Interprocedural Symbolic Range Propagation for Optimizing Compilers [*]

Hansang Bae and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
{baeh,eigenman}@purdue.edu

Abstract. We have designed and implemented an interprocedural algorithm to analyze symbolic value ranges that can be assumed by variables at any given point in a program. Our algorithm contrasts with related work on interprocedural value range analysis in that it extends the ability to handle symbolic range expressions. It builds on our previous work of intraprocedural symbolic range analysis. We have evaluated our algorithm using 11 Perfect Benchmarks and 10 SPEC floating-point benchmarks of the CPU 95 and CPU 2000 suites. We have measured the ability to perform test elision, dead code elimination, and detect data dependences. We have also evaluated the algorithm's ability to help detect zero-trip loops for induction variable substitution and subscript ranges for array reductions.

1 Introduction

The motivation for the present work is the pursuit of the long-term goal of developing higher-level programming languages. At the same time, we aim to increase the power of the present generation of optimizing compilers. One thrust towards these goals is to strengthen the capabilities of compilers in reasoning about and manipulating program sections in symbolic terms. Developing an algorithm for interprocedural range propagation is a small step in this direction. By knowing the value range that a variable may assume at any given program point, compiler techniques can make more informed optimization decisions. We have developed and used such techniques in the past for our Polaris parallelizing compiler [2, 13, 4]. Knowing symbolic value ranges has become key to detecting data dependences, privatizing variables, substituting induction variables, and parallelizing reduction operations. Polaris' Range Test [5] makes use of advanced symbolic expression manipulation capabilities, which exploit knowledge about possible value ranges of program variables. The privatization pass [17] is able to analyze and comprehend the meaning of certain compute patterns. The induction variable and reduction recognition passes [15] exploit value range information to prove zero-trip loops and to narrow array subscript ranges, respectively. In

[*] This work is supported in part by the National Science Foundation under Grants No. 0103582-EIA, and 0429535-CCF.

```
X = 1              X = [-INF,INF]
IF (X.LE.N) THEN   X = [1,1]
  X = 2*X          X = [1,1]
ELSE               X = [2,2]
  X = X+2          X = [MAX(1,1+N),1]
ENDIF              X = [MAX(3,3+N),3]
...                X = [2,3]
```

Fig. 1. Intraprocedural symbolic range propagation

all these cases, value range information significantly boosts Polaris' ability to detect parallelism. Currently, Polaris uses only *intraprocedural* range analysis. It operates in concert with interprocedural expression propagation and forward substitution, which we consider the best alternative to the new techniques and reference point for the evaluation section. In Section 4 we will explain why we have chosen this reference point over other related contributions. Briefly, related work either focuses on interprocedural expression propagation [11], where no range representation is used, or restricts the bounds of ranges to simple expressions [14]. A number of approaches have also considered range information in the context of pointer analysis for C-type languages.

2 Interprocedural Symbolic Range Propagation (ISRP)

Our framework for interprocedural analysis follows the classical fixed-point approach based on abstract interpretation, and uses existing intraprocedural range analysis techniques as the source of range information to propagate. During the analysis, ISRP collects symbolic range information within a subroutine and generates interprocedural ranges at every important program point – at subroutine returns and at call sites. Then it propagates the collected data towards the leaves of the call graph. This is similar to the jump function approach [9] and we use terms in that approach to present our algorithm[1]. We also apply procedure-cloning to enhance the accuracy of the analysis in every calling context. Before introducing our algorithm in detail, we briefly review our existing framework for intraprocedural symbolic range analysis.

The goal of symbolic range propagation [3] is to collect a valid set of symbolic value ranges for each variable at every program point. The collected ranges can be used for advanced compiler analyses such as data dependence testing, dead-code elimination, and program verification – the main application of the analysis in the current Polaris compiler is a nonlinear symbolic data dependence

[1] In contrast to [9], our jump function is defined for the value ranges of all variables of a statement, not for a single variable.

test [5]. Figure 1 gives an example of the value ranges for program variable X, valid before each statement, as analyzed by the existing intraprocedural analysis technique. To derive this information, ranges defined by individual statements are intersected along the control flow. At control flow merge points unions of ranges are computed. The ranges so analyzed supply the source of candidate interprocedural symbolic ranges for ISRP.

In presenting the ISRP algorithm, we make use of the following terms.

Definition 1. *A* symbolic range *is a mapping from a variable to its value range, V = [LB,UB], where LB is the lower bound and the UB is the upper bound. All variables contained in the expressions* {*V,LB,UB*} *belong to the scope of the subroutine enclosing the program point being analyzed. If either LB or UB is infinite, we call it an* open range*, otherwise it is a* closed range*.*

Definition 2. *An* interprocedural (symbolic) range *is a symbolic range that is gathered from information in one subroutine and inserted at relevant points in another subroutine – either at the subroutine entry or after a call statement.*

Definition 3. *For a caller P, a callee Q, and a call site S that calls Q in P, the* jump function *at S is the set of known symbolic ranges before S, expressed in terms of input variables to Q (actual parameters and global variables).*

Definition 4. *The return jump function at Q is the set of known symbolic ranges at the end of Q, expressed in terms of return variables to P (return parameters, reference parameters and global variables).*

Without loss of generality we use only integer type and logical type. The existing, intraprocedural framework did not allow logical types, but we added it to make our analysis more general. If the type of *V* is logical, *LB* should be equal to *UB*. Notice, that the jump functions express symbolic ranges in terms of variable names in the present subroutine. In order to create interprocedural ranges, these names may need to be changed to those used in the target subroutine. This also applies for the return jump functions.

Figure 2 gives a high-level description of our algorithm. It uses the following functions:

`Compute_Jump_Functions()`
This routine produces intraprocedural value ranges that are valid at each program point in a subroutine. The source of information are ranges as analyzed by the intraprocedural range analysis algorithms plus the inserted interprocedural ranges (at the beginning of the subroutine and after each call statement). The interprocedural ranges collected from `Get_Backward_Interprocedural_Ranges` are intersected with existing ranges before the call site. However, when a certain variable is modified in the callee or its descendants in the call graph, the previous range for that variable is discarded. Note that the resulting jump functions, as per Definition 3 are expressed in terms of variables in the current subroutine. In

```
Propagate_Interprocedural_Ranges()
{
  Initialize_Call_Graph()
  while (there is any change in interprocedural ranges)
  {
    foreach Subroutine (reverse topologically)
    {
      Get_Backward_Interprocedural_Ranges()
      Compute_Jump_Functions()
      Compute_Return_Jump_Functions()
    }
    Get_Forward_Interprocedural_Ranges()
  }
}
```

Fig. 2. Algorithm for interprocedural symbolic range propagation.

order to create interprocedural ranges, renaming of actual to formal parameters will need to be performed (by `Get_Forward_Interprocedural_Ranges()`).

`Compute_Return_Jump_Functions()`
This routine generates value ranges as per Definition 4 at the return point of the current subroutine. Information at multiple return points is merged – for simplicity, we just take unions.

`Get_Forward_Interprocedural_Ranges()`
This routine creates the interprocedural symbolic value ranges valid at the entry point of each subroutine – this is done by applying jump functions to each subroutine, appropriately converting actual parameters to formal parameters. In addition, this step performs procedure-cloning for different calling contexts.

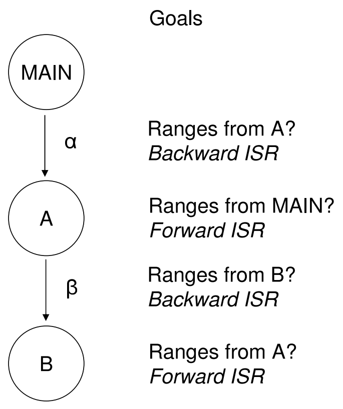
`Get_Backward_Interprocedural_Ranges()`
This routine propagates interprocedural symbolic value ranges backward from all callees to the current subroutine. It uses information from return jump functions in the callees. If a value range contains a formal parameter of the callee, it is mapped to the corresponding actual parameter. Note that the algorithm performs this step reverse topologically. Hence, intraprocedural analysis for each callee has already been done, including the generation of return jump functions. For leaves in the call graph, this subroutine performs no action. This reverse order is for speeding up a single iterative step, and in-order traversal would converge to the same solution eventually.

In summary, the algorithm computes the interprocedural symbolic ranges propagated backward from callees' contexts and ranges propagated forward from

```
  PROGRAM MAIN
  INTEGER X, Y
  X = 1
  Y = 2
α CALL A(X, Y)
  END

  SUBROUTINE A(T, U)
  INTEGER N, T, U
  DO N = 10, 40
β  CALL B(T, U, N)
  ENDDO
  END

  SUBROUTINE B(V, W, M)
  INTEGER M, V, W
  V = W + M
  END
```



(a)

1st iteration

Subroutine	Forward ISR	Return Jump	Callsite	Jump	Backward ISR
B	-	V=[W+M,W+M]	-	-	-
A	-	ϕ	β	N=[10,40]	T=[U+N,U+N]
MAIN	-	-	α	X=[1,1],Y=[2,2]	ϕ
A	T=[1,1],U=[2,2]	-	β	-	T=[U+N,U+N]
B	M=[10,40]	-	-	-	-

2nd iteration

Subroutine	Forward ISR	Return Jump	Callsite	Jump	Backward ISR
B	M=[10,40]	V=[W+M,W+M],M=[10,40]	-	-	-
A	T=[1,1],U=[2,2]	U=[2,2]	β	N=[10,40],U=[2,2]	T=[U+N,U+N],N=[10,40]
MAIN	-	-	α	X=[1,1],Y=[2,2]	Y=[2,2]
A	T=[1,1],U=[2,2]	-	β	-	T=[U+N,U+N],N=[10,40]
B	M=[10,40],W=[2,2]	-	-	-	-

3rd iteration

Subroutine	Forward ISR	Return Jump	Callsite	Jump	Backward ISR
B	M=[10,40],W=[2,2]	V=[W+M,W+M],M=[10,40] W=[2,2]	-	-	-
A	T=[1,1],U=[2,2]	U=[2,2]	β	N=[10,40],U=[2,2]	T=[U+N,U+N],N=[10,40] U=[2,2]
MAIN	-	-	α	X=[1,1],Y=[2,2]	Y=[2,2]
A	T=[1,1],U=[2,2]	-	β	-	T=[U+N,U+N],N=[10,40] U=[2,2]
B	M=[10,40],W=[2,2]	-	-	-	-

(b)

Fig. 3. Interprocedural symbolic range propagation on an example code. (a) An example program with three program units and two call sites. The goal is to compute backward/forward Interprocedural Symbolic Ranges (ISRs) at the entry to each subroutine and at the call sites. (b) Step-by-step process of ISRP on the code. Each row is completed in a single step by the algorithm in Figure 2.

callers' contexts until it reaches a fixed point. Each iteration performs range analysis within a subroutine, selects valid symbolic ranges across subroutine boundaries, and feeds those new data into the intraprocedural range analysis for the next iterative step.

The example illustrated in Figure 3 shows a simple program with three program units and two subroutine calls, and interprocedural symbolic range propagation on that program. The goal of the analysis is to compute Interprocedural Symbolic Ranges (ISR) that are valid at the entry to each subroutine and after

Table 1. Benchmark suite.

Code	Size	Subroutines	Call sites	Code	Size	Subroutines	Call sites
ARC2D	4650	36	100	applu	3868	13	26
BDNA	4843	38	162	apsi	7361	66	328
DYFESM	8446	57	204	fpppp	2784	13	52
FLO52Q	2324	27	86	hydro2d	4292	39	200
MDG	1430	12	42	mgrid	484	11	46
MIGRATION	3455	23	110	su2cor	2332	26	242
OCEAN	3198	34	490	swim	429	6	10
QCD2	2816	30	166	tomcatv	190	1	0
SPEC77	4870	39	232	turb3d	2101	19	206
TRACK	4628	29	106	wupwise	2184	22	284
TRFD	580	4	20				

the call sites. In other words, the analysis collects forward ISR at the entry of each subroutine and backward ISR for each call site as shown in Figure 3(a). Figure 3(b) presents step-by-step process of ISRP on the code example. Each column shows the result after performing each step in the algorithm in Figure 2. For example, during the first iteration, the analysis computes return jump function for B ($V = [W + M, W + M]$), backward ISR for β ($T = [U + N, U + N]$), jump function for β ($N = [10, 40]$), return jump function for A (ϕ), backward ISR for α (ϕ), and jump function for α ($X = [1, 1], Y = [2, 2]$) successively. Then it finally computes forward ISRs for A and B, which come directly from the jump functions for α and β after converting actual parameters into formal parameters. The resulting forward ISRs have changed since the start of the iteration, which triggers next iteration and the analysis continues with a new set of initial information. The analysis finally stops after third iteration where the starting forward ISRs are identical to the resulting forward ISRs.

3 Experiments

We have measured the effectiveness of the presented interprocedural symbolic range propagation algorithm on several aspects of optimizing compilers – data dependence analysis, test elision with dead code elimination, and other optimizations for automatic parallelization. We implemented our analysis in the Polaris parallelizing compiler and our reference point, to which we refer as *Base*, is the performance of the current version of Polaris with full optimization. That includes intraprocedural symbolic range propagation, interprocedural expression propagation with forward substitution, automatic partial inlining, and procedure cloning. The existing constant propagation pass also removes unreachable code sections due to control flows resolved at compile time. We switched this function off and implemented a stand-alone pass that can interface with the range information.

Table 2. The number of test elision and dead code elimination.

Codes	Base	ISRP	Codes	Base	ISRP
ARC2D	4	5	applu	4	4
BDNA	15	15	apsi	1	18
DYFESM	18	26	fpppp	12	7
FLO52Q	2	5	hydro2d	9	9
MDG	0	1	mgrid	0	0
MIGRATION	4	6	su2cor	7	8
OCEAN	67	72	swim	0	0
QCD2	0	3	tomcatv	0	0
SPEC77	3	3	turb3d	5	15
TRACK	4	10	wupwise	19	27
TRFD	2	8			

Another feature of our ISRP implementation is a substitution pass that substitutes a variable with its corresponding symbolic expression. This capability builds the interface with existing compiler passes that do not have the ability to query range information. For this substitution, we used simple decision heuristics to avoid unwanted chains of forward substitutions generating large expressions (a drawback of the current constant propagation and forward substitution technique). For example, replacing a variable with a known numeric value is always preferred, but replacing loop variables (indices, bounds) with complex expressions is not.

3.1 Benchmark Suite

We selected 21 scientific engineering codes from the Perfect Benchmarks, SPEC CPU95 floating point, and SPEC CPU2000 floating point suites. This set of codes includes most of the Fortran 77 codes in each benchmark suite except for some codes that fail to compile (for reasons other than ISRP). Table 1 shows the feature of each benchmark, such as code size, number of subroutines or function calls, and number of call sites. Our algorithm converts all function calls to subroutine calls as a preliminary step.

3.2 Test Elision and Dead Code Elimination

Test elision and dead code elimination are optimizations that can benefit from static analysis such as constant propagation and range propagation. As more information about conditions is known, the compiler may prove that the test condition is always true or always false and thus eliminate one branch. We measured how the interprocedural range information affects the compile-time resolution of branches and corresponding dead code elimination.

Table 2 presents the number of successfully resolved branches with the Base method and with ISRP. For a fair comparison, we only counted one instance

Table 3. The number of data dependence arcs.

Codes	Base	ISRP	Codes	Base	ISRP
ARC2D	801	459	applu	677	677
BDNA	1124	1081	apsi	11395	8222
DYFESM	1388	1108	fpppp	22064	20006
FLO52Q	263	279	hydro2d	110	66
MDG	1120	908	mgrid	83	19
MIGRATION	7180	6219	su2cor	10915	8712
OCEAN	433	370	swim	0	0
QCD2	17257	3579	tomcatv	45	45
SPEC77	2392	1652	turb3d	371	342
TRACK	2003	1781	wupwise	2013	1071
TRFD	93	40			

of cloned procedures. The results show that the number of statically resolved branches with ISRP is greater than or equal to that with Base for all benchmark codes except for `fpppp`. `Fpppp` contains a pattern that benefits from repeated information propagation and deadcode elimination, which is performed by the Base technique but not by ISRP. The type of information to be propagated is simple, however, and done equally well by both techniques.

Overall, Table 2 shows that ISRP provides more static information than Base and (except for the minor case in `fpppp`) subsumes the Base techniques.

3.3 Data Dependence Analysis

Data dependence analysis is of obvious importance in optimizing compilers. We counted the number of dependence arcs in each benchmark code with Base and with ISRP to see how effective ISRP is in breaking data dependence arcs.

Table 3 shows the resulting numbers for each benchmark code. ISRP reduced the number of dependence arcs up to 79% for all benchmark codes except for `FLO52Q`. The increased number in `FLO52Q` is due to limitations of our simple forward substitution heuristics. Those variables were marked as private variables with Base whereas they carry cross-iteration dependences with ISRP. Another observation is that the total number of dependence pairs to disprove is increased for some codes and decreased for other codes – numbers are not presented here. Limited forward substitution accounts for the former case and better test elision accounts for the latter case. However, in both cases, the number of data dependence arcs decreased for most codes. This shows that limited forward substitution is not a significant factor to achieve accuracy in the data dependence analysis.

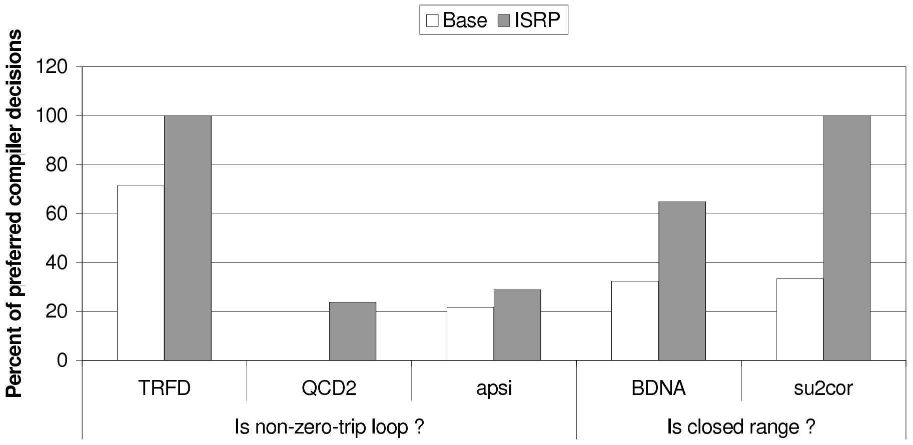


Fig. 4. Compiler's decisions for relevant questions with ISRP. If a loop is a non-zero-trip loop, the induction variable substitution pass can safely transform the code. If there is a closed subscript range for an array reduction, the compiler generates more efficient code for the reduction.

3.4 Detecting Zero-trip Loops and Array Bounds

Our parallelizing compiler additionally makes use of the collected range information when making decisions in several parallelization passes. One such case is the induction variable substitution pass, which tries to decide if a loop is a zero-trip loop. Not knowing that a loop is non-zero-trip may prevent the substitution of an induction variable with its closed form [15]. The Polaris compiler inserts a runtime test in this case. ISRP can potentially eliminate this runtime overhead.

Another pass benefiting from range analysis is the reduction parallelization technique. In absence of accurate information about index ranges used by an array reduction pattern, the compiler must consider the entire array a potential reduction variable. This conservative measure cost significant runtime overhead. Again, ISRP has the potential to reduce this cost.

Figure 4 shows the percentage of preferred compiler's decisions in those passes with Base and with ISRP. In the benchmarks not listed here, we did not find any significant differences. ISRP substantially increased the number of desirable decisions for the codes in Figure 4. The compiler could answer all the questions in favor of each optimization for `TRFD` and `su2cor`. In `TRFD`, a significant code section was statically parallelized with ISRP whereas Base relied on a run-time test.

4 Related Work

Range analysis in imperative programming languages has been addressed in several contexts over the last few decades, most of them stemming from a formal foundation – *Abstract Interpretation*[6, 7]. One of the major concerns of early work was how to make the analysis reach a fixed point at reasonable speed in the presence of loop-like program structures, and widening and narrowing[6] were then introduced to guarantee termination of the analysis. The demand for whole program analysis has also emerged and interprocedural analysis has become a key enabler of compiler optimizations in many contexts. The importance and effectiveness of range analysis or symbolic analysis have also been addressed in several contributions.

Havlak's work[11] served as an infrastructure for interprocedural symbolic analysis in the Parascope compilation system[1]. He divided a symbolic interprocedural analysis problem into four sub-problems, depending on if the analysis propagates symbolic values for variables or predicates, and if the information is passed to or returned from the callee. Two of the problems, returned values and passed predicates (linear equalities) were evaluated in his work. Our approach differs in two important regards. First, Havlak's work focuses on symbolic expression propagation, as opposed to value ranges. Second, our work is more general in that it can give solutions to all the four sub-problems. Before and after a call site, each symbolic value range with the same lower bound and the upper bound gives a set of passed expressions and returned expressions. Symbolic lower bounds and upper bounds can be used to infer a valid relationship between two expressions or variables. Including this work, interprocedural symbolic analysis was also applied in analyzing array accesses [8, 10] to be used for interprocedural parallelization and other optimizations. Although analyzing array subscripts is a major application of our framework, ISRP has more flexibility that enables many other potential optimizations.

Patterson[14] adopted value range propagation to statically predict if a certain branch is taken or not. The range representation used in his work carries the probability that a variable has a certain lower bound, an upper bound, and a stride. Because the analysis is intended for a single optimization, static branch prediction, he limited the complexity of the problem so that the analysis can trade-off accuracy and efficiency. For example, the interprocedural analysis only concerns about propagation through parameter mappings, and symbolic expressions for the value ranges can have at most one variable, which greatly simplifies the problem. Our analysis is intended for general use in several compiler optimizations and considers arbitrary symbolic expressions.

There are also efforts that adopt range analysis in the C language because of its applicability for non-numerical programs. While tackling issues that arise in C-type languages (primarily pointer analysis) these approaches have not shown or claimed progress for high-performance computing applications. The work by Verbrugge et. al.[18] expressed range analysis as Generalized Constant Propagation (GCP) and implemented it in the McCAT optimizing/parallelizing compiler[12]. They used the concept of an invocation graph that maintains context-sensitive information, and also utilized points-to information and read/write sets to minimize the loss of information during interprocedural analysis. They also introduced "stepping", which is a variation of widening and narrowing, to guarantee finite fixed-point iterations. The use of the invocation graph is similar to procedure cloning in that it maintains context-sensitive special information for each invocation of a function. One limitation of their work is that it only handles non-symbolic ranges.

Rugina's work[16] on symbolic bounds analysis took a different approach to achieve a similar goal. He did not adopt conventional concepts such as abstract interpretation and fixed-point algorithm. Instead, he set up a system of constraints within a region of interest and introduced a way of reducing the constraint system to a linear program under an assumption that the positivity of each coefficient is known. A framework for interprocedural analysis was also introduced, which describes mapping and unmapping actions at call sites. To avoid fixed-point iteration for recursive calls, he introduced a method of building a system of recursive constraints. The idea of not doing fixed-point iteration is an outstanding feature compared with other related work. This feature may improve the efficiency of the analysis but it left unclear how to compare the accuracy of this technique with that of a conventional fixed-point technique including ours.

The most recent work by Yong and Horwitz[19] also adopted range analysis to compute a safe approximation of the set of memory locations that may be accessed by each pointer dereference. To simplify the problem, they treated all memory accesses as pointer dereferences even for a scalar variable. Their work was focused on dealing with language-specific challenges such as pointer arithmetic and type mismatch due to union and casting by introducing advanced range description methods that embeds type information. Like other conventional techniques, they adopted the concept of widening and narrowing for convergence but their interprocedural analysis does not handle context-sensitive information and symbolic ranges.

5 Conclusion

We have designed and implemented an interprocedural symbolic range analysis technique and have shown that the resulting compiler pass substantially enhances the accuracy of other optimizations. The reference point we have chosen in our evaluation of 21 science/engineering benchmarks is the combination of interprocedural expression propagation, intraprocedural symbolic range analysis, forward substitution, and automatic partial inlining, as currently implemented in the Polaris parallelizing compiler. We believe this to be the best state-of-the-art symbolic range analysis framework for high-performance computing applications, among related contributions.

ISRP is an enabling technique for other optimizations. We can expect substantial performance improvement once we enhance existing optimization passes to take advantage of the new information. As is, we have already found 14 more parallel loops in our benchmark codes.

Advanced program analysis comes at the cost of longer compilation time. We have measured up to 150% increased compilation time in all but two cases.

Such increase seems acceptable, given the benefits and ever-increasing processor speeds. It is known that symbolic range analysis has exponential worst-case complexity [3], which explains substantial increases in compilation time in two of our codes – `OCEAN` and `TRACK`, which have a large number of call sites. In ongoing work we are considering optimizations of the algorithm to improve such behavior.

References

1. V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The parascope editor: an interactive parallel programming tool. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 540–550, New York, NY, USA, 1989. ACM Press.
2. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
3. William Blume and Rudolf Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, Santa Barbara, CA, April 1995.
4. William Blume and Rudolf Eigenmann. Demand-driven, Symbolic Range Propagation. *Lecture Notes in Computer Science, 1033: Languages and Compilers for Parallel Computing*, pages 141–160, 1996.
5. William Blume and Rudolf Eigenmann. Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1180–1194, December 1998.
6. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd Internatioal Symposium on Programming*, pages 106–130, April 1976.
7. Patrick Cousot and Rhadia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th ACM Symposium*, pages 238–252, 1977.
8. Béatrice Creusillet and Francois Irigoin. Interprocedural Array Region Analyses. In *Eighth International Workshop on Languages and Compilers for Parallel Computing (LCPC'95)*, pages 4–1 to 4–15, August 1995.
9. Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–99, 1993.
10. Mary W. Hall, Brian R. Murphy, Saman P. Amarasinghe, Shih-Wei Liao, and Monica S. Lam. Interprocedural analysis for parallelization. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 61–80, London, UK, 1996. Springer-Verlag.
11. Paul Havlak. *Interprocedural Symbolic Analysis.* PhD thesis, Dept. of Computer Science, Rice University, May 1994.
12. Laurie J. Hendren, C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. Designing the mccat compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420, London, UK, 1993. Springer-Verlag.

13. Seuing-Jai Min, Seon Wook Kim, Michael Voss, Sang-Ik Lee, and Rudolf Eigenmann. Portable compilers for OpenMP. In *OpenMP Shared-Memory Parallel Programming*, Lecture Notes in Computer Science #2104, pages 11–19, Springer Verlag, Heidelberg, Germany, July 2001.
14. Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the conference on Programming language design and implementation*, pages 67–78. ACM Press, 1995.
15. William M. Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, pages 444–448, 1995.
16. Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, Canada, June 2000.
17. Peng Tu and David Padua. Array privatization for shared and distributed memory machines (extended abstract). *SIGPLAN Not.*, 28(1):64–67, 1993.
18. Clark Verbrugge, Phong Co, and Laurie J. Hendren. Generalized constant propagation: A study in c. In *Proceedings of the Internatioal Conference on Compiler Construction*, pages 74–90, April 1996.
19. Suan Hsi Yong and Susan Horwitz. Pointer-range analysis. In *Proceedings of the 11th International Static Analysis Symposium (SAS '04)*, page 16 pages, August 2004.