulator driven by traces of programs' executions. In general, these studies do not manipulate the contents of traces and hence do not take into account program optimization.

For example, TRAPEDS is a system that traces the memory reference pattern of programs running on a parallel computer [20]. These traces are used to model the performance of cache memory systems. The Rice Parallel Processing Testbed (RPPT) traces a parallel program and uses the resulting trace to drive a simulation of a parallel computer [7]. Its goal is to permit investigations of changes to a computer's architecture and to the assignment of processes and data to processors. Unlike opp, RPPT requires the original program to be written for a parallel machine. RPPT also does not allow modifications of the trace to investigate the effect of optimization.

Kumar used his COMET system to study the effectiveness of one optimization on Fortran programs [13]. The optimization eliminated anti- and output data dependences by splitting a single memory location used by multiple statements into several locations. COMET's approach to simulating a program's parallel execution differs from the one described in this paper, although both dynamically detect data dependences. COMET also assumes a fine-grained, statement-level parallelism in considering this single

optimization. opp simulates a wider variety of optimizations and execution models.

Chen *et al.* used a Kumar's technique to study the effectiveness of different execution strategies on Fortran programs [6]. They simulated a variety of scheduling strategies and varied the cost and type of synchronization. However, they did not try to predict the effect of compiler optimizations.

3 What is an Optimization?

In this paper, we need to change perspective and view compiler optimizations not in terms of their prerequisite analysis or their transformational effect on a program's statements, but rather as transformers of a program's data reference pattern. Parallel program optimizations are valuable because they reduce the number or severity of data dependences by altering a program's behavior.

Data dependences constrain parallel execution. These dependences have three forms. *Flow dependences* arise when a statement produces a value that is subsequently consumed by another statement. *Anti-dependences* occur when an earlier statement reads a memory location that is subsequently rewritten by another statement. *Output dependences* happen when two statements write to the same location. Two statements *conflict* if they share a data dependence. A dependence is *loop-carried* if the conflicting statements execute in different iterations of a loop. The dependence is *loop-independent* otherwise. Data dependences are important because changing the execution order of conflicting statements can cause a program to produce a different result.

Depending on the programming language, the responsibility of ensuring that conflicts are not violated belongs either to a programmer or a compiler. In either case, conflicts reduce a program's parallelism in two ways. The most immediate effect is that certain statements cannot execute concurrently, which reduces the quantity of parallel tasks. In addition, the synchronization necessary to ensure that conflicting statements execute in the proper order introduces delays and increases the program's cost.

Optimization attempts to reduce these costs and to improve a program's performance by rearranging or eliminating conflicting data references. For example, loop interchange rearranges the order in which a statement accesses an array, so loop-carried dependences interfere less with parallel execution [2]. Consider the loops:

```
for i ← 1 to N do
    for j ← 1 to N do
        A[j, i + 1] ← A[j, i] * B[j, i]
```

The inner loop can execute concurrently since it contains no loop-carried dependences. However, on many multiprocessors, it would be more profitable to run the outer loop concurrently so fewer tasks perform larger computations. Interchanging loops produces an equivalent fragment that traverses the array perpendicularly to the original direction:

```
for j ← 1 to N do
    for i ← 1 to N do
        A[j, i + 1] ← A[j, i] * B[j, i]
```

Other optimizations eliminate conflicts entirely. For example, in the loop:

```
for i ← 1 to N do
    t ← A[i] * B[i]
    if t > 0 then C[i] ← t else C[i] ← −t
od
```

the variable t causes loop-carried output dependences, all of which can be eliminated by renaming distinct uses of the variable [8]:

```
for i ← 1 to N do
    t[i] ← A[i] * B[i]
    if t[i] > 0 then C[i] ← t[i] else C[i] ← −t[i]
od
```

4 Operation of opp

opp simulates the parallel execution of a C program by examining a detailed trace of the program's sequential execution. A trace describes the cost

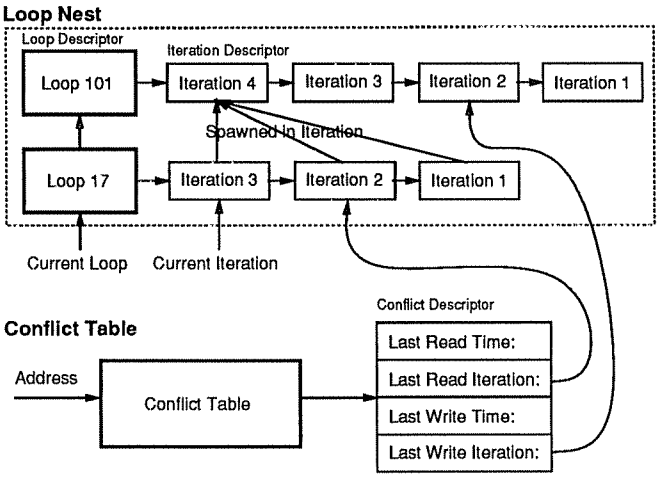

Figure 1: opp's loop nest and conflict table data structures. The loop nest keeps track of loop nesting and iterations. The table maps a memory address to a conflict descriptor, which records when the address was last read and written.

of instructions, the data dependences between them, and identifies higher-level program constructs such as function entry and loop iterations. By rearranging or ignoring events, opp can predict the effect of optimization on a program. In addition, by changing the details of the simulator, alternative execution models can be examined. opp is based on a simpler tool, `llpp`, which is described in detail elsewhere [15]. Since the operation of both tools is similar, the description below only outlines the technique for simulating programs.

The program traces come from AE, which is a compiler-based system that economically collects detailed traces of a program's execution [14]. opp reads a stream of events from a trace. These events indicate: an instruction executed (with its address and cost in cycles); a read or write of a memory location (with its address and information about the containing statement and referenced object); the initiation, iteration, and termination of a loop (with a unique loop identifier); or entry and exit of a function (with its address).

opp simulates the parallel execution of a program's loops with the aid of two data structures: the loop nest and conflict table. The *loop nest* is a stack of *loop descriptors* (top half of Figure 1). This stack contains a descriptor for every uncompleted loop. When a loop begins, a new descriptor is pushed

onto the nest. When a loop terminates, its descriptor is popped. Each loop descriptor points to a stack of *iteration descriptors*. When the top loop begins a new iteration, a new iteration descriptor is pushed on its stack. When the loop terminates, its iteration descriptors may not be deallocated since the conflict table can contain references to them. Iteration descriptors record the nesting of loops by maintaining a pointer to the iteration of the surrounding loop.

opp's other data structure is the *conflict table*, which is a hash table that maps a memory address into a *conflict descriptor* (bottom half of Figure 1). These descriptors record when a memory location was last read and written. They contain both the time of the access and the iteration descriptor of the innermost loop surrounding the statement that referenced the location. On each access to a memory location, opp compares the values stored in the location's descriptor against the current time and loop to detect loop-carried data dependences.

The algorithm for finding loop-carried data dependences is:

On a read of memory location M:

1. Record read time and iteration.
2. Find the loop in the nest, L, that is the least-common ancestor of the current iteration and the last write iteration.
3. If the read and write occur in different iterations of L, then record a Flow Dependence.

On a write to memory location M:

1. Record write time and iteration.
2. Find the loop, L, that is the least-common ancestor of current iteration and last read iteration.
3. If the read and write occur in different iterations of L, then record an Anti-Dependence.
4. Find the loop, L, that is the least-common ancestor of current iteration and last write iteration.
5. If the writes occur in different iterations of L, then record an Output Dependence.
6. Remove record of read of location.

For example, consider detecting loop-carried flow dependences. On a read of a memory location, opp examines the loop iterations in which the

location was last modified. If the location was written in a different iteration than the one currently executing, the loop containing both iterations has a loop-carried flow dependence. When a conflict occurs, opp uses the access times to calculate the delay necessary to ensure that the location is not read until after it is written.

This technique detects all loop-carried flow and output data dependences, but only a subset of the anti-dependences. Conflict descriptors record only the last read of a location, not all reads since the last modification. Therefore, they cannot detect anti-dependences between the earlier reads and a write. Another problem is that opp does not detect dependences carried by variables stored in registers since references to them do not appear in the address trace.

opp uses the cycle count of the executed instructions as a measure of time. Most instructions take 1 *tick*, except loads, which require 2 ticks, and some floating point operations, which require up to 20 ticks. The times are from the MIPS R2000 and are similar for most RISC computers. However, these numbers ignore the effect of cache misses on loads and stores.

When a loop terminates, opp examines each iteration descriptor to find the iteration that finished last. In addition, opp records a wealth of information about the loop, such as how many times it executed, the cost of each iteration, and the number and distance of the loop-carried dependences.

A loop's *speedup* is the ratio of its sequential to parallel execution time. This definition has a unusual aspect. If an inner loop has a large speedup, it will reduce the parallel execution time of every iteration of the surrounding loop, thereby permitting that loop's speedup to exceed the number of its iterations.

5 Measurements

This section presents some measurements of real programs that demonstrate the utility of opp and illustrate the existence of and differences between classes of programs. Numeric programs manipulate arrays of numbers and are the traditional subjects of optimizing parallel compilers. Symbolic programs are more difficult to characterize. Generally, they manipulate structured data in which the relations between items may be as important as the values themselves. However, the data may be numeric, as in the case of sparse-matrix or network-flow computations.

This paper uses six programs as examples. Three are symbolic applica-

Program	Purpose	Size (lines)	Time (ticks)
gcc	C compiler	87,838	24,522,714
xlisp	Lisp interpreter	7,741	20,220,999
espresso	PLA minimization	14,838	30,794,533
sgefa	Gaussian elimination	1,219	18,255,659
dcgc	Conjugate gradient	1,060	19,295,194
costScale	Feasible flows in networks	2,128	79,998,720

Table 1: Characteristics of the test programs.

tions that perform few floating-point operations: *gcc*, *xlisp*, and *espresso*. *gcc* is the GNU C compiler optimizing and compiling a 775-line file. *xlisp* is a lisp interpreter running a program that solves the 5-queens problem. *espresso* is a PLA minimization program running on a 7-input, 10-output PLA. These programs form most of the integer portion of the SPEC benchmark suite [19]. The other three programs perform arithmetic computations: *sgefa*, *dcgc*, *costScale*. *sgefa* is a gaussian elimination program running a variety of test cases. *dcgc* is a preconditioned conjugate gradient package running a variety of test cases. *costScale* finds a feasible flow in a network that minimizes a linear cost function. Although it performs many floating-point operations, it uses data structures similar to those used in the symbolic programs. The programs range in size from a thousand to a hundred thousand lines.

The primary execution model used by opp assumes a unbounded number of parallel processors that communicate and synchronize at no cost through a shared memory. Each loop iteration runs on a separate processor. A loop's iterations begin simultaneously when the loop starts executing. Synchronization introduces delays to serialize loop-carried data dependences. If statement S_1 conflicts with statement S_2 in a later iteration, synchronization delays the memory reference in S_2 until S_1 reads or writes the common memory location. A loop terminates when all iterations complete, so its parallel speedup is the ratio of time to execute the loop sequentially to the time spent in the iteration with the longest combined delay and execution time. Figure 2 illustrates this model, which is Cytron's *doacross* scheduling [9]. Later sections discuss other execution models.

The simulations below assume a unbounded number of processors and zero-cost task creation and synchronization. These quantities are parame-

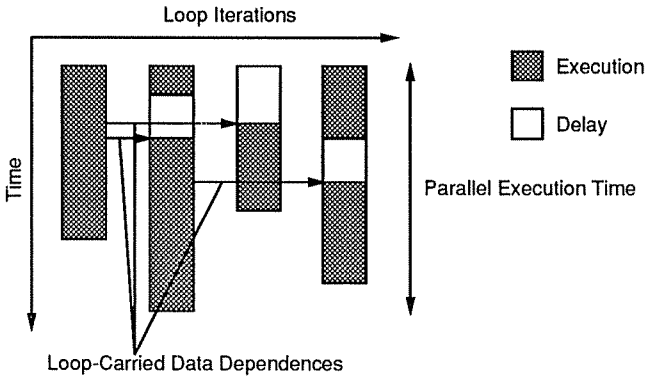

Figure 2: *Doacross* scheduling of parallel loop execution. All iterations of a loop begin execution simultaneously. Loop-carried data dependences are synchronized by delaying a conflicting statement in a later iteration until an earlier statement accesses a common memory location. The loop finishes when the iteration with the longest combined delay and execution time completes.

Program	Speedup
gcc	2.4
xlisp	1.4
espresso	4.5
sgefa	108.3
dcgc	308.5
costScale	5.8

Table 2: Speedups of test programs with *doacross* scheduling.

ters. For this study, they were assigned extreme values that produce maximal parallelism, which permitted examination of optimizations and strategies independent of a particular computer. If given values typical of a computer, opp could simulate the effect of optimization on a particular machine.

5.1 Optimizations

Optimization improves the performance of *doacross* loops by reducing the number of loop-carried data dependences. This change reduces the delay necessary to serialize conflicts and hence the time to complete the loop. As a first step in developing these optimizations, it is worth determining the

Program	Speedup	Improvement
gcc	219.1	91.3 x
xlisp	3.7	2.6 x
espresso	17.3	3.8 x
sgefa	6419.2	59.2 x
dcgc	3386.5	11.0 x
costScale	39043.0	6708.4 x

Table 3: Speedups if all loop-carried dependences were eliminated. The second column (Improvement) contains the ratio of the resulting speedups to the speedups in Table 2.

potential benefits of different optimizations. Table 2 contains the baseline parallel performance of the programs, namely the speed improvement over sequential execution when loops execute with *doacross* scheduling. These speedups include delays that serialize all loop-carried conflicts. Only the two numeric programs produced a significant speedup, despite the optimistic assumptions about available parallelism and low machine overhead. Reasons for the programs' performance are discussed elsewhere [15].

5.1.1 Eliminating Dependences

To obtain a upper bound on the benefits of *doacross* optimization, opp can ignore all loop-carried data dependences. Table 3 contains the resulting speedups. The second column is the improvement over the unoptimized, *doacross*-scheduled programs. This measure is the ratio of the optimized to unoptimized speedups, which is the inverse ratio of the two execution times. The programs can be divided into three categories. The first group (*gcc* and *costScale*) benefited significantly from eliminating dependences. Compiler optimizations that eliminate loop-carried dependences have the potential to improve significantly the execution of these programs. Optimization need not be effective for all loops since the improvements were due to changes in a handful of loops. The next group's (*sgefa* and *dcgc*) performance was already good, but they also benefited from elimination of dependences. However, much of the improvement came from executing several test cases simultaneously. The final group's (*xlisp* and *espresso*) performance was not significantly improved from its low level by these optimizations. The parallelism in these two programs is not expressed at the loop level. Note, however, the responses cannot be categorized along the numeric/symbolic dichotomy since *gcc* belongs in the first group.

Program	Flow		Anti-		Output	
	Speedup	Improvement	Speedup	Improvement	Speedup	Improvement
gcc	2.9	1.2 x	3.0	1.3 x	2.5	1.0 x
xlisp	1.6	1.1 x	1.4	1.0 x	1.4	1.0 x
espresso	5.0	1.1 x	5.0	1.1 x	4.6	1.0 x
sgefa	227.3	2.1 x	109.4	1.0 x	119.9	1.1 x
dcgc	322.0	1.0 x	308.5	1.0 x	312.4	1.0 x
costScale	98.1	16.9 x	5.8	1.0 x	5.8	1.0 x

Table 4: Speedups if flow, anti-, and output loop-carried dependences are eliminated.

The utility of these numbers is that they form a upper bound on the benefits that could result from optimizing the programs. In reality, a compiler cannot eliminate all dependences because of a variety of constraints: some dependences carry information between iterations; compiler analyses are never precise; and the resulting program may be too expensive to execute. Nevertheless, these measurements demonstrate that no optimization will significantly improve the *doacross* scheduling of *xlisp* and *espresso*. Improvements in their parallel performance must come from other sources. The measurements help a compiler writer identify and characterize loops for which optimization can produce a substantial performance improvement.

Reasons for the program's different responses can be inferred from other data collected by opp [15]. *xlisp* and *espresso* execute very few iterations per loop invocation (fewer than 10 iterations for 95.6% of *espresso*'s and 99.8% of *xlisp* loop invocations). In addition, each iteration executed in a short time (less than 100 ticks for over 80% of the iterations). By contrast, the numeric programs executed loops many more times and had a higher proportion of loops that constituted a substantial fraction of the program's cost.

To isolate further the effect of different types of dependences, we can eliminate particular loop-carried dependences. Table 4 shows the speedups that result when flow, anti-, or output dependences are eliminated. In general, eliminating a single type of dependence did not produce a large improvement in any program's performance. A typical loop is constrained by a variety of dependences, all of which need to be removed before it can freely execute in parallel. The exception was *costScale*, whose performance increased markedly without flow dependences. These measurements illustrate that optimizations specific to a particular type of dependence are valuable

Program	Speedup	Improvement
gcc	4.0	1.7 x
xlisp	1.4	1.0 x
espresso	5.5	1.2 x
sgefa	646.1	6.0 x
dcgc	354.6	1.1 x
costScale	5.8	1.0 x

Table 5: Speedups if anti- and output loop-carried dependences are eliminated.

only when combined with similar optimizations for other dependences.

Table 5 illustrates the point further by showing the speedup when all anti- and output dependences are eliminated, which is the effect of renaming [8]. Only *sgefa*'s performance noticeably improved. The remaining flow dependences still constrain other programs' loops. Kumar found that a similar optimization greatly increased the statement-level parallelism in scientific programs [13]. The two results are difficult to compare because of the different execution models. However, with a finer-grain execution model (Section 5.2.2), this optimization also produced a much larger speed improvement.

5.1.2 Classifying References

Another way to examine the potential for optimization is to classify conflicts by the type of statements involved and by the type of object referenced by the conflicting statements. This classification provides a simple measure of the difficulty of performing the program analysis that is a necessary prerequisite to correct optimization. Detecting and analyzing conflicts between statements that manipulate scalar variables is equivalent to the well-understood technique of reaching-definition dataflow analysis. Conflicts between array-manipulating statements are more difficult to analyze precisely because the exact locations accessed by a statement may be difficult to determine. Techniques such as Banerjee's test can eliminate many spurious dependences by inferring that two statements do not reference a common array element [4]. Conflicts between pointer variables are the most difficult to analyze precisely [5, 10, 11, 16]. Only the simplest cases (e.g., pointers into an array or a tree) can be accurately analyzed. The complexity of the analyses means that a compiler will detect more and eliminate fewer

Program	*Variable/Pointer*		*Array/Struct*	
	Speedup	Improvement	Speedup	Improvement
gcc	3.6	1.5 x	2.8	1.2 x
xlisp	2.4	1.6 x	1.4	1.0 x
espresso	5.0	1.1 x	5.8	1.3 x
sgefa	277.0	2.6 x	132.9	1.2 x
dcgc	2362.1	7.7 x	308.5	1.0 x
costScale	5.8	1.0 x	17178.3	2951.6 x

Table 6: Speedups that result if conflicts to pointers and structures (also arrays) are eliminated.

array conflicts than scalar variable conflicts and will be almost unable to alleviate conflicts caused by pointers.

Table 6 shows the speedups that result from eliminating all conflicts in which the second statement directly accesses the common location as a variable or through a pointer (*Variable/Pointer*) or in which the second statement accesses the location as an array or data structure element (*Array/Struct*). In most programs (except *costScale*), eliminating one type of conflict did not produce a large improvement in the program's speed. The style in which the programs were written is partially responsible for this result. For example, *sgefa* and *dcgc* use the curious mixture of array subscripting and pointer reference permitted by the C programming language. Careful analysis can find array pointers that can be translated to array references [3], in which case eliminating array references would be more effective. *costScale*, on the other hand, although it manipulates a graph, accesses graph nodes as elements in an array of structures. Rewriting these programs with another notation would not change their functionality or memory reference pattern, but could increase the precision of the analysis.

5.1.3 Object Lifetimes

Another way to classify dependences is by the dynamic extent of the object manipulated by the conflicting statements. Dynamically-allocated objects are more difficult to analyze precisely than statically-allocated objects because the lifetime of a dynamic object is unknown and must be determined by analysis before conflicts are identified. A static object persists for

Program	*Dynamic*		*Static*	
	Speedup	Improvement	Speedup	Improvement
gcc	3.8	1.6 x	2.8	1.2 x
xlisp	1.5	1.1 x	1.6	1.1 x
espresso	8.1	1.8 x	4.7	1.0 x
sgefa	6419.2	59.2 x	108.3	1.0 x
dcgc	3386.5	11.0 x	308.5	1.0 x
costScale	39043.0	6708.4 x	5.8	1.0 x

Table 7: Speedups if conflicts over dynamically- and statically-allocated objects are eliminated.

a program's execution and has a distinguishable name.[1] AE identifies memory references to statically-allocated objects and provides this information in the trace supplied to opp. Other memory references, to both heap and stack, refer to dynamic objects.

Table 7 contains the speedups that result when conflicts in which the second statement references a dynamic or static object are eliminated. Eliminating conflicts over static objects produces almost no improvement in the programs' performance because these programs allocate most data structures dynamically to accommodate different-sized problems. Eliminating dynamic conflicts produces maximal speedups (equal to those in which all dependences are eliminated) in the last three programs. However, speedups in the symbolic programs, particularly *gcc*, are lower. These programs must reference static data in inner loops. These results demonstrate that compilers cannot base optimization on the presumption that objects have unlimited extent and an easily determinable name.

5.1.4 Loop Interchange

opp can simulate loop-interchange optimization by reordering the memory references produced by AE. Consider the two loops:

[1]The name may not be unique because of aliasing. However, efficient interprocedural techniques for computing aliases exist.

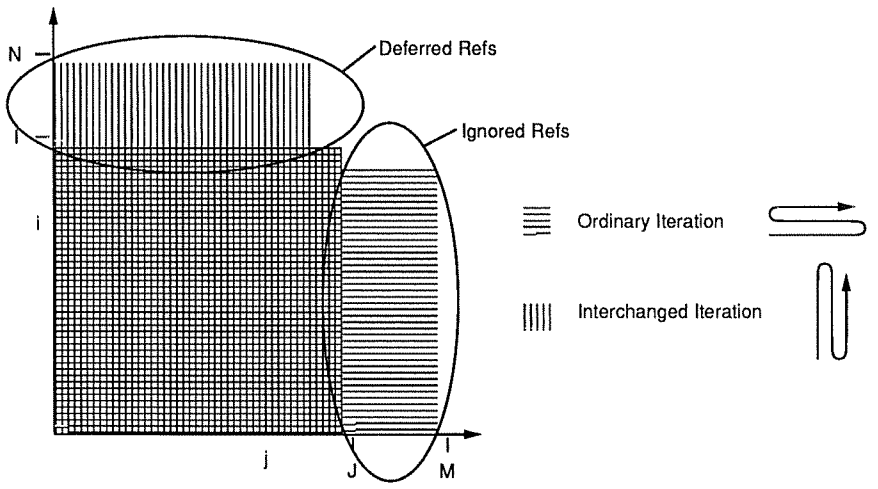

Figure 3: Memory references visible after loop interchange. After interchange the i and j loop, memory references from the deferred iterations are visible and those from the ignored iterations are not seen.

for $i \leftarrow 1$ **to** N **do**
  for $j \leftarrow 1$ **to** M **do**
    ...

At an arbitrary iteration, $i = I, j = J$, AE will have produced memory references from the iterations for which $i < I, j \leq J$ and $i = I, j < J$ (see Figure 3). It will not have generated references from the iterations for which $i > I, j < J$ (*deferred refs*). Finally, opp will have seen, but can discount references from the iterations for which $i < I, j > J$ (*ignored refs*) because these accesses occur after the current iteration in the interchanged program.

Because of the deferred refs, opp cannot compute the delay for iteration J, I (unless it is the first iteration) when AE produces the memory references. The immediate solution is to save the memory references (and the time at which they occurred) for each iteration until its deferred iterations execute. References from $M(N-1)-1$ iterations must be saved until the final iteration of the outer loop, at which point the delays can be properly calculated.

An alternative is to modify AE to produce the memory references in the order in which the interchanged loops would have executed. This modification is not difficult but would greatly increase the file system traffic as AE

randomly read pieces of the trace file. It is not yet clear which approach is faster or more practical.

5.2 Execution Strategies

An execution strategy is a technique for exploiting a particular form of parallelism in programs. For example, the model discussed above is *doacross* scheduling. A strategy has two components. The first is a conception where parallelism is likely to be found in programs. *doacross* scheduling assumes that loop iterations can execute concurrently and that doing so will reduce a program's execution time because loops account for most of a program's cost. The second component is a technique for exploiting the perceived parallelism on an actual computer. We cannot discuss this aspect in detail because a successful strategy requires careful attention to the peculiarity of a computer. opp is well-suited to finding potential sources of parallelism in programs and illustrating areas that should be exploited by language design, compiler transformations, and hardware support. With more realistic parameters, opp can also approximate the performance of a strategy on a particular computer.

5.2.1 Recursive Functions

Recursive functions form loops that can potentially execute concurrently. The Lisp community is particularly interested in executing this type of loop since recursion is a common iterative construct in Lisp programs. opp can identify simple recursive loops and simulate their parallel execution with the *doacross* mechanism used for program loops. A recursive function forms a *simple recursive loop* if the function does not execute a conventional loop around the recursive call. This restriction ensures that the recursive loop appears to be properly nested.

opp identifies these loops by maintaining a count of the number of executing invocations of each function. When this count is greater than one, the function is recursive. opp treats each recursive function invocation like a loop iteration. The invocations start executing concurrently with the first one and are delayed by "loop-carried" data dependences between statements in different invocations. The speedups produced by this model are optimistic since opp does not detect dependences carried in registers (for example, a function's arguments or result). However, this oversight is irrelevant for the test cases since the speedups are insignificant.

Program	Speedup	Improvement
gcc	2.4	1.0 x
xlisp	1.4	1.0 x
espresso	4.6	1.0 x
sgefa	108.3	1.0 x
dcgc	308.5	1.0 x
costScale	5.8	1.0 x

Table 8: Speedups if recursive loops execute in addition to conventional program loops.

Table 8 illustrates the speedup of the six programs when conventional and recursive loops execute concurrently. No program benefited from this model. In part, this poor performance is due to the absence of recursion in C programs: *espresso* contains 8, *gcc* contains 22, and *xlisp* contains 1 simple recursive loop. The other programs contained no recursive functions. In addition, the few recursive loops did not consume much execution time or result in large speedups when executed in parallel.

5.2.2 Dataflow Loop Execution

Another, more profitable, execution strategy is to initiate nested loops concurrently with the start of their surrounding loop. This model was developed for symbolic programs, which contain loops whose body encloses more than one loop. However, it is also a profitable optimization for numeric programs. For example, in the collection of loops:

for $i \leftarrow 1$ **to** 100 **do**
    ...
    for $j \leftarrow 1$ **to** 100 **do** ... **od**
    ...
    for $k \leftarrow 1$ **to** 100 **do** ... **od**
    ...
    for $l \leftarrow 1$ **to** 100 **do** ... **od**
od

this strategy would start the j, k, and l loops concurrently with the initiation of each iteration of the i loop, which also executes its iterations simultaneously.

Program	Speedup	Improvement
gcc	3.9	1.6 x
xlisp	8.3	5.8 x
espresso	8.8	1.9 x
sgefa	988.4	9.1 x
dcgc	1104.0	3.6 x
costScale	10.9	1.9 x

Table 9: Speedups resulting from dataflow loop scheduling.

The advantage of this technique is that inner loops overlap execution rather than execute sequentially. This reduces the cost of each iteration of the outer loop. This strategy is called *dataflow loop scheduling* since loop iterations execute as early as possible, subject to their data dependences.

opp simulates this model by starting an inner loop at the same time as the surrounding loop's iteration. Concurrent loop iterations are still scheduled with *doacross* scheduling. The statements within a loop are delayed by both loop-carried dependences, as in *doacross* scheduling, and by loop-independent dependences from earlier statements in the surrounding loop's body. Unfortunately, opp misses dependences carried in registers, so the numbers in the table may overstate the available parallelism.

This strategy improved the programs' parallel performance (see Table 9). The improvement was not uniformly distributed since this technique depends on the program's structure. Notably, this model improved the performance of *xlisp*, a program that was difficult to optimize in any other manner. The strategy was also profitable for *sgefa*.

5.2.3 Parallel Calls

The final strategy executes ordinary function calls asynchronously, subject to data dependences. The purpose of this strategy is to find large-grained parallelism in a program by executing function invocations concurrently. opp simulates this model by dividing the instruction stream into segments and attempting to overlap these code sequences as much as possible.

A *segment* is a sequence of instructions that begins with a call or return instruction and terminates immediately before the next call or return instruction. Upon entry to a function, all segments within the body of the function begin executing simultaneously. Data dependences between seg-

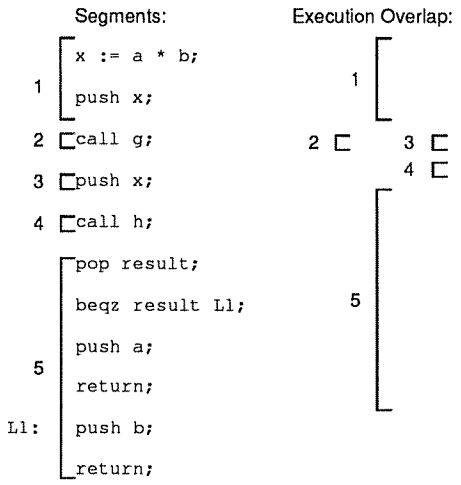

Figure 4: Example of overlapped execution of segments.

ments introduce delays that serialize the segments' execution. For example, consider the function:

```
function f (integer a, b)
    integer x := a * b;
    g (x);
    if (h(x)) then return a; else return b;
```

It contains five segments (see Figure 4). The first extends from function entry to the call on g. The second (which may contain other segments) encompasses the call on g. The third extends from the return to the call on h. The next includes the call on h. And the fifth extends from the return from h to the return from f.

Table 10 illustrates the benefits of this optimization. Note that the figures in this table, unlike those in other tables, are speedups without *doacross* loop execution. *xlisp*, in particular, benefited dramatically from this model. opp's inattention to register-carried dependences has a significant impact on these numbers since most function arguments and results are passed in registers. Nevertheless, these results indicate that some programs have significant course-grain parallelism that can be exploited at the function call level.

Program	Speedup
gcc	5.6
xlisp	590.8
espresso	3.5
sgefa	15.4
dcgc	1.2
costScale	1.0

Table 10: Speedups resulting from parallel call scheduling. Numbers do not include *doacross* loop scheduling.

6 Conclusion

Simulation will never supplant compilation, if only because programmers want their programs to produce answers and are rarely satisfied to know how fast the results could have been computed. Fortunately, compiler writers and computer architects have a different perspective. They are often more concerned with a program's performance than its results. Simulating a program's execution provides more information about the causes of poor performance than does executing it. In addition, simulation is often less expensive in time and effort, which permits ideas to be examined more quickly.

A future extension is to relate directly the speedup and dependence information to sections of the program. A tool of this type would help a programmer find portions of a program that are bottlenecks, both before and after optimization. The information for this mapping is already produced by AE. Its use awaits tools to manipulate large quantities of data and to identify the critical sections.

This paper describes a technique for simulating the effect of optimization on parallel programs. The essential prerequisite is a detailed trace of a program's execution. From this trace, opp simulates the program's parallel execution. By varying the details of the simulation and selectively ignoring portions of the trace, this system can predict the effect of different optimizations and execution strategies. Because this process is entirely mechanical, it can be applied rapidly to a large collection of programs.

The approach's drawback is that its results may not correspond to the program's behavior on a particular computer because the simulation is too abstract. This problem has two aspects. The first is that program traces

may not identify enough events to model all aspects of an optimization. In addition, execution simulation only provides a small amount of information about the analysis that is a prerequisite for safely applying an optimization. The second problem is that a simulation may omit relevant details, either to reduce complexity or because the details are mistakenly deemed unimportant. These problems are common to all simulation and can be partially alleviated by making explicit the assumptions on which a simulation is based.

Acknowledgments

Mark Hill generously provided disk space and a copy of the SPEC benchmarks, which facilitated this work.

References

[1] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[2] John R. Allen and Ken Kennedy. Automatic Loop Interchange. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, pages 233–246, June 1984.

[3] Randy Allen. Compiling C for Vectorization, Parallelization, and Inline Expansion. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 241–249, June 1988.

[4] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[5] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[6] Ding-Kai Chen, Hong-Men Su, and Pen-Chen Yew. The Impact of Synchronization and Granularity on Parallel Systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, May 1990.

[7] R.C. Covington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 4–11, May 1988.

[8] Ron Cytron and Jeanne Ferrante. What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[9] Ronald G. Cytron. Compile-Time Scheduling and Optimization for Asynchronous Machines. Technical Report UIUCDCS-R-84-1177, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1984. PhD thesis.

[10] William Ludwell Harrison III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation*, 2(3–4):179–396, 1989.

[11] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[12] David J. Kuck et al. The Effect of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 129–138, August 1984.

[13] Manoj Kumar. Effect of Storage Allocation/Reclamation Methods on Parallelism and Storage Requirements. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 197–205, June 1987.

[14] James R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice & Experience*, 1990. To appear.

[15] James R. Larus. Estimating the Potential Parallelism in Programs. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, California, August 1990.

[16] James R. Larus and Paul N. Hilfinger. Detecting Conflicts Between Structure Accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[17] Gyungho Lee, Clyde P. Kruskal, and David J. Kuck. An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors. *IEEE Transactions on Computers*, C-34(10):927–933, October 1985.

[18] Constantine D. Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, et al. The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 423–453. MIT Press, 1990.

[19] SPEC. SPEC Benchmark Suite Release 1.0, Winter 1990.

[20] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 70–78, May 1989.