Automatic Parallelization of Divide and Conquer Algorithms *

Radu Rugina and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{rugina, rinard}@lcs.mit.edu

Abstract

Divide and conquer algorithms are a good match for modern parallel machines: they tend to have large amounts of inherent parallelism and they work well with caches and deep memory hierarchies. But these algorithms pose challenging problems for parallelizing compilers. They are usually coded as recursive procedures and often use pointers into dynamically allocated memory blocks and pointer arithmetic. All of these features are incompatible with the analysis algorithms in traditional parallelizing compilers.

This paper presents the design and implementation of a compiler that is designed to parallelize divide and conquer algorithms whose subproblems access disjoint regions of dynamically allocated arrays. The foundation of the compiler is a flow-sensitive, context-sensitive, and interprocedural pointer analysis algorithm. A range of symbolic analysis algorithms build on the pointer analysis information to extract symbolic bounds for the memory regions accessed by (potentially recursive) procedures that use pointers and pointer arithmetic. The symbolic bounds information allows the compiler to find procedure calls that can execute in parallel without violating the data dependences. The compiler generates code that executes these calls in parallel. We have used the compiler to parallelize several programs that use divide and conquer algorithms. Our results show that the programs perform well and exhibit good speedup.

1 Introduction

Divide and conquer algorithms solve problems by breaking them up into smaller subproblems, recursively solving the subproblems, then combining the results to generate a solution to the original problem. A simple algorithm that works well for small problem sizes terminates the recursion. Good divide and conquer algorithms exist for a large variety of problems, including sorting, matrix manipulation, and many dynamic programming problems [5].

Divide and conquer algorithms have several appealing properties that make them a good match for modern parallel machines. First, they tend to have a lot of inherent parallelism. Once the division phase is complete, the subproblems are usually independent and can therefore be solved in parallel. Moreover, the recursive structure of the algorithm naturally leads to recursively generated concurrency, which means that even the divide and combine phases execute in parallel with divide and combine phases from other subproblems. This approach typically generates more than enough concurrency to keep the machine busy [3].

Second, divide and conquer algorithms also tend to have good cache performance. Once a subproblem fits in the cache, the standard recursive solution reuses the cached data until the subproblem has been completely solved. Because most of the work takes place deep in the recursive call tree, the algorithm usually spends most of its execution time running out of the cache. Furthermore, divide and conquer algorithms naturally work well with a range of cache sizes and at all levels of the memory hierarchy. As soon as a subproblem fits into one level of the memory hierarchy, the algorithm runs out of that level (or below) until the subproblem has been solved [7]. Divide and conquer programs therefore automatically adapt to different cache hierarchies, and tend to run well without modification on whatever machine is available.

It can be quite difficult, however, to parallelize programs that use divide and conquer algorithms. The natural formulation of these algorithms is recursive. For efficiency reasons, programs often use pointers into arrays and pointer arithmetic to identify subproblems. Our benchmark programs also tend to use dynamic memory allocation to match the sizes of the data structures to the problem size. All of these properties pose challenging analysis problems for the compiler. Moreover, traditional analyses for parallelizing compilers are of little or no use for this class of programs — they are designed to analyze loop nests that access arrays using affine array index expressions, not recursive procedures that use pointers and pointer arithmetic.

Inspired by the appealing properties of divide and conquer algorithms, we designed and implemented a parallelizing compiler for programs that use these algorithms. This paper presents analysis algorithms and experimental results from this effort. To successfully parallelize divide and conquer programs, we had to develop a new approach for parallelizing compilers and a new set of sophisticated analyses that realize this approach. These analyses reason symbolically about how (potentially recursive) procedures access specific regions of dynamically allocated memory.

*This research was supported in part by NSF Grant CCR-9702297.

1.1 Analysis Overview

Our compiler is designed primarily to parallelize algorithms whose subprograms use pointers and pointer arithmetic to access disjoint regions of dynamically allocated arrays. For these algorithms, the analysis usually proceeds as follows. The compiler first runs a flow-sensitive, context-sensitive, and interprocedural pointer analysis algorithm. The information extracted by this analysis is used in all successive analyses. The compiler then extracts symbolic expressions for the regions of memory accessed in the procedures that implement the base and combination phases of the divide and conquer algorithm. It uses these expressions in an interprocedural fixed-point analysis that extracts expressions for the regions of memory accessed by the recursive procedures that implement the divide phase of the algorithm. In effect, these expressions make explicit the memory access invariants that drive the recursive structure of the algorithm.

The compiler then runs a dependence testing analysis on the extracted expressions from adjacent calls. If the expressions represent disjoint regions of memory, the calls are independent and can execute in parallel. The dependence tester uses both pointer analysis information (to determine that expressions denote regions in different allocation blocks) and logical reasoning (to determine that expressions denote nonoverlapping regions in the same allocation block).

1.2 Other Applications

Although we developed these analyses to parallelize divide and conquer algorithms, we believe they will be useful in other contexts. Fundamentally, our analysis extracts expressions that characterize how pointer-based programs access regions of dynamically allocated memory. This technology could easily be applied to symbolic array bounds checking, to detect data races in explicitly parallel programs, and as part of a program understanding system that helps programmers understand the behavior of complex, pointer-based programs. We also believe we have developed the basic analysis technology necessary to fully extend traditional parallelizing compiler technology to heavily optimized programs whose loops access memory using pointers rather than array references.

Finally, we contrast the problem we are solving and our analysis techniques with the problem and analysis techniques for traditional parallelizing compilers. Traditional parallelizing compilers are designed to exploit loop-level parallelism in computations that access dense matrices using affine access functions. This problem naturally leads to integer programming algorithms that analyze potential interferences between loop iterations. Our compiler is designed to exploit recursively generated concurrency in divide and conquer computations that use pointers to identify subproblems and manipulate data. Faced with this problem, we developed a set of symbolic analysis algorithms. These algorithms use fixed-point techniques to extract invariants that describe the regions of memory that recursive procedures access.

This paper makes the following contributions:

- **Approach:** It identifies a new approach for automatically parallelizing divide and conquer algorithms whose subproblems access disjoint regions of dynamically allocated arrays. The approach fully supports recursion and the heavy use of pointers and pointer arithmetic.
- **Algorithms:** It presents a set of novel analysis algorithms that, together, enable a compiler to automatically parallelize divide and conquer programs. These algorithms are based on pointer analysis and symbolic analysis of the regions of dynamically allocated memory accessed by (potentially recursive) procedures.
- **Experimental Results:** It presents experimental results for several automatically parallelized programs. These results show that the compiler is capable of compiling divide and conquer algorithms and that the resulting parallel code performs well.

The remainder of the paper is organized as follows. Section 2 presents an example that illustrates the actions of the compiler. Sections 3, 4, 5, 6, and 7 present the analysis algorithms. Section 8 presents the experimental results from our parallelizing compiler. Section 9 discusses related work; we conclude in Section 10.

2 Example

Figure 1 presents an example of the kind of programs that our analysis is designed to handle. The `sort` procedure on line 18 implements a recursive, divide-and-conquer algorithm written in C. It takes an unsorted input array `d` of size `n`, and sorts it, using the array `t` (also of size `n`) as temporary storage. The algorithm is structured as follows.

In the divide part of the algorithm, the `sort` procedure divides the two arrays into four sections and, in lines 29 through 32, calls itself recursively to sort the sections. Once the sections have been sorted, the combination phase in lines 34 through 37 produces the final sorted array. It merges the first two sorted sections of the `d` array into the first half of the `t` array, then merges the last two sorted sections of `d` into the last half of `t`. It then merges the two halves of `t` back into `d`. The base case of the algorithm uses the insertion sort procedure in lines 9 through 17 to sort small sections.

For efficiency reasons, the `sort` program identifies subproblems using pointers into dynamically allocated memory blocks that hold the data and accesses these blocks via these pointers. This strategy leads to code containing significant amounts of pointer arithmetic and pointer comparison operators. Note, for example, the pointer arithmetic in lines 24 through 28 and the `<` pointer comparison operators in lines 3, 6, and 7. This code will usually run faster than code that uses integer array indices to identify and solve subproblems.

There are two sources of concurrency in this program: the four recursive calls to the `sort` procedure can execute in parallel, and the first two calls to the `merge` procedure can execute in parallel.

The basic problem a parallelizing compiler must solve is to determine the regions of memory that each procedure accesses. In our example, the compiler determines that a call to `merge(l1,h1,l2,h2,d)` reads the memory regions $[\texttt{l1}, \texttt{h1}-1]$ and and $[\texttt{l2}, \texttt{h2}-1]$ and writes the memory region $[\texttt{d}, \texttt{d}+(\texttt{h1}-\texttt{l1})+(\texttt{h2}-\texttt{l2})-1]$.[1] It also determines that a call to `insertionsort(l,h)` reads and writes $[\texttt{l}, \texttt{h}-1]$, and a call to `sort(d,t,n)` reads and writes $[\texttt{d}, \texttt{d}+\texttt{n}-1]$ and $[\texttt{t}, \texttt{t}+\texttt{n}-1]$.

[1] Here we use the notation $[\texttt{l}, \texttt{h}]$ to denote the region of memory between the addresses `l` and `h`, inclusive. If `h` is less than `l`, $[\texttt{l}, \texttt{h}]$ denotes the empty region. As is standard in C, we assume continguous allocation of arrays, and that the addresses of the elements increase as the array indices increase.

```
1:  void merge(int *l1, int *h1,
2:             int *l2, int *h2, int *d) {
3:    while ((l1 < h1) && (l2 < h2))
4:      if (*l1 < *l2) *d++ = *l1++;
5:      else *d++ = *l2++;
6:    while (l1 < h1) *d++ = *l1++;
7:    while (l2 < h2) *d++ = *l2++;
8:  }

9:  void insertionsort(int *l, int *h) {
10:   int *p, *q, k;
11:   for (p = l+1; p < h; p++) {
12:     k = *p;
13:     for (q = p-1; l <= q && k < *q; q--)
14:       *(q+1) = *q;
15:     *(q+1) = k;
16:   }
17: }

18: void sort(int *d, int *t, int n) {
19:   int *d1, *d2, *d3, *d4, *d5,
20:       *t1, *t2, *t3, *t4;
21:   if (n < CUTOFF) {
22:     insertionsort(d, d+n);
23:   } else {
24:     d1 = d; t1 = t;
25:     d2 = d1 + n/4; t2 = t1 + n/4;
26:     d3 = d2 + n/4; t3 = t2 + n/4;
27:     d4 = d3 + n/4; t4 = t3 + n/4;
28:     d5 = d4+(n-3*(n/4));

29:     sort(d1, t1, n/4);
30:     sort(d2, t2, n/4);
31:     sort(d3, t3, n/4);
32:     sort(d4, t4, n-3*(n/4));
33:
34:     merge(d1, d2, d2, d3, t1);
35:     merge(d3, d4, d4, d5, t3);
36:
37:     merge(t1, t3, t3, t1+n, d);
38:   }
39: }

40: void main() {
41:   int n;
42:   int *data, *temp;
43:   scanf("%d", & n);
44:   if (0 < n) {
45:     data = (int *) malloc(sizeof(int)*n);
46:     temp = (int *) malloc(sizeof(int)*n);
47:     /* code to initialize the data array */
48:     sort(data, temp, n);
49:     /* code that uses the sorted array */
50:   }
51: }
```

Figure 1: Divide and Conquer Sorting Example

2.1 Region Expressions

Roughly speaking, the compiler extracts these region expressions as follows. It first performs a flow-sensitive, context-sensitive, and interprocedural pointer analysis. The information from this pass is used at various places throughout the rest of the analyses. In some cases, the results are used to increase the precision of the analysis by determining that pointer assignments do not affect the values of local variables; in other cases they are used to disambiguate expressions that denote regions of memory in different allocation blocks.

The compiler next performs an analysis that extracts symbolic upper and lower bounds for each pointer or integer variable at each program point. These bounds are represented as expressions in the initial values of the procedure parameters. In our example, the analysis determines that at line 14, $\mathtt{l} \leq \mathtt{q} \leq \mathtt{h} - 2$; at line 15, $\mathtt{l} - 1 \leq \mathtt{q} \leq \mathtt{h} - 2$; and at line 12, $\mathtt{l} + 1 \leq \mathtt{p} \leq \mathtt{h} - 1$.

The compiler next examines all of the load or store instructions in the program. It uses the symbolic bounds to generate regions that the accesses must fall into. In our example, the analysis is able to place the reads via `p` at line 12 in the region $[\mathtt{l}+1, \mathtt{h}-1]$, the reads via `q` at lines 13 and 14 in the region $[\mathtt{l}, \mathtt{h}-2]$, the writes via `q` at line 14 in the region $[\mathtt{l}+1, \mathtt{h}-1]$, and the writes via `q` at line 15 in the region $[\mathtt{l}, \mathtt{h}-1]$. The compiler coalesces these regions to deduce that a call to `insertionsort(l,h)` reads and writes $[\mathtt{l}, \mathtt{h}-1]$. A similar analysis enables the compiler to determine that a call to `merge(l1,h1,l2,h2,d)` reads the memory regions $[\mathtt{l1}, \mathtt{h1}-1]$ and $[\mathtt{l2}, \mathtt{h2}-1]$ and writes the memory region $[\mathtt{d}, \mathtt{d}+(\mathtt{h1}-\mathtt{l1})+(\mathtt{h2}-\mathtt{l2})-1]$.

The compiler next uses the region expressions from the analysis of `merge` and `insertionsort` as the basis for a fixed-point algorithm that determines the memory locations that the `sort` procedure accesses. The algorithm repeatedly analyzes the `sort` procedure, incrementally deriving more precise information about the regions of memory that it accesses.

In our example, the analysis proceeds as follows. The algorithm uses the analysis results for `insertionsort` and `merge` to determine that the call to `insertionsort` on line 22 reads and writes $[\mathtt{d}, \mathtt{d}+\mathtt{n}-1]$, and the call to `merge` on line 34 reads $[\mathtt{d1}, \mathtt{d2}-1]$ and $[\mathtt{d2}, \mathtt{d3}-1]$ and writes $[\mathtt{t1}, \mathtt{t1}+(\mathtt{d2}-\mathtt{d1})+(\mathtt{d3}-\mathtt{d2})-1]$. It simplifies the upper and lower bound expressions to determine that the call to `merge` reads $[\mathtt{d}, \mathtt{d}+\mathtt{n}/4-1]$ and $[\mathtt{d}+\mathtt{n}/4, \mathtt{d}+\mathtt{n}/2-1]$ and writes $[\mathtt{t}, \mathtt{t}+\mathtt{n}/2-1]$. It then coalesces adjacent regions to derive a read region of $[\mathtt{d}, \mathtt{d}+\mathtt{n}/2-1]$ and a write region of $[\mathtt{t}, \mathtt{t}+\mathtt{n}/2-1]$. A similar analysis for the other `merge` call sites combined with the previously described information yields the final read and write regions $[\mathtt{d}, \mathtt{d}+\mathtt{n}-1]$ and $[\mathtt{t}, \mathtt{t}+\mathtt{n}-1]$ for a call to `sort(d,t,n)`.

The algorithm then analyzes the `sort` procedure under the assumption that each call reads and writes the regions described above. This analysis derives that a call to `sort(d,t,n)` reads and writes $[\mathtt{d}, \mathtt{d}+\mathtt{n}-1]$ and $[\mathtt{t}, \mathtt{t}+\mathtt{n}-1]$. The algorithm has therefore reached a fixed point and converges.

2.2 Parallelization

To parallelize the program, the algorithm uses the extracted region expressions to perform dependence tests between adjacent call sites. If there is no dependence, the compiler

generates code that executes the calls in parallel.[2] In our example, the dependence test between the recursive calls to `sort` in lines 29 and 30 proceeds as follows. The compiler uses the region expressions for `sort` to determine that the first call reads and writes $[\mathtt{d}, \mathtt{d}+\mathtt{n}/4-1]$ and $[\mathtt{t}, \mathtt{t}+\mathtt{n}/4-1]$, while the second call reads and writes $[\mathtt{d}+\mathtt{n}/4, \mathtt{d}+\mathtt{n}/2-1]$ and $[\mathtt{t}+\mathtt{n}/4, \mathtt{t}+\mathtt{n}/2-1]$. The compiler checks all pairs of region expressions from the two call sites to see if they are independent. If so, the calls are independent and can execute in parallel.

There are three ways for region expressions to be independent: either they denote regions in different allocation blocks, they both denote regions that are read, or they denote nonoverlapping regions of the same block. The compiler uses the pointer analysis information to determine if region expressions denote regions of different blocks and to determine if the region expressions both denote regions that are read. It reasons logically about the upper and lower bound expressions to determine if region expressions denote nonoverlapping regions of the same block. In our example, the compiler uses pointer analysis information to determine that $[\mathtt{d}, \mathtt{d}+\mathtt{n}/4-1]$ and $[\mathtt{t}+\mathtt{n}/4, \mathtt{t}+\mathtt{n}/2-1]$ denote memory regions in different blocks. It uses logical reasoning to determine that $[\mathtt{d}, \mathtt{d}+\mathtt{n}/4-1]$ and $[\mathtt{d}+\mathtt{n}/4, \mathtt{d}+\mathtt{n}/2-1]$ denote nonoverlapping regions of the same block. It can use similar strategies to determine that all of the other pairs are independent, and that the calls can execute in parallel.

Using this basic approach, the compiler can determine that all four recursive calls to `sort` can execute in parallel, and that the first two calls to `merge` can execute in parallel. It therefore generates Cilk code that executes these calls in parallel. Figure 2 contains the generated code for the `sort` procedure in our example. This code uses the Cilk `spawn` construct to execute calls in parallel, and the Cilk `sync` construct to synchronize after the parallel calls.

```
18: void sort(int *d, int *t, int n) {
19:   int *d1, *d2, *d3, *d4, *d5,
20:       *t1, *t2, *t3, *t4;
21:   if (n < CUTOFF) {
22:     insertionsort(d, d+n);
23:   } else {
24:     d1 = d; t1 = t;
25:     d2 = d1 + n/4; t2 = t1 + n/4;
26:     d3 = d2 + n/4; t3 = t2 + n/4;
27:     d4 = d3 + n/4; t4 = t3 + n/4;
28:     d5 = d4+(n-3*(n/4));

29:     spawn sort(d1, t1, n/4);
30:     spawn sort(d2, t2, n/4);
31:     spawn sort(d3, t3, n/4);
32:     spawn sort(d4, t4, n-3*(n/4));
33:     sync ;
34:     spawn merge(d1, d2, d2, d3, t1);
35:     spawn merge(d3, d4, d4, d5, t3);
36:     sync ;
37:     merge(t1, t3, t3, t1+n, d);
38:   }
39: }
```

Figure 2: Generated Parallel Code for Sorting Example

3 Analysis Overview

The primary goal of the analysis is to obtain, for each procedure, a set of symbolic region expressions that characterize how the procedure accesses memory. The compiler uses these region expressions to find independent procedure calls, then generates code that executes independent calls in parallel.

Each region expression contains a symbolic lower bound and a symbolic upper bound. These bounds are expressed in terms of a *reference set* of variables. The reference set for each procedure consists of a set of variables that denote the initial values, or values at the start of the execution of the procedure, of the parameters and referenced global variables. We denote the initial value of a parameter or global variable `p` by $\mathtt{p}_0$. For example, the reference set for the `sort` procedure in Figure 1 is $\{\mathtt{d}_0, \mathtt{t}_0, \mathtt{n}_0\}$. The analysis consists of several steps:

- **Pointer Analysis:** The pointer analysis extracts information used by all succeeding analyses.
- **Bounds Analysis:** The intraprocedural bounds analysis extracts symbolic upper and lower bounds for the values of pointer variables at each point in the program. These bounds are expressed in terms of the reference set of the enclosing procedure.
- **Region Analysis:** The region analysis extracts the set of regions accessed by each procedure. It first uses the results of the bounds analysis to extract a region expression for each pointer dereference. It then coalesces region expressions from the same procedure to obtain a minimal set of regions that each procedure accesses directly. An interprocedural fixed-point algorithm derives region expressions for the entire (potentially recursive) computation of each procedure.
- **Parallelization:** The concurrency extractor compares region expressions to find independent procedure calls (two calls are independent if neither's computation accesses memory that the other's computation writes). The code generator then generates code that executes independent calls in parallel.

Figure 3 illustrates how all of these analyses come together in the overall structure of the compiler.

4 Pointer Analysis

We use a flow-sensitive, context-sensitive, and interprocedural pointer analysis algorithm [18]. The analysis works for both sequential and multithreaded programs, although in the research presented in this paper, we use it only for sequential programs. The algorithm uses location sets to represent the memory locations accessed by statements that dereference pointers and caches the results of previous analyses to avoid performance problems caused by repeatedly analyzing the same procedure in the same context [21, 6].

The pointer analysis serves two main purposes. First, it provides the pointer disambiguation information required

[2] More precisely, the compiler generates code that exposes the concurrency to the run-time system. Actually creating a full-blown thread at each call site would generate an excessive amount of overhead. We generate code in the Cilk parallel programming language; the Cilk run-time system uses lazy task creation [15, 3] to generate only as many threads required to keep the machine busy.

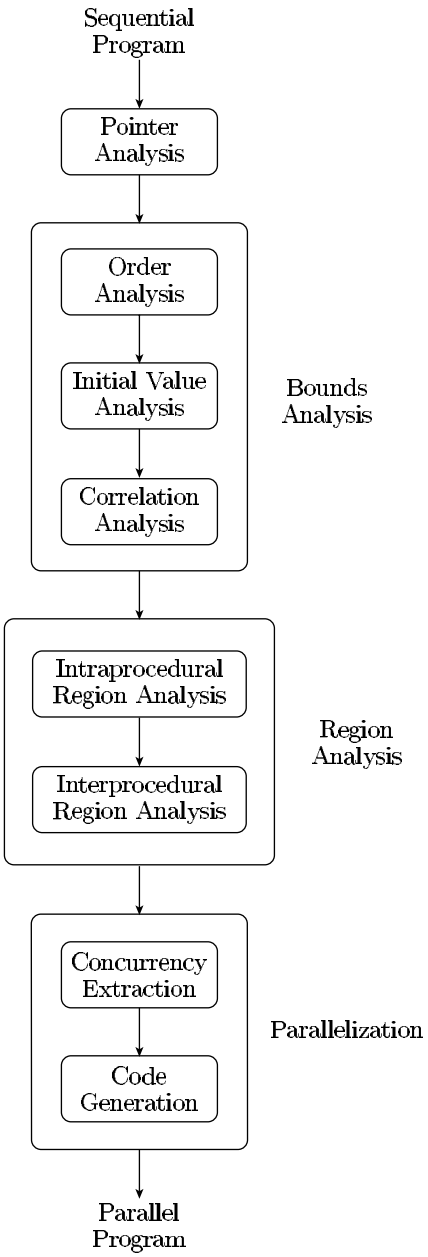

Figure 3: The Structure of the Compiler

for other dataflow analyses to give accurate results on programs that use pointers. Most of the succeeding analyses reason symbolically about the computed pointer values, and a single write via an unresolved pointer reference would destroy all of the extracted information. The analysis results are also used in the dependence testing phase to determine that region expressions denote regions in different allocation blocks.

5 Bounds Analysis

The bounds analysis consists of three subanalyses: an order analysis that extracts information about the order relationships between variables at each program point, an initial value analysis that expresses the order relationships in terms of the reference set, and a correlation analysis that improves the precision of the other analyses. At the end of the bounds analysis, the compiler has generated symbolic upper and lower bounds for each pointer dereference. These bounds are expressed in terms of the reference set of the enclosing procedure, and are used by the region analysis to derive region expressions for each procedure. The analysis uses the special lower bound $-\infty$ or the special upper bound $+\infty$ if it is unable to derive a lower or upper bound in terms of the reference set.

5.1 Order Analysis

The order analysis extracts two kinds of information. For integer variables, the *zero order* analysis maintains information about the values of the variables relative to zero. For pointer and integer variables, the *relative order* analysis maintains information about the values of the variables relative to other variables. Both analyses are predicated intraprocedural dataflow analyses with the order information generated both at assignments and at conditionals.

The zero order analysis maintains information for each integer variable `i`. It uses a lattice that can represent any disjunction (logical or) of the following atomic relations: $\texttt{i} \leq -2$, $\texttt{i} = -1$, $\texttt{i} = 0$, $\texttt{i} = 1$, and $\texttt{i} \geq 2$. The analysis formally represents these atomic relations using the set $O = \{O_{-2}, O_{-1}, O_0, O_1, O_2\}$. The lattice is the power set $\mathcal{P}(O)$ of O, and the meet operation is set union. At each program point p, the analysis produces a function $Zero_p : \mathtt{V} \rightarrow \mathcal{P}(O)$; $Zero_p(\texttt{i})$ represents the order relation for `i` relative to zero.

Similarly, the relative order analysis maintains information for each pair of integer or pointer variables `i` and `j`. It uses a lattice that can represent any disjunction (logical or) of the following atomic relations: $\texttt{i} \leq \texttt{j} - 2$, $\texttt{i} = \texttt{j} - 1$, $\texttt{i} = \texttt{j}$, $\texttt{i} = \texttt{j} + 1$, and $\texttt{i} \geq \texttt{j} + 2$. The analysis formally represents these atomic relations using the set $R = \{R_{-2}, R_{-1}, R_0, R_1, R_2\}$. The lattice is the power set $\mathcal{P}(R)$ of R, and the meet operation is set union. At each program point p, the analysis produces a function $Rel_p : \mathtt{V} \times \mathtt{V} \rightarrow \mathcal{P}(R)$; $Rel_p(\texttt{i},\texttt{j})$ represents the order relation between `i` and `j`.

When the analysis starts, both the zero order and relative order information is initialized to the empty set for all variables at all program points. The exception is the initial program point, which starts out with the zero order information initialized to O and the relative order information initialized to R for all variables.

We next consider how assignments affect the order information. Each assignment to a variable `i` kills all zero order relations for `i` and all relative order relations that involve `i`. If the assignment is of the form `i=`n or `i=j+`n, where `i` and

$\mathtt{j}$ are program variables and $n \in \mathbf{N}$ is an integer constant, the analysis uses an abstract interpretation to generate new order relations.

Figure 4 presents the abstraction functions used in the analysis. The functions g_O and g_R map integers to their abstract representations in the analysis lattices; the functions h_O and h_R are the corresponding abstractions for integer addition.

$$g_R : \mathbf{Z} \to R$$

$$g_R(n) = \begin{cases} R_{-2} & \text{if } n \leq -2 \\ R_n & \text{if } n \in \{-1, 0, 1\} \\ R_2 & \text{if } n \geq 2 \end{cases}$$

$$h_R : \mathbf{Z} \times \{-2, -1, 0, 1, 2\} \to \mathcal{P}(R)$$

$$h_R(n, m) = \begin{cases} \{g_R(k) \mid k \leq n + m\} & \text{if } m = -2 \\ \{g_R(n + m)\} & \text{if } m \in \{-1, 0, 1\} \\ \{g_R(k) \mid k \geq n + m\} & \text{if } m = 2 \end{cases}$$

$$g_O : \mathbf{Z} \to O$$

$$g_O(n) = \begin{cases} O_{-2} & \text{if } n \leq -2 \\ O_n & \text{if } n \in \{-1, 0, 1\} \\ O_2 & \text{if } n \geq 2 \end{cases}$$

$$h_O : \mathbf{Z} \times \{-2, -1, 0, 1, 2\} \to \mathcal{P}(O)$$

$$h_O(n, m) = \begin{cases} \{g_O(k) \mid k \leq n + m\} & \text{if } m = -2 \\ \{g_O(n + m)\} & \text{if } m \in \{-1, 0, 1\} \\ \{g_O(k) \mid k \geq n + m\} & \text{if } m = 2 \end{cases}$$

Figure 4: Abstraction Functions for Order Analysis

Figure 5 presents the analysis rules for assignment statements. These rules assume that p is the program point before the assignment and $p + 1$ is the program point after the assigment. The basic idea is that the rules reconstruct as much of the order information as the abstraction allows. For example, the analysis of the assignment `i=5` will produce, by rule 1, $Zero_{p+1}(\mathtt{i}) = O_2$. In other words, $\mathtt{i} \geq 2$ after the execution of `i=5`. Moreover, if before the execution of `i=5`, $O_1 \in Zero_p(\mathtt{j})$ (i.e., $\mathtt{j} = 1$), then rules 2 and 3 require that $O_2 \in Rel_{p+1}(\mathtt{i,j})$ and $O_{-2} \in Rel_{p+1}(\mathtt{j,i})$. In in other words, $\mathtt{i} \geq \mathtt{j} + 2$ and $\mathtt{j} \leq \mathtt{i} - 2$ after the assignment.

Next consider the analysis of the assignment `i=j+1` with $\mathtt{i} \neq \mathtt{j}$. Rules 1 and 2 require that $R_1 \in Rel_{p+1}(\mathtt{i,j})$ and $R_{-1} \in Rel_{p+1}(\mathtt{j,i})$ — i.e., after the execution of `i=j+1`, $\mathtt{i} = \mathtt{j} + 1$ and $\mathtt{j} = \mathtt{i} - 1$. If $O_{-1} \in Zero_p(\mathtt{j})$ (i.e., $\mathtt{j} = -1$), then by rule 3, $O_0 \in Zero_{p+1}(\mathtt{i})$ (i.e., $\mathtt{i} = 0$ after the execution of `i=j+1`). If $R_{-2} \in Rel_p(\mathtt{j,k})$ (i.e., $\mathtt{j} \leq \mathtt{k} - 2$) then by rules 4 and 5, $\{R_{-2} \cup R_{-1}\} \in Rel_{p+1}(\mathtt{i, k})$ and $\{R_2 \cup R_1\} \in Rel_{p+1}(\mathtt{k, i})$. In other words, $\mathtt{i} \leq \mathtt{k} - 1$ and $\mathtt{k} \geq \mathtt{i} + 1$ at the program point after the assignment.

We next consider how conditionals affect the order information. Conceptually, the order information that flows into the true branch is the conjunction (logical and) of the order information in the condition and the order information flowing into the conditional. The order information that flows into the false branch is the conjunction of the negation of the order information in the condition and the order information flowing into the conditional. In our lattices $\mathcal{P}(O)$ and $\mathcal{P}(R)$, the conjunction corresponds to the set intersection operation.

The analysis extracts additional order information from conditionals of the form $\mathtt{i} \leq n$, $\mathtt{i} \geq n$ and $\mathtt{i} \leq \mathtt{j} + n$, where `i` and `j` are program variables and $n \in \mathbf{N}$ is an integer constant. Other conditionals such as $\mathtt{i} < n$ or $\mathtt{i} \geq \mathtt{j} + n$ can easily be reduced to these conditionals. The analysis also supports conditionals with equality tests by replacing them with two conditionals with inequality tests: $\mathtt{i} = n$ is replaced by $\mathtt{i} \leq n$ and $\mathtt{i} \geq n$, and $\mathtt{i} = \mathtt{j} + n$ is replaced by $\mathtt{i} \leq \mathtt{j} + n$ and $\mathtt{j} \leq \mathtt{i} - n$. Figure 6 shows the analysis rules for a conditional statement at program point p with a true branch at program point t and a false branch at program point f. For conditionals of the form $\mathtt{i} \leq n$ and $\mathtt{i} \geq n$, only the zero ordering information of `i` is modified; for conditionals of the form $\mathtt{i} \leq \mathtt{j} + n$, only the relative ordering information of `i` and `j` is modified. In other words, the analysis does not perform the full transitive closure of the additional ordering information generated at the conditional.

The analysis rules in Figures 5 and 6 define analyses that are monotonic under the subset inclusion ordering — if the analysis extracts more information about the ordering at a program point p (i.e., it can use fewer atomic relations to represent the order information), it generates more information at $p + 1$ (if there is an assignment at p) and at t and f (if there is a conditional at p).

5.2 Initial Value Analysis

The order analysis produces relations that can be used to derive upper and lower bounds for each variable at each program point. But the order relations are expressed in terms of the values of the variables at the current program point. The region analysis needs the bounds to be expressed in terms of the reference set of the procedure (i.e., the initial values of the parameters and the globals).

The initial value analysis propagates the initial values of the parameters and the globals into the procedure. Whenever possible, it generates, for each variable at each program point, a mapping from that variable to an expression with variables from the reference set. The analysis is structured as a dataflow analysis on the flat lattice of expressions with least element $\bot$ and greatest element $\top$. If the analysis is unable to represent the value of a variable using an expression with variables from the reference set, it maps the variable to $\top$.

The transfer function for a statement `p=exp` generates a new mapping for `p`. It first examines all of the variables in `exp`. If any of these variables are currently mapped to $\bot$, the analysis maps `p` to $\bot$. If none of the variables are mapped to $\bot$, but at least one is mapped to $\top$, the analysis maps `p` to $\top$. Otherwise, it derives a new expression from `exp` by mapping all of the variables in `exp` to their current expressions. The analysis maps `p` to this new expression.

The merge operation is defined as follows. The merge of $\bot$ with any expression `exp` is `exp`, the merge of two identical expressions `exp` is `exp`, the merge of two different expressions is $\top$, and the merge of $\top$ with any expression is $\top$.

When the analysis starts, the mapping at the first program point maps each parameter and global variable to its corresponding variable from the reference set. All other variables are mapped to $\top$. The mappings at all of the other program points start out mapping all of the variables to $\bot$.

Rules for statement $\mathtt{i} = \mathtt{j} + n$:

1. if $\mathtt{i} \neq \mathtt{j}$ then $Rel_{p+1}(\mathtt{i},\mathtt{j}) = \{g_R(n)\}$
2. if $\mathtt{i} \neq \mathtt{j}$ then $Rel_{p+1}(\mathtt{j},\mathtt{i}) = \{g_R(-n)\}$
3. if $O_m \in Zero_p(\mathtt{j})$ then $h_O(n,m) \subseteq Zero_{p+1}(\mathtt{i})$
4. if $R_m \in Rel(\mathtt{j},\mathtt{k})$, $\mathtt{i} \neq \mathtt{k}$ then $h_R(n,m) \subseteq Rel_{p+1}(\mathtt{i},\mathtt{k})$
5. if $R_m \in Rel(\mathtt{j},\mathtt{k})$, $\mathtt{i} \neq \mathtt{k}$ then $h_R(-n,-m) \subseteq Rel_{p+1}(\mathtt{k},\mathtt{i})$

Rules for statement $\mathtt{i} = n$:

1. $Zero_{p+1}(\mathtt{i}) = \{g_O(n)\}$
2. if $O_m \in Zero_p(\mathtt{j})$, $i \neq j$ then $h_R(n,-m) \subseteq Rel_{p+1}(\mathtt{i},\mathtt{j})$
3. if $O_m \in Zero_p(\mathtt{j})$, $\mathtt{i} \neq \mathtt{j}$ then $h_R(-n,m) \subseteq Rel_{p+1}(\mathtt{j},\mathtt{i})$

Figure 5: Analysis rules for an assignment at program point p.

Rules for condition $\mathtt{i} \leq n$:

1. $Zero_t(\mathtt{k}) = \begin{cases} Zero_p(\mathtt{i}) \cap h_R(n+2,-2) & \text{if } \mathtt{k} = \mathtt{i} \\ Zero_p(\mathtt{k}) & \text{if } \mathtt{k} \neq \mathtt{i} \end{cases}$
2. $Rel_t(\mathtt{k},\mathtt{j}) = Rel_p(\mathtt{k},\mathtt{j})$ for any $\mathtt{k} \neq \mathtt{j}$
3. $Zero_f(\mathtt{k}) = \begin{cases} Zero_p(\mathtt{i}) \cap h_R(n-1,2) & \text{if } \mathtt{k} = \mathtt{i} \\ Zero_p(\mathtt{k}) & \text{if } \mathtt{k} \neq \mathtt{i} \end{cases}$
4. $Rel_f(\mathtt{k},\mathtt{j}) = Rel_p(\mathtt{k},\mathtt{j})$ for any $\mathtt{k} \neq \mathtt{j}$

Rules for condition $\mathtt{i} \geq n$:

1. $Zero_t(\mathtt{k}) = \begin{cases} Zero_p(\mathtt{i}) \cap h_R(n-2,2) & \text{if } \mathtt{k} = \mathtt{i} \\ Zero_p(\mathtt{k}) & \text{if } \mathtt{k} \neq \mathtt{i} \end{cases}$
2. $Rel_t(\mathtt{k},\mathtt{j}) = Rel_p(\mathtt{k},\mathtt{j})$ for any $\mathtt{k} \neq \mathtt{j}$
3. $Zero_f(\mathtt{k}) = \begin{cases} Zero_p(\mathtt{i}) \cap h_R(n+1,-2) & \text{if } \mathtt{k} = \mathtt{i} \\ Zero_p(\mathtt{k}) & \text{if } \mathtt{k} \neq \mathtt{i} \end{cases}$
4. $Rel_f(\mathtt{k},\mathtt{j}) = Rel_p(\mathtt{k},\mathtt{j})$ for any $\mathtt{k} \neq \mathtt{j}$

Rules for condition $\mathtt{i} \leq \mathtt{j} + n$:

1. $Rel_t(\mathtt{k},\mathtt{l}) = \begin{cases} Rel_p(\mathtt{i},\mathtt{j}) \cap h_R(n+2,-2) & \text{if } \mathtt{k} = \mathtt{i}, \mathtt{l} = \mathtt{j} \\ Rel_p(\mathtt{j},\mathtt{i}) \cap h_R(-n-2,2) & \text{if } \mathtt{k} = \mathtt{j}, \mathtt{l} = \mathtt{i} \\ Rel_p(\mathtt{k},\mathtt{l}) & \text{otherwise} \end{cases}$
2. $Zero_t(\mathtt{k}) = Zero_p(\mathtt{k})$ for any $\mathtt{k}$
3. $Rel_f(\mathtt{k},\mathtt{l}) = \begin{cases} Rel_p(\mathtt{j},\mathtt{i}) \cap h_R(-n+1,-2) & \text{if } \mathtt{k} = \mathtt{j}, \mathtt{l} = \mathtt{i} \\ Rel_p(\mathtt{i},\mathtt{j}) \cap h_R(n-1,2) & \text{if } \mathtt{k} = \mathtt{i}, \mathtt{l} = \mathtt{j} \\ Rel_p(\mathtt{k},\mathtt{l}) & \text{otherwise} \end{cases}$
4. $Zero_f(\mathtt{k}) = Zero_p(\mathtt{k})$ for any $\mathtt{k}$

Figure 6: Analysis rules for a conditional at program point p, with true branch at program point t and false branch at program point f.

5.3 Pointer Disambiguation in Order and Initial Value Analyses

The order and initial value analyses use the pointer analysis information to maintain precision in the face of pointer dereferences. Consider, for example, an assignment `*p=exp`. If the pointer analysis determines that `p` always points to a specific variable `v`, the compiler can conceptually replace `*p` with `v` in the assignment. This conceptual transformation allows the compiler to analyze `*p=exp` as `v=exp`. The order analysis can therefore generate precise order information for `v` and the initial value analysis can map `v` to an accurate expression derived from `exp`. A similar approach preserves precision in the presence of reads via pointers.

The compiler falls back on conservative approaches if it is unable to completely disambiguate a pointer. Assume that the compiler is only able to determine that, at the assignment `*p=exp`, `p` points to one of several variables. In this case, the initial value analyis conservatively maps all of the potentially updated variables to $\top$ when it analyzes the assignment. Similarly, the order analysis kills all of the order information involving any of the potentially updated variables.

Finally, it is possible for a callee procedure to change the order or initial value information in the caller. This can happen, for example, if the caller passes a pointer variable by reference, and the callee modifies the pointer variable. An unmapping process similar to that used in standard pointer analysis algorithms ensures that the analyses conservatively model this possibility.

5.4 Correlation Analysis

The compiler uses correlation analysis to improve the precision of the bounds analysis in cases, such as for the variable `d` in the `merge` procedure from Figure 1, when the order and initial value analyses fail to derive accurate bounds. Correlation analysis is designed to detect relationships of the following form: "whenever `p` is incremented, exactly one of `q`, `r`, or `s` is also incremented". In this case, we say that `p` is the *target* variable, and that `q`, `r`, and `s` are the *correlated* variables. The compiler uses this information to derive

bounds for the target variable in terms of the bounds and initial values of the correlated variables.

The analysis is triggered whenever the compiler is unable to derive bounds for a target variable using the order and initial value analyses. For each target variable and each program point, the analysis produces two sets of variables: a set of correlated variables with which the target variable is correlated, and a set of uncorrelated variables with which the target variable is known to be not correlated. The analysis maintains the invariant that these two sets are disjoint. Once a variable enters the set of uncorrelated variables, it never moves back into the set of correlated variables at any subsequent point in the program.

5.4.1 The Analysis Algorithm

The analysis starts by examining the basic blocks to match each increment of the target variable with an increment in the same basic block of a correlated variable whose symbolic bounds are already known from the order and initial value analyses. The initial value analysis must also have successfully extracted a symbolic initial value for the correlated variable at the start of the procedure in terms of the reference set of the procedure. In the example in Figure 1, each increment of `d` is matched with the increment of `l1` or `l2` from the same line in the program. Because `l1` and `l2` are parameters, their initial values are simply their values at the start of the `merge` procedure.

This matching is used to define the transfer functions for basic blocks. The output set of uncorrelated variables that flows out of a basic block is the input set of uncorrelated variables that flows into the basic block plus all variables updated in the block whose updates are not matched. The output set of correlated variables is computed as follows. The compiler first augments the input set of correlated variables to include all variables with matched increments in the basic block. It then removes all variables that are in the output set of uncorrelated variables.

The merge operation is defined as follows. The output set of uncorrelated variables is the union of the input sets of uncorrelated variables. The output set of correlated variables is the union of the input sets of correlated variables minus the output set of uncorrelated variables. The sets of correlated and uncorrelated variables are empty when the analysis starts, and monotonically increase to their final values as the analysis proceeds.

For the target variable `d` in the example in Figure 1, the sets of correlated variables for the basic blocks in lines `4` and `5` are {`l1`} and {`l2`}, respectively. The sets of uncorrelated variables are empty. The merge operation at the top of the **while** loop in line `3` will compute {`l1,l2`} as the correlated set for `d`, and will leave the uncorrelated set empty. The analysis reaches a fixed point with {`l1, l2`} as the correlated set for `d` at the beginning of each basic block that dereferences `d`.

5.4.2 Using Correlation Information

The correlation information establishes equations for the values of reference variables in terms of the correlated variables. Whenever a target variable is correlated with a set of variables at a certain program point, the difference between the value of the target variable and its initial value is equal to the sum of differences between the values of the correlated variables and their initial values. The compiler uses this equation to derive bounds for the target variable in terms of the bounds for the correlation variables.

In the example in Figure 1, the equation between the values of the target variable `d` and the correlated variables `l1` and `l2` is $\mathtt{d} - \mathtt{d}_0 = (\mathtt{l1} - \mathtt{l1}_0) + (\mathtt{l2} - \mathtt{l2}_0)$ at the start of each basic block that dereferences `d`. This equation can be combined with the bounds $\mathtt{l1}_0 \leq \mathtt{l1} < \mathtt{h1}_0$ for `l1` and $\mathtt{l2}_0 \leq \mathtt{l2} < \mathtt{h2}_0$ for `l2` to obtain the bounds $\mathtt{d}_0 \leq \mathtt{d} < \mathtt{d}_0 + (\mathtt{h1}_0 - \mathtt{l1}_0) + (\mathtt{h2}_0 - \mathtt{l2}_0)$ for `d`.

The compiler uses the bounds at the start of each basic block to obtain bounds at each program point within the basic block. It simply propagates the bounds from the start of the block into the block, incrementing the lower and upper bound whenever the target variable is incremented.

6 Region Analysis

The region analysis extracts symbolic region expressions that characterize how each statement and the computation rooted at each call site access data. Throughout the analysis, the compiler keeps reads and writes separate, generating a set of read region expressions and a separate set of write region expressions for each statement and each call site. Because the variables in these expressions are all from the reference set of the enclosing procedure, the dependence tester can directly compare the region expressions to see if they overlap.

The region analysis starts with the results of the bounds analysis, which generates an upper and lower bound for each pointer dereference in terms of the reference set of the enclosing procedure. For each procedure, the region analysis coalesces adjacent and overlapping regions from the procedure's pointer dereferences to obtain a minimal set of regions that the procedure directly accesses. It then uses an interprocedural fixed-point algorithm to extract the regions accessed by the entire computation of the procedure. This algorithm analyzes call sites and propagates region expressions from callees to callers.

When the interprocedural analysis terminates, it has computed a set of region expressions for each procedure. These region expressions are given in terms of the reference set of the procedure and characterize how that procedure accesses data. As a byproduct of the interprocedural analysis, the compiler also generates a set of symbolic region expressions that characterize how each statement and call site in the program access data; these region expressions are given in terms of the reference set of the enclosing procedure. The dependence testing phase uses these region expressions to extract the concurrency.

6.1 Region Expressions

Each region expression is represented in the form $[\mathtt{l}, \mathtt{u}]$, where `l` is the lower bound and `u` is the upper bound. Both `l` and `u` are symbolic expressions in terms of the reference set of the currently analyzed procedure. These expressions are of the form $\mathtt{p} + \mathtt{exp}$, where `p` is a pointer into the accessed allocation block and `exp` is an integer expression representing the pointer offset.

If the region expression summarizes how a statement or procedure reads data, it is marked as a read expression; if it summarizes how a statement or procedure writes data, it is marked as a write expression. It is important to realize that each region expression identifies a region of memory *within a specific set of allocation blocks.* The pointer analysis determines the set of allocation blocks for each region expression. So even if the symbolic analysis is unable to generate symbolic bounds for a region expression in terms of the reference set, the region expression does not denote

all of memory. It merely denotes all of the memory in the allocation blocks that the pointer analysis extracted for that region expression.

6.2 Intraprocedural Region Analysis

The intraprocedural region analysis generates the *local region set* of the procedure, or a minimal set of region expressions that characterize how the procedure accesses data. The local region set is expressed in terms of the procedure's reference set. The analysis starts by using the results of the bounds analysis to extract a region expression for each pointer dereference in the procedure. At each pointer dereference, it uses the order information (as translated by the initial value analysis into the reference set of the procedure) to obtain upper and lower bounds for the region that the dereference accesses. Together, these bounds make up the region expression for that dereference. The special bound $-\infty$ is used as the lower bound for dereferences with no lower bound from the bounds analysis; $+\infty$ is used as the upper bound for dereferences with no upper bound from the bounds analysis.

The local region set is initialized to the union of the region expressions from all of the pointer dereferences in the procedure. An iterative algorithm repeatedly finds two region expressions in this set whose upper and lower bounds are either adjacent or overlap. It then merges the regions into a new region as follows. The lower bound of the new region is the minimum of the lower bounds of the original regions, and the upper bound is the maximum of the upper bounds of the original regions. The original regions are removed from the local region set, the new region is inserted into the set, and the algorithm iterates until there are no more adjacent or overlapping regions.

This algorithm assumes that the analysis can compare the upper and lower bounds of region expressions and generate the minimum and maximum of two bounds. These bounds are symbolic expressions in the reference set of enclosing procedure. During the bounds analysis, each bounds expression is transformed into a sum of terms; each term is a product of a coefficient and a variable from the reference set. The algorithm compares two such expressions by comparing corresponding terms: one expression is larger than another if all of its terms are larger. The algorithm compares terms by comparing their coefficients. If the variable from the reference set is positive or zero, the compiler chooses the term with the larger coefficient as the larger term. If the variable is negative, the term with the larger coefficient is the smaller term. This approach requires that the compiler know the sign of the integer variables in the reference set. The current implementation relies on the programmer to declare all such variables as C `unsigned` variables, which forces them to be nonnegative. It would be straightforward to implement an interprocedural abstract analysis to determine the sign of variables in the reference set.

The algorithm that computes the minimum and maximum of two bounds expressions also operates at the granularity of terms. The minimum of two bounds expressions is the sum, over all pairs of corresponding terms, of the smaller term in the pair; the maximum is the sum of the larger terms in corresponding pairs.

For the example in Figure 1, the local region sets for `main` and `sort` are empty. The local region set for `merge` reads $[\mathtt{l1}_0, \mathtt{h1}_0 - 1]$ and $[\mathtt{l2}_0, \mathtt{h2}_0 - 1]$ and writes $[\mathtt{d}_0, \mathtt{d}_0 + (\mathtt{h1}_0 - \mathtt{l1}_0) + \mathtt{h2}_0 - \mathtt{l2}_0 - 1]$. The local region set for `insertionsort` reads and writes $[\mathtt{l1}_0, \mathtt{h1}_0 - 1]$.

6.3 Interprocedural Region Analysis

For each procedure, the interprocedural region expression analysis uses the local region sets to derive a *global region set*, or a minimal set of region expressions that characterize how the entire execution of the procedure accesses data.

6.3.1 Non-Recursive Procedures

For non-recursive procedures, the analysis extracts the global region sets by propagating region sets up from the leaves of the call graph towards the root. For each procedure, the global region set is initialized to the procedure's local region set. At each propagation step, the analysis performs a *symbolic unmapping* as follows. It first translates the region expressions from the reference set of the callee to expressions in the variables of the caller. It then translates the expressions from the variables of the caller to expressions in the reference set of the caller. The resulting expressions are added to the current global region set of the caller, with adjacent or overlapping regions coalesced as discussed in Section 6.2.

Consider, for example, the call site at line `34` in Figure 1 where `sort` calls `merge`. The global region set for `merge` contains the read region expression $[\mathtt{l1}_0, \mathtt{h1}_0 - 1]$. The analysis first unmaps this region expression into the variables of the `sort` procedure to obtain the region expression $[\mathtt{d1}, \mathtt{d2} - 1]$. It then uses the bounds analysis information to obtain the lower bound $\mathtt{d}_0$ for `d1` and the upper bound $\mathtt{d}_0 + \mathtt{n}_0/4 - 1$ for $\mathtt{d2} - 1$. Note that both bounds expressions are in terms of the reference set for `sort`. The compiler combines these bounds to obtain the region expression $[\mathtt{d}_0, \mathtt{d}_0 + \mathtt{n}_0/4 - 1]$, which is the translation of the original region expression from the global region set of `merge` into a region expression that characterizes, in part, how a specific call to `merge` accesses data.

6.3.2 Recursive Procedures

The analysis uses a fixed point algorithm to handle recursive procedures. For each recursive procedure, the analysis initially sets the procedure's global region set to its local region set. It then applies the bottom-up symbolic unmapping algorithm described above to propagate region expressions from callees to callers. It terminates the recursion by using the procedure's current global region set in the unmapping as an approximation of its actual global region set. Whenever possible, the unmapped region expressions are coalesced into the current global region set of the caller. The analysis uses the coalescing algorithm discussed above in Section 6.2. The algorithm continues until it reaches a fixed point.

In some cases, this analysis generates an unbounded number of region expressions that cannot be coalesced. This may happen, for instance, when the recursive function accesses a statically unbounded number of disjoint regions. In this case, the analysis as described above will not terminate. Even if the analysis is always able to coalesce the region expressions from recursive calls into the current global region set, the analysis as described above may not terminate if the bounds always increase or decrease.

The compiler therefore imposes a finite bound on the number of analysis iterations. If the analysis fails to converge within this bound, we replace the extracted region expressions with corresponding region expressions that identify the entire allocation blocks as potentially accessed.

In the example in Figure 1, the global region set for `sort` starts out empty. The interprocedural region analysis for non-recursive procedures propagates the region expressions from the calls to `insertionsort` and `merge` into the

`sort` procedure, and coalesces the resulting region expressions to add the read region $[\mathtt{d}_0, \mathtt{d}_0 + \mathtt{n}_0 - 1]$ and the write region $[\mathtt{t}_0, \mathtt{t}_0 + \mathtt{n}_0 - 1]$ to the global region set for `sort`. The interprocedural region analysis for recursive procedures uses this global region set as an approximation to derive region expressions that characterize how the recursive calls to `sort` access data. After coalescing these region expressions back into the current global region set for `sort`, the analysis reaches a fixed point.

7 Parallelization

The goal of the compiler is to find sequences of procedure calls that can execute in parallel. The primitives the compiler works with are the Cilk spawn and sync primitives [3]. The spawn primitive generates a parallel call — the spawned procedure executes in parallel with the rest of caller, including any subsequent parallel calls. The sync primitive blocks until all of the caller's outstanding parallel calls terminate. The output of the concurrency extraction phase of the compiler consists of a set of spawn points (each spawn point corresponds to a parallel call) and a set of sync points. The compiler inserts these constructs to maximize concurrency subject to the constraint that the parallel program preserve the data dependences of the original serial program.

7.1 Dependence Testing

Given the Cilk primitives, the relevant data dependences exist between a callee and subsequent statements in the caller, and between multiple callees. The compiler enforces these dependences by comparing the region expressions from callees with region expressions from statements or other callees. If a write region from one of the two has a nonempty intersection with a write or read region from the other, then there is a potential data dependence. Otherwise, there is no dependence.

The intersection test between two regions is performed as follows. The compiler first uses the pointer analysis information to check if the expressions denote regions in different allocation blocks. If so, their intersection is empty and there is no dependence. If not, the compiler does a symbolic check of the expressions for the lower and upper bounds of the two regions. If the upper bound for one region is less than the lower bound of the other, then the regions have an empty intersection and there is no dependence. If the compiler is unable to determine that the upper bound for one region is less than the lower bound of the other, it must conservatively assume that the intersection is nonempty. The bounds comparison checks are done symbolically using the expression comparison algorithm described in Section 6.2.

7.2 Concurrency Extraction

The compiler uses the dependence test outlined above as the foundation of an algorithm that identifies *parallel sections* of the program, or sections in which each procedure call can execute in parallel with all subsequent statements and all other procedure calls in the section.

The parallel section algorithm starts with an initial call site. It then traverses the control flow graph of the program to grow the parallel section as follows. It repeatedly visits a candidate statement or call site on the control flow frontier of the parallel section. To visit a candidate, the compiler performs a dependence test between the symbolic region expressions of of the candidate and the symbolic region expressions of the call sites in the current parallel section. These symbolic region expressions are generated by the region analysis, and are all expressed in terms of the reference set of the enclosing procedure.

If all of the dependence tests indicate that there is no dependence, the candidate statement or call site is added to the parallel section. Otherwise, the program point before the statement or call site is marked as a sync point. The algorithm continues until all of the statements or call sites on the frontier either have been visited or are the end node of the procedure.

The analysis of the procedure generates multiple (potentially overlapping) parallel sections as follows. The compiler first traverses the call graph to find an initial call site. It then performs the parallel section algorithm described above to generate a set of sync points. The algorithm next finds another call site that is not yet in any parallel section, and repeats the parallel section algorithm using that call site as the initial site. The algorithm terminates when every call site is in a parallel section. All call sites in parallel sections that contain at least two call sites are identified as spawn points.

7.3 Code Generation

Once the compiler has determined the spawn and sync points, code generation is straightforward. The compiler inserts a spawn construct at each spawn point and a sync construct at each sync point.

8 Experimental Results

We have implemented a parallelizing compiler based on the analysis algorithms presented in this paper. This compiler was implemented using the SUIF compiler infrastructure [1]. We implemented all of the analyses, including the pointer analysis, from scratch starting with the standard SUIF distribution. The compiler generates code in the Cilk parallel programming language [3]. We present experimental results for two programs: a version of the sorting program presented in Section 2 and a divide and conquer matrix multiplication program. The matrix multiplication program is representative of matrix manipulation programs; the sorting program is representative of less regular divide and conquer algorithms.

We ran the generated programs on an eight processor Sun Ultra Enterprise Server. Table 1 presents the execution times and self-relative speedups for the automatically parallelized matrix multiply program. The input is a 1024 by 1024 matrix of doubles. For comparison purposes, the execution time of the standard naive, triply-nested matrix multiply loop is 316 seconds, as opposed to 28.5 seconds for the automatically parallelized version running on one processor. We attribute the performance difference to cache improvements from blocking and from an efficient, hand-unrolled implementation of the base case in the automatically parallelized version.

Table 2 presents the execution times and self-relative speedups for the automatically parallelized sort program presented in Section 2. The input is four million randomly generated integers. For comparison purposes, the execution time of the sequential version of this program is 9.23 seconds.

	1	2	4	6	8
Time (seconds)	28.5	14.8	7.4	5.1	3.8
Speedup	1.0	1.9	3.8	5.6	7.5

Table 1: Execution Times and Speedups for Divide and Conquer Matrix Multiply

	1	2	4	6	8
Time (seconds)	9.24	4.94	2.99	2.36	2.11
Speedup	1.0	1.9	3.1	3.9	4.4

Table 2: Execution Times and Speedups for Divide and Conquer Sort

9 Related Work

Many tree traversal programs can be viewed as divide and conquer programs. For this class of programs there is a significant body of research in the area of shape analysis, which is designed to discover when a data structure has a certain "shape" such as a tree or list [4, 19, 8]. Several researchers have used shape analysis algorithms as the basis for compilers that automatically parallelize divide and conquer programs that manipulate linked data structures [9, 13, 14]. We are aware, however, of no previous research on parallelizing compilers for divide and conquer programs (such as those in our benchmark set) that use pointers to access disjoint regions of large, contiguously allocated blocks of memory.

Several researchers have developed symbolic analysis techniques for various parallelization approaches. Blume and Haghighat [2, 10] have independently developed symbolic analysis techniques for parallelizing loop nests that manipulate dense matrices [2, 10]. Rinard and Diniz have developed symbolic analysis techniques for detecting commuting operations on objects and using the commutativity information to automatically parallelize irregular, object-based programs [17].

Moon, Hall and Murphy have developed a data-flow analysis that uses the conditions in flow of control statements to obtain extra precision. They use the analysis to generate conditions that guard conditionally optimized code, and to generate conditions that use run-time information to identify parallel loops [16].

There has been a significant amount of research on extracting array sections in scientific programs that manipulate dense matrices [20, 12, 11]. These techniques are all designed to work for programs with loop nests that access matrices using affine access functions. The techniques presented in this paper, on the other hand, are designed to work for pointer references in recursive procedures with general control flow.

10 Conclusion

Traditional parallelizing compilers have focused on an important, but narrow, form of concurrency: the concurrency available in loop nests that manipulate dense matrices using affine access functions. This paper presents algorithms and experimental results from a parallelizing compiler that focuses on a more general and no less important form of concurrency: the recursively generated concurrency available in divide and conquer algorithms.

To exploit this form of concurrency, we found that we had to implement both pointer analysis and a set of new symbolic analysis algorithms. These algorithms allow the compiler to reason statically about the regions of memory accessed in (potentially recursive) procedures that heavily use pointers and pointer arithmetic. The compiler uses this region access information to detect independent calls to these procedures and to generate code that executes the independent calls in parallel.

We have implemented a parallelizing compiler based on this general approach; our experimental results show that this compiler is capable of automatically extracting concurrency from optimized implementations of divide and conquer algorithms.

Acknowledgements

We would like to thank Nate Kushman, Darko Marinov, and Don Dailey for their help in generating the experimental results.

References

[1] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[2] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, Santa Barbara, CA, April 1995. IEEE Computer Society Press, Los Alamitos, Calif.

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995. ACM, New York.

[4] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990. ACM, New York.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introductions to Algorithms.* The MIT Press, Cambridge, Mass., Cambridge, MA, 1990.

[6] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994. ACM, New York.

[7] J. Frens and D. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[8] R. Ghiya and L. Hendren. Is is a tree, a DAG or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 1–15, January 1996.

[9] V. Guarna. A technique for analyzing pointer and structure references in parallel restructuring compilers. *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[10] M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[11] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press, Los Alamitos, Calif.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[13] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[14] J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988. ACM, New York.

[15] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197. ACM, New York, June 1990.

[16] S. Moon, M. Hall, and B. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1993.

[17] M. Rinard and P. Diniz. Commutativity analysis: A new framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, pages 54–67, Philadelphia, PA, May 1996. ACM, New York.

[18] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[20] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[21] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.