Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions*

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

Abstract

This paper presents a novel framework for the symbolic bounds analysis of pointers, array indices, and accessed memory regions. Our framework formulates each analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to a linear program. The solution to the linear program provides symbolic lower and upper bounds for the values of pointer and array index variables and for the regions of memory that each statement and procedure accesses. This approach eliminates fundamental problems associated with applying standard fixed-point approaches to symbolic analysis problems. Experimental results from our implemented compiler show that the analysis can solve several important problems, including static race detection, automatic parallelization, static detection of array bounds violations, elimination of array bounds checks, and reduction of the number of bits used to store computed values.

1 Introduction

This paper presents a new algorithm for statically extracting information about the regions of memory that a program accesses. To obtain accurate information for programs whose memory access patterns depend on the input, our analysis is symbolic, deriving polynomial expressions that bound the ranges of the pointers and array indices used to access memory. Our prototype compiler uses the analysis information to solve a range of problems, including automatic race detection for parallel programs, automatic parallelization of sequential programs, static detection of array bounds violations, static elimination of array bounds checks, and (when it is possible to derive precise numeric bounds) automatic computation of the minimum number of bits required to hold the values that the program computes.

We have applied our techniques to divide and conquer programs that access disjoint regions of dynamically allocated arrays [16, 13, 8]. These programs present a challenging set of program analysis problems: they use recursion as their primary control structure, they use dynamic memory allocation to match the sizes of the data structures to the problem size, and they access data structures using pointers and pointer arithmetic, which complicates the static disambiguation of memory accesses.

The straightforward application of standard program analysis techniques to this class of programs fails because the domain of symbolic expressions has infinite ascending chains. This paper presents a new framework that eliminates this problem. Instead of using traditional fixed-point algorithms, it formulates each analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to a linear program. The solution to the linear program provides symbolic lower and upper bounds for the values of pointer and array index variables and for the regions of memory that each statement and procedure accesses. The analysis solves one symbolic constraint system per procedure, then one symbolic constraint system for each strongly connected component in the call graph.

1.1 Static Race Detection

Explicitly parallel languages give programmers the control they need to produce extremely efficient programs. The drawback is that explicit parallelism can significantly complicate the development process. One of the main complications is the possibility of *data races*, or unanticipated interactions that occur at memory locations accessed by parallel threads. A divide and conquer program has a data race when one thread writes a location that another parallel thread accesses. Our analysis statically compares the regions of memory accessed by parallel threads to determine if there may be a data race. If not, the programmer is guaranteed that the program is race-free and will execute deterministically with no unanticipated interactions. Our analysis therefore eliminates a primary complication associated with parallel programming.

1.2 Automatic Parallelization

The difficulty of developing parallel programs has led to a large research effort devoted to automatically parallelizing sequential programs [1, 3, 17, 23]. Our analysis is capable of automatically parallelizing sequential divide and conquer programs. We emphasize the fact that traditional parallelization techniques are of little or no use for this class of

*This research was supported in part by NSF Grant CCR-9702297. We intend to maintain a full, updated version of this paper at www.cag.lcs.mit.edu/~rinard/paper/pldi00.updated.ps.

PLDI 2000, Vancouver, British Colombia, Canada.

programs — they are designed to analyze loop nests that access dense matrices using affine index expressions, not recursive procedures that use pointers and offsets into dynamically allocated arrays.

Our analysis allows us to be neutral on the issue of explicit parallelism versus automatic parallelization. If the programmer prefers to write a parallel program, our compiler will help eliminate one of the key problems, data races. If the programmer prefers to write a sequential program, our compiler will automatically parallelize the program so that it executes efficiently on parallel machines.

1.3 Detecting Array Bounds Violations

For efficiency reasons, low-level languages like C do not check that array accesses fall within the array bounds. But array bounds violations are a serious potential problem, in large part because they introduce unanticipated and difficult to understand interactions between statements that violate the array bounds and the data structures that they incorrectly access.

Because our algorithms characterize the regions of memory accessed by statements and procedures, they allow the compiler to certify that a program will never violate the array bounds. This is true even for programs that dynamically allocate arrays and use pointer arithmetic to obtain long-lived pointers into the middle of arrays.

1.4 Eliminating Array Bounds Checks

Safe languages like Java eliminate the possibility of undetected array bounds violations by dynamically checking that each array access falls within the array bounds. A problem with this approach is the cost of executing the extra instructions that check for array bounds violations.

Because our algorithms characterize the regions of memory accessed by statements and procedures, they allow the compiler to eliminate array bounds checks. If the regions accessed by a statement or procedure fall within the array bounds, the compiler can safely eliminate any associated checks. Once again, our analysis allows us to be neutral on the issue of array bounds checks. If the programmer would prefer to use an unsafe language, the compiler can certify that the program does not violate the array bounds. If the programmer would prefer to use a safe language, the compiler can automatically eliminate the array bounds checks.

1.5 Bitwidth Analysis

Although our analysis is designed to derive symbolic bounds, it extracts precise numeric bounds when it is possible to do so. In this case, it can bound the number of bits required to represent the values that the program computes. These bounds can be used to eliminate superfluous bits from the structures used to store the values, reducing the memory and energy consumption of hardware circuits automatically generated from programs written in standard programming languages [2, 6, 27].

1.6 Contributions

This paper makes the following contributions:

- **Analysis Framework:** It presents a novel framework for the symbolic bounds analysis of pointers, array indices, and accessed memory regions. This framework formulates the analysis problem using systems of symbolic inequality constraints.
- **Solution Mechanism:** Standard program analyses use iterative fixed-point algorithms to solve systems of inclusion constraints or dataflow equations [21]. But fixed-point methods fail to solve our constraint systems because the domain of symbolic expressions has infinite ascending chains. Instead of attempting to iterate to a solution, our new approach reduces each system of symbolic constraints to a linear program. The solution of this linear program translates directly into a solution for the symbolic constraint system. There is no iteration and no possibility of nontermination.
- **Pointer Analysis:** It shows how to use pointer analysis to enable the application of the analysis framework to programs that heavily use dynamic allocation, pointers into the middle of dynamically allocated memory regions, and pointer arithmetic.
- **Analysis Uses:** It presents algorithms that use the analysis results to solve several important problems, including static race detection, automatic parallelization, detection of array bounds violations, elimination of array bounds checks, and reduction of the number of bits used to store the values computed by the program.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of the algorithms on a set of benchmark programs. Our results show that the algorithms can verify the absence of data races in our benchmark parallel programs, detect the available parallelism in our benchmark serial programs, and verify that both sets of benchmark programs do not violate their array bounds. They can also significantly reduce the number of bits required to store the state of our benchmark bitwidth analysis programs.

The remainder of the paper is organized as follows. Section 2 presents a running example that we use throughout the paper. Section 3 presents the analysis algorithms, while Section 4 presents some extensions to these algorithms. Section 5 presents experimental results from our implementation. Section 6 discusses related work. We conclude in Section 7.

2 Example

Figure 1 presents a simple example that illustrates the kinds of programs that our analysis is designed to handle. The `dcInc` procedure implements a recursive, divide-and-conquer algorithm that increments each element of an array. The example is written in Cilk, a parallel dialect of C [14].

2.1 Parallelism in the Example

In the divide part of the algorithm, the `dcInc` procedure divides each array into two subarrays. It then calls itself recursively to increment the elements in each subarray. Because the two recursive calls are independent, they can execute concurrently. The program generates this parallel execution using the Cilk `spawn` construct, which executes its argument function call in parallel with the rest of the computation in the procedure. The program then executes a `sync` instruction, which blocks the caller procedure until the parallel calls have finished. After the execution of several recursive levels, the subarray size becomes as small as `CUTOFF`, at which point the algorithm uses the base case procedure `baseInc` to sequentially increment each element of the subarray.

```
 1: #define CUTOFF 16
 2:
 3: void baseInc(int *q, int m) {
 4:   int i;
 5:   i = 0;
 6:   while(i <= m-1) {
 7:     *(q+i) += 1;
 8:     i = i+1;
 9:   }
10: }
11: void dcInc(int *p, int n) {
12:   if (n <= CUTOFF) {
13:     baseInc(p, n);
14:   } else {
15:     spawn dcInc(p, n/2);
16:     spawn dcInc(p+n/2, n-n/2);
17:     sync;
18:   }
19: }
20: void main(int argc, char *argv[]) {
21:   int size, *A;
22:   scanf("%d", &size);
23:   if (size > 0) {
24:     A = malloc(size * sizeof(int));
25:     /* code that initializes A */
26:     dcInc(A, size);
27:     /* code that uses A */
28:   }
29: }
```

Figure 1: Divide and Conquer Array Increment Example

This example reflects the structure of most of the Cilk programs discussed in Section 5 in that it identifies subproblems using pointers into dynamically allocated memory blocks. This strategy leads to code containing significant amounts of pointer arithmetic. Arguably better programming practice would use integer indices instead of pointers. Our pointer analysis algorithm and formulation of the symbolic analysis allows us to be neutral on this issue. Our algorithm can successfully analyze programs that identify subproblems using any combination of pointer arithmetic and array indices. Also note that the exclusive use of array indices instead of pointer arithmetic does not significantly simplify the analysis problem — the compiler must still reason about recursively generated accesses to regions of dynamically allocated memory blocks.

2.2 Required Analysis Information

The basic problem that our symbolic analysis must solve is to determine the regions of memory that each procedure accesses. The analysis represents regions of memory using two abstractions: *allocation blocks* and *symbolic regions*. There is an allocation block for each allocation site in the program, with the memory locations allocated at that site merged together to be represented by the site's allocation block. In our example, the allocation block a represents the array `A` allocated at line 24 in Figure 1.

Symbolic regions identify a contiguous set of memory locations within an allocation block. Each symbolic region has a lower bound and an upper bound; the bounds are symbolic polynomials with rational coefficients. In the analysis results for each procedure, the variables in each bound represent the initial values of the parameters of the procedure. In our example, the compiler determines that each call to `baseInc` reads and writes the symbolic region $[\mathtt{q}_0, \mathtt{q}_0 + \mathtt{m}_0 - 1]$ within the allocation block a and that each call to `dcInc` reads and writes the symbolic region $[\mathtt{p}_0, \mathtt{p}_0 + \mathtt{n}_0 - 1]$ within a.[1]

The compiler can use this information to detect data races and array bounds violations as follows. To check for data races, it compares the symbolic regions from parallel call sites to see if a region written by one call overlaps with a region accessed by a parallel call. If so, there is a potential data race. If not, there is no race. To compare the regions accessed by the two recursive calls to `dcInc`, for example, the compiler substitutes the actual parameters at the call site in for the corresponding formal parameters in the extracted symbolic regions. It computes that the first call reads and writes $[\mathtt{p}, \mathtt{p}+(\mathtt{n}/2)-1]$ in a and that the second call reads and writes $[\mathtt{p}+(\mathtt{n}/2), \mathtt{p}+\mathtt{n}-1]$ in a. The compiler compares the bounds of these regions to verify that neither call writes a region that overlaps with a region accessed by the other call, which implies that the program has no data races. Note that the rational coefficients in the bound polynomials allow the compiler to reason about the calculations that divide each array increment problem into two subproblems of equal size.

To detect array bounds violations, the compiler compares the sizes of the arrays against the expressions that tell which regions of the array are accessed by each procedure. In the example, the appropriate comparison is between the size of the array when it is allocated in the `main` procedure and the regions accessed by the top-level call to `dcInc` in the `main` procedure. The top-level call to `dcInc` reads and writes $[\mathtt{A}, \mathtt{A}+\mathtt{size}-1]$ in a. This symbolic region is contained within the dynamically allocated block of memory that holds the array, so the program contains no array bounds violations.

In the remainder of the paper, we present the algorithms that the compiler uses to analyze the program and solve the set of problems discussed in the introduction. We use the array increment program in Figure 1 as a running example to illustrate how our algorithms work.

3 Analysis Algorithm

The analysis has two goals: to compute an upper and lower bound for each pointer and array index variable at each program point and, for each procedure and each allocation block, to compute a set of symbolic regions that represent the memory locations that the entire computation of the procedure accesses. It computes the bounds as a polynomials with rational coefficients and variables that represent the initial values of the parameters of the enclosing procedure.

3.1 Structure of the Compiler

Figure 2 presents the general structure of the compiler, which consists of the following analysis phases:

- **Pointer and Read-Write Sets Analysis**: The compiler first runs an interprocedural, context-sensitive, flow-sensitive *pointer analysis* that analyzes both sequential and parallel programs [25]. It then performs an interprocedural *read-write sets analysis*, which uses the extracted information to compute the allocation blocks accessed by each instruction and each procedure in the program. The remaining phases rely on this phase to disambiguate references via pointers.

[1] Here we use the notation $[l, h]$ to denote the region of memory between the addresses l and h, inclusive. As is standard in C, we assume contiguous allocation of arrays, and that the addresses of the elements increase as the array indices increase. We also use the notation p_0 to denote the initial value of the parameter p.

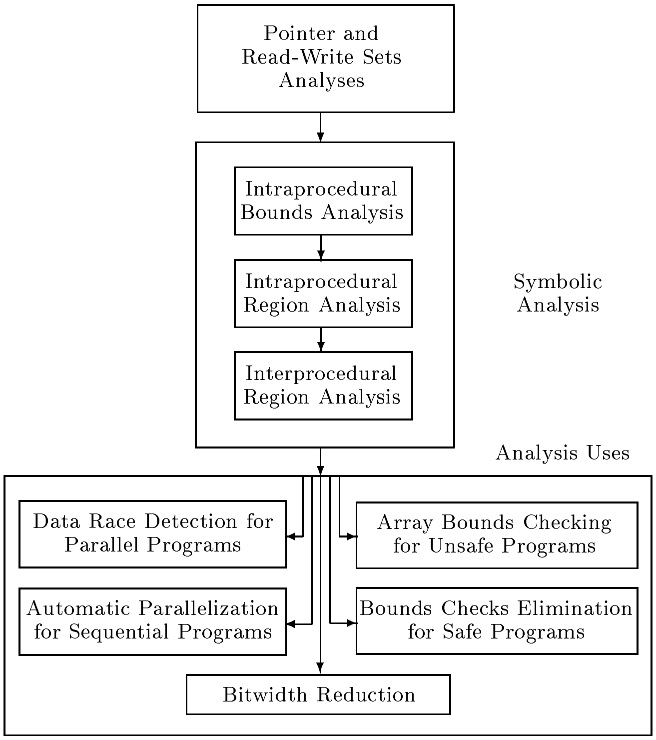

Figure 2: Structure of the Compiler

- **Symbolic Analysis**: This phase produces sets of symbolic regions that characterize how each procedure accesses memory. It first extracts symbolic bounds for each pointer and array index variable, then uses this information to compute symbolic bounds for the accessed regions within each allocation block.
- **Uses of the Analysis Results**: This phase uses the symbolic memory access information computed by the earlier stages to solve the problems discussed above in the introduction.

The symbolic analysis consists of the following subphases:

- **Intraprocedural Bounds Analysis**: This phase derives symbolic bounds for each pointer and array index variable at each program point.
- **Intraprocedural Region Analysis**: For each allocation block, this phase computes a set of symbolic regions that characterizes how the procedure directly reads or writes the allocation block.
- **Interprocedural Region Analysis**: For each allocation block, this phase computes a set of symbolic regions that characterizes how the entire computation of the procedure reads or writes the allocation block.

Both the bounds analysis and the interprocedural region analysis use a general symbolic analysis framework for building and solving systems of symbolic inequality constraints between polynomials. Recursive constraints may be generated by loops in the control flow (in the case of the bounds analysis), or by recursive calls (in the case of the region analysis). By solving arbitrary systems of recursive constraints, the compiler is able to handle arbitrary flow of control at both the intraprocedural and interprocedural level.

3.2 Basic Concepts

The analysis uses the following mathematical objects to represent the symbolic bounds and accessed memory regions:

- **Allocation Blocks:** There is an allocation block a for each static or dynamic allocation site in the program, with the variable declaration sites considered to be the static allocation sites. All of the elements of each array are merged together to be represented by the allocation block from the array's allocation site. For programs with structures, each field of each structure has its own allocation block.
- **Program Variables:** V_f is the set of pointer and array index variables from the procedure f. In our example, $V_{\texttt{baseInc}} = \{\texttt{q}, \texttt{m}, \texttt{i}\}$. v_p denotes the value of the variable v at the program point p; v_0 is the initial value of a parameter v of a given procedure.
- **Reference Sets:** C_f is the set of initial values of the parameters of the procedure f. C_f is called the *reference set* of f. In our example, $C_{\texttt{baseInc}} = \{\texttt{q}_0, \texttt{m}_0\}$.
- **Polynomials:** P_S^* is the set of multivariate polynomials with rational coefficients and variables in S. Also, $P_S = P_S^* \cup \{+\infty, -\infty\}$. P_{C_f} is the *analysis domain* for the procedure f; all symbolic analysis results for f are computed as elements of C_f.

  Note that even though the polynomials represent integer values, they have rational coefficients, not integer coefficients. Rational coefficients enable the compiler to reason about address computations that contain division operators. These kinds of address computations are common in our target class of divide and conquer computations, which use them to divide a problem into several subproblems of equal size.
- **Symbolic Bounds:** For each variable v and program point p, the analysis computes a symbolic lower bound $l_{v,p}$ and upper bound $u_{v,p}$ for the value of v at p. The analysis computes these bounds as symbolic polynomials with rational coefficients and variables from the reference set of the enclosing procedure.

  In our example, the analysis computes $l_{\texttt{i},p} = \texttt{0}$ and $u_{\texttt{i},p} = \texttt{m}_0 - 1$, where p is the program point before line 7 in Figure 1.
- **Symbolic Regions:** A symbolic region R in the domain of a procedure f is a pair of symbolic bounds from the analysis domain of f: $R \in P_{C_f} \times P_{C_f}$, $R = [l, u]$, with $l, u \in P_{C_f}$; l is the lower bound and u is the upper bound. Each symbolic region represents a contiguous set of memory locations within an accessed allocation block.
- **Symbolic Region Sets:** A symbolic region set RS in the domain of a procedure f is a set of symbolic regions from f: $RS \subseteq P_{C_f} \times P_{C_f}$. For each procedure f and allocation block a, the analysis computes two symbolic region sets to represent the locations that the entire computation of f accesses: $RW_{f,a}$, which represents the locations that f writes in a, and $RR_{f,a}$, which represents the locations that f reads in a. In our example the analysis computes:

$$\begin{array}{lll} RW_{\texttt{baseInc},a} & = RR_{\texttt{baseInc},a} & = \{[\texttt{q}_0, \texttt{q}_0 + \texttt{m}_0 - 1]\} \\ RW_{\texttt{dcInc},a} & = RR_{\texttt{dcInc},a} & = \{[\texttt{p}_0, \texttt{p}_0 + \texttt{n}_0 - 1]\} \end{array}$$

  where a is the allocation block for the array allocated at line 24 in Figure 1.

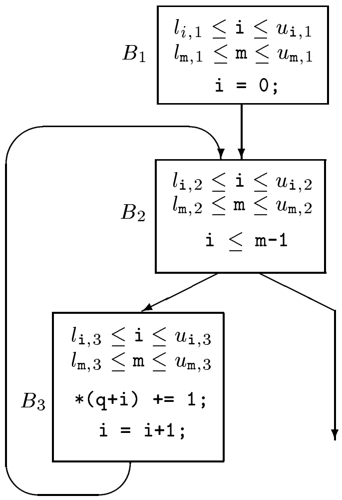

Figure 3: Symbolic Bounds at the Start of Basic Blocks

3.3 Intraprocedural Bounds Analysis

In this phase, the compiler computes symbolic lower and upper bounds for each pointer and array index at each program point. The bounds are expressed as polynomials with rational coefficients. The variables in the polynomials represent the initial values of the formal parameters of the enclosing procedure. We illustrate the operation of this phase by showing how it analyzes the procedure `baseInc` from Figure 1.

3.3.1 Initial Symbolic Bounds

Let $B = \{B_j | 1 \le j \le l\}$ be the set of basic blocks in the control-flow graph of the procedure f. For each variable $v \in V_f$ and basic block B_j, the compiler generates a symbolic lower bound $l_{v,j}$ and a symbolic upper bound $u_{v,j}$ for the value of v at the start of B_j. Figure 3 presents the control-flow graph and initial symbolic bounds for the procedure `baseInc` from our example.

3.3.2 Symbolic Analysis of Basic Blocks

The compiler next symbolically executes the instructions in each basic block to produce new symbolic bounds for each variable at the end of the block and at all intermediate program points within the block. These bounds are expressed as linear combinations of the symbolic bounds from the start of the block. Figure 4 presents the results of this step in our example.[2] We next explain how the compiler extracts these bounds.

During the analysis of individual instructions, the compiler must be able to compute bounds of expressions. The analysis computes the lower bound $L(e,p)$ and upper bound $U(e,p)$ of an expression e at a program point p as follows. If e contains at least one variable with infinite bounds, then $L(e,p) = -\infty$ and $U(e,p) = +\infty$. Otherwise, the following equations define the bounds. Note that in these expressions, $+$, $-$, and $\cdot$ operate on polynomials. Each expression is a linear combination of the symbolic bounds $l_{v,p}$ and $u_{v,p}$.

[2] Our compiler decouples the analysis of `i` and `m` from the analysis of `q` (see Section 4.3). We therefore present the analysis only for `i` and `m`.

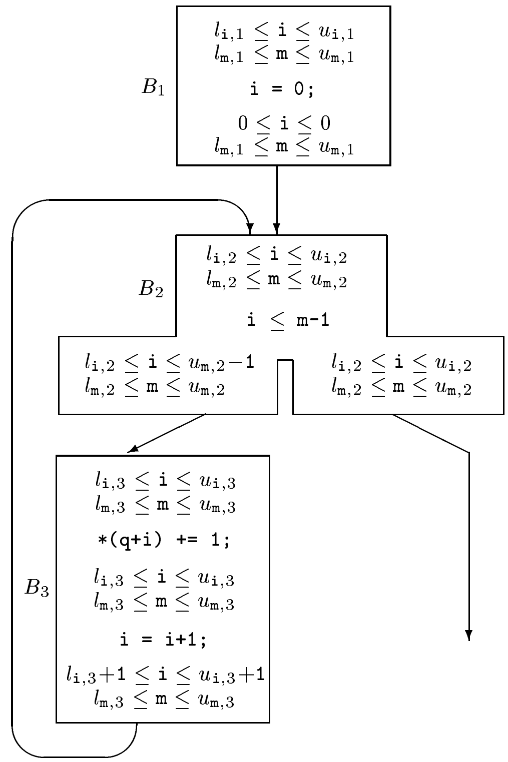

Figure 4: Symbolic Bounds at the End of Basic Blocks

$$
\begin{aligned}
L(c,p) &= c \\
L(v,p) &= v_{l,p} \\
L(e_1 + e_2, p) &= L(e_1,p) + L(e_2,p) \\
L(c \cdot e, p) &= \begin{cases} c \cdot L(e,p) & \text{if } c > 0 \\ c \cdot U(e,p) & \text{if } c \le 0 \end{cases} \\
U(c,p) &= c \\
U(v,p) &= v_{u,p} \\
U(e_1 + e_2, p) &= U(e_1,p) + U(e_2,p) \\
U(c \cdot e, p) &= \begin{cases} c \cdot U(e,p) & \text{if } c > 0 \\ c \cdot L(e,p) & \text{if } c \le 0 \end{cases}
\end{aligned}
$$

For an assignment instruction i of the form $v = e$, where $v \in V_f$ and e is a linear expression in the program variables, the analysis updates the bounds of v to be the bounds of e. Formally, if p is the program point before i and p' is the program point after i, then:

$$
\begin{aligned}
l_{v,p'} &= L(e,p) \\
u_{v,p'} &= U(e,p)
\end{aligned}
$$

For a conditional instruction i of the form $v \le e$ or $v \ge e$, where $v \in V_f$ and e is a linear expression in the program variables, the analysis generates a new upper or lower bound for v on the true branch of the conditional. If the conditional is of the form $v \ge e$, the new lower bound of v is the lower bound of e. If the conditional is of the form $v \le e$, the new upper bound of v is the upper bound of e. Formally, if p is the program point before the conditional and t is the program point on the true branch of the conditional, then:

$$
\begin{aligned}
l_{v,t} &= L(e,p) \text{ if } i \text{ is of the form } v \ge e \\
u_{v,t} &= U(e,p) \text{ if } i \text{ is of the form } v \le e
\end{aligned}
$$

Initialization Conditions:

$$
\begin{array}{ll}
l_{\mathtt{i},1} = -\infty & u_{\mathtt{i},1} = +\infty \\
l_{\mathtt{m},1} = \mathtt{m}_0 & u_{\mathtt{m},1} = \mathtt{m}_0
\end{array}
$$

Symbolic Constraints:

$$
\begin{array}{lr}
l_{\mathtt{i},2} \leq 0 & 0 \leq u_{\mathtt{i},2} \\
l_{\mathtt{i},2} \leq l_{\mathtt{i},3} + 1 & u_{\mathtt{i},3} + 1 \leq u_{\mathtt{i},2} \\
l_{\mathtt{m},2} \leq l_{\mathtt{m},1} & u_{\mathtt{m},1} \leq u_{\mathtt{m},2} \\
l_{\mathtt{m},2} \leq l_{\mathtt{m},3} & u_{\mathtt{m},3} \leq u_{\mathtt{m},2} \\
l_{\mathtt{i},3} \leq l_{\mathtt{i},2} & u_{\mathtt{m},2} - 1 \leq u_{\mathtt{i},3} \\
l_{\mathtt{m},3} \leq l_{\mathtt{m},2} & u_{\mathtt{m},2} \leq u_{\mathtt{m},3}
\end{array}
$$

Objective Function:

$$
\begin{array}{ll}
\min: & (u_{\mathtt{i},2} - l_{\mathtt{i},2}) + (u_{\mathtt{m},2} - l_{\mathtt{m},2}) + \\
& (u_{\mathtt{i},3} - l_{\mathtt{i},3}) + (u_{\mathtt{m},3} - l_{\mathtt{m},3})
\end{array}
$$

Figure 5: Symbolic Constraint System for Bounds Analysis

All other bounds remain the same as the corresponding bounds from before the conditional.

For all other instructions, such as assignments of expressions that are not linear in the program variables (but see Section 4.3 for an extension that enables the analysis to support polynomial expressions in certain cases), call instructions, or more complicated conditionals, the analysis generates conservative bounds. All of the variables that the analyzed instruction writes have infinite bounds, and all of the other variables have unchanged bounds. The pointer and read-write sets analyses compute the set of written variables.

3.3.3 Constraint Generation

The algorithm next builds a symbolic constraint system over the lower and upper bounds. The system consists of a set of initialization conditions, a set of symbolic constraints, and an objective function to minimize:

- The *initialization conditions* require that at the start of the entry basic block B_1, the bounds of each pointer or array index parameter $v \in V_f$ must be equal to v_0 (the value of that variable at the beginning of the procedure). For all other variables, the lower bounds are set to $-\infty$ and the upper bounds to $+\infty$.

- The *symbolic constraints* require that the range of each variable at the beginning of each basic block must include the range of that variable *at the end* of the predecessor basic blocks. Formally, if B_j is a predecessor of B_k, $l'_{v,j}$ and $u'_{v,j}$ are the bounds of v at the end of B_j, and $l_{v,k}$ and $u_{v,k}$ are the bounds of v at the beginning of B_k, then $l_{v,k} \leq l'_{v,j}$ and $u'_{v,j} \leq u_{v,k}$.

- The *objective function* minimizes the upper bounds and maximizes the lower bounds. Therefore the objective function is: $\min: \sum_{v \in V_f} \sum_{j=2}^{l} (u_{v,j} - l_{v,j})$.

The initialization conditions and symbolic constraints ensure the safety of the computed bounds. The objective function ensures a tight solution that minimizes the symbolic ranges of the variables. Figure 5 presents the constraint system in our example.

The analysis next extracts the bounds in terms of the reference set (the initial values of the parameters). The algorithm does not know what the bounds are, but it proceeds under the assumption that they are polynomials with variables in the reference set. It therefore expresses the bounds as symbolic polynomials. Each term of the polynomial has a rational coefficient variable c_j. The goal of the analysis is to find a precise numerical value for each coefficient variable c_j. In our example, the bounds are expressed using coefficient variables and the variables from the reference set $\{\mathtt{q}_0, \mathtt{m}_0\}$:

$$
\begin{array}{ll}
l_{\mathtt{i},2} = c_1\mathtt{q}_0 + c_2\mathtt{m}_0 + c_3 & u_{\mathtt{i},2} = c_{13}\mathtt{q}_0 + c_{14}\mathtt{m}_0 + c_{15} \\
l_{\mathtt{m},2} = c_4\mathtt{q}_0 + c_5\mathtt{m}_0 + c_6 & u_{\mathtt{m},2} = c_{16}\mathtt{q}_0 + c_{17}\mathtt{m}_0 + c_{18} \\
l_{\mathtt{i},3} = c_7\mathtt{q}_0 + c_8\mathtt{m}_0 + c_9 & u_{\mathtt{i},3} = c_{19}\mathtt{q}_0 + c_{20}\mathtt{m}_0 + c_{21} \\
l_{\mathtt{m},3} = c_{10}\mathtt{q}_0 + c_{11}\mathtt{m}_0 + c_{12} & u_{\mathtt{m},3} = c_{22}\mathtt{q}_0 + c_{23}\mathtt{m}_0 + c_{24}
\end{array}
$$

The initialization conditions define the bounds at the beginning of the starting basic block B_1:

$$l_{\mathtt{m},1} = \mathtt{m}_0 \quad u_{\mathtt{m},1} = \mathtt{m}_0 \quad l_{\mathtt{i},1} = -\infty \quad u_{\mathtt{i},1} = +\infty$$

3.3.4 Solving the Symbolic Constraint System

This step solves the symbolic constraint system by deriving a rational numeric value for each coefficient variable. We first summarize the starting point for the algorithm.

- The algorithm is given a set of symbolic lower and upper bounds. These bounds are expressed as a set of symbolic bound polynomials $P_i \in P_C$, where C is the reference set of the currently analyzed procedure. Each term in each symbolic bound polynomial consists of a coefficient variable and a product of reference set variables: $P_i = \sum_{i=0}^{t} c_i \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$.

- The algorithm is also given a set of inequality constraints between polynomial expressions and an objective function to minimize. Formally, the symbolic constraint system can be expressed as a pair (I, O), where $I \subseteq \{\ Q \leq R \mid Q, R \in P_C\ \}$ is the set of symbolic constraints and $O \in P_C$ is the objective function. The polynomial expressions Q, R, and O are linear combinations of the symbolic bound polynomials. The analysis described in Sections 3.3.2 and 3.3.3 produces these expressions.

The algorithm solves the constraint system by reducing it to a linear program over the coefficient variables from the symbolic bound polynomials. It generates the linear program by reducing each symbolic inequality constraint to several linear inequality constraints over the coefficient variables of the symbolic bound polynomials. Formally, if $Q = \sum_{i=0}^{t} c_i^Q \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$ and $R = \sum_{i=0}^{t} c_i^R \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$, then:

$$(Q \leq R) \in I \quad \text{is reduced to:} \quad c_i^Q \leq c_i^R, \quad \text{for all } 0 \leq i \leq t$$

Because the polynomial expressions are linear combinations of the symbolic bound polynomials, the coefficients c_i^R and c_i^Q are linear combinations of the coefficient variables from the symbolic bound polynomials.

The algorithm also reduces the symbolic objective function to a linear objective function in the coefficient variables. This reduction minimizes the sum of the coefficients in the polynomial expression. Formally, if the objective function is $O = \sum_{i=0}^{t} c_i^O \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$, then:

$$\min: O \quad \text{is reduced to:} \quad \min: \sum_{i=0}^{t} c_i^O$$

Symbolic Constraints		*Generated Linear Constraints*		
$l_{\mathtt{i},2} \leq 0$	:	$c_1 \leq 0$	$c_2 \leq 0$	$c_3 \leq 0$
$l_{\mathtt{i},2} \leq l_{\mathtt{i},3} + 1$	:	$c_1 \leq c_7$	$c_2 \leq c_8$	$c_3 \leq c_9 + 1$
$l_{\mathtt{m},2} \leq \mathtt{m}_0$	:	$c_4 \leq 0$	$c_5 \leq 1$	$c_6 \leq 0$
$l_{\mathtt{m},2} \leq l_{\mathtt{m},3}$	:	$c_4 \leq c_{10}$	$c_5 \leq c_{11}$	$c_6 \leq c_{12}$
$l_{\mathtt{i},3} \leq l_{\mathtt{i},2}$	:	$c_7 \leq c_1$	$c_8 \leq c_2$	$c_9 \leq c_3$
$l_{\mathtt{m},3} \leq l_{\mathtt{m},2}$	:	$c_{10} \leq c_4$	$c_{11} \leq c_5$	$c_{12} \leq c_6$
$u_{\mathtt{i},2} \geq 0$	:	$c_{13} \geq 0$	$c_{14} \geq 0$	$c_{15} \geq 0$
$u_{\mathtt{i},2} \geq u_{\mathtt{i},3} + 1$	:	$c_{13} \geq c_{19}$	$c_{14} \geq c_{20}$	$c_{15} \geq c_{21} + 1$
$u_{\mathtt{m},2} \geq \mathtt{m}_0$	:	$c_{16} \geq 0$	$c_{17} \geq 1$	$c_{18} \geq 0$
$u_{\mathtt{m},2} \geq u_{\mathtt{m},3}$	:	$c_{16} \geq c_{22}$	$c_{17} \geq c_{23}$	$c_{18} \geq c_{24}$
$u_{\mathtt{i},3} \geq u_{\mathtt{m},2} - 1$	:	$c_{19} \geq c_{16}$	$c_{20} \geq c_{17}$	$c_{21} \geq c_{18} - 1$
$u_{\mathtt{m},3} \geq u_{\mathtt{m},2}$	:	$c_{22} \geq c_{16}$	$c_{23} \geq c_{17}$	$c_{24} \geq c_{18}$

Objective Function:

$$\begin{aligned} \min: \quad & ((c_{13} + c_{14} + c_{15}) - (c_1 + c_2 + c_3)) + \\ & ((c_{16} + c_{17} + c_{18}) - (c_4 + c_5 + c_6)) + \\ & ((c_{19} + c_{20} + c_{21}) - (c_7 + c_8 + c_9)) + \\ & ((c_{22} + c_{23} + c_{24}) - (c_{10} + c_{11} + c_{12})) \end{aligned}$$

Figure 6: Linear Program for Bounds Analysis

At this point the analysis has generated a linear program. The solution to this linear program directly gives the solution to the symbolic constraint system.

We emphasize that the symbolic constraint system is reduced to a **linear program**, not to an integer linear program. The coefficient variables in the linear program are rational numbers, not integer numbers.

Figure 6 shows the generated linear program in our example. It presents the symbolic constraints on the left hand side and the generated linear constraints on the right hand side. For example, the constraint $l_{\mathtt{i},2} \leq l_{\mathtt{i},3} + 1$ means that $(c_1\mathtt{q}_0 + c_2\mathtt{m}_0 + c_3) \leq (c_7\mathtt{q}_0 + c_8\mathtt{m}_0 + c_9) + 1$, which in turn generates the following constraints: $c_1 \leq c_7$, $c_2 \leq c_8$ and $c_3 \leq c_9 + 1$. Solving the linear program yields the following values of the coefficient variables:

$c_1 = 0$	$c_2 = 0$	$c_3 = 0$	$c_{13} = 0$	$c_{14} = 1$	$c_{15} = 0$
$c_4 = 0$	$c_5 = 1$	$c_6 = 0$	$c_{16} = 0$	$c_{17} = 1$	$c_{18} = 0$
$c_7 = 0$	$c_8 = 0$	$c_9 = 0$	$c_{19} = 0$	$c_{20} = 1$	$c_{21} = -1$
$c_{10} = 0$	$c_{11} = 1$	$c_{12} = 0$	$c_{22} = 0$	$c_{23} = 1$	$c_{24} = 0$

This gives the following polynomials for the lower and upper bounds:

$$\begin{array}{llll} l_{\mathtt{i},2} = 0 & u_{\mathtt{i},2} = \mathtt{m}_0 & l_{\mathtt{m},2} = \mathtt{m}_0 & u_{\mathtt{m},2} = \mathtt{m}_0 \\ l_{\mathtt{i},3} = 0 & u_{\mathtt{i},3} = \mathtt{m}_0 - 1 & l_{\mathtt{m},3} = \mathtt{m}_0 & u_{\mathtt{m},3} = \mathtt{m}_0 \end{array}$$

Finally, these bounds are used to compute the symbolic bounds of the variables at each program point, giving the final result shown in Figure 7. Note that the analysis detects that the symbolic range of the index variable `i` before the store instruction `*(q+i) += 1` is $[0, \mathtt{m}_0 - 1]$. In a similar manner, the bounds analysis is able to determine that the range of the pointer `q` at this program point is $[\mathtt{q}_0, \mathtt{q}_0]$, which means that $\mathtt{q} = \mathtt{q}_0$ before the store instruction.

3.3.5 Positivity Analysis

Both of the transformations that reduce the symbolic constraint system to a linear program assume that the variables

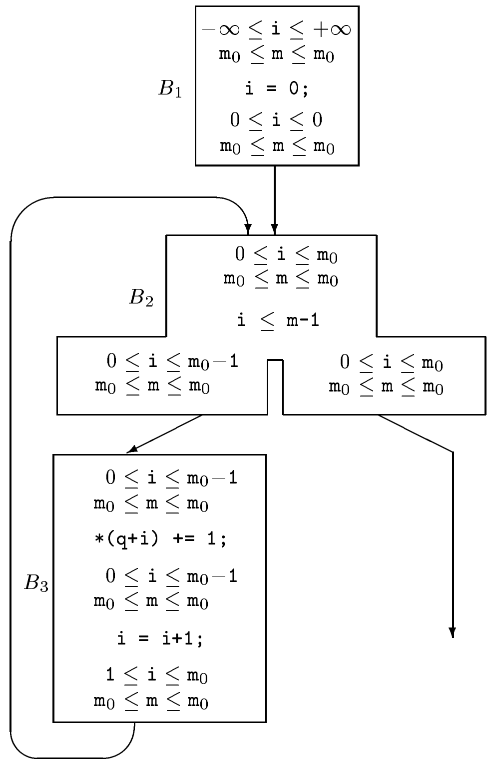

Figure 7: Results of the Bounds Analysis

in the reference set C_f are *positive*. We can apply similar transformations if we know that a variable is negative. But if we do not know the sign of a variable in C_f, we cannot reduce the symbolic system to a linear program. The algorithm therefore performs a simple interprocedural *positivity analysis* to compute the sign of the array index variables in the reference set. It does not check pointer variables, since they always represent positive addresses.

3.4 Region Analysis

For each procedure f and allocation block a, the region analysis computes a symbolic region set that represents the regions of a that f reads or writes. An intraprocedural analysis first builds the regions that each procedure accesses directly. An interprocedural analysis then uses these symbolic regions to build symbolic constraint systems that specify the regions accessed by the complete computation of each procedure. The algorithm solves each constraint system by reducing it to a linear program. This approach solves the hard problem of computing symbolic access regions for recursive procedures.

3.4.1 Region Coalescing

At certain points in the analysis, the algorithm must coalesce overlapping regions. Figure 8 presents the region coalescing algorithm. It first tries to coalesce the new region with some other overlapping region in the region set, in which case the bounds of the overlapping region are adjusted to accommodate the new region. If no overlapping region is found, the algorithm adds the new region to the region set.

Algorithm *Coalesce*(Region R, RegionSet RS)
  let $R = [l, u]$
  if $(\exists [l', u'] \in RS,\ l \le l' \le u' \le u)$ **then**
    return $(RS - \{[l', u']\}) \cup \{[l, u]\}$
  else if $(\exists [l', u'] \in RS,\ l \le l' \le u \le u')$ **then**
    return $(RS - \{[l', u']\}) \cup \{[l, u']\}$
  else if $(\exists [l', u'] \in RS,\ l' \le l \le u' \le u)$ **then**
    return $(RS - \{[l', u']\}) \cup \{[l', u]\}$
  else if $(\exists [l', u'] \in RS,\ l' \le l \le u \le u')$ **then**
    return RS
  else return $RS \cup \{[l, u]\}$

Figure 8: Region Coalescing Algorithm

Algorithm *IntraproceduralRegionAnalysis()*
  for (each procedure f and each allocation block a) **do**
    $RW^{local}_{f,a} = RR^{local}_{f,a} = \emptyset$
  for (each allocation block access in the program) **do**
    let f = the current procedure;
    let p = the current program point;
    let a = the accessed allocation block;
    let e = the address expression of the access;
    $R_{new} = [L(e,p), U(e,p)]$
    if (write access)
    then $RW^{local}_{f,a} = Coalesce(R_{new}, RW^{local}_{f,a})$
    else $RR^{local}_{f,a} = Coalesce(R_{new}, RR^{local}_{f,a})$

Figure 9: Intraprocedural Region Analysis Algorithm

3.4.2 Intraprocedural Region Analysis

Figure 9 presents the pseudo-code for the intraprocedural region analysis. The algorithm first initializes the read region sets $RR^{local}_{f,a}$ and write region sets $RW^{local}_{f,a}$. These sets characterize the regions of memory directly accessed by f. It then scans the instructions to extract the region expressions, using the bounds analysis results to build the region expression for each instruction. It also coalesces overlapping region expressions from different instructions.

If a is the memory block dynamically allocated in the `main` program in the example, the result of the intraprocedural analysis for `baseInc` and `dcInc` is:

$$RW^{local}_{\texttt{baseInc},a} = \{[\mathtt{q}_0, \mathtt{q}_0 + \mathtt{m}_0 - 1]\} \qquad RR^{local}_{\texttt{dcInc},a} = \emptyset$$
$$RR^{local}_{\texttt{baseInc},a} = \{[\mathtt{q}_0, \mathtt{q}_0 + \mathtt{m}_0 - 1]\} \qquad RW^{local}_{\texttt{dcInc},a} = \emptyset$$

The region analysis uses the pointer analysis information to determine that the store instruction in `baseInc` accesses the allocation block a.

3.4.3 Interprocedural Region Analysis

The interprocedural region analysis uses the results of the intraprocedural region analysis to compute a symbolic region set for the entire computation of each procedure, including all of the procedures that it invokes. It first builds the call graph of the computation and identifies the strongly

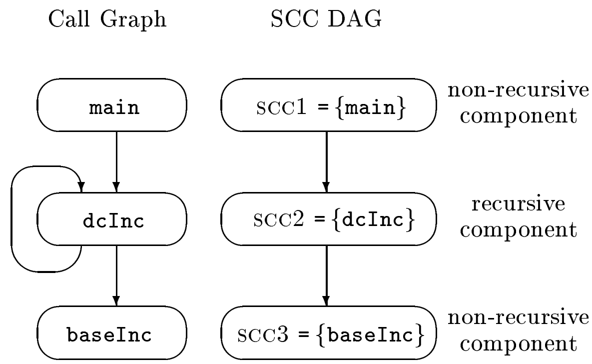

Figure 10: Call Graph and SCC DAG in Example

connected components. It then traverses the strongly connected components in reverse topological order, propagating the access region information between strongly connected components from callee to caller. Within each strongly connected component with recursive calls, it generates a symbolic constraint system and solves it using the algorithm from Section 3.3.4. Figure 10 shows the call graph and its strongly connected components for our example.

3.4.4 Symbolic Unmapping

At each call site, the analysis models the assignments of actual parameters to formal parameters, then uses this model to propagate access region information from the callee to the caller. The analysis of the callee produces a result in terms of the initial values of the callee's parameters. But the result for the caller must be expressed in terms of the caller parameters, not the callee parameters. The *symbolic unmapping* algorithm performs this change of analysis domain for each accessed region R from the callee.

- The algorithm first transforms the region R from the callee domain to a new region R' by replacing the formal parameters from the callee with the actual parameters from the call site. The new region $[l, u] = R'$ expresses the bounds in terms of the variables of the caller.

- The algorithm next uses the results of the intraprocedural bounds analysis presented in Section 3.3 to compute a lower bound for l and an upper bound for u in terms of the reference set of the caller. These two new bounds are the symbolic lower and upper bounds for the unmapped region $SU_{cs}(R)$, the translation of the region R from the callee domain to the caller domain at the call site cs.

Let CallSites(f, g) be the set of all call sites with caller f and callee g. We formalize the symbolic unmapping as follows:

- **Mapping:** For two sets of variables S and T, a *mapping* M from S to T is either a function $M \in S \to P^*_T$ (a function from S to symbolic polynomials in T), or a special mapping M_{unk}, called the *unknown mapping*.

- **Call Site Mapping:** For a call site $cs \in$ CallSites(f, g), we define a *call site mapping* $M_{cs} \in (C_g \to P^*_{V_f}) \cup \{M_{unk}\}$ as follows:

– if the actual parameters can be expressed as polynomials $p_1, \ldots, p_m \in P^*_{V_f}$ then $M_{cs}(v_i) = p_i$, where $v_1, \ldots, v_m$ are the formal parameters of g.

– otherwise, $M_{cs} = M_{unk}$.

- **Symbolic Unmapping of a Polynomial:** Given a polynomial $P \in P_S$ and a mapping $M \in S \to P^*_T$, we define the *symbolic unmapping* $SU_M(P) \in P^*_T$ *of the polynomial* P *using* M as follows:

$$P = \sum c_i \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$$
$$SU_M(P) = \sum c_i \cdot M(x_1)^{r_{i,1}} \cdots M(x_s)^{r_{i,s}}$$

- **Symbolic Unmapping of a Region at a Call Site:** Given a region $R = [l, u] \in P_{C_f} \times P_{C_f}$ and a call site $cs \in \text{CallSites}(f, g)$ with call site mapping $M \neq M_{unk}$, we define the *symbolic unmapping* $SU_{cs}(R) \in P_{C_f} \times P_{C_f}$ *of the region* R *at call site* cs as follows:

$$SU_{cs}(R) = [L(SU_M(l), cs), U(SU_M(u), cs)]$$

If $M = M_{unk}$, then $SU_{cs}(R) = [-\infty, +\infty]$.

- **Symbolic Unmapping of a Region Set at a Call Site:** Given a region set RS and a call site cs, we define the *symbolic unmapping* $SU_{cs}(RS)$ *of the region set* RS *at call site* cs as follows:

$$SU_{cs}(RS) = \{SU_{cs}(R) \mid R \in RS\}$$

- **Symbolic Unmapping of Region Sets for Allocation Blocks:** Given an accessed allocation block a and a call site $cs \in \text{CallSites}(f, g)$, the accessed region sets $RR^{cs}_{f,a} = SU_{cs}(RR_{g,a})$ and $RW^{cs}_{f,a} = SU_{cs}(RW_{g,a})$ describe the regions of a accessed by g at call site cs, in terms of the reference set of f.

Given these definitions, the interprocedural analysis for non-recursive procedures is straightforward. The algorithm simply traverses the call graph in reverse topological order, using the unmapping algorithm to propagate region sets from callees to callers.

Figure 11 shows the symbolic unmapping process at call site $cs1$, where `dcInc` invokes `baseInc` in our example. The compiler starts with the region expression $RW_{\mathtt{baseInc},a} = [\mathtt{q}_0, \mathtt{q}_0 + \mathtt{m}_0 - 1]$ computed by the intraprocedural region analysis. Here a is the accessed allocation block from `baseInc`. The compiler creates a call site mapping M that maps the formal parameters $\mathtt{q}_0$ and $\mathtt{m}_0$ to the symbolic expressions representing the corresponding actual parameters at the call site: $M(\mathtt{q}_0) = \mathtt{p}$ and $M(\mathtt{m}_0) = \mathtt{n}$. The analysis uses this mapping to translate $RW_{\mathtt{baseInc},a}$ into the new region $R' = [\mathtt{p}, \mathtt{p} + \mathtt{n} - 1]$. Finally, the compiler uses the bounds of `p` and `n` at the call site to derive the unmapped region:

$$\begin{aligned} SU_{cs1}(RW_{\mathtt{baseInc},a}) &= [L(\mathtt{p}, cs1), U(\mathtt{p} + \mathtt{n} - 1, cs1)] \\ &= [\mathtt{p}_0, \mathtt{p}_0 + \mathtt{n}_0 - 1] \end{aligned}$$

The unmapped region $SU_{cs1}(RW_{\mathtt{baseInc},a}) = [\mathtt{p}_0, \mathtt{p}_0 + \mathtt{n}_0 - 1]$ characterizes the regions in a accessed by the call instruction `baseAdd(p,n)` in terms of the initial values of the parameters of `dcAdd`, $\mathtt{p}_0$ and $\mathtt{n}_0$.

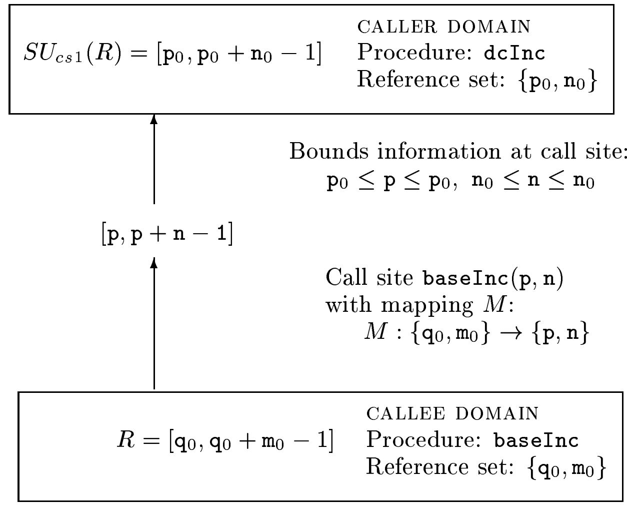

Figure 11: Symbolic Unmapping Example

3.4.5 Analysis of Recursive Procedures

One way to attack the analysis of recursive procedures is to use a fixed-point algorithm to propagate region sets through cycles in the call graph. But this approach fails because the domain of multivariate polynomials has infinite ascending chains, which means that the algorithm may never reach a fixed point. First, the bounds in some divide and conquer programs form a convergent geometric series. There is no finite number of iterations that would find the limit of such a series. Second, recursive programs can generate a statically unbounded number of regions in the region set.

Our algorithm avoids these problems by generating a system of recursive symbolic constraints whose solution delivers a region set specifying the regions of memory accessed by the entire strongly connected component. The symbolic constraint system is solved by using the algorithm presented in Section 3.3.4 to reduce the symbolic constraint system to a linear program. The main idea is to generate a set of constraints that, at each call site, requires the caller region sets to include the unmapped region sets of the callee. We next discuss how the compiler computes the region sets for a set S of mutually recursive procedures.

Step 1. *Define the target symbolic bounds.* The compiler first defines, for each recursive procedure $f \in S$ and allocation block a, a finite set of read regions and a finite set of write regions. An analysis of the base cases of the recursion determines the number of regions in each set.

$$\begin{aligned} RR_{f,a} &= \{[l^{rd}_{f,a,1}, u^{rd}_{f,a,1}], \ldots, [l^{rd}_{f,a,j}, u^{rd}_{f,a,j}]\} \\ RW_{f,a} &= \{[l^{wr}_{f,a,1}, u^{wr}_{f,a,1}], \ldots, [l^{wr}_{f,a,k}, u^{wr}_{f,a,k}]\} \end{aligned}$$

The bounds of these regions are the target bounds in our analysis framework. For each procedure $f \in S$, these bounds are polynomial expressions in P_{C_f}. To guarantee the soundness of the unmapping, the constraint system requires the coefficients of the variables in these bounds to be positive.

Consider, for example, the computation of the region sets for the strongly connected component $S = \{\mathtt{dcInc}\}$ from our example.[3] Since the base case for this procedure writes

[3] The compiler decouples the computation of write region sets from the computation of read region sets (see Section 4.3). We therefore present the analysis only for the write set of `dcInc`.

a single region within the allocation block a, the compiler generates a single write region for `dcInc`:

$$RW_{\texttt{dcInc},a} = \{[l^{wr}_{\texttt{dcInc},a}, u^{wr}_{\texttt{dcInc},a}]\}$$

The bounds of this region are polynomials with variables in $C_{\texttt{dcInc}} = \{\mathtt{p}_0, \mathtt{n}_0\}$. The algorithm uses the following bounds:

$$\begin{aligned} l^{wr}_{\texttt{dcInc},a} &= C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3 \\ u^{wr}_{\texttt{dcInc},a} &= C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6 \end{aligned}$$

where C_1, C_2, C_4, and C_5 are positive rational coefficients.

Step 2. *Generate the symbolic system of constraints.* The analysis next generates the constraint system for the region bounds defined in the previous step. The system must ensure that two conditions are satisfied. First, the local region sets must be included in the global region sets:

$$\begin{aligned} RR^{local}_{f,a} &\subseteq RR_{f,a} \quad \forall f \in S \\ RW^{local}_{f,a} &\subseteq RW_{f,a} \quad \forall f \in S \end{aligned}$$

Second, for each call site, the unmapped region sets of the callee must be included in the region sets of the caller:

$$\begin{aligned} SU_{cs}(RR_{g,a}) &\subseteq RR_{f,a} \quad \forall f \in S,\ \forall cs \in \text{CallSites}(f,g) \\ SU_{cs}(RW_{g,a}) &\subseteq RW_{f,a} \quad \forall f \in S,\ \forall cs \in \text{CallSites}(f,g) \end{aligned}$$

Figure 12 summarizes the constraints in the generated system. As in the intraprocedural case, the objective function minimizes the sizes of the symbolic regions.

Figure 13 presents the system of symbolic constraints for the interprocedural analysis of `dcInc`. Because `dcInc` does not directly access any allocation block, there are no local constraints. The analysis generates call site constraints for the call sites $cs1$, $cs2$, and $cs3$, at lines `13`, `15`, and `16` in Figure 1 respectively. At call site $cs1$, which invokes `baseInc`, the analysis symbolically unmaps the callee region $[\mathtt{q}_0, \mathtt{q}_0 + \mathtt{m}_0 - 1]$ to generate the unmapped region $[\mathtt{p}_0, \mathtt{p}_0 + \mathtt{n}_0 - 1]$. The bounds of this unmapped region generate the first two constraints in Figure 13. The recursive call sites $cs2$ and $cs3$ generate similar constraints, except that now the analysis unmaps the region $[l^{wr}_{\texttt{dcInc},a}, u^{wr}_{\texttt{dcInc},a}] = [C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3, C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6]$. The positivity of C_1, C_2, C_4, and C_5 ensures the correctness of the symbolic unmapping for this region. The last four constraints correspond to the call site constraints for $cs2$ and $cs3$ after performing the symbolic unmapping.

Step 3. *Reduce the symbolic constraints to a linear program and solve the linear program.* The analysis next uses the reduction method presented in Section 3.3.4 to reduce the symbolic constraint system to a linear program. This linear program contains constraints that explicitly ensure the positivity of the rational coefficients of the variables in the bounds. The solution of the linear program directly yields the symbolic region sets of the recursive procedures in S. As for the bounds analysis, if the linear program does not have a solution, the compiler conservatively sets the regions of all the procedures in S to $[-\infty, +\infty]$. In our example, the algorithm reduces the symbolic constraints to the linear program in Figure 14. The solution of this linear program yields the following expressions for the bounds defined in the first step:

$$l^{wr}_{\texttt{dcInc},a} = \mathtt{p}_0\ , \quad u^{wr}_{\texttt{dcInc},a} = \mathtt{p}_0 + \mathtt{n}_0 - 1$$

The write region for `dcInc` is therefore $[\mathtt{p}_0, \mathtt{p}_0 + \mathtt{n}_0 - 1]$. The compiler similarly derives the same read region for `dcInc`.

Local Access Constraints:

$$\begin{aligned} & l^{rd}_{f,a} \le l \ \wedge\ u \le u^{rd}_{f,a} \quad \forall f \in S,\ \forall [l,u] \in RR^{local}_{f,a} \\ & l^{wr}_{f,a} \le l \ \wedge\ u \le u^{wr}_{f,a} \quad \forall f \in S,\ \forall [l,u] \in RW^{local}_{f,a} \end{aligned}$$

Call Site Constraints:

$$\begin{aligned} & l^{rd}_{f,a} \le L(SU_{M_{cs}}(l), cs) \ \wedge\ U(SU_{M_{cs}}(u), cs) \le u^{rd}_{f,a} \\ & \quad \forall f \in S,\ \forall cs \in \text{CallSites}(f,g),\ \forall [l,u] \in RR_{g,a} \\ & l^{wr}_{f,a} \le L(SU_{M_{cs}}(l), cs) \ \wedge\ U(SU_{M_{cs}}(u), cs) \le u^{wr}_{f,a} \\ & \quad \forall f \in S,\ \forall cs \in \text{CallSites}(f,g),\ \forall [l,u] \in RW_{g,a} \end{aligned}$$

Objective Function:

$$\min:\ \sum_{f \in S} [\ (u^{rd}_{f,a} - l^{rd}_{f,a}) + (u^{wr}_{f,a} - l^{wr}_{f,a})\ ]$$

Figure 12: Interprocedural Symbolic Constraints for a Set S of Mutually Recursive Procedures

Call Site Constraints:

$$\begin{aligned} C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3 &\le \mathtt{p}_0 \\ C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6 &\ge \mathtt{p}_0 + \mathtt{n}_0 - 1 \\[1ex] C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3 &\le C_1\mathtt{p}_0 + C_2\tfrac{\mathtt{n}_0}{2} + C_3 \\ C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6 &\ge C_4\mathtt{p}_0 + C_5\tfrac{\mathtt{n}_0}{2} + C_6 \\[1ex] C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3 &\le C_1(\mathtt{p}_0 + \tfrac{\mathtt{n}_0}{2}) + C_2(\mathtt{n}_0 - \tfrac{\mathtt{n}_0}{2}) + C_3 \\ C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6 &\ge C_4(\mathtt{p}_0 + \tfrac{\mathtt{n}_0}{2}) + C_5(\mathtt{n}_0 - \tfrac{\mathtt{n}_0}{2}) + C_6 \end{aligned}$$

Objective Function:

$$\min: (C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6) - (C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3)$$

Figure 13: Interprocedural Symbolic Constraints for Write Regions of $S = \{\texttt{dcInc}\}$ in Example

$$\begin{array}{llll} C_1 \le 1 & C_2 \le 0 & & C_3 \le 0 \\ C_4 \ge 1 & C_5 \ge 1 & & C_6 \ge -1 \\[1ex] C_1 \le C_1 & C_2 \le C_2/2 & & C_3 \le C_3 \\ C_4 \ge C_4 & C_5 \ge C_5/2 & & C_6 \ge C_6 \\[1ex] C_1 \le C_1 & C_2 \le C_1/2 + C_2/2 & & C_3 \le C_3 \\ C_4 \ge C_4 & C_5 \ge C_4/2 + C_5/2 & & C_6 \ge C_6 \\[1ex] C_1 \ge 0 & C_2 \ge 0 & C_4 \ge 0 & C_5 \ge 0 \end{array}$$

$$\min:\quad (C_4 + C_5 + C_6) - (C_1 + C_2 + C_3)$$

Figure 14: Generated Linear Program for Write Regions of $S = \{\texttt{dcInc}\}$ in Example

Finally, the compiler analyzes the `main` procedure. Here the call site mapping for call site $cs4$, where procedure `main` calls `dcInc`, is unknown $M_{cs4} = M_{unk}$, so the symbolic unmapping generates the whole-array region $[-\infty, +\infty]$ for the procedure `main`. The interprocedural analysis therefore derives the following region sets for the array allocated in `main`:

$$\begin{array}{lll} RW_{\texttt{baseInc},a} & = RR_{\texttt{baseInc},a} & = \{[\mathtt{q}_0, \mathtt{q}_0 + \mathtt{m}_0 - 1]\} \\ RW_{\texttt{dcInc},a} & = RR_{\texttt{dcInc},a} & = \{[\mathtt{p}_0, \mathtt{p}_0 + \mathtt{n}_0 - 1]\} \\ RW_{\texttt{main},a} & = RR_{\texttt{main},a} & = \{[-\infty, +\infty]\} \end{array}$$

This information enables the compiler to determine that there are no array bounds violations because of the calls to `baseInc` and `dcInc`. It can also use the analysis results to determine that there are no data races in these procedures. If it had been given a sequential version of the program, the analysis results would allow the compiler to automatically parallelize it.

4 Extensions

We next present some extensions to the basic symbolic analysis algorithm presented so far. These extensions are designed to improve the precision or efficiency of the basic algorithm, or to extend its functionality.

4.1 Correlation Analysis

The compiler uses *correlation analysis* to improve the precision of the bounds analysis. Correlated variables are integer or pointer variables with matching increments in loops.[4] When the loop increments correlated variables but the loop condition specifies bounds only for some of the correlated variables, the compiler can use correlation analysis to automatically derive bounds for the other correlated variables.

```
void Merge(int *l1, int *h1,
           int *l2, int *h2, int *d) :

   while ((l1 < h1) && (l2 < h2))
   if (*l1 < *l2) { *d = *l1; d++; l1++; }
   else           { *d = *l2; d++; l2++; }
```

Figure 15: Correlated Variables in Mergesort

Figure 15 shows an example of correlated variables in the main loop of the `Merge` procedure in Mergesort. Here, the variables `d`, `l1`, and `l2` are correlated, but the loop condition specifies upper bounds only for `l1` and `l2` ($\mathtt{l1} < \mathtt{h1}_0$ and $\mathtt{l1} < \mathtt{h2}_0$). Correlation analysis enables the compiler to automatically derive an upper bound for `d` at the top of the loop body: $\mathtt{d} \leq \mathtt{d}_0 + (\mathtt{h1}_0 - \mathtt{l1}_0) + (\mathtt{h2}_0 - \mathtt{l2}_0) - 2$.

The compiler detects correlated variables by generating a *correlation expression* e with the property that the execution of the loop body does not decrease e if and only if the variables in e have matching increments. In general, e is a linear combination of the correlated variables. In our example, $e = \mathtt{l1} + \mathtt{l2} - \mathtt{d}$. Note that because the loop does not decrease e, every increment of `d` has a matching increment of either `l1` or `l2`.

To prove that the loop does not decrease e, the compiler extracts a new lower bound for e at the program point q at the top of the loop body (flow of control passes through q once for every loop iteration). The compiler generates a new *correlation variable* v with the property that $v = e$ throughout the loop, and treats v specially in the bounds analysis, ensuring that its bounds are valid bounds for e. At the program point p before the loop, the analysis sets the bounds of v to $L(e,p)$ and $U(e,p)$. When an instruction in the loop body increments one of the variables in e, the analysis either increments or decrements both bounds of v, depending on the sign of the coefficient of the variable in the correlation expression. In our example, the analysis increments the bounds of v when an instruction increments `l1` or `l2`, and decrements the bounds of v when an instruction increments `d`. For all other instructions that write one of the variables in e, the analysis sets the bounds of v to the bounds of e. In our example, this special treatment of v enables the analysis to compute $L(v,q) = \mathtt{l1}_0 + \mathtt{l2}_0 - \mathtt{d}_0$, where q is the program point at the top of the loop body.

This lower bound $\mathtt{l1}_0 + \mathtt{l2}_0 - \mathtt{d}_0 \leq v = \mathtt{l1} + \mathtt{l2} - \mathtt{d}$ translates immediately into an upper bound for `d`: $\mathtt{d} \leq (\mathtt{l1} + \mathtt{l2}) - (\mathtt{l1}_0 + \mathtt{l2}_0 - \mathtt{d}_0)$. The compiler uses the test conditions $\mathtt{l1} < \mathtt{h1}_0$ and $\mathtt{l2} < \mathtt{h2}_0$ to eliminate `l1` and `l2` from the inequality, deriving the upper bound for `d` at the top of the loop: $\mathtt{d} \leq (\mathtt{h1}_0 - 1 + \mathtt{h2}_0 - 1) - (\mathtt{l1}_0 + \mathtt{l2}_0 - \mathtt{d}_0) = \mathtt{d}_0 + (\mathtt{h1}_0 - \mathtt{l1}_0) + (\mathtt{h2}_0 - \mathtt{l2}_0) - 2$.

4.2 Integer Division

As presented so far, the algorithm assumes that division is exact, i.e., it is identical to real division. But address calculations in divide and conquer programs often use integer division. For positive expressions e, the compiler uses the following equations to eliminate integer division operations from lower and upper bound polynomials:

$$\begin{array}{rcl} L\left(\left\lfloor \frac{e}{n} \right\rfloor, p\right) & = & \frac{L(e,p) - n + 1}{n} \\ U\left(\left\lfloor \frac{e}{n} \right\rfloor, p\right) & = & \frac{U(e,p)}{n} \end{array}$$

The compiler uses similar equations if e is negative, or if there is no information about the sign of e. With these modifications, the symbolic analysis algorithm correctly handles programs with integer division.

4.3 Constraint System Decomposition

As presented so far, the algorithm generates a single linear program for each symbolic constraint system. If it is not possible to statically bound one of the pointer or array index variables (in the intraprocedural bounds analysis) or one of the symbolic regions (in the interprocedural region analysis), the linear program solver will not deliver a solution, and the algorithm will set *all bounds* in the system to conservative, infinite values, even though it may be possible to statically compute some of the bounds. We avoid this form of imprecision by decomposing the symbolic constraint system into smaller subsystems. This decomposition isolates, as much as possible, variables and regions without statically computable bounds.

The decomposition proceeds as follows. We first build a *bounds dependence graph*. The nodes in this directed graph are the unknown symbolic bounds $l_{v,p}$ and $u_{v,p}$ in the constraint system (in the intraprocedural bounds analysis) or the lower and upper bounds of the symbolic regions (in the interprocedural region analysis). The edges reflect the

[4] The analysis also correctly handles increments and decrements by arbitrary constants.

dependences between the bounds. For each symbolic constraint of the form $b \leq e$ or $b \geq e$, where b is a symbolic bound and e is an expression containing symbolic bounds, the graph contains an edge from each bound in e to b. Intuitively, there is an edge from one bound to another if the second bound depends on the first bound. The algorithm uses the bounds dependence graph to decompose the original constraint system into subsystems, with one subsystem for each strongly connected component in the graph. It solves the subsystems in the topological order of the corresponding strongly connected components, with the solutions flowing along the edges between the strongly connected components of the bounds dependence graph. The system decomposition ensures that:

- *The bounds of unrelated variables and regions fall into different and unrelated subsystems.* The analysis therefore computes the bounds independently, and a failure to compute a bound for one variable or region does not affect the computation of the bounds for the other variables and regions.
- *The bounds of a variable at different program points fall into different subsystems if the program points are not in the same loop.* Thus, outside loops, the system decomposition separates the computation of bounds at different program points.
- *Unrelated lower and upper bounds of the same variable at the same program point fall into different and unrelated subsystems.* The analysis can therefore compute a precise lower bound of a variable even if there is no information about the upper bound of that variable, and vice versa.

The decomposition also improves the efficiency of the algorithm. The smaller, decomposed subsystems solve much faster than the original system. Most of the subsystems are trivial systems with only one target bound, and we use specialized, fast solvers for these cases.

Finally, the system decomposition enables the algorithm to extend the intraprocedural analysis to handle nonlinear polynomial bounds. We first extend the basic block analysis from Section 3.3 to handle assignments and conditionals with nonlinear polynomial expressions in the program variables. These nonlinear expressions generate nonlinear combinations of the bounds $l_{v,p}$ and $u_{v,p}$ in the symbolic constraint system. Unfortunately, the linear program reduction cannot be applied in this case because of the presence of terms that contain products of the coefficient variables.

The system decomposition allows the compiler to use a simple substitution algorithm to solve this problem and support nonlinear bounds expressions provided that the relevant bounds are not part of a cycle in the bounds dependence graph. Once the compiler has solved one subsystem, it can replace bounds in successor subsystems with the solution from the solved subsystem. This substitution eliminates nonlinear combinations of bounds in the successor subsystems, enabling the analysis to use the linear program reduction or specialized solvers to obtain a solution for the successor subsystems.

4.4 Analysis Contexts

As presented so far, the symbolic interprocedural analysis generates a single result for each procedure. In reality, the pointer analysis generates a result for each context in which each procedure may be invoked [25]. The symbolic analysis also generates a result for each context rather than a result for each procedure.

The pointer analysis algorithm uses *ghost allocation blocks* to avoid reanalyzing procedures for equivalent contexts [25]. We therefore extend the unmapping algorithm discussed in Section 3.4.4 to include a translation from the ghost allocation blocks in the analysis result to the actual allocation blocks at the call site.

5 Experimental Results

We used the SUIF compiler infrastructure [1] to implement a compiler based on the analysis algorithms presented in this paper. We extended the SUIF system to support programs written in Cilk, a parallel version of C [4]. Given a sequential C program, our compiler will automatically parallelize it. Given a Cilk program, our compiler will determine if it is free of data races. For both kinds of programs, our compiler will determine if it may violate its array bounds. Our compiler would also eliminate array bounds checks if the underlying language (C) had them. We present experimental results for several divide and conquer programs:

- **Fibonacci:** Standard recursive Fibonacci benchmark.
- **Quicksort:** Divide and conquer quicksort.
- **Mergesort:** Divide and conquer mergesort.
- **Heat:** Solves heat diffusion on a mesh.
- **Knapsack:** Solves the 0/1 knapsack problem.
- **BlockMul:** Divide and conquer blocked matrix multiply. Allocates temporary arrays on the stack.
- **NoTempMul:** Divide and conquer blocked matrix multiply with no temporary arrays.
- **LU:** Divide and conquer LU decomposition.

We have sequential and parallel versions of all programs. We would like to emphasize the challenging nature of the programs in this benchmark set. Most of them contain multiple mutually recursive procedures, and have been heavily optimized by hand to extract the maximum performance. As a result, they heavily use low-level C features such as pointer arithmetic and casts. Our analysis handles all of these low-level features correctly.

5.1 Data Race Detection and Array Bounds Violations

The analysis verifies that all of the parallel programs except Knapsack are free of data races. The data race in Knapsack is used to prune the search space. This data race is intentional and part of the algorithm design, but causes nondeterministic behavior. The analysis was also able to verify that none of the benchmarks violates the array bounds.

5.2 Automatic Parallelization

The analysis was able to automatically parallelize all of the sequential programs except Knapsack, whose parallelization would have a data race. In general, the analysis detected the same sources of parallelism as in the Cilk programs. We ran the benchmarks on an eight processor Sun Ultra Enterprise Server. Table 1 presents the speedups of the benchmarks with respect to the sequential versions, which execute with no parallelization overhead. In some cases the parallelized version running on one processor runs faster than the sequential version, in which case the absolute speedup is above one for one processor. We ran Quicksort and Mergesort on a randomly generated file of 8,000,000 numbers, and BlockMul, NoTempMul and LU on a 1024 by 1024 matrix.

Program	Number of Processors				
	1	2	4	6	8
Fibonacci	0.76	1.52	3.03	4.55	6.04
Quicksort	1.00	1.99	3.89	5.68	7.36
Mergesort	1.00	2.00	3.90	5.70	7.41
Heat	1.03	2.02	3.89	5.53	6.83
BlockMul	0.97	1.86	3.84	5.70	7.54
NoTemp	1.02	2.01	4.03	6.02	8.02
LU	0.98	1.95	3.89	5.66	7.39

Table 1: Absolute Speedups for Parallelized Programs

Program	Percentage of Eliminated Register Bits	Percentage of Eliminated Memory Bits
convolve	35.94%	25.76%
histogram	30.56%	73.86%
intfir	36.72%	1.59%
intmatmul	47.32%	35.42%
jacobi	42.71%	75.00%
life	65.92%	96.88%
median	43.75%	3.12%
mpegcorr	58.20%	53.12%
pmatch	59.38%	47.24%

Table 2: Bitwidth Analysis Results

5.3 Bitwidth Analysis

Bitwidth analysis has recently been identified as a concrete value range problem [27]. Even though our algorithm is designed to extract symbolic bounds, it extracts exact numeric bounds when it is possible to do so. By adjusting our algorithm to compute bounds for all variables and not just pointers and array indices, we are able to apply our algorithm to the bitwidth analysis problem. Table 2 presents experimental results for several programs. We report reductions in two kinds of program state: register state, which holds scalar variables, and memory state, which holds array variables. Our analysis is able to significantly reduce the number of bits required to hold the state of the program.

6 Related Work

We discuss several areas of related work: research in symbolic memory access region analysis, array bounds check elimination, race detection, and parallelizing compilers.

6.1 Symbolic Memory Access Region Analysis

Researchers have previously proposed several algorithms for the symbolic analysis of accessed memory regions in sequential programs [18, 24, 15]. These algorithms use fixed-point approaches to analyze recursive programs, employing a variety of ad-hoc techniques (such as artificially limiting the number of iterations or using imprecise widening operators) to avoid the problem of infinite ascending chains in the domain of symbolic expressions. In addition, previous techniques tended to exploit the loop structure of the program to determine the regions of memory directly accessed by each procedure, rather than providing a general framework for arbitrary control flow. This paper replaces these limited techniques with a clean, general formulation of the problem. As a result, our techniques can successfully analyze a wider range of programs. And we have generalized our techniques to analyze both sequential and parallel programs.

6.2 Array Bounds Check Elimination

There is a long history of research on array bounds check elimination [5, 19, 22, 20]. A typical goal is to move checks out of loops or to detect that the loop termination condition ensures that array accesses within the loop do not violate the array bounds. The goal of optimizing checks in loops tends to produce intraprocedural analyses that focus on the simple conditionals that occur frequently in programs that use loops as their primary control structure.

Our approach is designed to handle programs that use recursion as their primary control structure. It is possible to use the extracted pointer and index variable bounds to eliminate checks at individual array access sites. Our compiler instead checks the extracted symbolic regions against the size of the array at the allocation site to verify that invoked procedures do not violate the array bounds. Because our source language does not have array bounds checks, our implemented compiler uses this information to detect array bounds violations. But for languages with array bounds checks, the information would enable the compiler to eliminate all checks in the call graph rooted at the invoked procedure. To make this approach work for our target class of applications, we had to use symbolic polynomials with rational coefficients instead of the simpler expressions often used in previous approaches that are designed to work well for loop-based programs. Unlike some previous researchers, we have not attempted to eliminate partially redundant checks or move checks to less frequently executed program points.

6.3 Race Detection

Data races are widely recognized as a serious problem for parallel programmers. Several researchers have attacked the problem by developing packages that dynamically record information about the memory locations that parallel threads access, then use the information to determine the presence or absence of races in a specific execution of the program [11, 9]. In this context, a data race occurs when one thread writes a memory location, another thread accesses the same memory location, and the accesses are not protected by mutual exclusion or signal/wait synchronization. Our algorithms differ in that they can statically certify that all executions of the program are free of data races, and in that they are designed for programs that use fork/join synchronization, not mutual exclusion or signal/wait synchronization.

Researchers have also developed static analyses that allow programmers to declare the correspondence between locks and the pieces of data that they protect [28, 10, 12]. The analysis then checks that the program correctly holds the appropriate lock when it accesses data. We view these techniques as orthogonal to ours. Our techniques determine when parallel threads access disjoint regions of dynamically allocated arrays; lock-based analyses ensure that parallel threads hold the correct lock when they access shared data.

6.4 Parallelizing Compilers

Previous research in parallelizing compilers has focused on parallelizing loop nests that access dense matrices using affine access functions [1, 3, 17]. The techniques presented in this paper, on the other hand, are designed for programs with recursive procedures, dynamic memory allocation, and pointer arithmetic. On a more philosophical level, we have generalized our algorithms to the point where they unify the automatic parallelization problem for sequential programs and the static race detection problem for parallel programs.

Many parallel tree traversal programs can be viewed as divide and conquer programs. Shape analysis is designed to discover when a data structure has a certain "shape" such as a tree or list [7, 26]. Several researchers have used shape analysis as the basis for compilers that automatically parallelize divide and conquer programs that manipulate linked data structures. Commutativity analysis views computations as sequences of operations on objects [23]. It generates parallel code if all pairs of operations commute. We view both commutativity analysis and shape analysis as orthogonal to our analyses.

7 Conclusion

This paper presents a new analysis framework for the symbolic bounds analysis of pointers, array indices, and accessed memory regions. Standard program analysis techniques fail for this problem because the analysis domain has infinite ascending chains. Instead of fixed point algorithms, our analysis uses a framework based on symbolic constraints reduced to linear programs. Our pointer analysis algorithm enables us to apply our framework to complicated recursive programs that use dynamic memory allocation and pointer arithmetic. Experimental results from our implemented compiler show that our analysis can successfully solve several important program analysis problems, including static race detection, automatic parallelization, static detection of array bounds violations, elimination of array bounds checks, and reduction of the number of bits used to store computed values.

Acknowledgements

We would like to thank Alex Salcianu and Brian Demsky for their help in generating the experimental results. We would also like to thank Darko Marinov for discussions regarding the bounds dependence graph and Mark Stephenson for providing us with the bitwidth analysis benchmarks.

References

[1] S. Amarasinghe, J. Anderson, M. Lam, and A. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[2] C. Scott Ananian. Silicon C: A hardware backend for SUIF. Available from http://flex-compiler.lcs.mit.edu/SiliconC/paper.pdf, May 1998.

[3] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Raughwerger, P. Tu, and S. Weatherford. Effective automatic parallelization with Polaris. In *International Journal of Parallel Programming*, May 1995.

[4] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[6] M. Budiu, S. Goldstein, M. Sakr, and K. Walker. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*. Munich, Germany, August 2000.

[7] 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.

[8] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint Malo, France, June 1999.

[9] G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998.

[10] D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.

[11] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991.

[12] C. Flanagan and S. Freund. Type-based race detection for java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[13] 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.

[14] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[15] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *Proceedings of the 1999 Conference on Parallel Algorithms and Compilation Techniques (PACT) '99*, Newport Beach, CA, October 1999.

[16] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[17] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992. IEEE Computer Society Press, Los Alamitos, Calif.

[18] 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.

[19] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, San Diego, CA, June 1995.

[20] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, June 1982.

[21] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[22] J. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, San Diego, CA, June 1995. ACM, New York.

[23] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):941–992, November 1997.

[24] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[25] 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.

[26] 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.

[27] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[28] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the 1993 Winter Usenix Conference*, January 1994.