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

RADU RUGINA
Cornell University
and
MARTIN C. RINARD
Massachusetts Institute of Technology

This article 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.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Analysis, Languages

Additional Key Words and Phrases: Symbolic analysis, parallelization, static race detection

1. INTRODUCTION

This article 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

An earlier version of this article appeared in the Proceedings of the 2000 Conference on Programming Language Design and Implementation (Vancouver, B.C., Canada, June).
Authors' addresses: R. Rugina, Cornell University, Computer Science Department, 4141 Upson Hall, Ithaca, NY 14853; email: rugina@cs.cornell.edu; M. C. Rinard, MIT Laboratory for Computer Science, 545 Technology Square, NE43-420A, Cambridge, MA 02139; email: rinard@lcs.mit.edu.

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 [Gustavson 1997; Frens and Wise 1997; Chatterjee et al. 1999]. 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 based on dataflow analysis or abstract interpretation to this class of programs fails because the domain of symbolic expressions has infinite ascending chains. This article 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.

Using this framework to analyze several divide-and-conquer programs, our compiler was able to extract the precise memory access information required to perform the transformations and safety checks briefly described below.

(1) *Static Race Detection.* Explicitly parallel languages give programmers the control they need to produce extremely efficient programs. But they also significantly complicate the development process because of 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 will execute deterministically with no unanticipated interactions.

(2) *Automatic Parallelization.* The difficulty of developing parallel programs has led to a large research effort devoted to automatically parallelizing sequential programs. Our analysis is capable of automatically parallelizing recursive procedures in sequential divide-and-conquer programs. It compares the access regions of statements and procedures in the program to determine if they refer to disjoint pieces of memory and can safely execute in

parallel. We emphasize the fact that traditional parallelization techniques are of little or no use for this class of 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.

(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 determine if array accesses in the program may violate the array bounds.

(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 bound checking instructions. Because our algorithms characterize the regions of memory accessed by statements and procedures, they can allow the compiler to eliminate redundant 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.

(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.

1.1 Contributions

This article 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 [Nielson et al. 1999]. But these 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 shows how to use the symbolic 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 this article is organized as follows: Section 2 presents a running example that we use throughout the article. Section 3 presents the analysis algorithms, while Section 4 presents some extensions to these algorithms. We next discuss the scope of the analysis in Section 6 and give a complexity evaluation of the algorithm in Section 5. Section 7 presents experimental results from our implementation. Section 8 discusses related work. We conclude in Section 9.

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 [Frigo et al. 1998].

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. This example reflects the structure of most of the Cilk programs discussed in Section 7 in that it identifies subproblems using pointers into dynamically allocated memory blocks.

```
 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:  }
```

Fig. 1. Divide-and-conquer array increment example.

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]

[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.

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.

2.3 Pointers Versus Array Indices

As mentioned above, this example 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. 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.

Figure 2 shows a slightly modified version of the array increment example that identifies subproblems using array indices instead of pointers in the middle of the arrays. The divide and conquer procedure `dcInc` recursively increments `n` elements in the array `p`, starting with element at index `l` and the base case procedure `baseInc` iteratively increments `m` elements in the array, starting with element at index `k`. Our algorithm can succcesfully analyze this program and determine that the whole execution of the recursive procedure `dcInc` reads and writes a region $[\mathtt{p}_0+\mathtt{l}_0, \mathtt{p}_0+\mathtt{l}_0+\mathtt{n}_0-1]$ and that the iterative computation in the base case procedure `baseInc` reads and writes a region $[\mathtt{q}_0+\mathtt{k}_0, \mathtt{q}_0+\mathtt{k}_0+\mathtt{m}_0-1]$. Extracting these regions is no easier than computing the corresponding access regions for the program in Figure 1.

```
 1:  #define CUTOFF 16
 2:
 3:  void baseInc(int *q, int k, int m) {
 4:    int i;
 5:    i = 0;
 6:    while(i <= m-1) {
 7:      q[k+i] += 1;
 8:      i = i+1;
 9:    }
10:  }
11:  void dcInc(int *p, int l, int n) {
12:    if (n <= CUTOFF) {
13:      baseInc(p, l, n);
14:    } else {
15:      spawn dcInc(p, l, n/2);
16:      spawn dcInc(p, l+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, 0, size);
27:      /* code that uses A */
28:    }
29:  }
```

Fig. 2. Array increment program with array indices instead of pointer arithmetic.

In the remainder of the article, we present the algorithms that the compiler uses to extract symbolic bounds 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 these bounds as polynomials with rational coefficients; the variables in these polynomials represent the initial values of the parameters of the enclosing procedure.

3.1 Structure of the Compiler

Figure 3 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

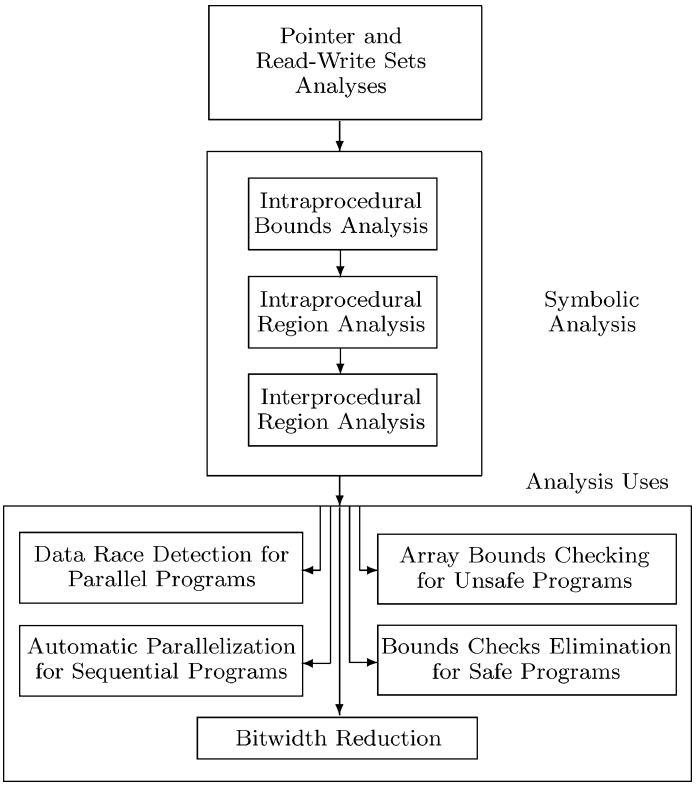

Fig. 3. Structure of the compiler.

sequential and parallel programs [Rugina and Rinard 1999b]. 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.

—*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 pointer and symbolic analysis phases are independent building blocks in the compiler: the particular pointer analysis algorithm used in the first phase doesn't affect the functioning of the subsequent symbolic analysis. In particular, the precise interprocedural, context-sensitive, flow-sensitive pointer analysis algorithm can be replaced with a more efficient, but less precise, context- and flow-insensitive algorithm, without affecting the functioning of the symbolic analysis. The difference when using such an imprecise pointer analysis would

be that the computed symbolic regions characterize accesses in a larger number of allocation blocks.

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 P_{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 $l_{\mathtt{i},p} = 0$ and $u_{\mathtt{i},p} = \mathtt{m}_0 - 1$, 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 continguous 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:

$$RW_{\mathtt{baseInc},a} = RR_{\mathtt{baseInc},a} = \{[\mathtt{q}_0, \mathtt{q}_0 + \mathtt{m}_0 - 1]\}$$
$$RW_{\mathtt{dcInc},a} \quad = RR_{\mathtt{dcInc},a} \quad = \{[\mathtt{p}_0, \mathtt{p}_0 + \mathtt{n}_0 - 1]\}$$

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

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 $B = \{B_j | 1 \leq j \leq l\}$ 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 4 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 5 presents the results of this step in our example.[2] We next explain how the compiler extracts these bounds.

[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`.

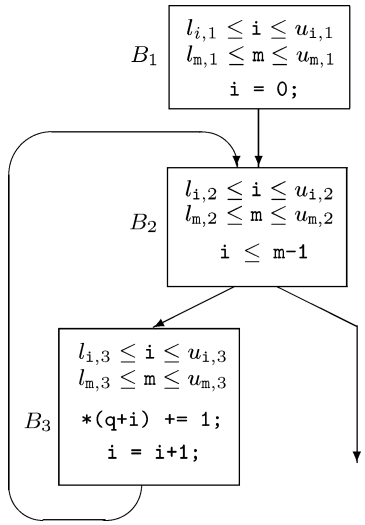

Fig. 4. Symbolic bounds at the start of basic blocks.

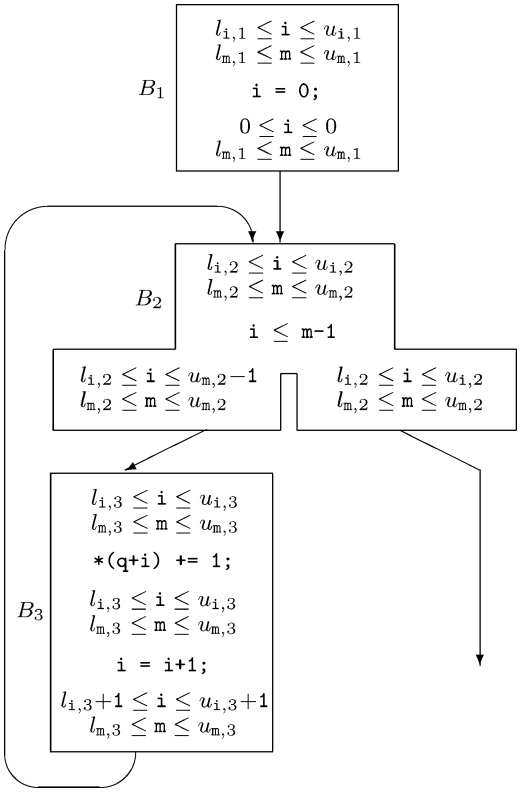

Fig. 5. Symbolic bounds at the end of basic blocks.

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 $u_{v,p}$.

The following equations give a recursive definition of $L(e, p)$ and $U(e, p)$ based on the structure of expression e:

$$
\begin{aligned}
L(c, p) &= c \\
L(v, p) &= l_{v,p} \\
L(e_1 + e_2, p) &= L(e_1, p) + L(e_2, p) \\
L(c \cdot e, p) &= \begin{cases} c \cdot L(e, p) & \text{if } c > 0 \\ c \cdot U(e, p) & \text{if } c \leq 0 \end{cases} \\
U(c, p) &= c \\
U(v, p) &= u_{v,p} \\
U(e_1 + e_2, p) &= U(e_1, p) + U(e_2, p) \\
U(c \cdot e, p) &= \begin{cases} c \cdot U(e, p) & \text{if } c > 0 \\ c \cdot L(e, p) & \text{if } c \leq 0 \end{cases} .
\end{aligned}
$$

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:

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

The analysis also takes into account the bounds in the condition expressions of branching constructs. For a conditional instruction i of the form $v \leq e$ or $v \geq e$, 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 $v \geq e$, the new lower bound of v is the lower bound of e. If the conditional is of the form $v \leq e$, 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:

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

All other bounds remain the same as the corresponding bounds from before the conditional. Finally, strict conditional instructions of the form $v < e$ or $v > e$ can be similarly analyzed, since $v < e$ is equivalent to $v \leq e - 1$ and $v > e$ is equivalent to $v \geq e + 1$.

Initialization Conditions:

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

Symbolic Constraints:

$$l_{\mathtt{i},2} \leq 0 \qquad 0 \leq u_{\mathtt{i},2}$$
$$l_{\mathtt{i},2} \leq l_{\mathtt{i},3} + 1 \qquad u_{\mathtt{i},3} + 1 \leq u_{\mathtt{i},2}$$
$$l_{\mathtt{m},2} \leq l_{\mathtt{m},1} \qquad u_{\mathtt{m},1} \leq u_{\mathtt{m},2}$$
$$l_{\mathtt{m},2} \leq l_{\mathtt{m},3} \qquad u_{\mathtt{m},3} \leq u_{\mathtt{m},2}$$
$$l_{\mathtt{i},3} \leq l_{\mathtt{i},2} \qquad u_{\mathtt{m},2} - 1 \leq u_{\mathtt{i},3}$$
$$l_{\mathtt{m},3} \leq l_{\mathtt{m},2} \qquad u_{\mathtt{m},2} \leq u_{\mathtt{m},3}$$

Objective Function:

$$\min : (u_{\mathtt{i},2} - l_{\mathtt{i},2}) + (u_{\mathtt{m},2} - l_{\mathtt{m},2}) + (u_{\mathtt{i},3} - l_{\mathtt{i},3}) + (u_{\mathtt{m},3} - l_{\mathtt{m},3})$$

Fig. 6. Symbolic constraint system for bounds analysis.

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 6 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:

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

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 symbolic bound polynomial consists of a number t of terms; each term consists of a coefficient variable and a product of reference set variables: $P_i = \sum_{i=1}^{t} c_i \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$, with $r_{i,j} \geq 0$.

—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 $Q = \sum_{i=1}^{t} c_i^Q \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$ and $R = \sum_{i=1}^{t} c_i^R \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$, then:

$$(Q \leq R) \in I \quad \text{is reduced to:} \quad c_i^Q \leq c_i^R, \quad \text{for all } 1 \leq i \leq t. \tag{1}$$

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 $O = \sum_{i=1}^{t} c_i^O \cdot x_1^{r_{i,1}} \cdots x_s^{r_{i,s}}$, then:

$$\min : O \quad \text{is reduced to:} \quad \min : \sum_{i=1}^{t} c_i^O. \tag{2}$$

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.

The above transformations assume that all of the variables in the reference set are *positive*. In this case, the reduction is *safe*—the reduced linear inequalities in Eq. (1) represent sufficient conditions for the symbolic inequalities to hold. In general, the compiler can relax the positivity condition and apply similar safe transformations if it can determine a constant lower bound or a constant upper bound for each variable in the reference set. For instance, if the compiler determines that $v \geq c$, where v is a variable and c is an arbitrary constant, the analysis can introduce a new variable $v' = v - c$, with $v' \geq 0$, and substitute $v' + c$ for each occurence of v in the symbolic inequalities. It can then apply the above reductions for v' since this is a positive variable. Similarly, if the analysis can determine that $v \leq c$, it can apply the same substitution technique using the positive variable $v' = c - v$ and replacing v with $c - v'$ in each symbolic inequality. However, if the compiler cannot compute constant upper or lower bounds for some variable in C_f, then it cannot reduce the symbolic system to a linear program. In practice, most of the variables used in array index expressions are either positive or negative. Therefore, in our implementation, the compiler 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.

We would like to emphasize that in many cases the reduction is also *precise*: if the symbolic polynomials are such that each term contains a different variable, then the reduced linear inequalities in Eq. (1) also represent necessary conditions. In particular, if the symbolic polynomials are linear combinations in the parameters of the enclosing function, then the reduction in this step doesn't lose precision. The theorem below characterizes the precision of the reduction transformation.

THEOREM PRECISION. *If the symbolic polynomials are such that each term contains a different variable:* $P = \sum_{i=1}^{t-1} c_i \cdot x_i^{r_i} + c_t$, *with* $r_i \geq 1$, *then the reduction of symbolic inequalities to linear inequalities between the coefficients is safe and precise:*

$$\sum_{i=1}^{t-1} a_i \cdot x_i^{r_i} + a_t \leq \sum_{i=1}^{t-1} b_i \cdot x_i^{r_i} + b_t, \quad \forall\, x_1 \geq 0, \ldots, x_{t-1} \geq 0 \tag{3}$$

$$\textit{if and only if} \quad a_i \leq b_i, \quad \forall\, 1 \leq i \leq t. \tag{4}$$

PROOF. We separately prove the two implications: safety and precision.

Safety: (4) ⇒ (3). This implication is trivial, because the corresponding terms in the left and in the right hand side of Eq. (3) satisfy the inequality. For each i, $1 \leq i \leq t-1$, and each $x_i \geq 0$, we have: $x_i^{r_i} \geq 0$ and $a_i \leq b_i$ (by hypothesis). Hence $a_i \cdot x_i^{r_i} \leq b_i \cdot x_i^{r_i}$. For the free term, $a_t \leq b_t$ is true by hypothesis.

Precision: (3) ⇒ (4). We know that Eq. (3) holds for all $x_1 \geq 0, \ldots, x_{t-1} \geq 0$ and we want to prove that Eq. (4) holds for all $1 \leq i \leq t$.

Since Eq. (3) holds for any positive variables, let $x_1 = \cdots = x_{t-1} = 0$. Then, Eq. (3) becomes $a_t \leq b_t$, which proves the inequality for the free term.

Now consider an arbitrary index j between 1 and $t-1$. If we set all variables except x_j to 0: $x_1 = \cdots = x_{j-1} = x_{j+1} = x_{t-1} = 0$, then Eq. (3) becomes:

$$a_j \cdot x_j^{r_j} + a_t \leq b_j \cdot x_j^{r_j} + b_t, \ \forall x_j \geq 0. \tag{5}$$

Hence:

$$(a_j - b_j) \cdot x_j^{r_j} \leq b_t - a_t, \ \forall x_j \geq 0. \tag{6}$$

We will prove now that $a_j \leq b_j$ by contradiction. Assume that $a_j > b_j$, that is, $a_j - b_j > 0$. With this assumption, Eq. (6) becomes:

$$x_j^{r_j} \leq \frac{b_t - a_t}{a_j - b_j}, \ \forall x_j \geq 0. \tag{7}$$

The fraction in the right hand side has a positive numerator $b_t - a_t \geq 0$, as we just proved above, and a strictly positive denominator $a_j - b_j > 0$, by our assumption. Hence the fraction is positive and has a positive, real r_j-th root. At this point, we get a contradiction, because the inequality in Eq. (7) cannot hold for any $x_j \geq 0$. For instance, it doesn't hold for $x_j = \lfloor \sqrt[r_j]{\frac{b_t - a_t}{a_j - b_j}} \rfloor + 1$. Hence, our assumption was incorrect, so $a_j \leq b_j$. Since j was chose arbitrarily, this completes our proof. □

Figure 7 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:

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

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

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

Finally, these bounds are used to compute the symbolic bounds of the variables at each program point, giving the final result shown in Figure 8. Note that the analysis detects that the symbolic range of the index variable `i` before

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:

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

Fig. 7. Linear program for bounds analysis.

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.

3.4 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 9 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.

3.4.2 *Intraprocedural Region Analysis.* Figure 10 presents the pseudocode for the intraprocedural region analysis. The algorithm first initializes the read region sets $RR_{f,a}^{local}$ and write region sets $RW_{f,a}^{local}$. These sets characterize

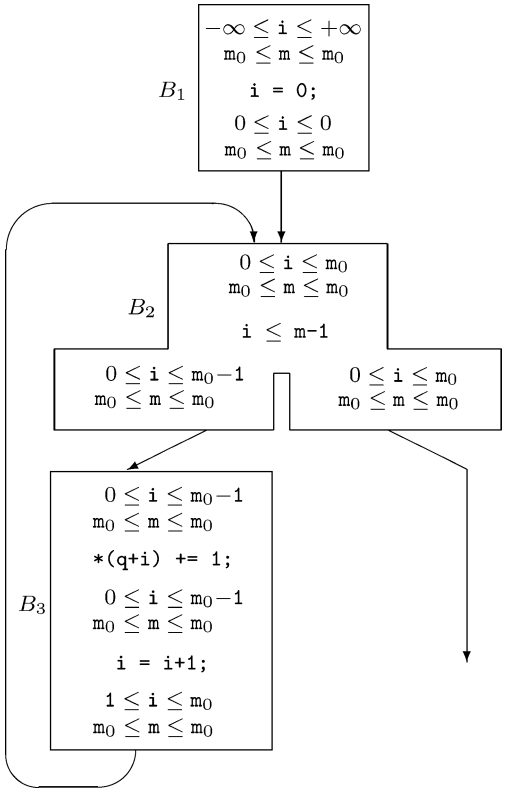

Fig. 8. Results of the bounds analysis.

```
Algorithm Coalesce(Region R, RegionSet RS)
    let R = [l, u]
    if        (∃[l', u'] ∈ RS, l ≤ l' ≤ u' ≤ u) then
            return (RS − {[l', u']}) ∪ {[l, u]}
    else if (∃[l', u'] ∈ RS, l ≤ l' ≤ u ≤ u') then
            return  (RS − {[l', u']}) ∪ {[l, u']}
    else if (∃[l', u'] ∈ RS, l' ≤ l ≤ u' ≤ u) then
            return (RS − {[l', u']}) ∪ {[l', u]}
    else if (∃[l', u'] ∈ RS, l' ≤ l ≤ u ≤ u') then
            return RS
    else   return RS ∪ {[l, u]}
```

Fig. 9. Region coalescing algorithm.

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

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 $RR^{local}_{f,a} = Coalesce(R_{new}, RR^{local}_{f,a})$

Fig. 10. Intraprocedural region analysis algorithm.

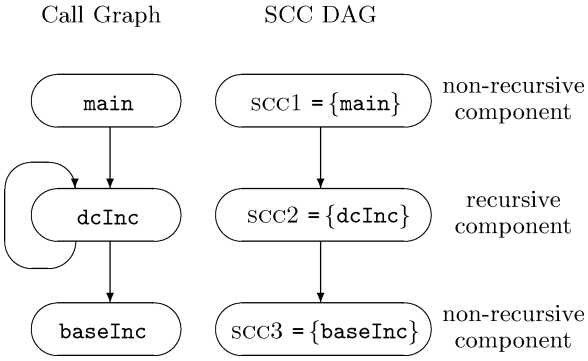

Fig. 11. Call graph and SCC DAG in example.

`dcInc` is:

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

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 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 11 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 $M \in S \rightarrow P_T^*$ (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* $M_{cs} \in (C_g \rightarrow P_{V_f}^*) \cup \{M_{unk}\}$ 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 $M \in S \rightarrow P_T^*$, 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 $cs \in$ 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:

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

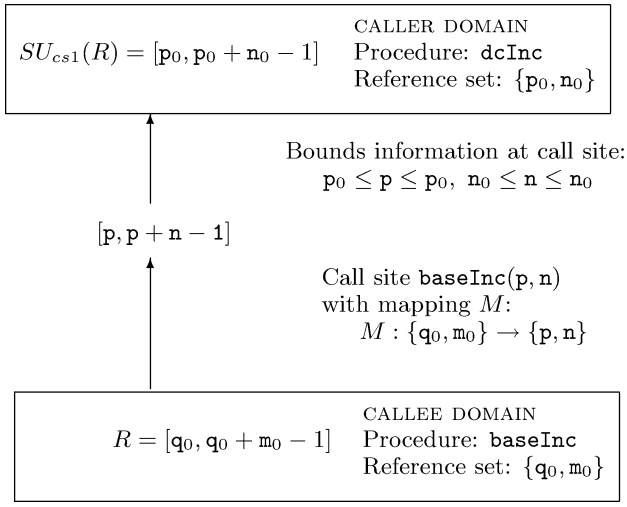

Fig. 12. Symbolic unmapping example.

—*Symbolic Unmapping of Region Sets for Allocation Blocks.* Given an accessed allocation block a and a call site $cs \in$ 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 nonrecursive 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 12 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 baseInc(p,n) in terms of the initial values of the parameters of dcInc, $\mathtt{p}_0$ and $\mathtt{n}_0$.

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.

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

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 $S = \{\texttt{dcInc}\}$ from our example.[3] Since the base case for this procedure writes 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 $C_{\texttt{dcInc}} = \{\texttt{p}_0, \texttt{n}_0\}$. The algorithm uses the following bounds:

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

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:

$$RR^{local}_{f,a} \subseteq RR_{f,a} \quad \forall f \in S$$
$$RW^{local}_{f,a} \subseteq RW_{f,a} \quad \forall f \in S.$$

[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`.

Local Access Constraints:

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

Call Site Constraints:

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

Objective Function:

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

Fig. 13. Interprocedural symbolic constraints for a set S of mutually recursive procedures.

Call Site Constraints:

$$\begin{aligned} C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3 &\le \mathtt{p}_0 \\ C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6 &\ge \mathtt{p}_0 + \mathtt{n}_0 - 1 \end{aligned}$$

$$\begin{aligned} C_1\mathtt{p}_0 + C_2\mathtt{n}_0 + C_3 &\le C_1\mathtt{p}_0 + C_2\tfrac{\mathtt{n}_0}{2} + C_3 \\ C_4\mathtt{p}_0 + C_5\mathtt{n}_0 + C_6 &\ge C_4\mathtt{p}_0 + C_5\tfrac{\mathtt{n}_0}{2} + C_6 \end{aligned}$$

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

Objective Function:

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

Fig. 14. Interprocedural symbolic constraints for write regions of $S = \{\texttt{dcInc}\}$ in example.

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

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

Figure 13 summarizes the constraints in the generated system. As in the intraprocedural case, the objective function minimizes the sizes of the symbolic regions.

Figure 14 presents the system of symbolic constraints for the interprocedural analysis of `dcInc`. Because `dcInc` does not directly access any allocation

$$
\begin{array}{lll}
C_1 \leq 1 & C_2 \leq 0 & C_3 \leq 0 \\
C_4 \geq 1 & C_5 \geq 1 & C_6 \geq -1 \\
\\
C_1 \leq C_1 & C_2 \leq C_2/2 & C_3 \leq C_3 \\
C_4 \geq C_4 & C_5 \geq C_5/2 & C_6 \geq C_6 \\
\\
C_1 \leq C_1 & C_2 \leq C_1/2 + C_2/2 & C_3 \leq C_3 \\
C_4 \geq C_4 & C_5 \geq C_4/2 + C_5/2 & C_6 \geq C_6 \\
\\
C_1 \geq 0 & C_2 \geq 0 & C_4 \geq 0 \quad C_5 \geq 0 \\
\\
\multicolumn{3}{l}{\min : \quad (C_4 + C_5 + C_6) - (C_1 + C_2 + C_3)}
\end{array}
$$

Fig. 15. Generated linear program for write regions of $S = \{\texttt{dcInc}\}$ in example.

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 14. 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 15. 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`.

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`:

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

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 described 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 or decrements in loops. When the loop increments (or decrements) several correlated variables, but the loop condition specifies explicit bounds only for a subset of these variables, the compiler can use correlation analysis to automatically derive bounds for the other correlated variables.

We will refer to the correlated variables whose bounds are explicitly specified in the loop condition as *test variables*, and to the other variables, which are updated in the loop body but whose bounds are not given in the test condition, as *target variables*. The goal of correlation analysis is to compute bounds for the target variables using the known bounds of the test variables; the compiler can derive such bounds only if each increment or decrement of a target variable is matched by at least one increment or decrement of a test variable in the loop body.

The correlation analysis presented in this section can handle increments or decrements by arbitrary constants. An increment is a statement that increases the value of a variable by a positive constant; a decrement is one that decreases the value of a variable by a positive constant. We will generally refer to increment or decrement statements as *incremental updates*.

4.1.1 *Example* 1: *Two Correlated Variables.* Figure 16 shows a simple example of correlated variables. The loop in this program increments two variables `i` and `j` at each iteration, but the loop condition specifies an upper bound only for `i`: $\mathtt{i} \leq \mathtt{m}_0 - 1$. Correlation analysis enables the compiler to automatically derive an upper bound for target variable `j` at the top of the loop body: $\mathtt{j} \leq 2\mathtt{m}_0 - 4$.

The compiler detects correlated variables and computes bounds for the target variables as follows: It first generates a *correlation expression ce* which counts the number of incremental updates for test and target variables. The compiler builds *ce* as a linear combination of test and target variables such that the value

```
void foo(int m) {
   int i,j;
   i = 1;
   j = 0;
   while (i <= m-1) {
      i = i+1;
      j = j+2;
   }
}
```

Fig. 16. Simple example of correlated variables.

of *ce* increases by 1 whenever the loop incrementally updates a test variable and decreases by 1 whenever the loop incrementally updates a target variable. Intuitively, the correlated expression counts the difference between the number of incremental updates of test variables and the number of incremental updates of target variables. At this point, the compiler doesn't know if the variables in *ce* are correlated—this depends on the control flow in the body of the loop. In our example, the correlation expression $ce = \mathtt{i} - \mathtt{j}/2$ counts the difference between the number of increments of `i` and the number of increments of `j`.

Next, the compiler tries to prove that the variables in *ce* are correlated. For this, it computes a lower bound for the correlation expression at program point q, at the top of the loop body (the flow of control passes through q once for each iteration). Note that the expression *ce* has a finite lower bound at program point q only if the variables in this expression are correlated; in this case, the lower bound of the correlation expression is the value of that expression at program point p, right before the loop. If the variables in *ce* are not correlated, then the value of *ce* strictly decreases on some paths in the loop body, hence *ce* will have an infinite lower bound at program point q. In our example $ce \geq 1$ at program point q; this finite bound shows that `i` and `j` are correlated.

Finally, the compiler can use the lower bound for *ce* at program point at the top of the loop body and the bounds of the test variables from the loop condition to derive new bounds for the target variables. In our example, the analysis uses the upper bound $\mathtt{i} \leq \mathtt{m}_0 - 1$ of the test variable `i` and the lower bound $ce \geq 1$ of the correlation expression $ce = \mathtt{i} - \mathtt{j}/2$ to compute an upper bound for the target variable `j`: $\mathtt{j} = 2\mathtt{i} - 2ce \leq 2(\mathtt{m}_0 - 1) - 2 \cdot 1 = 2\mathtt{m}_0 - 4$.

We have integrated the above correlation analysis mechanism in our symbolic analysis framework by generating additional constraints in our symbolic system; the solution to the system automatically yields the new bounds for the correlation expressions and for the target variables. Consider a target variable v and the correlation expression *ce* which counts the difference between the incremental updates of all test variables and the incremental updates of v. To compute the new bounds of *ce* and v, the analysis builds the additional constraints as follows:

—It generates a correlation variable cv with the property that $cv = ce$ at each program point. As for other variables, the analysis assigns initial bounds

for cv at the beginning of each basic block in the loop body. The analysis also computes the bounds of cv at program point p, right before the loop: $l_{cv,p} = L(ce, p)$ and $u_{cv,p} = U(ce, p)$. Then, during the analysis of each basic block, the analysis treats the correlation variable specially: when an instruction in the loop body incrementally updates a test variable, the analysis increments cv by 1; similarly, when an instruction incrementally updates v, the analysis decrements cv by 1. The compiler next generates symbolic constraints between the lower and upper bounds of cv at the beginning and end of basic blocks, in the same manner as for all the other variables in the program.

—The analysis also generates a new upper or lower bound for v on the true branch of the conditional instruction of the loop. The analysis derives the new bound for v a the top of the loop body using the lower bound of cv and the bounds of all the test variables at that program point, as presented above for the example program. If the coefficient of the target variable in the correlation expression is negative (i.e., if the target variable is incremented in the loop), then the computed bound is an upper bound; otherwise it is a lower bound.

Figure 17 shows how correlation analysis how correlation analysis works for our example. It first generates a correlation variable cv for the expression $ce = \mathtt{i} - \mathtt{j}/2$ and computes the bounds of cv at each program point. Before the loop, the lower and upper bounds of cv are 1 because $1 \leq \mathtt{i} \leq 1$ and $0 \leq \mathtt{j} \leq 0$ at this point. The analysis then computes the bounds of cv in the loop body: after `i = i+1`, it increases the bounds of cv by 1; after `j = j+2` it decreases the bounds of cv by 1. The bounds of cv are therefore unchanged after the execution of the loop body.

The analysis next derives a new bound for target variable `j` on the true branch of the conditional instruction. Since $cv = ce = \mathtt{i} - \mathtt{j}/2$, the compiler deduces that $\mathtt{j} = 2\mathtt{i} - 2cv$. The test condition of the loop provides an upper bound for `i` at the beginning of the loop: $\mathtt{i} \leq u_{\mathtt{m},2} - 1$. The compiler combines these relations to derive a new upper bound for `j` on the true branch: $\mathtt{j} \leq 2(u_{\mathtt{m},2} - 1) - 2l_{cv,2}$.

After the analysis of each basic block, the algorithm proceeds as before. It generates a symbolic system of inclusion constraints between the ranges of each variable, including the correlation variable cv, and then reduces this system to a linear program. The solution to the linear program automatically gives the symbolic bounds at each program point, for each variable, as shown in Figure 18. The computed bounds include the lower bound of correlation variable cv at the top of the loop body: $cv \geq 1$ and the upper bound of the target variable `j` at the same program point: $\mathtt{j} \leq 2\mathtt{m}_0 - 4$.

We would like to emphasize that, in our symbolic framework, the analysis computes the bounds for the correlation variable cv and for the target variable `j` at the same time. The solution to the constraint system simultaneously yields the bounds for `j` and cv, despite the fact that the bound computation for `j` requires the lower bound of cv and conceptually takes place after computing the bound of cv.

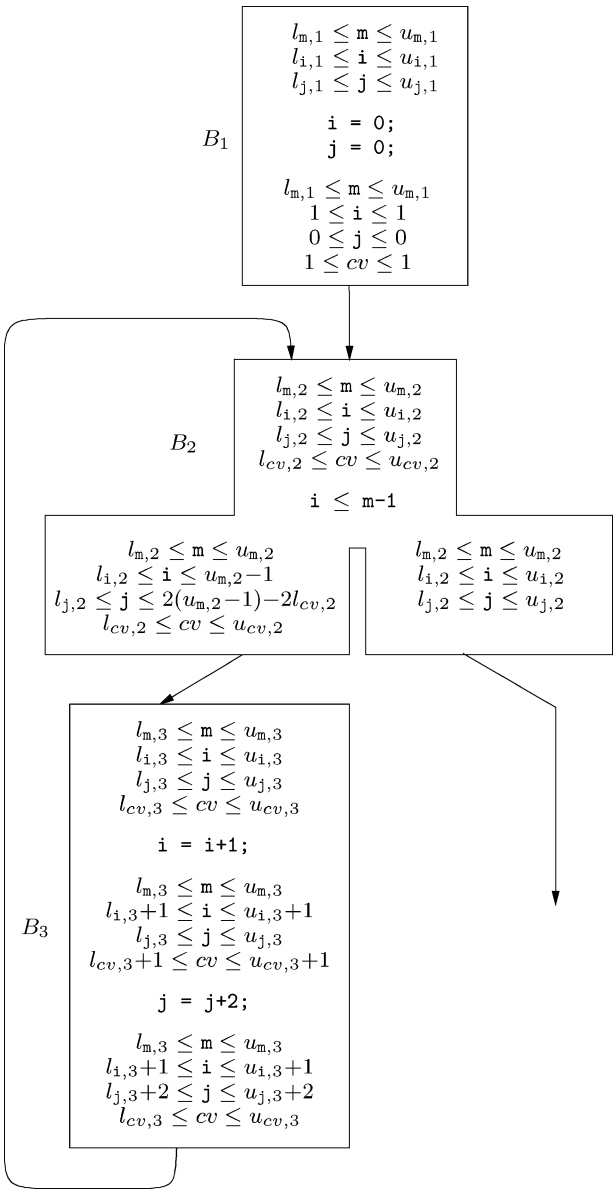

Fig. 17. Bounds at each program point for correlation analysis using correlation variable cv, which corresponds to expression $ce = \mathtt{i} - \mathtt{j}/2$.

4.1.2 *Example* 2: *Mergesort.* Figure 19 shows an example of correlated variables in the main loop of the `Merge` procedure in a Mergesort program. Here, the variables `d`, `l1`, and `l2` are correlated, but the loop condition specifies upper bounds only for test variables `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 target variable `d` at the top of the loop body: $\mathtt{d} \le \mathtt{d}_0 + (\mathtt{h1}_0 - \mathtt{l1}_0) + (\mathtt{h2}_0 - \mathtt{l2}_0) - 2$.

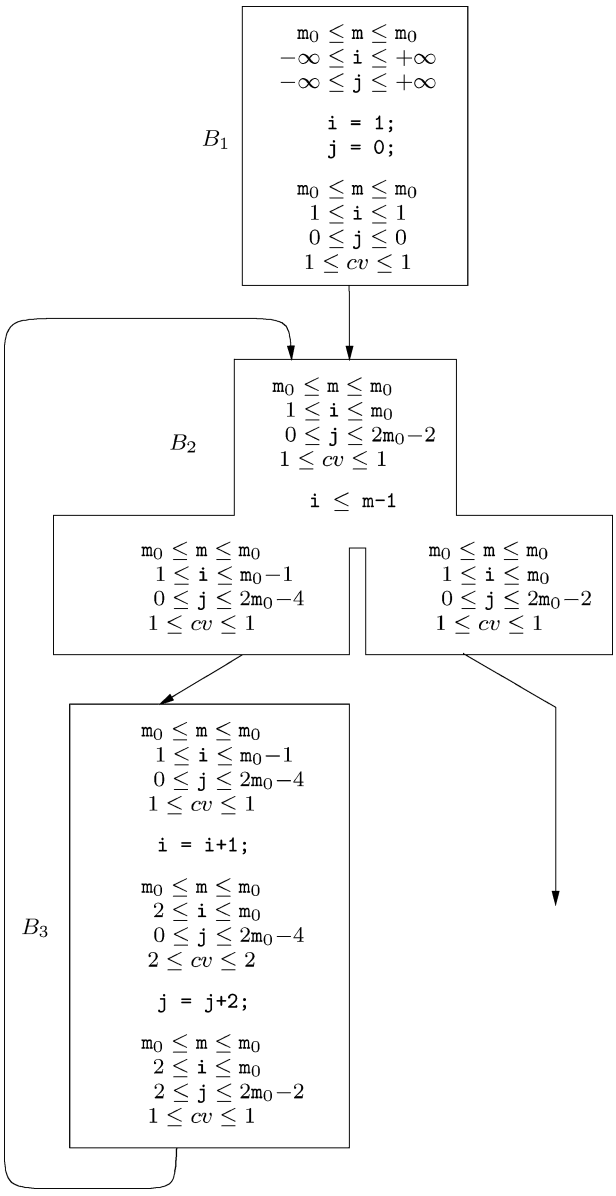

Fig. 18. Result symbolic bounds for correlation example.

In this example, each increment of d matches an increment of either l1 or l2, depending on the execution path at each iteration. To detect the correlation between these variables, the analysis derives a correlation expression $ce = \mathtt{l1}+\mathtt{l2}-\mathtt{d}$. The compiler then computes a finite bound for this expression at the top of the loop body: $ce \ge \mathtt{l1}_0+\mathtt{l2}_0-\mathtt{d}_0$. Because the bound is finite, d, l1, and l2 are correlated. Finally, this lower bound translates immediately into an upper bound for d, using the upper bounds for the test variables $\mathtt{l1} < \mathtt{h1}_0$ and $\mathtt{l2} < \mathtt{h2}_0$

```
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++; }

   ...
}
```

Fig. 19. Correlated variables in Mergesort.

```
void QuickSort(int *A, int low, int high) {
   int p, i, j;

   if (low < high) {
      p = A[low];
      j = low;

      i = low+1;
      while(i <= high) {
         if (A[i] < p) {
            j = j+1;
            swap(&A[j], &A[i]);
         }
         i = i+1;
      }

      swap(&A[low], &A[j]);
      QuickSort(A, low, j);
      QuickSort(A, j+1, high);
    }
}
```

Fig. 20. Correlated variables in Quicksort.

from the test condition: $\mathtt{d} = \mathtt{l1} + \mathtt{l2} - ce \leq (\mathtt{h1}_0 - 1) + (\mathtt{h2}_0 - 1) - (\mathtt{l1}_0 + \mathtt{l2}_0 - \mathtt{d}_0) = \mathtt{d}_0 + \mathtt{h1}_0 - \mathtt{l1}_0 + \mathtt{h2}_0 - \mathtt{l2}_0 - 2$.

4.1.3 *Example* 3: *Quicksort.* Figure 20 shows an examples of correlated variables in the recursive procedure of a Quicksort program. In the while loop of this procedure, the integer variables `i` and `j` are correlated, but the condition of the loop specifies an upper bound only for `i`: $\mathtt{i} \leq \mathtt{high}_0$. In this example, each increment of the target variable `j` matches one or more increments of `i`, depending on how many times the true branch of the inner if statement is executed in the loop. Again, the compiler uses correlation analysis to deduce an upper bound for `j` at the beginning of the loop: $\mathtt{j} \leq \mathtt{high}_0 - 1$.

To compute this bound, the compiler uses the same mechanism as in the previous examples. It generates a correlation expression $ce = \mathtt{i} - \mathtt{j}$. It then computes a lower bound $ce \geq 1$ at the top of the loop body. Finally, this lower bound and the test condition $\mathtt{i} \leq \mathtt{high}_0$ translate into an upper bound for `j` at the top of the loop: $\mathtt{j} = \mathtt{i} - ce \leq \mathtt{high}_0 - 1$.

4.1.4 *General Method.* In this section, we present a general method for detecting correlated variables and for computing symbolic bounds for these variables. Consider a loop in the program and let p be the program point before the loop and q the program point at the top of the loop body. Consider that the condition C in the loop test is a conjunction of inequalities that specify upper bounds for a set of test variables $v_1, \ldots, v_n$ and lower bounds for a set of test variables $u_1, \ldots, u_m$:

$$\begin{aligned} C: \quad & (v_1 \leq E_1) \wedge \cdots \wedge (v_n \leq E_n) \wedge \\ & (u_1 \geq F_1) \wedge \cdots \wedge (u_m \geq F_m), \end{aligned}$$

where E_i and F_i are expressions containing variables not modified in the loop body. The bounds of these expressions at program points p and q are therefore identical. Consider that the only updates of the test variables v_i in the loop body are increments by integer constants $a_i > 0$ and the only updates of variables u_i in the loop body are decrements by integer constants $b_i > 0$:

$$\begin{aligned} v_i &= v_i + a_i \qquad \text{for } 1 \leq i \leq n \\ u_i &= u_i - b_i \qquad \text{for } 1 \leq i \leq m. \end{aligned}$$

Finally, a set of target variables $x_1, \ldots, x_k$ whose only updates in the loop body are increments by integer constants $c_i \geq 0$ and a set of target variables $y_1, \ldots, y_l$ whose only updates are decrements by integer constants $d_i > 0$:

$$\begin{aligned} x_i &= x_i + c_i \qquad \text{for } 1 \leq i \leq k \\ y_i &= y_i - d_i \qquad \text{for } 1 \leq i \leq l. \end{aligned}$$

Target variables do not occur in the loop condition C; the sets of test and target variables are therefore disjoint. Test and target variable may be updated multiple times in the loop body. For simplicity, we assume that each test and target variable is updated at most once within each basic block and is always updated with the same constant in different blocks. Both of these conditions can be relaxed to allow more general correlation patterns. The goal of the compiler is to detect if the target variables are correlated with the test variables and then derive bounds for the target variables at program point q.

If all of the above conditions are met, the compiler proceeds as follows: It first generates a correlation expression for each target variable x_i and y_i:

$$\begin{aligned} ce_i &= \sum_{j=1}^{n} \frac{1}{a_j} \cdot v_j - \sum_{j=1}^{m} \frac{1}{b_j} \cdot u_j - \frac{1}{c_i} \cdot x_i \qquad \text{for } 1 \leq i \leq k \\ ce_{k+i} &= \sum_{j=1}^{n} \frac{1}{a_j} \cdot v_j - \sum_{j=1}^{m} \frac{1}{b_j} \cdot u_j + \frac{1}{d_i} \cdot y_i \qquad \text{for } 1 \leq i \leq l. \end{aligned}$$

It then computes new bounds for the correlation expressions ce_i and for the target variables x_i, y_i by generating new constraints in the symbolic system:

—For each correlation expression ce_i, the compiler generates a corresponding *correlation variable* cv_i with the property that $cv_i = ce_i$ throughout the loop.

As in the case of program variables, the analysis generates unknown bounds for each of the correlation variables cv_i at the beginning of each basic block. The analysis also derives bounds for the correlation variables at program point p as follows: $l_{cv_i,p} = L(ce_i, p)$, and $u_{cv_i,p} = U(ce_i, p)$.

The analysis next treats each variable cv_i specially during the analysis of individual statements in the loop body, ensuring that its bounds are valid bounds for ce_i at each program point: if the current statement is an incremental update of a source variable $v_i = v_i + a_i$ or $u_i = u_i - b_i$, the analysis increments the bounds of all correlation variables cv_i by 1; if the current statement is an incremental update of a target variable $x_j = x_j + c_j$ or $y_i = y_i - d_i$, the analysis decrements the bounds of the corresponding correlation variable cv_i by 1 and leaves the bounds of all the other correlation variables unchanged.

The analysis then generates symbolic constraints for each correlation variable cv_i, in the same manner as for all other program variables, using the techniques from Section 3. It imposes that the range of cv_i at the beginning of each basic block should conservatively include the ranges of cv_i at the end of each predecessor basic block. It also constructs an objective function that minimizes the computed ranges for all the correlation variables cv_i.

—The analysis also derives new bounds each target variable on the true branch of the conditional instruction of the loop, as follows. The compiler uses the definitions of the correlation variables to derive an expression for each target variable:

$$x_i = \sum_{j=1}^{n} \frac{c_i}{a_j} \cdot v_j - \sum_{j=1}^{m} \frac{c_i}{b_j} \cdot u_j - c_i \cdot ce_i \qquad \text{for } 1 \le i \le k$$

$$y_i = \sum_{j=1}^{n} \frac{d_i}{a_j} \cdot v_j - \sum_{j=1}^{m} \frac{d_i}{b_j} \cdot u_j + d_i \cdot ce_{k+i} \qquad \text{for } 1 \le i \le l.$$

The analysis combines the above expressions, the bounds of the test variables v_i and u_i from the test condition C, and the lower bound of the correlation variables ce_i to derive new bounds for the test variables at program point q. At this point, it computes the following upper bounds for x_i and lower bounds for y_i:

$$U(x_i, q) = \sum_{j=1}^{n} \frac{c_i}{a_j} \cdot U(E_j, q) - \sum_{j=1}^{m} \frac{c_i}{b_j} \cdot L(F_j, q) - c_i \cdot L(cv_i, q)$$
$$\text{for } 1 \le i \le k$$

$$L(y_i, q) = -\sum_{j=1}^{n} \frac{d_i}{a_j} \cdot U(E_j, q) + \sum_{j=1}^{m} \frac{d_i}{b_j} \cdot L(F_j, q) + d_i \cdot L(cv_{k+i}, q)$$
$$\text{for } 1 \le i \le l.$$

The compiler then proceeds as presented in Section 3. It reduces the generated symbolic constraint system to a linear program. The solution to this linear program automatically yields the symbolic bounds for each variable at each program point. In particular, it yields symbolic bounds for the correlation variables and for the target variables. Note that the solution to the system yields

all these bounds at the same time, despite the fact that computing the bounds for the target variables conceptually takes place after computing the bounds for the correlation variables.

4.2 Integer Division

As presented so far, the algorithm assumes that division is exact, that is, it is identical to real division. But address calculations in divide and conquer programs often use integer division, for instance when dividing problems into subproblems. In the general case, the integer division of two numbers m and n is the floor of the real division: $\lfloor m/n \rfloor$.

The compiler handles integer division by approximating the integer division calculations with real division bounds. In general, integer division can occur either at the intraprocedural level, when integer variables are assigned expressions containing integer division, or at the interprocedural level, in the actual parameters at call sites. To handle both of these cases, the compiler extends the functionality of the bound operators L and U as follows. For a given expression E and for an integer constant $n \geq 2$, if the positivity analysis detects that E is positive at program point p, then the compiler uses the following equations to compute the lower and upper bounds of the expression $\lfloor \frac{E}{n} \rfloor$ at this program point:

$$L\left(\left\lfloor \frac{E}{n} \right\rfloor, p\right) = \frac{L(E,p) - n + 1}{n}$$
$$U\left(\left\lfloor \frac{E}{n} \right\rfloor, p\right) = \frac{U(E,p)}{n}.$$

Similarly, if E is negative, then:

$$L\left(\left\lfloor \frac{E}{n} \right\rfloor, p\right) = \frac{L(E,p)}{n}$$
$$U\left(\left\lfloor \frac{E}{n} \right\rfloor, p\right) = \frac{U(E,p) + n - 1}{n}.$$

Finally, if there is no information about the sign of E, then:

$$L\left(\left\lfloor \frac{E}{n} \right\rfloor, p\right) = \frac{L(E,p) - n + 1}{n}$$
$$U\left(\left\lfloor \frac{E}{n} \right\rfloor, p\right) = \frac{U(E,p) + n - 1}{n}.$$

With these modifications, the symbolic analysis algorithm will correctly handle programs with integer division, both at the intraprocedural level, for the bounds analysis, and at the interprocedural level, during the symbolic unmapping.

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

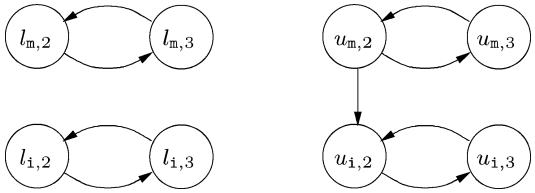

Fig. 21. Bounds dependence graph for example.

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 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. Figure 21 shows the bounds dependency graph for our running example. The compiler derives this graph from the symbolic inequalities from Figure 6. For instance, it generates the edge $l_{\mathtt{i},3} \rightarrow l_{\mathtt{i},2}$ from the symbolic inequality $l_{\mathtt{i},2} \leq l_{\mathtt{i},3} + 1$.

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 then 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. In our example, the graph from Figure 21 has four strongly connected components: $(l_{\mathtt{m},2}, l_{\mathtt{m},3})$, $(u_{\mathtt{m},2}, u_{\mathtt{m},3})$, $(l_{\mathtt{i},2}, l_{\mathtt{i},3})$, and $(u_{\mathtt{i},2}, u_{\mathtt{i},3})$. The compiler generates a linear system for each of these components, as shown in Figure 22. It then solves the linear system according to the reverse topological order of the corresponding strongly connected components, substituting the solutions into the successor systems. For instance, the compiler first solves system 2, then substitutes the solution for $u_{\mathtt{m},2}$ from system 2 into system 4, and finally solves system 4.

In general, we say that two decomposed systems are *unrelated* if the corresponding components are not connected to each other. Hence, there is no particular order of solving unrelated systems. By construction of the bounds dependence graph, if two decomposed systems are unrelated, the solution of one system doesn't affect the result of the other. In our example, system 1 and system 3 are unrelated, but system 2 and system 4 are not.

System 1:

$$
\begin{array}{l}
l_{\mathtt{m},2} \leq \mathtt{m}_0 \\
l_{\mathtt{m},2} \leq l_{\mathtt{m},3} \\
l_{\mathtt{m},3} \leq l_{\mathtt{m},2} \\
\min : -l_{\mathtt{m},2} - l_{\mathtt{m},3}
\end{array}
$$

System 2:

$$
\begin{array}{l}
u_{\mathtt{m},2} \geq \mathtt{m}_0 \\
u_{\mathtt{m},2} \geq u_{\mathtt{m},3} \\
u_{\mathtt{m},3} \geq u_{\mathtt{m},2} \\
\min : u_{\mathtt{m},2} + u_{\mathtt{m},3}
\end{array}
$$

System 3:

$$
\begin{array}{l}
l_{\mathtt{i},2} \leq 0 \\
l_{\mathtt{i},2} \leq l_{\mathtt{i},3} + 1 \\
l_{\mathtt{i},3} \leq l_{\mathtt{i},2} \\
\min : -l_{\mathtt{i},2} - l_{\mathtt{i},3}
\end{array}
$$

System 4:

$$
\begin{array}{l}
u_{\mathtt{i},2} \geq 0 \\
u_{\mathtt{i},2} \geq u_{\mathtt{i},3} + 1 \\
u_{\mathtt{i},3} \geq u_{\mathtt{m},2} - 1 \\
\min : u_{\mathtt{i},2} + u_{\mathtt{i},3}
\end{array}
$$

Fig. 22. Decomposed linear systems for example.

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. As shown in the example, 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 [Rugina and Rinard 1999b]. 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 [Rugina and Rinard 1999b]. 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. COMPLEXITY OF AND SCALABILITY OF ANALYSIS

In this section, we discuss the complexity of each phase in our algorithm: pointer analysis, intraprocedural bounds analysis, and interprocedural region analysis.

In the first phase, the compiler uses a flow-sensitive, context-sensitive pointer analysis algorithm. At the intraprocedural level, this algorithm runs in polynomial time in the program size. As most context-sensitive analyses, the interprocedural analysis has an exponential worst-case complexity. However, the use of partial transfer functions [Wilson and Lam 1995] significantly reduces the number of analyzed contexts in practice. As we mentioned at the beginning of Section 3, the compiler can alternatively use any other pointer analysis. For instance, it can use an efficient (but imprecise) flow-insensitive pointer analysis algorithm which runs in almost linear time [Steensgaard 1996].

The complexity of the bounds and region analyses are more relevant since the symbolic analysis phase is the main focus of this article. We first derive upper bounds for the size of the generated linear programs, that is, for the number of inequalities and number of variables in these systems. Consider a program with n statements, v integer and pointer variables, b basic blocks, f procedures, a dynamic allocation sites, and s call sites. Note that the program size n is an upper bound for b, f, a and s. We assume that the program consists only of three-address instructions (instructions which refer to at most three variables), so $v \leq 3n$. Without restricting the generality, we assume that there are no multibranch instructions in the program. Finally, consider that each symbolic polynomial has t terms, and that the analysis computes at most r regions for each allocation block during the interprocedural analysis; t and r are fixed, constant parameters for the analysis.

We derive an upper bound for the size of the linear programs generated during the intraprocedural bounds analysis, as follows. Since there are no multibranch instructions, each basic block has at most two successor blocks, hence the number of edges in the control flow graph is at most $2b$. For each of these edges and each variable in the program, the analysis derives a symbolic range constraint. Each range constraint then translates in two symbolic inequalities,

one for the upper bound and one for the lower bound. Hence, the number of symbolic inequalities in the constraint system for each procedure is at most $4bv$. Finally, the analysis reduces each symbolic inequality to t linear inequalities between the coefficients. Therefore, the generated linear system has at most $4t \cdot bv$ inequalitites. The number of variables in the linear system is the number of coefficients in the symbolic expressions. Since the analysis generates symbolic bounds for each variable at the beginning of each basic block and each such bound has t coefficients, there will be at most $2t \cdot bv$ variables in the linear system for each procedure. Hence, the intraprocedural bounds analysis solves f linear systems, each having at most $4t \cdot bv$ inequalitites and at most $2t \cdot bv$ variables. Since n is an upper bound for b, $v \leq 3n$, and t is a constant parameter for the analysis, the size of each linear system generated during the bounds analysis has quadratic complexity in the program size: $O(n^2)$.

We next derive an upper bound for the size of the linear programs generated by the interprocedural region analysis. The analysis generates a symbolic range constraint for each call site, each allocation block, each region for that allocation block, and each kind of access (read or write). Since the allocation blocks represent a memory abstraction which models stack an heap locations, the number of allocation blocks is at most $v + a$. Hence, the number of range constraints is at most $2r \cdot (v + a)s$. Each range constraint consists of an upper bound constraint and a lower bound constraint and each of them is reduced to t linear inequalities between coefficient variables. The number of inequalities in the constructed linear system is therefore at most $4tr \cdot (v + a)s$. Again, the number of variables in the linear system is the number of coefficients in the symbolic expressions. Since each symbolic bound has t coefficients and the analysis generates symbolic bounds for each kind of access (read or write), each kind of bound (upper or lower), each allocation block, each of the r regions of an allocation block, and each procedure in the current recursive cycle, the number of variables in each linear system is at most $4tr \cdot (v + a) f$. The interprocedural region analysis solves one linear system per recursive and there may be at most f recursive cycles in the program, hence this analysis solves at most f linear systems, each having at most $2r \cdot (v + a)s$ inequalitites and at most $4tr \cdot (v + a) f$ variables. Since t and r are constant parameters for the analysis, $v \leq 3n$, and n is an upper bound for a, s, and f, the size of each linear system generated during the region analysis also has quadratic complexity in the program size: $O(n^2)$.

Hence, our symbolic analysis solves $O(n)$ linear systems, each of size $O(n^2)$. Since linear programming has known polynomial time complexity, we finally conclude that our symbolic analysis also has polynomial complexity. In practice, however, non-polynomial linear programming solver algorithms such as the simplex method perform better than the polynomial-time algorithms for many linear systems.

We believe that the symbolic algorithm also scales in practice, because the compiler generates and solves exactly one linear system per procedure during the bounds analysis and one linear system per recursive cycle during the region analysis. Unlike most context-sensitive interprocedural analyses, the compiler does not analyze procedures multiple times. Instead, it is compositional and derives, for each procedure, a set of access regions parameterized by the procedure

arguments. These symbolic regions match any calling context and the compiler can use them to derive access regions at each call site by symbolically unmapping the parameterized regions of the invoked procedure.

6. SCOPE OF ANALYSIS

The analysis presented in this paper is designed to extract symbolic access ranges for a general class of programs, including pointer-based programs with arbitrary intraprocedural control-flow and procedures whose recursive invocations access disjoint pieces of the same arrays. However, there are several aspects of the program that the algorithm is not designed to handle. The discussion below summarize these cases and presents how the analysis can be extended to handle many of them.

First, the computed access regions characterize element ranges within contiguous blocks of memory—the algorithm is not designed to compute sets of elements within dynamic structures, such as sets of tree nodes or list nodes. For this, the compiler requires shape analysis techniques, which can determine the shape of dynamic structures. For instance, shape analysis can detect that a data structure is a tree rather than a general graph. The compiler can then use this information to conclude that the sets of nodes reachable from the left and the right children of each node are always disjoint. We consider that shape analysis techniques are orthogonal to the symbolic analysis presented in this article.

As presented, the algorithm derives unidimensional, contiguous regions for each index variable. We believe that the algorithm can be easily extended to handle multidimensional arrays, by extending the region analysis presented in Section 3.4 as follows: the intraprocedural analysis must use the the bounds of the variables in each dimension to derive intraprocedural multidimensional regions; the interprocedural analysis must propagate such regions across procedure calls, performing the symbolic unmapping on each dimension; and the analysis of recursive procedures must generate separate constraints for each dimension. The analysis can also be extended to detect strided access regions by recording, for each pointer and index variable, the greatest common division of all its increment values. Finally, we believe that the analysis can also be extended to handle non-rectangular multidimensional access regions within nested loops, by augmenting the region representation with relations between the index variables. However, we consider that the algorithm requires substantial changes to detect nonrectangular multidimensional regions accessed by recursive functions.

Next, the analysis relies on positivity analysis, that is, determining that the variables in the symbolic expressions have positive values. As presented in Section 3, this condition can be relaxed by imposing that each of these variables has either a constant lower bound or a constant upper bound. If this condition is not met, the analysis can only derive numeric bounds, in which case the symbolic expressions are constants and contain no variables.

Another necessary requirement to allow reducing the problem to a linear program is that the values assigned to integers and pointers must be linear

expressions. However, as we discussed in Section 4.3, the analysis can also handle assignments of nonlinear expressions only as long as they do not belong to a loop or to a recursive cycle in the program.

Another issue is that the analysis of conditional statements uses the bound given by the conditional on the true branch and leaves the bounds on the false branch unchanged. However, the bound before the conditional is also a valid bound on the true branch; and the negated condition can provide a new bound on the false branch. Hence, for each branch there are two possibilities: either use the bounds before the branch or use the bounds given by the conditional. This choice introduces disjunctions in our constraint system—for instance, if u_1 is the upper bound of v before the branch and u_2 is the upper bound given by the conditional, then the bound of v on the true branch is characterized by the disjunction $(v \leq e_1) \vee (v \leq e_2)$, depending on which of e_1 or e_2 provides a tighter upper bound. Note that using either of these bounds provides a safe result, but using the disjunction provides a more precise result. In general, disjunctions in the system lead to a case analysis which must examine each of the possibilities. To avoid this problem, we use the heuristic that the bounds of the true branches are characterized by the conditional expressions and the bounds on the false branches are characterized by the bounds before the conditional instruction. This heuristic works well for our benchmarks, allowing the compiler to extract the precise bounds information required to enable the transformations and safety checks discussed in Section 1. We would like to emphasize that, unlike the case of conditional statements, joins in control flow result in conjunctions of inequalities for the lower and the upper bounds.

Finally, library routines that manipulate arrays require a special treatment by the analysis. As in the case of any other whole-program analysis in the presence of library functions, our symbolic analysis must encode the behavior of such functions in the algorithm. The analysis of each library function must characterize the regions accessed by the function and the values of any updated pointer or index variable. For instance, the standard Unix utilities make heavy use of string manipulation routines. The application of our symbolic techniques to these programs requires the analysis to characterize how each of these functions accesses the strings passed as parameters and to characterize the returned string values.

7. EXPERIMENTAL RESULTS

We used the SUIF compiler infrastructure [Amarasinghe et al. 1995] to implement a compiler based on the analysis algorithms presented in this article. We extended the SUIF system to support programs written in Cilk, a parallel version of C [Blumofe et al. 1996]. We also used a freely distributed linear program solver, `lp_solve`,[4] which uses the simplex method to solve the linear programs generated by our analysis.

Table I presents our set of divide and conquer benchmarks; for each of these benchmarks, we have a sequential C version and a multithreaded Cilk version. In this section, we present results for the following experiments: automatic

[4]Available at `ftp://ftp.es.ele.tue.nl/pub/lp_solve/`.

Table I. Divide and Conquer Benchmark Programs

Program	Description
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	Blocked matrix multiply with temporary arrays
NoTemp	Blocked matrix multiply without temporary arrays
LU	Divide and conquer LU decomposition

parallelization for the sequential versions; data race detection for the multithreaded versions; and array bounds violation detection for the sequential and multithreaded versions. Our compiler would also eliminate array bounds checks if the underlying language (C) had them.

We would like to emphasize the challenging nature of the programs in our 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. The list below highlight several challenges in these benchmarks, all of which our analysis succesfully handles:

—The main loops in Quicksort and Mergesort contain correlated variables, as described in Section 4. The compiler needs to apply correlation analysis to accurately characterize memory accesses for these benchmarks.

—The main recursive procedure in Heat swaps its arguments at even iterations, but preserves their order at odd iterations. To enable transformations and safety checks for this benchmark, the compiler needs to apply the context-sensitive pointer analysis techniques; the compiler uses one context of the procedure for even iterations and another context for odd iterations.

—The main recursive procedure in the matrix multiply algorithm NoTemp uses eight recursive calls; these calls access disjoint pieces of the same matrix.

—BlockMul, a similar matrix multiply program, also uses a main recursive procedure with eight recursive calls. But it also dynamically allocates memory on stack (using the C library call `alloca`), to store temporary submatrices. At each invocation, the procedure accesses memory dynamically allocated in its own stack frame, but also accesses memory allocated in two other stack frames of the same procedure, at one and two levels up in the recursion. The compiler needs context sensitive analysis techniques to distinguish between the memory allocated in all these stack frames of the same procedure.

—The recursive structure in LU is significantly more complex. It uses six recursive procedures which invoke each other: four of them have mutually recursive calls, one has eight direct recursive calls, and the other, which is the main recursive procedure, has one direct recursive call and three calls to the other recursive procedures. All of these procedures access multiple regions within the same memory block. For the multithreaded version of LU, all of these recursive calls execute in parallel. This complex recursive

Table II. Absolute Speedups for Parallelized Programs

	Number of Processors				
Program	1	2	4	6	8
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

structure poses a significant challenge to a program analysis system if it is to accurately characterize the memory accesses for each of these procedures.

7.1 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 II 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.

7.2 Data Race Detection

The analysis verifies that the multithreaded versions of all programs except Knapsack are free of data races. The data race in Knapsack, a branch and bound algorithm, is used to prune the search space, as follows. The program uses an integer variable records the current best solution. When a thread finds a better solution, it updates this variable. During the search, parallel threads compare their partial solution to the current best; if a thread detect that the current best is better than its partial solution, it aborts its search. Hence, the current best is a shared variable concurrently read and written by parallel threads. Therefore, the data race in Knapsack is intentional and part of the algorithm design, but causes nondeterministic behavior.

7.3 Array Bounds Violations and Checks

The analysis was also able to verify that none of our benchmarks (including both the sequential and the multithreaded versions) violates the array bounds.

To get some idea of the magnitude of the array bounds check overhead, we manually added array bounds checks to the BlockedMul and the two sort programs. BlockedMul executes approximately 2.50 times slower with array bounds checks. Mergesort executes 1.14 times slower with array bounds checks, and Quicksort executes 1.09 times slower.

Table III. Bitwidth Analysis Results

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%

7.4 Bitwidth Analysis

Bitwidth analysis has recently been identified as a concrete value range problem [Stephenson et al. 2000]. 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 III presents experimental results for several programs presented in Stephenson et al. [2000]. We report reductions in two kinds of program state: register state, which holds scalar variables, and memory state, which holds array variables. These results show that our analysis is able to significantly reduce the number of bits required to hold the state of the program.

Our analysis extracts ranges from arithmetic operations and control flow only; it does not extract information from bitwise operations, it does not infer width information from the size of return types of procedures, and it does not infer bounds information by assuming that array accesses fall within bounds. Other bitwidth analyses [Budiu et al. 2000; Stephenson et al. 2000] use such techniques and are able to eliminate a larger number of bits. Nonetheless, our results show a substantial reduction in the number of bits, which makes our algorithm a good candidate for extracting bitwidth information.

7.5 Analysis Running Times

Table IV presents the running times of the pointer analysis and symbolic analysis phases for the divide and conquer benchmarks. These results were collected on a Pentium III at 450 MHz, running RedHat Linux. In general, the running times of the context-sensitive pointer analysis depend on the number of generated contexts. For our benchmarks, the running times of pointer analysis for the two matrix multiplication programs (BlockMul and NoTemp) are larger than those for the other programs, because the analysis generates multiple contexts in these cases. In contrast, the running times for the bounds and region analysis show smaller fluctuations across our set of benchmarks. We attribute this behavior to the fact that our symbolic treatment performs exactly one analysis per procedure and per recursive cycle.

Table IV. Running Times (in Seconds)

Program	Pointer Analysis	Bounds Analysis	Region Analysis
Quicksort	0.03	0.25	0.03
Mergesort	0.08	0.49	0.16
Heat	0.23	2.39	0.13
Knapsack	0.07	0.17	0.07
BlockMul	8.75	0.53	0.33
NoTemp	4.57	0.61	0.16
LU	0.39	1.09	0.35

As discussed in Section 5, the complexity of the bounds analysis is proportional to the number of basick blocks, that is, to the amount of intraprocedural control flow, while the complexity of the region analysis depends instead on the number of call sites in the program. Hence, the bounds analysis time is larger for programs with more complex intraprocedural control flow, such as Heat; the region analysis time is larger for programs with more procedure calls, such as LU; and, for each benchmark, the bounds analysis takes longer than the region analysis because programs usually have more complex intraprocedural control-flow than interprocedural control-flow (i.e., more branch instructions than call instructions).

8. RELATED WORK

We discuss the related work in the area of array bound checking elimination and value range analysis, which is the most relevant to the analysis presented in this paper. We will then present related work in the areas of data race detection and automatic parallelization.

8.1 Array Bound Checking Elimination and Value Range Analysis

There is a long history of research on array bound check elimination. A large part of the work in this area concentrated on detecting and eliminating *partially redundant* array bound checks [Markstein et al. 1982; Gupta 1990, 1993; Asuru 1992; Austin et al. 1994; Kolte and Wolfe 1995]. An array bound check is partially redundant if it is implied by another bound check during the execution of the program. The two dynamic checks can either be different statements in the program or can be different instances of the same statement in a loop (usually the check in the first iteration makes the checks in all the subsequent iterations redundant). Consequently, two major techniques have been developed for detecting partially redundant bound checks: removal of bound checking statements made redundant by other checking statements and hoisting bound checks out of the loops. These optimizations are essentially generalizations of common subexpression elimination and loop invariant code motion for array bound checks. All the techniques developed for discovering partially redundant checks are intraprocedural analyses; in particular, they are not able to detect partially redundant checks in recursive procedures.

Hence, the detection of partially redundant checks requires the compiler to analyze the relation between the bound checks in the program. In contrast, our

analysis is aimed at detecting and eliminating *fully redundant* array bound checks, where the compiler analyzes the computation in the program (not the checking statements) and determines if the array subscripts always fall within the array bounds, regardless of the other bound checks in the program. These two techniques are therefore orthogonal and can complement each other.

Researchers have attacked the problem of detecting fully redundant bound checks from many perspectives, including theorem proving, iterative value range analysis, constraint-based analyses, and type systems.

The techniques based on theorem proving generate, for each array access, the logic predicate necessary to ensure the safety of that access. The compiler then traverses the statements in the program backwards and derives the weakest preconditions that must be satisfied for the predicates to hold. If the compiler can prove that on each path to an array access there is a precondition which holds, then the bound check for that access is redundant and can be eliminated [Suzuki and Ishihata 1977]. Theorem proving is a powerful technique because it operates on the general domain of expressions in predicate logic. However, it is usually expensive and not guaranteed to terminate. For instance, the weakest preconditions may grow exponentially for a sequence of if statements and the computation of weakest preconditions for loops may not reach a fixed point. Therefore, the analysis has to artificially bound the number of iterations for loop constructs.

Value range analysis aims at determining the ranges of variables at each program point. The compiler can compare the computed ranges with the array bounds at each array access to determine if the access is fully redundant. The range information is also valuable for many other optimizations and verifications; for instance, range analysis was studied in the context of automatic parallelization of programs with affine array accesses in nested loops [Triolet et al. 1986; Balasundaram and Kennedy 1989b; Havlak and Kennedy 1991; Hall et al. 1995; Blume and Eigenmann 1995], automatic parallelization of recursive procedures [Gupta et al. 1999; Rugina and Rinard 1999a], static branch prediction [Patterson 1995], bitwidth reduction in C-to-hardware compilers [Stephenson et al. 2000; Budiu et al. 2000], or program debugging [Bourdoncle 1993]. Several other researchers have developed range analyses without specific target applications [Harrison 1977; Verbrugge et al. 1996]. All of these analyses compute value ranges using iterative approaches, such as dataflow analysis or abstract interpretation, over the lattice domain of integer or symbolic ranges. Because these lattices have infinite ascending chains, the iterative analysis of loops or recursive procedures is not guaranteed to terminate; to ensure termination, these analyses artificially limit the number of iterations or use imprecise widening operators. In contrast, the work presented in this article replaces these limited techniques with a clean, general framework which can successfully analyze arbitrary intra- and inter-procedural control flow and is guaranteed to terminate.

The constraint-based approaches for elimination of fully redundant bound checks are the most relevant for this article and were developed simultaneously with the work presented in this article [Wagner et al. 2000; Bodik et al. 2000]. They reduce the range computation problem to a constraint system consisting of linear inequalities and then solve the system to determine if the bound checks

are redundant. In the first approach [Wagner et al. 2000], the analysis derives derives a range constraint for each statement in the program. Each constraint is an inclusion between linear combinations of range variables. Because this work focuses the security vulnerabilities through string buffer overruns, the analysis derives special constraints for each string manipulation routine. The authors also present an algorithm for solving the range inclusion constraints; the algorithm is essentially equivalent to a linear program solver when all the bounds are finite. However, the range analysis presented in this work computes numeric ranges for variables, is intraprocedural, and is unsound. In contrast, our analysis is more general and more precise: it computes symbolic ranges, is flow-sensitive, is interprocedural, can handle recursive procedures, and is sound. The other constraint-based approach to array bound checking elimination [Bodik et al. 2000] captures the constraints between program variables in the form of an inequality graph, where the nodes represent program variables and the edges represent linear inequality constraints; an edge between variables x and y, labeled with integer constant c, encodes a linear inequality $y \leq x + c$. Hence, the graph represents a system of linear inequalities of this form. But instead of first solving the system to get bounds for the variables and then using these bounds to show that array accesses are within bounds, the analysis directly determines the safety of accesses from the constraint system, by reducing the problem to the shortest path problem: an array access $a[i]$ is within bounds if the weight of the shortest path between the array index node i and the array length node $a.length$ is less than zero. The analysis uses the SSA representation of the program (so it is flow-sensitive) and has extensions for detecting partially redundant checks. The main advantage of this algorithm is its simplicity and efficiency due to the sparse representation using the SSA form and the inequality graph. However, the analysis requires a special treatment of cyclic paths in the graph to ensure the termination of the shortest path computation—this technique is conceptually equivalent to the application of the widening operator in abstract interpretation. Also, the analysis is intraprocedural and we believe it requires complex modifications to be able to analyze the recursive divide-and-conquer programs presented in this paper. Finally, the analysis can only analyze assignments of constants and increments by constants; it cannot handle the assignment of other common linear expressions such as multiplication or division by constants and especially addition or subtraction of variables (because the inequality graph can only capture relations between two variables). The algorithm presented in this article doesn't suffer of any of these problems.

In all of the above analyses, the main difficulty is the extraction of the loop invariants and recursion invariants necessary to guarantee that array accesses are fully redundant. In general, invariant discovery is usually the main obstacle in any static analysis designed for checking arbitrary safety properties of programs. Recognizing this fact, researchers have recently developed a range of language extensions that facilitate the discovery or explicitly give these invariants. Examples of language extensions designed to facilitate the safety checking of memory accesses include preconditions and postconditions in the Eiffel language [Meyer 1992], dependent types [Xi and Pfenning 1998],

procedure annotations in a certifying compiler [Necula and Lee 1998], program annotations in the LCLint tool [Larochelle and Evans 2001], annotations in the Extended Static Checking system [Flanagan et al. 2002], and region types in the Vault language [DeLine and Fahndrich 2001] and in the Cyclone language [Grossman et al. 2002]. As language extensions, all of these techniques move the burden of invariant detection from the compiler to the programmer. In fact, programming systems could use our analysis to automatically derive some of the annotations in the above systems.

8.2 Race Detection

A data race occurs when two parallel threads access the same memory location without synchronization and one of the accesses is a write. Because data races are almost always the result of programmer error, many researchers have developed tools designed to find and help eliminate data races. Several approaches have been developed to attack this problem, including static program specifications and verifications [Sterling 1994; Detlefs et al. 1998], augmented type systems to enforce the synchronized access to shared data [Flanagan and Abadi 1999; Flanagan and Freund 2000; Boyapati and Rinard 2001; Flanagan and Qadeer 2003], and dynamic detection of data races based on program instrumentation [Steele 1990; Dinning and Schonberg 1991; Netzer and Miller 1991; Mellor-Crummey 1991; Min and Choi 1991; Savage et al. 1997; Cheng et al. 1998; Praun and Gross 2001; Choi et al. 2002; O'Callahan and Choi 2003; Pozniansky and Schuster 2003; Praun and Gross 2003]. The most relevant techniques are those that use program analysis to statically identify data races. Some of these techniques concentrate on the analysis of synchronization, and rely on the fact that detecting conflicting accesses is straightforward once the analysis determines which statements may execute concurrently [Taylor 1983; Balasundaram and Kennedy 1989a; Duesterwald and Soffa 1991]. Other analyses focus on parallel programs with affine array accesses in loops, and use techniques similar to those from data dependence analysis for sequential programs [Emrath and Padua 1988; Emrath et al. 1989; Callahan et al. 1990]. However, none of these analyses is able to statically detect data races in recursive programs where the recursive invocations access disjoint regions of the same array; or in multithreaded programs which manipulate shared pointers. To the best of our knowledge, the work presented in this article is the first attempt to address the static data race detection problem in recursive, multithreaded programs that manipulate shared pointers and shared arrays.

8.3 Parallelizing Compilers

Most of the previous research in parallelizing compilers has focused on parallelizing loop nests that access dense matrices using affine access functions. Such compilers [Hall et al. 1992, 1995; Blume et al. 1994] use range analysis for the array indices, as discussed in Section 8.1, or use data dependence analysis to determine if two array accesses in a loop nest may refer to the same location [Banerjee 1979, 1988; Wolfe 1982; Kong et al. 1990; Maydan et al. 1991; Goff et al. 1991; Pugh 1991]. The techniques presented in this article, on

the other hand, are designed for a more general class of programs, with arbitrary intraprocedural control flow, with recursive procedures, multiple threads, dynamic memory allocation, and pointer arithmetic.

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 [Chase et al. 1990; Ghiya and Hendren 1996; Sagiv et al. 1998, 2002]. 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 [Rinard and Diniz 1997]. It generates parallel code if all pairs of operations commute. We view both commutativity analysis and shape analysis as orthogonal to our analyses.

9. CONCLUSION

This article 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.

REFERENCES

AMARASINGHE, S., ANDERSON, J., LAM, M., AND TSENG, C. 1995. The SUIF compiler for scalable parallel machines. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing* (San Francisco, Calif.).

ASURU, J. 1992. Optimization of array subscript range checks. *ACM Lett. Prog. Lang. Syst. 1*, 2 (June), 109–118.

AUSTIN, T., BREACH, S., AND SOHI, G. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation* (Orlando, Fla.). ACM, New York.

BALASUNDARAM, V. AND KENNEDY, K. 1989a. Compile-time detection of race conditions in a parallel program. In *Proceedings of the 1989 ACM International Conference on Supercomputing* (Crete, Greece). ACM, New York.

BALASUNDARAM, V. AND KENNEDY, K. 1989b. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation* (Portland, Ore.). ACM, New York.

BANERJEE, U. 1979. Speedup of ordinary programs. Ph.D. dissertation. Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana-Champaign, Ill.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, Mass.

BLUME, W. AND EIGENMANN, R. 1995. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium* (Santa Barbara, Calif.).

BLUME, W., EIGENMANN, R., FAIGIN, K., GROUT, J., HOEFLINGER, J., PADUA, D., PETERSEN, P., POTTENGER, B., RAUCHWERGER, L., TU, P., AND WEATHERFORD, S. 1994. Polaris: The next generation in parallelizing compilers. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing* (Ithaca, N.Y.).

BLUMOFE, R., JOERG, C., KUSZMAUL, B., LEISERSON, C., RANDALL, K., AND ZHOU, Y. 1996. CILK: An efficient multithreaded runtime system. *J. Parall. Distrib. Comput. 37,* 1 (Aug.), 55–69.

BODIK, R., GUPTA, R., AND SARKAR, V. 2000. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation* (Vancouver, B.C., Canada). ACM, New York.

BOURDONCLE, F. 1993. Abstract debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation* (Albuquerque, N.M.). ACM, New York.

BOYAPATI, C. AND RINARD, M. 2001. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications* (Tampa Bay, Fla.).

BUDIU, M., GOLDSTEIN, S., SAKR, M., AND WALKER, K. 2000. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing* (Munich, Germany).

CALLAHAN, D., KENNEDY, K., AND SUBHLOK, J. 1990. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seattle, Wash.). ACM, New York.

CHASE, D., WEGMAN, M., AND ZADEK, F. 1990. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation* (White Plains, N.Y.). ACM, New York.

CHATTERJEE, S., LEBECK, A., PATNALA, P., AND THOTTETHODI, M. 1999. Recursive array layouts and fast matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures* (Saint Malo, France). ACM, New York.

CHENG, G., FENG, M., LEISERSON, C., RANDALL, K., AND STARK, A. 1998. Detecting data races in CILK programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico). ACM, New York.

CHOI, J.-D., LEE, K., LOGINOV, A., O'CALLAHAN, R., SARKAR, V., AND SRIDHARAN, M. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation* (Berlin, Germany).

DELINE, R. AND FAHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation* (Snowbird, Ut.).

DETLEFS, D., LEINO, K. R., NELSON, G., AND SAXE, J. 1998. Extended static checking. Tech. Rep. 159, Compaq Systems Research Center.

DINNING, A. AND SCHONBERG, E. 1991. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (Santa Cruz, Calif.). ACM, New York.

DUESTERWALD, E. AND SOFFA, M. 1991. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the 1991 International Symposium on Software Testing and Analysis* (Victoria, B.C., Canada). ACM, New York.

EMRATH, P., GHOSH, S., AND PADUA, D. 1989. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '89* (Reno, Nev.).

EMRATH, P. AND PADUA, D. 1988. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (Madison, Wisc.).

FLANAGAN, C. AND ABADI, M. 1999. Types for safe locking. In *Proceedings of the 1999 European Symposium on Programming*. Amsterdam, The Netherlands.

FLANAGAN, C. AND FREUND, S. 2000. Type-based race detection for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation* (Vancouver, B.C., Canada). ACM, New York.

FLANAGAN, C., LEINO, R., LILLIBRIDGE, M., NELSON, G., SAXE, J., AND STATA, R. 2002. Extended static checking for Java. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation* (Berlin, Germany). ACM, New York.

FLANAGAN, C. AND QADEER, S. 2003. A type and effect system for atomicity. In *Proceedings of the SIGPLAN '03 Conference on Program Language Design and Implementation* (San Diego, Calif.). ACM, New York.

FRENS, J. AND WISE, D. 1997. 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, Nev.). ACM, New York.

FRIGO, M., LEISERSON, C., AND RANDALL, K. 1998. The implementation of the CILK-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation* (Montreal, Que., Canada). ACM, New York.

GHIYA, R. AND HENDREN, L. 1996. 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* (St. Petersburg Beach, Fla.). ACM, New York.

GOFF, G., KENNEDY, K., AND TSENG, C. 1991. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation* (Toronto, Ont., Canada). ACM, New York.

GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. 2002. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation* (Berlin, Germany). ACM, New York.

GUPTA, M., MUKHOPADHYAY, S., AND SINHA, N. 1999. Automatic parallelization of recursive procedures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '99)* (Newport Beach, Calif.).

GUPTA, R. 1990. A fresh look at optimizing array bound checking. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation* (White Plains, N.Y.). ACM, New York.

GUPTA, R. 1993. Optimizing array bound checks using flow analysis. *ACM Lett. Prog. Lang. Syst. 2*, 1–4 (Mar.-Dec.), 135–150.

GUSTAVSON, F. 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Devel. 41*, 6 (Nov.), 737–755.

HALL, M., AMARASINGHE, S., MURPHY, B., LIAO, S., AND LAM, M. 1995. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95* (San Diego, Calif.).

HALL, M. W., HIRANANDANI, S., KENNEDY, K., AND TSENG, C. 1992. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92* (Minneapolis, Minn.).

HARRISON, W. H. 1977. Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng. SE-3*, 3 (May), 243–250.

HAVLAK, P. AND KENNEDY, K. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. Paral. Distrib. Syst. 2*, 3 (July), 350–360.

KOLTE, P. AND WOLFE, M. 1995. Elimination of redundant array subscript range checks. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation* (San Diego, Calif.). ACM, New York.

KONG, X., KLAPPHOLZ, D., AND PSARRIS, K. 1990. The I test: A new test for subscript data dependence. In *Proceedings of the 1990 International Conference on Parallel Processing* (St. Charles, Ill.).

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium* (Washington, D.C.).

MARKSTEIN, V., COCKE, J., AND MARKSTEIN, P. 1982. Optimization of range checking. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, Mass.). ACM, New York.

MAYDAN, D., HENNESSY, J., AND LAM, M. 1991. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation* (Toronto, Ont., Canada). ACM, New York.

MELLOR-CRUMMEY, J. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing '91* (Albuquerque, N.M.).

MEYER, B. 1992. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, N.J.

MIN, S. AND CHOI, J. 1991. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Williamsburg, Va.). ACM, New York.

NECULA, G. AND LEE, P. 1998. The design and implementation of a certifying compiler. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation* (Montreal, Que., Canada). ACM, New York.

NETZER, R. AND MILLER, B. 1991. Improving the accuracy of data race detection. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Williamsburg, Va.). ACM, New York.

NIELSON, F., NIELSON, H., AND HANKIN, C. 1999. *Principles of Program Analysis*. Springer-Verlag, Heidelberg, Germany.

O'CALLAHAN, R. AND CHOI, J.-D. 2003. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, Calif.). ACM, New York.

PATTERSON, J. 1995. Accurate static branch prediction by value range propagation. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation* (San Diego, Calif.). ACM, New York.

POZNIANSKY, E. AND SCHUSTER, A. 2003. Efficient on-the-fly data race detection in multihreaded C++ programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, Calif.). ACM, New York.

PRAUN, C. AND GROSS, T. 2001. Object race detection. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications* (Tampa Bay, Fla.).

PRAUN, C. AND GROSS, T. 2003. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the SIGPLAN '03 Conference on Program Language Design and Implementation* (San Diego, Calif.). ACM, New York.

PUGH, W. 1991. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91* (Albuquerque, N.M.).

RINARD, M. AND DINIZ, P. 1997. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Prog. Lang. Syst. 19*, 6 (Nov.), 941–992.

RUGINA, R. AND RINARD, M. 1999a. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Atlanta, Ga.). ACM, New York.

RUGINA, R. AND RINARD, M. 1999b. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation* (Atlanta, Ga.). ACM, New York.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst. 20*, 1 (Jan.), 1–50.

SAGIV, M., REPS, T., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst. 24*, 3 (May).

SAVAGE, S., BURROWS, M., NELSON, G., SOLBOVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic race detector for multi-threaded programs. *ACM Trans. Comput. Syst. 15*, 4, 391–411.

STEELE, G. 1990. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Programming Languages* (San Francisco, Calif.). ACM, New York.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages* (St. Petersburg Beach, Fla.). ACM, New York.

STEPHENSON, M., BABB, J., AND AMARASINGHE, S. 2000. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation* (Vancouver, B.C., Canada). ACM, New York.

STERLING, N. 1994. Warlock: A static data race analysis tool. In *Proceedings of the 1993 Winter* USENIX *Conference* (San Diego, Calif.).

SUZUKI, N. AND ISHIHATA, K. 1977. Implementation of an array bound checker. In *Conference Record of the 4th Annual ACM Symposium on the Principles of Programming Languages* (Los Angeles, Calif.). ACM, New York.

TAYLOR, R. N. 1983. A general purpose algorithm for analyzing concurrent programs. *Commun. ACM 26*, 5 (May), 362–376.

TRIOLET, R., IRIGOIN, F., AND FEAUTRIER, P. 1986. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (Palo Alto, Calif.). ACM, New York.

VERBRUGGE, C., CO, P., AND HENDREN, L. 1996. Generalized constant propagation: A study in C. In *Proceedings of the 1996 International Conference on Compiler Construction* (Linköping, Sweden).

WAGNER, D., FOSTER, J., BREWER, E., AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium* (San Diego, Calif.).

WILSON, R. AND LAM, M. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation* (La Jolla, Calif.). ACM, New York.

WOLFE, M. J. 1982. Optimizing supercompilers for supercomputers. Ph.D. dissertation. Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign.

XI, H. AND PFENNING, F. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation* (Montreal, Que., Canada). ACM, New York.

Received February 2001; revised July 2002; accepted June 2004