PARALLELISM IN NUMERIC AND SYMBOLIC PROGRAMS

by

James R. Larus

Computer Sciences Technical Report #929

April 1990
(Revised June 1990)

Parallelism in Numeric and Symbolic Programs [1]

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

June 4, 1990

[1]This is a revised version of University of Wisconsin Computer Sciences Technical Report #929 (April 1990). It will appear in the *Third Workshop on Programming Languages and Compilers for Parallel Computing,* August 1990.

Abstract

This paper explains a new technique for estimating and understanding the speed improvement that results when programs execute on a parallel computer. This approach traces a sequential program to record a small set of significant events. From this compact trace, a parallelism analyzer (`llpp`) regenerates a full address trace that also identifies events such as loop initiation and termination. The analyzer uses this information to simulate parallel execution of the program's loops.

In addition to predicting a program's parallel performance, `llpp` measures many aspects of the program's dynamic behavior. This paper presents measurements of six substantial C programs. These results indicate that the three symbolic (non-numeric) programs differ substantially from the numeric programs and, as a consequence, cannot be parallelized automatically with the same compilation techniques.

Contents

List of Figures

List of Tables

1 Introduction

Programmers often want to know whether a program would benefit from parallel execution. One way to answer this question is to write a parallel version of the program and run it on a parallel computer. This approach has several disadvantages. First, it requires a parallel program. Writing such a program, or even modifying a sequential program to run concurrently, may require considerable effort. Similarly, compiler writers need to know whether a large class of programs (e.g., Fortran or C programs) could benefit from parallel execution—before they write a restructuring compiler. Again, this information is difficult to collect without actually writing the compiler.

Another disadvantage of the traditional approach is that the speed of a program on a parallel computer says little about the program's inherent parallelism. Implementation details may hobble the program's performance. Errors of this kind are difficult to detect and correct because the program functions correctly. A programmer needs detailed information about loop speedups and data dependences to identify portions of the program that fail to take proper advantage of parallelism. Few tools provide this information.

This paper presents a simple, mechanizable technique for estimating a program's speed improvement on a parallel computer and for understanding limits on its performance. The technique requires no additional programming and minimal effort by a programmer. A modified compiler adds tracing code to the sequential program to record a small set of significant events. When the program runs, it produces a compact trace file that serves as input to an analyzer (`llpp`), which determines the potential speedup of the program. The process uses an innovative technique to reduce the cost of tracing and the size of the trace file, which permits long executions to be economically measured. `llpp` also measures and reports many characteristics of the program—for example, the dynamic size of loop bodies, number of loop iterations, and number and type of loop-carried data dependences—that form a basis for improving parallel execution.

This paper contains measurements of six substantial C programs. Three are symbolic applications and two are array-manipulating (*numeric*) programs. The sixth is an optimization program that performs many floating point operations, but uses graph data structures and could be classified as numeric or symbolic, although its behavior is similar to the symbolic programs. The measurements clearly demonstrate that the array-manipulating programs are very different from the other programs in two ways: the number and size of loop iterations and the quantity of loop-carried data dependences. The differences make the compilation techniques developed for FORTRAN programs difficult to apply to produce parallel versions of the symbolic programs.

Needless to say, the technique has limitations. Since the analysis is driven by a program trace, it reflects a particular execution of the program and does not necessarily predict the program's performance with other input data. Of course, empirical speedup measurements share this problem. Also, the analysis currently looks for parallelism between loop iterations and assumes an idealized parallel computer with an unbounded number of processors and no-cost synchronization. If desirable, the analyzer easily could be modified to make the execution model more realistic. However, the current assumptions ensure that the estimated parallelism is an upper bound on the program's performance on a real computer. Finally, and perhaps most significantly, the analysis currently does not account for the potential benefits of optimizations that a compiler or a programmer could use to increase the program's parallelism. However, `llpp` could estimate these benefits by ignoring some of the data dependences in the program under the assumption that an optimization eliminated them.

This paper contains four sections. The next section briefly discusses related work. Section 3 describes the model of parallel execution. Section 4 shows how `llpp` analyzes a program's parallelism. Finally, Section 5 presents some measurements of C programs. The appendix explains the technique for tracing a program.

2 Related Work

Several lines of research are related to this work. Sarkar directly measured the execution frequency of basic blocks and used the results to estimate the execution cost of portions of a program. The PTRAN parallel compiler uses this information to partition the program for parallel execution [8]. Unlike PTRAN, `llpp` does not estimate a program's execution cost or produce a parallel program. Instead, it measures the program's actual cost—and other features such as its memory reference pattern—and uses this data to estimate how the program would execute on a parallel computer.

Several groups have built tools for tracing parallel programs. TRAPEDS is a system that traces the memory reference pattern of programs running on a parallel computer [11]. MPTrace produces address traces of programs running on shared-memory multiprocessors [3]. Both systems capture the memory reference behavior of an existing parallel program. Unlike llpp, they do not record program structures (such as loops) that would permit them to simulate a program's behavior on other computers. Also, llpp's tracing mechanism is much more efficient than either system's, which permits tracing of longer program executions.

The Rice Parallel Processing Testbed (RPPT) has goals similar to llpp's [1]. That system measures a parallel program and uses the resulting trace to drive a parallel computer simulator. The major difference is that RPPT requires the original program to be written for a parallel machine and permits a programmer to investigate changes to the architecture of the computer and the assignment of processes and data to processors. llpp has a more static model of the target computer but captures more information about the program and is more efficient. It also does not require a parallel source program.

Kumar and So used address traces from a parallel program running on an uniprocessor to simulate the memory reference behavior of the program running on a parallel computer [5]. They used the memory references to model the parallel computer's memory system. Unlike llpp, their system requires a parallel source program and simulates the underlying hardware rather than measuring the characteristics of programs.

3 Parallel Execution Model

This paper concentrates exclusively on parallel loop execution and ignores other opportunities for parallelism.[1] The idealized execution model used by llpp assumes an 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, then 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 the time spent in the iteration with the longest combined delay and execution time

[1] However, the framework described below can accommodate other parallel execution strategies—for example, fine-grained data flow—by modifying the analyzer described in Section 4.

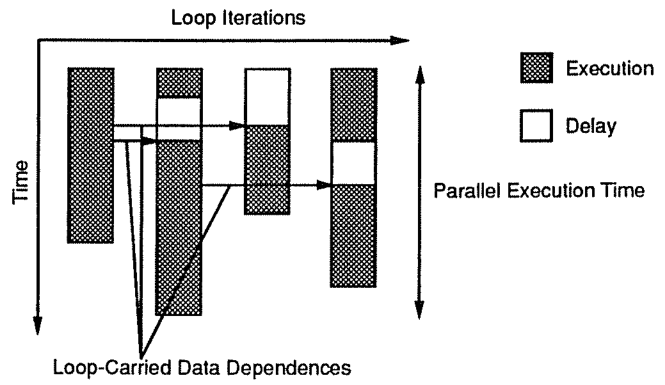


Figure 1: Doacross scheduling of parallel loop execution. All iterations of a loop begin execution simultaneously. Loop-carried data dependences are synchronized by delaying a conflicting statement in a later iteration until an earlier statement accesses a common memory location. The loop finishes when the iteration with the longest combined delay and execution time completes.

to the time required to execute the loop sequentially. Figure 1 illustrates this model, which is Cytron's *doacross* scheduling [2].

4 Analyzing Parallelism

AE is a system that economically collects detailed traces of a program's execution (see Appendix A). We can use the traces collected by AE to simulate a program's execution on a parallel machine. The parallelism analyzer (`llpp`) receives a stream of events from the trace regeneration program. These events indicate: an instruction execution (with its address); a read or write of a memory location (with its address); the initiation, iteration, and termination of a loop (with an unique loop identifier); or entry and exit of a function (with its address).

`llpp` 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 2). This stack contains a descriptor for every uncompleted loop. When a loop begins, a new descriptor is pushed on 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.

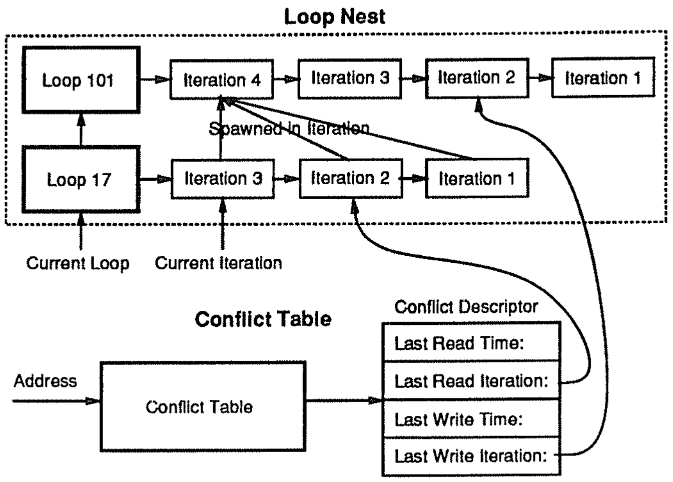


Figure 2: llpp's loop nest and conflict table data structures. The loop nest keeps track of loop nesting and iterations. The table maps a memory address to a conflict descriptor, which records when the address was last read and written.

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.

llpp'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 2). 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, llpp compares the values stored in the location's descriptor against the current time and loop to detect loop-carried data dependences.[2]

For example, consider detecting loop-carried flow dependences. On a read of a memory location, llpp 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

[2]The idea is analogous to the abstract interpretation used by Horwitz, Pfieffer, and Reps to detect data dependences in pointer-manipulating programs [4]. However, their technique is applied by a compiler to the static text of the program to determine potential—not actual—dependences.

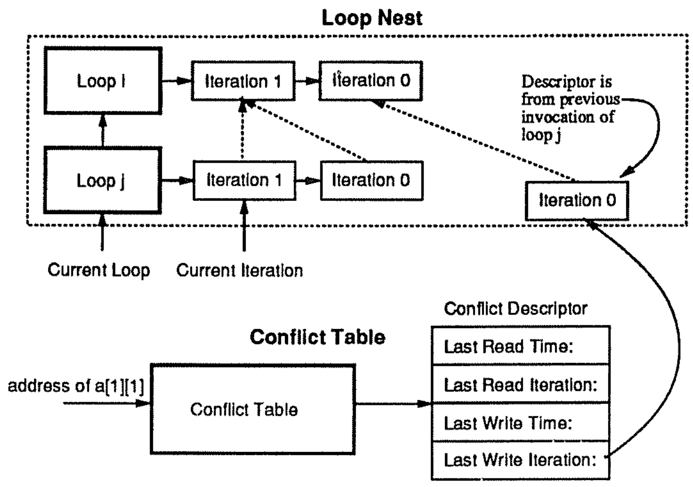


Figure 3: Example of detecting a loop-carried flow dependence.

loop-carried flow dependence. When a conflict occurs, `llpp` uses the access times to calculate the delay necessary to ensure that the location is not read until after it is written.

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.
```

---

sgefa	Number of Dependences (% of Memory References)							
	Flow		Anti		PAnti		Output	
regs	1,012,253	(28.7%)	23,893	(0.7%)	688,341	(19.5%)	957,253	(27.1%)
no regs	3,756,769	(31.0%)	1,079,983	(8.9%)	2,906,008	(23.8%)	3,283,637	(27.1%)

Table 1: Frequency of loop-carried data dependences with and without registers.

To make this discussion more concrete, consider the loops:

```
for (i = 0; i < 100; i = i + 1)
  for (j = 0; j < 100; j = j + 1)
    a[i+1][j+1] = a[i][j];
```

Both have loop-carried flow dependences. For example, when $i = j = 1$, the location read (`a[1][1]`) was written in the previous iteration of the outer loop. The iteration descriptor for this loop is still in the loop nest (see Figure 3). The iteration descriptor for the `j` loop is not on the stack since the first invocation of that loop has finished. However, the descriptor persists since it is referenced by the conflict descriptor for the array location. Since the outer loop is the least-common ancestor of both accesses and the read and write occur in different iterations, the loop has a loop-carried flow dependence.

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. However, by counting reads since the previous write to a location, we can compute an upper bound to the number of anti-dependences—called *panti-dependences* for potential anti-dependences. This bound may be conservative since some of the reads may have occurred outside of a loop currently executing and therefore not cause a loop-carried anti-dependence. In computing the delays, `llpp` uses anti-, not panti-, dependences.

Another problem is that `llpp` does not detect dependences carried by variables stored in registers since references to them do not appear in the address trace. This problem is less important for anti- and output dependences, which can be eliminated by renaming variables. As an experiment, the program sgefa (see Section 5) was compiled without registers, but with optimization still enabled. Table 1 shows the number of dependences and the proportion of memory references that cause a dependence. Allocating variables on the program stack greatly increased the number of memory references and dependences. The proportion of flow and output dependences did not change appreciably, although anti-dependences increased ten times. It is a hypothesis that most additional conflicts involved loop indices. In the measurements below, variables are register-allocated because the speedup is extremely limited by variable conflicts. A compiler for a parallel machine can precisely analyze variable-carried dependences and eliminate most conflicts—in particular, those involving loop indices—without much effort.

`llpp` does not find spurious dependences caused by reuse of locations on the program stack. This problem arises when a loop repeatedly invokes a function. Stack locations referenced in the first invocation will likely be used by subsequent invocations and will cause dependences that would not occur if the calls used separate stacks—as they would if executed concurrently. `llpp` avoids this problem by removing all stack locations accessed by a function from the conflict table when the function terminates.

`llpp` uses the cycle count of the executed instructions as a measure of time. Most instruction 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 to most RISC computers. However, these numbers ignore the effect of cache misses on loads and stores.

When a loop terminates, `llpp` examines each iteration descriptor to find the iteration that finished last. In addition, `llpp` 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 parallel to sequential execution time. This definition has an 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. A program's speedup can be estimated from its loop speedups. Since loops nest, `llpp` cannot simply add the parallel execution times to compute the program's cost. This cost is the sum of the cost of the top-level (non-nested) loops and the top-level non-looping code.

This analysis is not particularly expensive. On a DECstation 3100 (a 14 MIPS computer), `llpp` analyzes about 80,000 simulated ticks per second of DECstation time (175 times slower). A slightly more serious problem is the memory cost of `llpp`'s data structures. Their size is proportional to the number of referenced locations and the number of iterations in loops. With

Program	Purpose	Size (lines)	Time (ticks)	# Loops
gcc	C compiler	87,838	24,522,714	1139
xlisp	Lisp interpreter	7,741	20,220,999	126
espresso	PLA minimization	14,838	30,794,533	750
sgefa	Gaussian elimination	1,219	18,255,659	73
dcgc	Conjugate gradient	1,060	19,295,194	55
costScale	Feasible flows in networks	2,128	79,998,720	50

Table 2: Characteristics of the test programs.

virtual memory and large physical memories, this overhead is not exorbitant, although it clearly limits `llpp`'s ability to analyze extremely large programs.

Several schemes could reduce both costs. First, many memory references are irrelevant to parallelism analysis since they correspond to memory accesses that cannot cause loop-carried dependences. These references can be eliminated from the schema file. Another possibility is to group together several locations (e.g., by truncating the low-order bits of the address). However, neither technique has yet proven necessary.

5 Measurements of C Programs

This section presents measurements of six C programs. Three are symbolic applications 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 [9]. 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. Table 2 describes the characteristics of the programs in more detail.

Table 3 contains the ratio of the programs' parallel to sequential execution times—the speedup—calculated by `llpp`. Although `llpp`'s parallel

Program	Parallel/Sequential Time (Speedup)
gcc	2.5
xlisp	1.4
espresso	4.5
sgefa	106.3
dcgc	259.4
costScale	5.0

Table 3: Calculated speedup for the test programs.

execution model is optimistic, only the two numeric programs that manipulated arrays (*sgefa* and *dcgc*) significantly benefited from parallelism. The discussion below shows that this difference is due not only to differences in programs' data structures and data dependences, but also because of differences in programs' use of loops.

Figure 4 shows the distribution of loop speedups in the programs. As can be seen, most loop invocations in the two numeric programs have a speedup of 50 or more. Almost all symbolic loops (90–95%) have a speedup of 10 or less and almost no invocations ($< 0.25\%$) have a speedup of 100 or more. In addition, in numeric programs, the large speed improvements occurred in fairly expensive loops and consequently caused a large reduction in total time. In symbolic programs, the large speedups occurred in less expensive loops, which did not reduce the programs' total costs as much.

Two factors limit a loop's speedup: the number of iterations and the frequency and distance of loop-carried data dependences. Figure 5 shows the number of iterations per loop invocation. The figure again illustrates a difference between numeric and symbolic programs. Numeric loops iterate many times. Symbolic loops generally iterate 10 or fewer times, which limits their potential speedup in a doacross model.[3] A possible objection to these measurements is that the number of iterations is proportional to the size of the input and that with larger input, symbolic programs would iterate more. However, the inputs were chosen so that all programs executed for roughly the same time. Increasing the input size would also benefit numeric programs. In addition, increasing the length of the input to programs such

[3]These measurements confirm the compiler folklore that loops iterate 10 times.

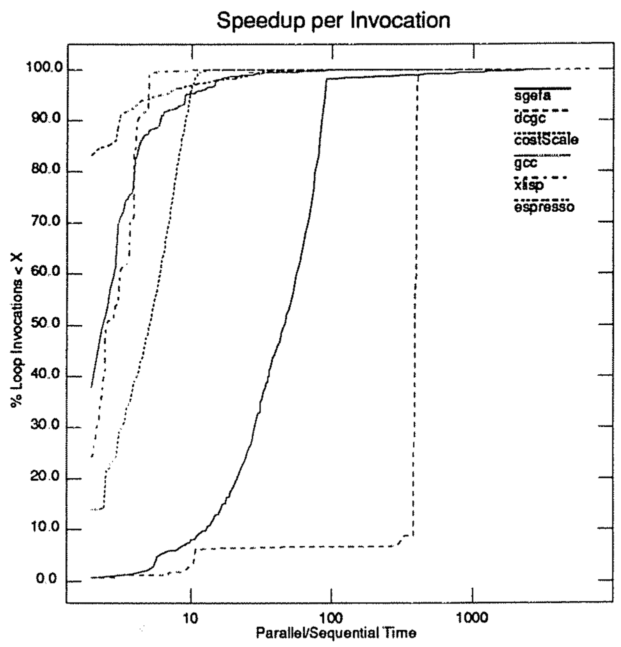


Figure 4: Distribution of loop speedups. This graph shows the percentage of loop invocations in each program that had a speedup less than the x-value.

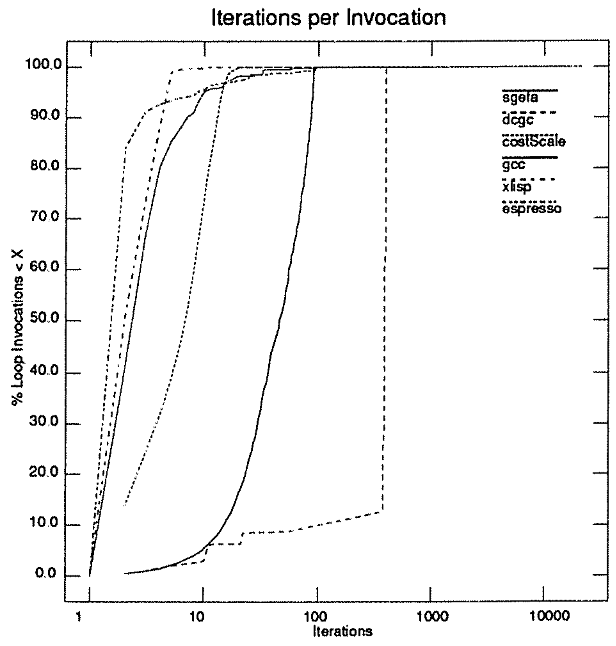
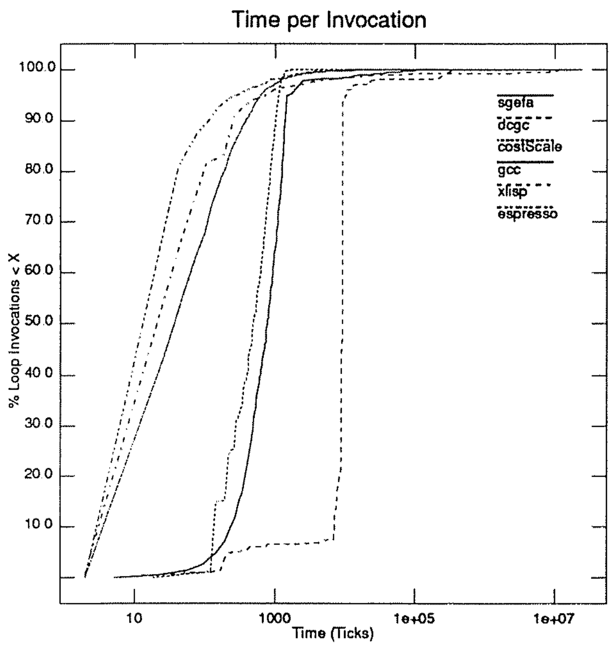


Figure 5: Distribution of iterations.



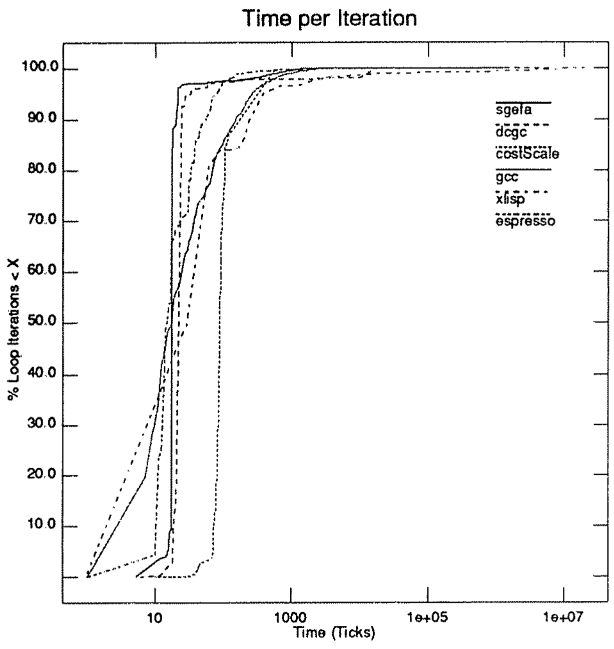


Figure 6: Sequential execution cost of loops.

as *gcc* and *xlisp* would not cause most loops to iterate more times, unless the input also changed qualitatively to include larger functions.

Another way to compare loops is to examine their sequential execution cost. Loops that execute few instructions—even if they iterate many times—have little potential for parallel execution, particularly on real computers in which initiating a parallel task has non-trivial overhead. Figure 6 shows the sequential time per loop invocation. Most loops in symbolic programs (70–90%) require fewer than 100 ticks to execute. By contrast, only 1–3% of the loops in numeric programs (including *costScale*) executed in fewer than 100 ticks. Most loops in these programs execute in 500–10000 ticks. However, the situation changes when we examine the cost per iteration (not invocation). Figure 6 shows that iterations in both symbolic and numeric programs typically execute in 100 or fewer ticks. *dgcg* and *sgefa* appear to have particularly short iterations. The difference in invocation costs is primarily due to the number of iterations per loop invocation. However, the symbolic programs also have a significant number of iterations that execute only a few instructions. The long tails on the distributions are due to programs' top-level loops, which encompass most of their computation.

Program		Sum of Per-Loop Coefficient of Variances	
		Unweighted	Weighted
gcc	Time	274.8	0.9
	Iterations	183.0	1.1
xlisp	Time	16.8	1.3
	Iterations	19.2	323.8
espresso	Time	111.7	0.9
	Iterations	101.7	1.8
sgefa	Time	13.4	0.5
	Iterations	18.9	1.6
dcgc	Time	14.1	0.2
	Iterations	17.0	1.2
costScale	Time	2.3	0.4
	Iterations	2.7	0.8

Table 4: Variance in execution time and the number of iterations per loop invocation.

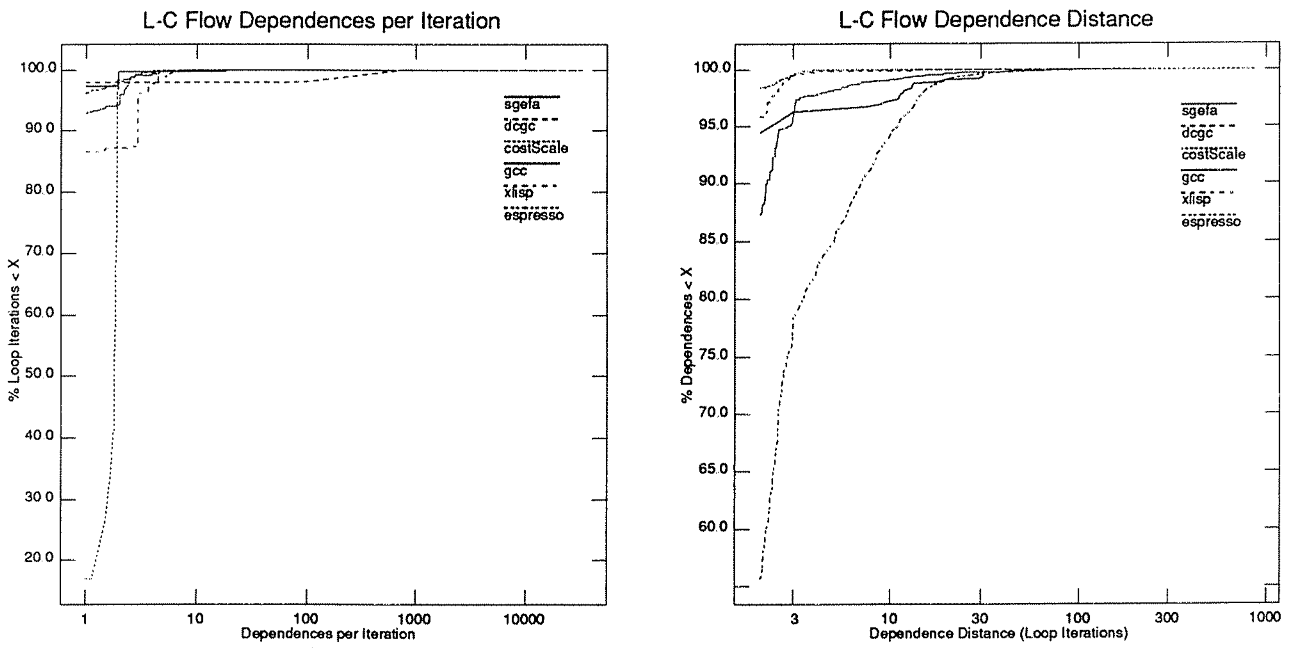


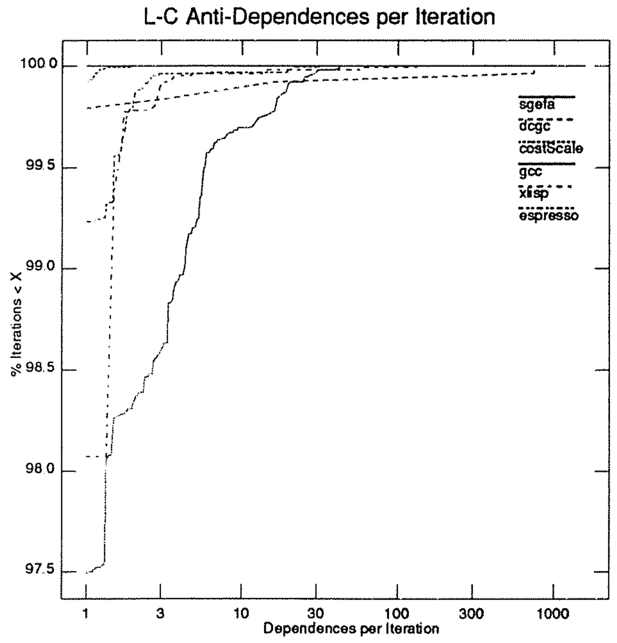
Figure 7: Flow dependences.

Another difference is the variance in both the cost and number of loop iterations. Table 4 shows the variance in the time and number of iterations per loop invocation. The first column is the sum of the coefficient of variation for each loop. The second column contains the sum weighted by the loop's contribution to the program's execution time. Unweighted variance is a better metric since the weighted numbers can be dominated by an outer loop, which may be invoked only once and have no variance. The unweighted variances of symbolic programs (except *xlisp*) are much larger than those of the numeric programs. In addition, in the symbolic (but not numeric) programs, the time variance is larger than the variance in the number of iterations. Hence, these programs are performing different actions in different iterations. Symbolic programs execute their loops less systematically than numeric programs: they have more variance in both the number of times that a loop iterates as well as the amount and type of work done in each iteration.

The other constraint on a program's speedup is the data dependences that inhibit concurrent execution of loops. Figure 7 illustrates the number of loop-carried flow dependences per loop iteration. In numeric programs, 97% of loop iterations have no loop-carried flow dependences and most other iterations have only one dependence. In symbolic programs, more iterations have one flow dependence, although few have more than 10 such dependences. The distance of loop-carried dependences also limits a loop's speedup. Figure 7 also illustrates the distances of the loop-carried flow dependences. The difference between numeric and symbolic programs is not as clear for this measure. Numeric programs have a higher percentage of flow dependences of distance one (for example, all of *dcgc*'s dependences). However, *sgefa* has more dependences of distance ten or more than any program except *espresso*.

The data dependences in *espresso* are very different from those in other programs. This program has more dependences of all types and these dependences extend over many loop iterations. Most of *espresso*'s time is spent sorting arrays with a quicksort algorithm. The conflicts arise as values are moved between array locations in a data-dependent pattern.

Figure 8 illustrates that anti-dependences are less common than flow dependences and are extremely uncommon in numeric programs. They also extend over a smaller number of iterations. Both differences may be artifacts of recording only the last read of a location. Figure 9 shows that output dependences are almost as frequent as flow dependences. However, unlike flow dependences, output dependences almost always have a distance of one.



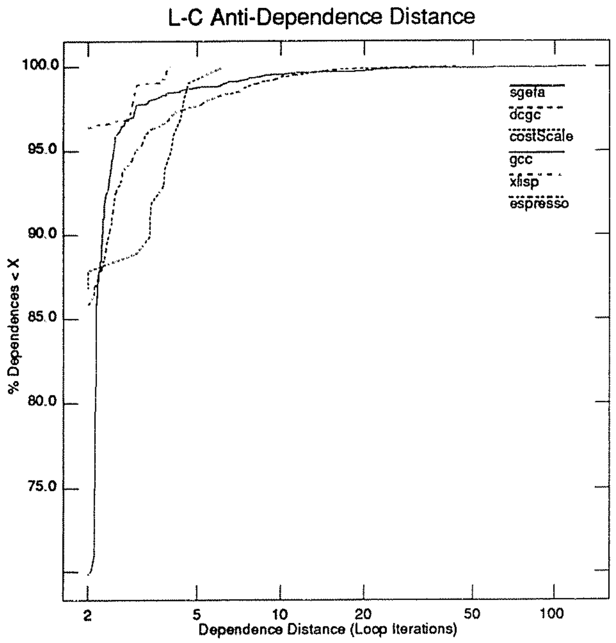
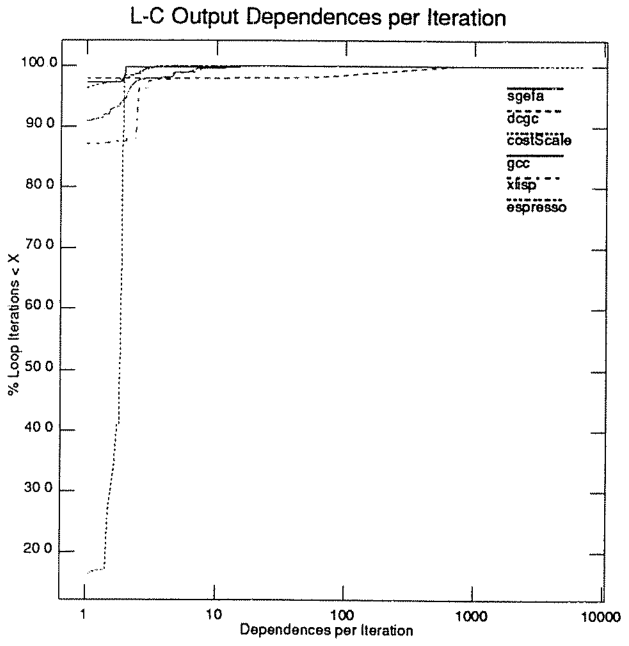


Figure 8: Anti-dependences.

Program	Number of Dependences (% of Memory References)							
	Flow		Anti		PAnti		Output	
gcc	822,503	(11.4%)	124,202	(1.7%)	1,245,036	(17.3%)	460,732	(6.4%)
xlisp	461,156	(6.5%)	17,987	(0.3%)	181,166	(2.6%)	299,282	(4.2%)
espresso	544,205	(7.4%)	99,151	(1.4%)	515,655	(7.0%)	519,029	(7.1%)
sgefa	1,012,253	(28.7%)	23,893	(0.7%)	688,341	(19.5%)	957,253	(27.1%)
dcgc	1,148,072	(17.8%)	22,550	(0.3%)	45,102	(0.7%)	774,227	(12.0%)
costScale	5,658,096	(23.1%)	129,727	(0.5%)	204,531	(0.8%)	2,341,341	(9.6%)

Table 5: Frequency of loop-carried data dependences.



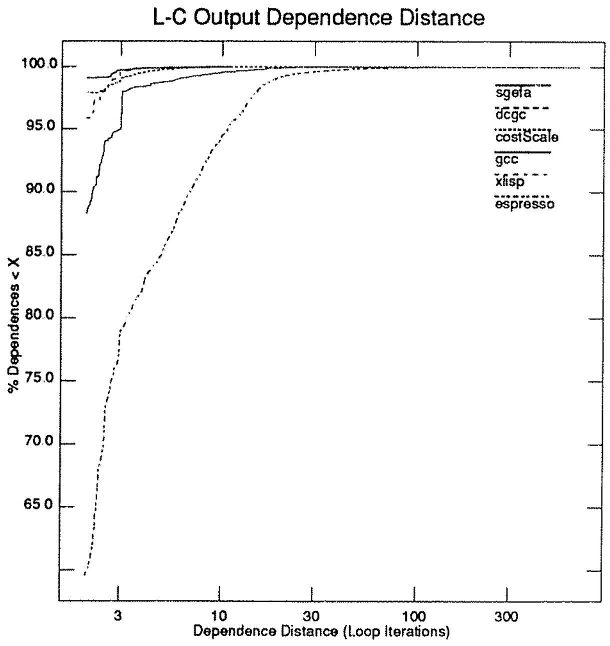


Figure 9: Output dependences.

	gcc	*xlisp*	*espresso*	*sgefa*	*dcgc*	*costScale*
Flow						
Heap Static	10.5%	33.1%	0.5%	0.0%	0.0%	0.0%
Heap Dynamic	78.0%	50.8%	98.1%	100.0%	100.0%	99.8%
Stack	11.5%	16.1%	1.4%	0.0%	0.0%	0.2%
Heap Struct	77.5%	50.8%	60.4%	5.7%	0.0%	99.8%
Heap Pointer	10.9%	33.1%	38.3%	94.3%	100.0%	0.0%
Stack Struct	9.0%	10.5%	0.3%	0.0%	0.0%	0.0%
Stack Pointer	2.6%	5.5%	1.1%	0.0%	0.0%	0.2%
Anti						
Heap Static	51.2%	89.5%	97.1%	99.6%	99.7%	100.0%
Heap Dynamic	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Stack	48.8%	10.5%	2.9%	0.4%	0.3%	0.0%
Heap Struct	38.1%	89.1%	76.3%	0.0%	0.0%	100.0%
Heap Pointer	13.2%	0.4%	20.9%	99.6%	99.7%	0.0%
Stack Struct	18.1%	0.9%	0.0%	0.3%	0.0%	0.0%
Stack Pointer	30.6%	9.6%	2.8%	0.0%	0.3%	0.0%
Output						
Heap Static	50.1%	74.3%	98.4%	100.0%	100.0%	99.8%
Heap Dynamic	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Stack	49.9%	25.7%	1.6%	0.0%	0.0%	0.2%
Heap Struct	35.8%	28.2%	73.3%	5.0%	0.0%	99.8%
Heap Pointer	14.4%	46.0%	25.0%	95.0%	100.0%	0.0%
Stack Struct	28.7%	17.3%	0.3%	0.0%	0.0%	0.0%
Stack Pointer	21.2%	8.4%	1.3%	0.0%	0.0%	0.2%

Table 6: Breakdown of loop-carried data dependences.

Data dependences, although they occur in few loop iterations, occur on a high percentage of memory references. Table 5 shows the frequency of loop-carried dependences and the percentage of memory references that result in a loop-carried dependence. Flow dependences are the most common, followed by output dependences, and by anti-dependences. Anti-dependences relatively are more common in symbolic programs. *dcgc* and *costScale* execute almost no anti-dependences. The table also illustrates that memory references that cause dependences are not evenly distributed among loop iterations, but tend to cluster so a few iterations account for most conflicting references.

Table 6 categorizes loop-carried dependences by the referenced object's type and the form of the reference. The first three entries for each dependence identify the referenced object. *Heap static* objects are global data whose size is known to the compiler. *Heap dynamic* objects are allocated by `malloc`. *Stack* objects are variables, structures, and arrays that are local to a procedure. The next four entries classify the type of memory reference in the second-executed statement in a conflict. *Heap struct* are references to structures or arrays that residing in the heap (both dynamically and statically allocated). *Heap pointer* are references to scalar variables or references through a C pointer. Note that the first category may not include all array references, since *gcc* does not try to determine that a pointer references an array. *Stack struct* and *stack pointer* are similar, except that the objects reside on the program stack.

Different programming styles cause conflicts to appear in different places in the programs. Typically, flow dependences are centered in dynamically allocated objects, which, curiously, never cause anti- or output dependences. Unfortunately, data dependence analysis for dynamically allocated objects (in particular, structures) is more complex and less precise than array analysis [6]. In symbolic programs (except *espresso*), the anti- and output dependences are divided between static and stack objects. In the numeric programs and *espresso*, almost all of these dependences are due to static objects. This difference is the product of different programming styles. However, it is worth noting that conflicts over static objects may be easier to analyze since the compiler knows the size of the underlying object.

The program statements causing conflicts also vary among the programs. In numeric programs (except *costScale*), most conflicting references are *heap pointer*, which implies that *gcc*'s simple analysis could not determine that the reference object was an array. In the symbolic programs, a wider range of statements cause the conflicts. However, the high percentage of pointer references implies that analysis of these programs would be difficult.

6 Conclusion

This paper has shown how to estimate the parallel speedup of a program without creating and running a parallel version of the program. The system currently computes an optimistic upper bound to the speedup. However, it could use a more realistic execution model and produce accurate estimates. The current measurements not only provide a basis for predicting the performance of the program on a parallel computer—and hence a baseline against which the program's actual performance can be compared—but they also provide a wealth of detail about the program's dynamic behavior. This information is valuable to compiler writers as well as people who construct languages and systems for parallel programming.

Because of the AE profiling system, these measurements require little effort on a programmer's part and do not consume much time during execution or require large amounts of disk space. By adapting techniques from AE to other compilers, it would be possible to analyze the parallelism in programs written in other languages, for example Fortran or Lisp.

The measurements of these C programs demonstrate a profound difference between the two array-manipulating numeric programs and the other programs. This difference has two components. First, the symbolic programs have smaller loops that execute fewer iterations. Second, the symbolic programs have many more loop-carried data dependences. Many existing compilation techniques (including the doacross model) were developed for programs similar to the array-manipulating programs and do not perform well for the symbolic programs. New compilation techniques, execution models, and even programming languages, are likely to prove necessary to compile symbolic programs for parallel machines.

Acknowledgments

Mark Hill generously provided disk space and a copy of the SPEC benchmarks, which facilitated this work. He and Diana Stone greatly improved this paper with their perceptive comments.

References

[1] 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.

[2] 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.

[3] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. Technical Report 89-09-18, Department of Computer Science, University of Washington, September 1989. Accepted for SIGMETRICS 1990.

[4] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence Analysis for Pointer Variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[5] Manoj Kumar and Kimming So. Trace Driven Simulation for Studying MIMD Parallel Computers. In *Proceedings of the 1989 International Conference on Parallel Processing (Vol. I Architecture)*, pages I68–I72, August 1989.

[6] James R. Larus. Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors. Technical Report UCB/CSD 89/502, Computer Science Division (EECS), University of California at Berkeley, May 1989. PhD thesis.

[7] James R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. Technical Report 912, Computer Sciences Department, University of Wisconsin–Madison, February 1990.

[8] Vivek Sarkar. Determining Average Program Execution Times and their Variance. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 298–309, June 1989.

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

[10] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, September 1989.

[11] 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.

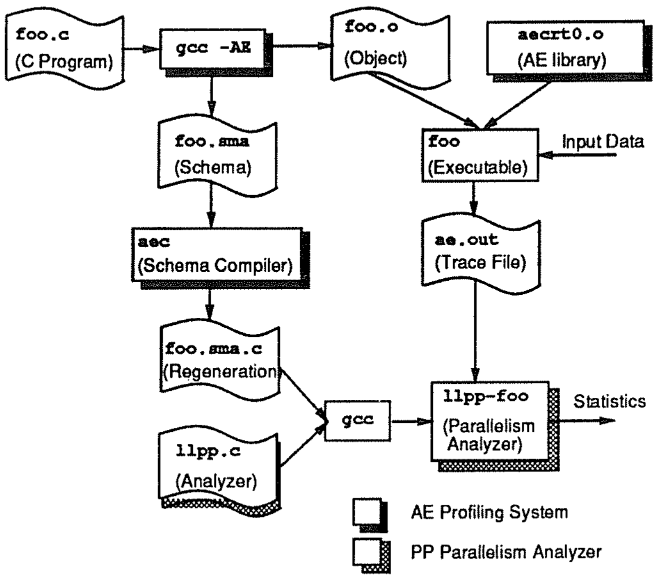


Figure 10: Overview of the AE program measurement system. The -AE flag to the GNU C compiler (gcc) causes it to add tracing code to the compiled program and to produce a schema file (`foo.sma`). The schema file describes how to interpret the trace file (`ae.out`) produced by running the program. The schema compiler (aec) translates a schema into a C program (`foo.sma.c`) that reads the trace file and generates a full program trace. This program is linked with the parallelism analyzer (`llpp.c`), which uses the trace to estimate the program's speedup.

Appendix

A Tracing Programs

The parallelism analyzer (`llpp`) depends on the program measurement system AE [7] to record the events during a program's execution that are necessary to predict parallel behavior. `llpp` needs two kinds of information. The first is details of a loop invocation—how many times the loop iterates and which instructions execute in each iteration. The other information is a complete record of the memory locations referenced by the program. This information is voluminous and could be expensive to collect and store. However, AE uses a new technique that greatly reduces the size of traces. Below is a brief description of AE.

AE is composed of two pieces. The first is a modified version of the GNU C compiler gcc [10]. This compiler performs two tasks beyond compiling a program. First, it adds a small amount of code to the compiled program to record *significant events* in a trace file. As illustrated by the example below, these events form a basis for reconstructing the program's control flow and its memory accesses. Not all branches and address calculations need to be recorded. Most can be recalculated later when producing the full trace. gcc also produces a condensed version of the program, called a *schema*, that describes how to interpret the trace file to regenerate a full program trace. Regeneration is carried out by a C program produced from the schema by the schema compiler aec. Figure 10 shows the pieces of the system.

An execution of the traced program produces a file (ae.out) containing the significant events that fully characterize the execution. These files can be interpreted only with the aid of the program's schema since they are highly compressed and omit information that can be recalculated. aec translates schema files into a C program that regenerates a full trace from the significant event trace. A simple example demonstrates this process. Consider the program:

```
main ()
{
  int i;
  int *a = (int *) malloc (sizeof (int) * 100);
  for (i = 0; i < 100; i = i + 1)  a[i] = i * i;
}
```

gcc will produce machine code for the loop similar to:

```
      move R4, 100
      call malloc
      move R5, R2         # R2 is result
      move R3, 0
      branch gt, R3, 100, end
loop: shift-left-logical R2, R3, 2
      add R2, R5, R2
      mult R6, R3, R3
      store R6, 0(R2)
      add R3, R3, 1
test: branch lt, R3, 100, loop
end:
```

In the program above, AE records only two types of significant events. First, to reproduce the program's control flow, AE adds tracing code at the beginning of basic blocks that are the target of a conditional branch. Each time such a block executes, this code records the block's number. These numbers permit the regeneration program to follow a conditional branch. Unconditional control flow is fully described by the schema and does not require significant events. AE also records the address of array a since its location cannot be determined by the regeneration program. The schema describes both the instructions produced by gcc and the significant events.

```
start_block 0
uneventful_inst 2 4
call_inst malloc
record_defn R2
compute_defn_0 R5 R2
compute_defn_0 R3 0
uneventful_inst 2 4
end_block_cjump 0 1 3

start_block_target 1
uneventful_inst 1 4
end_block_next_target 1 %loop_entry(2 0)

start_block_target 2
compute_defn_2 R2 R3 << 2
compute_defn_2 R2 R5 + R2
uneventful_inst 1 4
store_inst R2 + 0
compute_defn_2 R3 R3 + 1
end_block_cjump 2 3 2 %loop_exit(3 0) %loop_back(2 0)
```

The schema omits details of the portions of the program that produce values, as opposed to addresses. It describes instructions that compute memory addresses, reference memory, or affect control flow. The operation `record_defn` means that a value used in an address calculation (such as the result of `malloc`) cannot be regenerated. The value is recorded in the trace file. By contrast, the regeneration program can recalculate the expressions in `compute_defn`'s, so they do not record anything during execution. `uneventful_inst` are placeholders for instructions that do not contribute to memory references or control flow—in general, instructions that produce and consume values. The schema also indicates which arcs between basic blocks enter and leave loops and which begin new iterations.

The trace file for this program contains 103 items:

Stack Pointer Upon Entry	4 bytes
Result from `malloc`	4 bytes
Block 1	1 byte
⋮	
Block 1	1 byte
Block 3	1 byte

This file contains 109 bytes. The full address trace contains 3610 bytes (618 instructions and 104 memory references at 5 bytes each), so AE reduces the size of the trace by a factor of 33.[4]

[4]This ratio can be increased to 106 times by compressing `ae.out` to 34 bytes with the Unix utility `compress`.