International Journal of Parallel Programming (2022) 50:65–88
https://doi.org/10.1007/s10766-021-00722-1

Enhancing the Effectiveness of Inlining in Automatic Parallelization

Jichi Guo[1] · Qing Yi[1] · Kleanthis Psarris[2]

Received: 24 April 2021 / Accepted: 19 June 2021 / Published online: 6 August 2021

Abstract
The emergence of multi-core architectures makes it essential for optimizing compilers to automatically extract parallelism for large scientific applications composed of many subroutines residing in different files. Inlining is a well-known technique which can be used to erase procedural boundaries and enable more aggressive loop parallelization. However, conventional inlining cannot be applied to external libraries where the source code is not available, and when overly applied, it can degrade the effectiveness of compiler optimizations due to excessive code complexity. This paper highlights some obstacles we encountered while applying conventional inlining combined with automatic loop parallelization using the Polaris optimizing compiler and presents a new approach, annotation-based inlining, to effectively overcome these obstacles. Our experimental results show that the annotation-based inlining approach can eliminate negative impact of conventional inlining while enhancing the effectiveness of interprocedural parallelization for a majority of applications from the PERFECT benchmark suite.

Keywords Inlining · Automatic parallelization · Dependence analysis · Compiler optimization

✉ Kleanthis Psarris
kpsarris@brooklyn.cuny.edu

Qing Yi
qyi@uccs.edu

[1] Department of Computer Science, University of Colorado Colorado Springs, Colorado Springs, CO 80918, USA

[2] Department of Computer and Information Science, City University of New York - Brooklyn College, Brooklyn, NY 11210, USA

1 Introduction

As multi-core architectures become ubiquitous in modern computing, optimizing compilers need to automatically extract parallelism for large scale scientific applications composed of many subroutines. Inlining is a well-known program transformation which substitutes procedure invocations with their corresponding implementations to erase artificial procedural boundaries [9]. However, conventional inlining cannot be applied to recursive procedures or subroutines defined in external libraries where the source code is not available. Further, when excessively applied, it can cause code size explosion and curtail the compiler's effectiveness in applying optimizations (e.g., automatic loop parallelization and register allocation) due to the increased code complexity resulted from inlining.

This paper investigates techniques that enhance the effectiveness of inlining to support more aggressive loop parallelization by optimizing compilers. In particular, after identifying limitations of conventional inlining while using the Polaris compiler [7] to parallelize a collection of Fortran77 applications from the PERFECT benchmark suit [6], we present a new inlining approach to overcome these limitations. Our results show that the new approach can eliminate negative impact of conventional inlining while enhancing the effectiveness of interprocedural parallelization for a majority of the PERFECT benchmarks.

Figure 1 shows the workflow of our enhanced inlining approach. In contrast to conventional inlining, which substitutes a procedure invocation with the complete implementation of the callee, we use annotations, which summarize the computational structure and side effects of the callee, to replace the invocation. The inlined code is then optimized by the Polaris compiler which applies sophisticated loop dependence analysis to automatically parallelize loops via OpenMP when safe. The optimized code from Polaris is then piped into a *reverse inliner*, which reverts the earlier inlined code back to using the original procedure invocations but leaves the OpenMP pragma intact. The output from the reverse inliner is essentially the original input code optimized with automatic parallelization, where the annotations, currently manually provided by developers, have been used to enable more aggressive loop optimization by the Polaris compiler in spite of opaque procedure calls. Our annotation-based inlining approach offers the following advantages over conventional inlining.

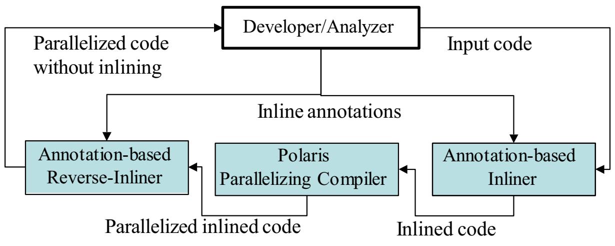

Developer/Analyzer
Parallelized code without inlining
Input code
Inline annotations
Annotation-based Reverse-Inliner
Polaris Parallelizing Compiler
Annotation-based Inliner
Parallelized inlined code
Inlined code

Fig. 1 Workflow of our annotation-based inlining approach

- Inlining can be applied even for subroutines defined in external libraries without their source code and for recursive subroutines because developers can provide a high-level summary of the semantics of these subroutines.
- The potential code size explosion problem can be avoided entirely as the inlining transformations will be reverted back to the original call statements after optimization.
- The user-supplied annotations do not need to include irrelevant implementation details of the subroutines of interest. After inlining, the code within the caller is much easier to analyze compared to when it contains the complete implementation of the callee.

The rest of this paper is organized as follows. Section 2 summarizes limitations of conventional inlining when used to support automatic loop parallelization by the Polaris compiler. Section 3 presents details of our annotation-based inlining approach. Section 4 presents experimental results. Sections 5 and 6 present related work and conclusions.

2 Automatic Parallelization Using Polaris

Polaris is a source-to-source Fortran 77 compiler which supports automatic parallelization of loops based on sophisticated dependence analysis techniques [7]. It uses simple heuristics controlled via command-line options to govern whether each procedure call is inlined before parallelization analysis [17]. The default strategy inlines a procedure call only when the procedure contains no I/O and not many statements (less than 150 by default) and when the invocation is inside a loop nest.

Polaris includes a number of sophisticated dependence analysis techniques which are fairly effective when analyzing regular Fortran DO loops operating on array subscripts that are linear combinations of the surrounding loop index variables. However, it becomes overly conservative when encountering non-linear array subscripts, which could be introduced by the inlining transformation applied before the analysis. The following summarizes the main issues we found to significantly hinder the effectiveness of Polaris loop parallelization analysis when combined with conventional procedure inlining.

2.1 Loss of Parallelism Due to Inlining

In languages such as Fortran/C/C++, arrays are treated as pointers into regions of data, and the same data operated by different subroutines can be declared as arrays of different shapes. Further, when optimizing Fortran subroutines, compilers can assume different array parameters are not aliased to each other. When subroutine invocations are inlined, the abstraction layer is broken, and the inlined implementations may become harder to analyze due to excessive code complexity introduced by inlining. As a result, loops that can be automatically parallelized by compilers when inside their respective subroutines may become no longer parallelizable after inlining, as discussed in the following.

2.1.1 Forward Substitution of Non-linear Subscripts

Figures 2 and 3 illustrate a situation where non-linear array subscripts are introduced by inlining the invocation of subroutine *PCINIT* at line 3 of Fig. 3 with its implementation in Fig. 2. Here the actual parameters used in the invocation are indirect references pointing to different regions of a global array *T*. When using these indirect array references to instantiate the formal parameters $X2$, $Y2$, and $Z2$ of *PCINIT* in Fig. 2, the array references $X2(I)$, $Y2(I)$, and $Z2(I)$ at lines 8–10 of Fig. 2 become $T(IX(7) + I)$, $T(IX(8) + I)$ and $T(IX(9) + I)$, respectively. Because the values of $IX(7)$, $IX(8)$, and $IX(9)$ are unknown at compile time, the inlining transformation has created subscripted subscripts (array subscripts that contain additional subscripted array references) which are non-linear and considered non-analyzable by most dependence analysis techniques. As a result, the loops at lines 3 and 6 of Fig. 2 can no longer be automatically parallelized after inlining, although Polaris dependence analysis can safely parallelize them inside the *PCINIT* subroutine before inlining.

2.1.2 Linearization of Array Dimensions

Figures 4 and 5 illustrate a situation where common arrays operated by two different subroutines are declared with different shapes. In particular, multi-dimensional arrays *PP, PHIT*, and *TM1* are used at line 5 of Fig. 5 to invoke the subroutine *MATMLT* defined in Fig. 4. However, the corresponding formal parameters $M1$, $M2$, and $M3$ are declared as single-dimensional arrays in Fig. 4. To inline the subroutine invocation, Polaris reconciles the mismatched array declarations by linearizing *PP, PHIT*, and *TM1* in Fig. 5 into single dimensional arrays without any explicit shape information. After inlining, the compiler can no longer precisely recognize the dependence constraints of the inlined loops. As a result the three loops at lines 22, 23, and 26 of Fig. 4 can no longer be parallelized after inlining.

Fig. 2 A subroutine with automatically parallelizable loops at lines 3 and 6

```
 1      SUBROUTINE  PCINIT(X2,Y2,Z2,..)
 2      DIMENSION X2(*),Y2(*),Z2(*)
      ......
 3      DO 200 N=1,NTYPES
 4      NSP=NSPECI(N)
 5      NS=NSITES(N)
 6      DO 200 J=1,NSP
 7      I=I+1
 8      X2(I)=FX(I)*TSTEP**2/2.D0/DSUMM(N)
 9      Y2(I)=FY(I)*TSTEP**2/2.D0/DSUMM(N)
10      Z2(I)=FZ(I)*TSTEP**2/2.D0/DSUMM(N)
      ......
    200 CONTINUE
        ...
```

```
1     DIMENSION T(*),S(*),W(*)
2     COMMON/WINDEX/ IX(99)
      ......
3     CALL PCINIT(T(IX(7)),T(IX(8)),T(IX(9)),..)
      ......
```

Fig. 3 A call site of the subroutine in Fig. 2 (loops in PCINIT become no longer automatically parallelizable after inlining)

Fig. 4 A subroutine with automatically parallelizable loops at lines 22, 23, and 26

```
20        SUBROUTINE MATMLT(M1,M2,M3,L,M,N)
21        DOUBLE PRECISION M1(L*M),M2(M*N),M3(L*N)
22        DO 120 JL=1,L
23        DO 110 JN=1,N
24          J=JL+L*(JN-1)
25          M3(J)=0.0
26          DO 100 JM=1,M
27          K1=JL+L*(JM-1)
28          K2=JM+M*(JN-1)
29 100      M3(J)=M3(J)+M1(K1)*M2(K2)
30 110 CONTINUE
31 120 CONTINUE
32        RETURN
33        END
```

```
       ...
       DIMENSION PP(4,4,15),PHIT(4,4),TM1(4 4),...
       ...
 3     DO 160 KS=1,15
 4     KSM=KS-1
 5     IF(KS.GT.1) CALL MATMLT(PP(1,1,KSM),PHIT,TM1,4,4,4)
 6     ENDIF
       ...
11 160 CONTINUE
       ...
```

Fig. 5 A call site which invokes the subroutine in Fig. 4 (loops in *MATMLT* are no longer automatically parallelizable after inlining)

2.2 Missed Opportunities

Conventional inlining substitutes a subroutine invocation with the entire implementation of the callee, where excessive complexity in the callee's implementation can force compiler optimizations, e.g., automatic loop parallelization, to be overly conservative due to the lack of domain-specific knowledge and runtime information.

The following discusses situations where the complexity of subroutine implementations prevents them from being effectively inlined or optimized.

2.2.1 Opaque Compositional Subroutines

Conventional inlining typically leaves out subroutines that make additional non-trivial procedure calls, as inlining a chain of subroutine invocations could result in serious code explosion. For example, the subroutine *FSMP* in Fig. 6 is excluded from inlining by Polaris as it invokes a fair number of other subroutines. This subroutine serves to initialize a single column of five arrays, FE (lines 12–13), SE (line 18), ME (line 19), MNLE (line 20), and PE (line 23), using a large number of

```
 1    SUBROUTINE FSMP(ID, IDE)
      ...
 2    CALL GETCR(ID)
 3    IRECT = IEGEOM(ID)
 4    ICT   = IECURV(ID)
 5    K1    = AK1(ICT)
 6    K2    = AK2(ICT)
 7    K12   = AK12(ICT)
 8    ISTRES = 0
 9    CALL SHAPE1
10    IF (IDEDON(IDE).EQ.0) THEN
11      IDEDON(IDE) = 1
12      CALL FORMF(FE(1,IDE))
13      CALL CHOFAC(FE(1,IDE), NSFE, IERR)
14      IF (IERR.NE.0) THEN
15         WRITE(6,*) ' F ELEMENT ',IDE,' IS SINGULAR '
16         STOP ' F SINGULAR '
17      ENDIF
18      CALL  FORMS(SE(1,IDE))
19      CALL  FORMM(ME(1,IDE))
20      CALL FORMNL(MNLE(1,IDE))
21    ENDIF
22    CALL GETLD(ID)
23    CALL FORMP(PE(1,ID))
24    RETURN
25    END
```

Fig. 6 A subroutine excluded from inlining by Polaris

global variables, including both scalar variables and arrays, some of which are modified to hold intermediate results of the internal computation. In spite of the complexity of computation, distinct columns of the five arrays are modified when invoking *FSMP* with different values for *ID* and *IDE*. Figure 7 shows an example loop nest which invokes *FSMP* with values for *ID* obtained from a global array *IDBEGS* which returns a unique integer for each given value of *ISS*. After feeding such information to the Polaris compiler via annotations, the compiler is able to tell that distinct values of *ID* and *IDE* are used at different iterations of the inner *K* loop at line 4. As a result, it can automatically parallelize this loop after annotation-based inlining is applied, discussed in Sect. 3.2.2.

2.2.2 Debugging and Error Checking

In practical applications, debugging and error checking statements are often used inside subroutines to ensure proper termination of the application when processing erroneous input data. This situation is illustrated by lines 14–17 of Fig. 6, where the

```
 1C    . LOOP OVER THE SUBSTRUCTURES .
 2    DO 35 ISS = 1, NSS
 3C      . LOOP OVER THE ELEMENTS IN THIS SUBSTRUCTURE .
 4      DO 30 K = 1, NEPSS(ISS)
 5C        . FORM THE ELEMENTAL ARRAYS .
 6        ID = IDBEGS(ISS) - 1 + K
 7        IDE = K
 8        CALL FSMP(ID, IDE)
 9 30   CONTINUE
10 35 CONTINUE
```

Fig. 7 A loop nest invoking the subroutine in Fig. 6 (the inner *K* loop at line 4 can be automatically parallelized after annotation-based inlining)

whole program would abort if previous evaluation has resulted in logical errors (indicated by the global *IERR* variable). Since debugging and error checking conditionals typically contain program I/O and early termination of the program, conservative compilers need to disable optimizations of the surrounding loops. However, pre-tested input data are often known to not trigger erroneous conditions at runtime, where the error handling statements are never executed. Even when errors do occur, replication of error messages is often acceptable. Such application-specific knowledge can be incorporated in our annotation-based inlining mechanism to support more aggressive loop parallelization, discussed in Sect. 3.2.3.

2.2.3 Use of Temporary Arrays

Many subroutines use temporary arrays to store intermediate results of computation, where each temporary array is first modified with new values before being used for additional computation. When the whole computation is inside a surrounding loop, compilers can apply array kill analysis to determine whether any value of the array comes from the previous iterations. If the whole array is killed (reinitialized) at each iteration, the temporary array can be privatized (duplicated within each thread) when parallelizing the surrounding loop. However, the array kill analysis may fail when only a subset of the array elements are modified, and those being used later are not obviously covered by the modifications. To illustrate such situations, Figs. 8 and 9 provide two subroutine definitions invoked by the *FSMP* routine in Fig. 6. Here a global array *XY* is used as a temporary array which is modified by the *GETCR* subroutine in Fig. 8 and then used by the *SHAPE1* subroutine in Fig. 9. Although *GETCR* modifies only a subset of the elements in *XY* (specifically, it modifies *XY(1:2,1:NNPED)*, where *NNPED* < = *ZNNPED*), only those elements being modified by *GETCR* are used in *SHAPE1*. However, due to the complexity of the multiple conditionals in Fig. 9, a typical optimizing compiler would fail to discover the coverage relation even after both subroutines are successfully inlined. We resolve this issue by declaring that the whole temporary array is reinitialized via user-supplied annotations, shown in Fig. 13 and discussed in Sect. 3.2.4.

2.2.4 Indirect References in Array Subscripts

Due to the lack of knowledge about runtime values of different arrays, conventional loop dependence analysis techniques are overly conservative when array references

Fig. 8 Definition of subroutine *GETCR* invoked by *FSMP* in Fig. 6

```
      SUBROUTINE GETCR(K)
      ...
      COMMON /ECOORD/ XY(2,ZNNPED)
      ...
  100 CONTINUE
      DO 200 I = 1, NNPED
         J = ICOND(I,K)
         XY(1,I) = XYG(1,J)
         XY(2,I) = XYG(2,J)
  200 CONTINUE
    ...
      RETURN
      END
      SUBROUTINE SHAPE1
      ...
      COMMON /ECOORD/ XY(2,ZNNPED)
      ...
      IF (NNPED.EQ.9) THEN
         AX = XY(1,3) - XY(1,1)
         AY = XY(2,5) - XY(2,3)
      ELSE IF (NNPED.EQ.4) THEN
         AX = XY(1,2) - XY(1,1)
         AY = XY(2,3) - XY(2,2)
      ELSE IF (NNPED.EQ.16) THEN
         AX = XY(1,4) - XY(1,1)
         AY = XY(2,7) - XY(2,4)
      ELSE
         CALL ERRORS(' NNPED NOT IMPLEMENTED IN SHAPE1')
      ENDIF
      ...
```

Fig. 9 Definition of subroutine *SHAPE1* invoked by *FSMP* in Fig. 6

```
 1       SUBROUTINE ASSEMR(ID, RHSE, RHSI, RHSB)
         ...
 2       DO 40 IN = 1, NNPED
 3          NODE = ABS(ICOND(IN,ID))
 4          IBLOCK  = IWHERD(NODE,1)
 5          IREL    = IWHERD(NODE,2)
 6          IF (IBLOCK.EQ.NBLOCK) THEN
 7             DO 10 I = 1, NDDF
 8                RHSB(IREL+I-1) = RHSB(IREL+I-1) + RHSE(I,IN)
 9 10          CONTINUE
10          ELSE
11             DO 20 I = 1, NDDF
12                RHSI(IREL+I-1) = RHSI(IREL+I-1) + RHSE(I,IN)
13 20          CONTINUE
14         ENDIF
15 40 CONTINUE
16    RETURN
```

Fig. 10 A subroutine which contains indirect references in array subscripts

are used inside the subscripts of accessing other arrays. Figure 10 illustrates such an example, where two global arrays, *ICOND* and *IWHERD*, which serve to save one-to-one relations between data in different arrays, are used to compute the subscripts of modifying two other arrays *RHSH* and *RHSI*. After forward substitution of variables, the subscripts used to modify *RHSB* and *RHSI* at lines 6–13 become *IWHERD(ABS(ICOND(IN,ID), 2)* + *I—1*, which will always yield different values when given distinct values of *IN*, *ID* and *I*. Therefore, different elements of the arrays *RHIB* and *RHSI* are modified when given distinct values of *IN*, *ID*, and *I*. However, such application-specific information is not available to the compiler, which must assume both *ICOND* and *IWHERD* could have arbitrary unknown values. Consequently the compiler must conservatively assume that arbitrary elements of *RHSH* and *RHSI* could be modified and must refrain from optimizing any loop that invokes *ASSEMR* even if the implementation of *ASSEMR* has been inlined. Figure 10 illustrates such a call site, where the *K* loop invokes *ASSEMR* at line 6 with the values of *ID* uniquely determined by the loop index variable *K*. We discuss how to enable Polaris to safely parallelize this loop via annotation-based inlining in Sect. 3.2.5.

3 Enhancing the Role of Inlining

To improve the effectiveness of automatic parallelization when encountering situations discussed in Section 2, we seek to enhance the role of inlining so that higher-level semantics of subroutine invocations can be made readily available to compilers. In particular, we use the following steps to enhance the effectiveness of conventional inlining supporting automatic parallelization across procedural boundaries.

(1) Annotate important subroutines to summarize their side effects and loop structures required for accurate dependence analysis. Then, substitute subroutine invocations with the corresponding annotations instead of the actual detailed implementations.
(2) Use Polaris to apply conventional loop dependence analysis and automatically parallelize loops when safe by inserting OpenMP directives.
(3) Apply a *reverse inlining* step which substitutes all the inlined annotations with appropriate invocations of the original subroutines. After this step, the only remaining transformation to the original input code is the parallelization of loops via OpenMP.

The reverse inlining step essentially reverses all the transformations introduced by annotation-based inlining so that the original input code can benefit from advanced compiler optimizations without sacrificing its modularity. The correct application of this step requires all the inlined annotations be recognized and mapped back to correct invocations of the original subroutines, which can be easily accomplished when minimal transformations, e.g., insertion of OpenMP directives, have been made to the inlined code. However, the task is more challenging when interacting with more drastic program transformations, e.g., loop blocking and software pipelining. Section 3.3 discusses these issues in more detail.

The following first discusses our annotation language and then illustrates how to use annotations to summarize the higher-level semantics of subroutines and enable more effective parallelization after inlining. Section 3.3 presents our algorithm for enhanced inlining. Section 3.4 discusses the correctness and generality of the overall approach.

3.1 The Annotation Language

Figure 11 summarizes the syntax of our annotation language, which can be used by developers to describe the side effects and loop structures of important subroutines. When these annotations are used to support subroutine inlining, a compiler can correctly recognize the dependence constraints carried by each subroutine invocation and subsequently successfully parallelize surrounding loops when safe.

The statements supported by our language include loops, if-conditionals, assignments, variable declarations, and return statements. They are used to summarize the control-flow structure and memory side effects of each subroutine. For implementation details that cannot be expressed using these statements, we

Fig. 11 The annotation language

```
s : {s1 s2 ... sn}
    | if (e) s1 [else s2]
    | do (id=e1:e2[:e3]) s
    | var=e;
    | vars=unknown(e1, e2, ...);
    | vars=unique(e1, e2, ...);
    | type var1,...,varn;
    | return e;
var : id | id '[' e1, ..., en ']'
vars : var | (var, ..., var)
```

s, $s1$, ..., s_n: statements;
e, $e1$, ..., e_n: expressions;
id : variable name;
id[$e1$, ..., e_n]: multi-dimensional array reference;
type var_1,...,var_n : declaring types of variables;

provide two special operators, *unique* and *unknown*, to summarize approximate relations between variables while omitting details of the computation. In particular, y=*unique*(x_1, ..., x_n) specifies that the value of y is uniquely computed (determined) from those of variables x_1, ..., x_n; that is, if y_1 is computed from (x_1=v_1, ..., x_n=v_n), y'_1 is computed from (x_1=v'_1, ..., x_n=v'_n), and (v_1, ..., v_n) $\neq$ (v'_1, ..., v'_n), then $y_1 \neq y'_1$.

Therefore, if the values of x_1, ..., x_n are different at distinct iterations of a loop surrounding y=*unique*(x_1, ..., x_n), then the values of y are guaranteed to be similarly different. In contrast, y=*unknown*(x_1, ..., x_n) specifies that the value of variable y is computed from reading those of variables x_1, ..., x_n, but the relationship could be arbitrary. These special-purpose operators serve to abstract away complex implementation details which degrade the effectiveness of compiler analysis, while keeping essential relations among variables visible to the compiler.

Expressions supported by our annotation language include most of the arithmetic operations in Fortran 77 combined with memory references via scalar and array variables. The Fortran 90 notation of array regions are supported so that collective operations can be applied to arrays without requiring explicit loops. The two special operators, *unknown* and *unique*, can also be used directly inside expressions, where their results do not need to be saved in variables before used.

3.2 Writing Annotations

Our annotation language serves to accurately summarize the side effects and loop structures of subroutines without exposing their local implementation details that are irrelevant to the surrounding calling context. In particular, for each subroutine, the annotations aim to summarize relations between its input parameters and output values as well as global variables modified by the subroutine while omitting intermediate results and variables that are local to the subroutine. The goal is to minimize adverse side effects of conventional inlining which may result in accidental loss of parallelism in the inlined code. To demonstrate the capacity of this approach, the following illustrates how to use user-supplied annotations to overcome inefficiencies of automatic parallelization discussed in Section 2.

3.2.1 Avoiding Loss of Parallelism

As discussed in Sect. 2.1, when conventional inlining breaks procedural boundaries by substituting subroutine invocations with detailed implementations, some parallel loops inside the inlined code may become no longer parallelizable by compilers due to unexpected interactions between the caller and callee. Our annotation-based approach resolves this issue by preserving all the original procedural boundaries and thereby entirely eliminating the adverse side effects of conventional inining. In particular, after we enable the compiler to perform interprocedural dependence analysis by substituting subroutine invocations with user-supplied annotations, the reverse inlining step will replace the inlined annotations with the original subroutine invocations, thereby preserving the original procedural boundaries as well as their optimizations.

3.2.2 Summarizing Opaque Subroutines

Since we substitute subroutine invocations with summaries of their semantics supplied by developers, our approach can be easily applied to opaque subroutines with arbitrarily complex implementations. As example, Fig. 12 illustrates our annotations for the *FSMP* subroutine in Fig. 6, which was excluded from inlining by Polaris due to its excess code complexity. In particular, these annotations summarize the regions of global arrays (*FE, ME, SE, MNLE, PE*) modified by all the subroutines invoked from *FSMP*, the temporary global variables (*XY, IRECT, K1, K2, K12, ISTRES, NDX, NDY, WTDET*) modified in the process, and the read-only global variables (*IEGEOMI, IECURV*, among others) used in the computation. The *unknown* operator is used extensively to omit local implementation details (e.g., intermediate results) of relevant computation, allowing invocations of the

```
subroutine FSMP(ID,IDE) {
  XY=unknown(XYG[:,ICOND[:,ID]],NSYMM);
  IRECT=IEGEOM[ID];
  K1=AK1[IECURV[ID]];
  K2=AK2[IECURV[ID]];
  K12=AK12[IECURV[ID]];
  ISTRES=0;
  (NDX,NDY,WTDET)=unknown(IRECT,XY,NDXI,NDETA,QDWGHT,
                          NQD,NNPED);
  if (IDEDON[IDE]==0) {
    IDEDON[IDE]=1;
    FE[:,IDE]=unknown(WTDET,C,NB,NNPES,NQD,NSFE);
    ME[:,IDE]=unknown(WTDET,N,NSYMM,RHO,NQD,NNPED);
    SE[:,IDE]=unknown(WTDET,NB,N,NDX,NDY,K1,K2,K12,
                      NNPES,NQD,NNPED);
    MNLE[:,IDE]=unknown(WTDET,NB,NDX,NDY,KNONLN,
                        NNPES,NQD,NNPED);
  }
  P=unknown(PXYZ[:,ABS(ICOND[:,ID])],NNPED,GX,NSYMM);
  PE[:,ID]=unknown(P,WTDET,N,NQD,NNPED);
  }
```

Fig. 12 Annotations for the opaque subroutine defined in Fig. 6

subroutine to be more accurately handled by dependence analysis of their surrounding loops.

3.2.3 Debugging and Error Handling

Many subroutines in large applications contain program output statements used for debugging and error handling purposes. The presence of these *exception handling* statements are typically treated with extreme caution by compilers, where all reordering transformations of the surrounding loops are subsequently disabled. However, since these statements are used for debugging/error handling only, they are not triggered at runtime in most cases, and even when triggered, precise exception handling is often not required. Using our annotation-based inlining approach, developers can choose to relax the consistency requirement of exception handling when parallelizing their program, by omitting these situations in the subroutine annotations. For example, in Fig. 12, the error checking conditional at lines 14–17 of Fig. 6 has been omitted in the annotations. Therefore it no longer prevents loops surrounding invocations of the *FSMP* subroutine from being safely parallelized.

3.2.4 Use of Temporary Arrays

Complex subroutines often use temporary arrays to hold intermediate results of the internal computation. When these arrays are declared as local variables, our annotations will omit their existence entirely as they do not incur any visible side effects to the outside. However, sometimes these arrays are declared in the global scope and used to pass values from one subroutine to another. An example global array used for this purpose is shown in Fig. 8, where the global array *XY* is modified in subroutine *GETCR* to hold intermediate results and then used in the subroutine *SHAPE1* in Fig. 9. It is conceptually a temporary array within the *FSMP* subroutine as only those elements defined in *GETCR* are used in the subsequent calls to *SHAPE1* and other subroutines. Similar global temporary arrays in *FSMP* include *NDX, NDY, WTDET* and *P*, shown in Fig. 12. In our annotations, these arrays are modified and used as if they are atomic scalar variables. Since modifications to these variables precede all their uses in the annotations of subroutine *FSMP* in Fig. 12, they can be treated as private variables when parallelizing a loop surrounding the invocation of *FSMP*, shown in Fig. 7. In particular, when parallelizing the *K* loop in Fig. 7, Polaris would peel *the last iteration* of the loop before parallelizing all the other iterations by privatizing temporary arrays in those iterations, so all the global arrays have the same values as their original sequential computation after the entire loop is finished.

3.2.5 Indirect References in Array Subscripts

Many Fortran applications use global arrays to store dynamic relations between different data structures. Most of these arrays are initialized only once throughout the entire program to save a one-to-one unique mapping between the related data.

An example subroutine using these special-purpose arrays is shown in Fig. 10, where *ICOND* and *IWHERD* are global arrays which contain one-to-one relations between elements stored in different places. When they are used as subscripts to access *RHSB* and *RHSI* in Fig. 10, the subscript expressions are non-linear, and compliers have to be overly conservative when parallelizing the surrounding loops. To overcome these difficulties, in Fig. 13, we use the *unique* operator to summarize the values of these arrays in terms of the relevant input parameters and loop index variables (*ID, IN,* and *I*). The declaration of the *unique* relation comes from domain-specific knowledge of the developer. When using the annotations in Fig. 13 to substitute for the invocation of subroutine *ASSEMR* in Fig. 14, each unique operator will be replaced with a linear expression which uniquely combines the involved integer variables *ID, IN,* and *I*. As a result the compiler can easily recognize that unique elements of arrays *RHSB* and *RHSI* are modified at each distinct iteration of the surrounding loop and thereby can safely parallelize the loop in Fig. 14.

3.3 The Enhanced Inlining Algorithm

Figure 15 shows the key steps of our algorithm for applying automatic parallelization with enhanced inlining support. The algorithm is comprised of three main phases: annotation-based inlining, automatic parallelization, and reverse inlining. The following explains each phase in detail.

3.3.1 Annotation-Based Inlining

The implementation of this step is similar to conventional inlining, except that subroutine invocations are substituted with user-supplied annotations instead of detailed implementations of the callees. Translating annotations to the underlying programming language (e.g., Fortran) is trivial, except for the two special purpose operators, *unknown* and *unique*. To translate each *unknown* operator, we define a new uninitialized global array, modify the array with all the operands of the *unknown* operator, and then replace the *unknown* invocation with an access to the new array. To translate each *unique* operator, we replace it with a linear expression which uniquely combines all the relevant integer variables. After inlining, a pair of special tags are placed surrounding the inlined source code to support reverse inlining at a later stage. Figure 18 shows the result of applying annotation-based inlining to an invocation of the *MATMLT* subroutine in Fig. 5, using annotations in Fig. 16.

In our algorithm, only subroutines with annotations are considered for inlining. Note that although the original implementation of *MATMLT* in Fig. 4 declares the

Fig. 13 Annotations for the subroutine in Fig. 10

```
subroutine ASSEMR(ID,RHSE, RHSI, RHSB) {
  do (IN=1:NNPED)
    do (I=1:NDDF)
      if (unique(ID,IN) == NBLOCK)
        RHSB[unique(ID,IN,I)] += RHSE[I,IN];
      else
        RHSI[unique(ID,IN,I)] += RHSE[I,IN];
}
```

```
      DO 10 K = 1, NEPSS(ISS)
         ID = IDBEGS(ISS) + K - 1
         IDE = IESMNO(ID)
         CALL GETEU(ID, XE, X)
         CALL MATMUL(ME(1,IDE), XE, MXE, NDFE, NDFE, 1)
         CALL ASSEMR(ID, MXE, MXI, MXB)
   10 CONTINUE
```

Fig. 14 A parallelizable loop invoking the subroutine in Fig. 10

```
Input: program source code and annotations for selected subroutines
Output: optimized source code
Algorithm:
  1) Annotation-based inlining: For each call statement within the input program
     where annotations are provided for the callee
      a) Instantiate the corresponding annotations with actual parameters;
      b) Replace the call statement with the instantiated annotations;
      c) insert tags surrounding the instantiated code fragment from inlining.
  2) Automatic parallelization: invoke Polaris to optimize the inlined code.
  3) Reverse inlining: For each tagged code segment in the optimized source code:
      a) Find the corresponding subroutine annotations from looking into the tag;
      b) Match the tagged code segment against the corresponding annotations to
         compute instantiation parameters;
      c) Replace the tagged code segment with a subroutine call using the instan-
         tiation parameters.
```

Fig. 15 Automatic parallelization with annotation-based inlining

Fig. 16 Annotations for the *MATMLT* subroutine in Fig. 4

```
subroutine MATMLT(M1,M2,M3,L,M,N) {
  dimension M1[L,M], M2[M,N], M3[L,N];
  M3 = 0.0;
  do (JN=1:N)
    do (JM=1:M)
      M3[:,JN] += M1[:,JM] * M2[JM,JN];
}
```

array parameters $M1$,$M2$,$M3$ as having single dimensions, our annotations declare them as two-dimensional matrices. Subsequently our annotation-based inlining can avoid the unnecessary linearization of array dimensions which may degrade the precision of compiler analysis, as discussed in Sect. 2.1.

3.3.2 Automatic Parallelization

After applying annotation- based inlining to the input program source, we optimize the inlined code using the Polaris compiler (with conventional inlining disabled), which performs advanced dependence analysis of the inlined code and automatically inserts OpenMP directives to parallelize loops when safe and profitable (the profitability is determined based on simplistic heuristics, e.g., all parallelized loop needs to exceed a certain number of iterations). Figure 17 shows the result of applying loop parallelization to our inlined code in Fig. 18. Note that the pair of special tags surrounding the inlined annotations remain intact after the parallelization optimizations.

Fig. 17 Call site of MATMLT in Fig. 18 after parallelization

```
            ...
!$OMP PARALLEL
!$OMP+DEFAULT(SHARED)
!$OMP DO
      DO KS=1,15
      IF(KS.GT.1) THEN
* //@; BEGIN(Code)
*     @annot inline MATMLT {
!$OMP PARALLEL
!$OMP+DEFAULT(SHARED)
!$OMP DO
       DO JL=1,4,1
        DO JN=1,4,1
         TM1(JL,JN)=0.0
         DO JM=1,4,1
          TM1(JL,JN)=TM1(JL,JN)
    *+PP(JL,JM,KS-1)*PHIT(JM,JN)
         ENDDO
        ENDDO
       ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
*     @}
      ENDIF
      ...
      ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

```
      ...
      DIMENSION PP(4,4,15),PHIT(4,4),TM1(4,4),...
      ...
      DO KS=1,15
      KSM=KS-1
      IF(KS.GT.1) THEN
*//@; BEGIN(Code)
     @annot inline MATMLT {
       DO JL=1,4,1
         DO JN=1,4,1
           TM1(JL,JN)=0.0
           DO JM=1,4,1
             TM1(JL,JN)=TM1(JL,JN)+
    *PP(JL,JM,KSM)*PHIT(JM,JN)
           ENDDO
         ENDDO
       ENDDO
     @}
      ENDIF
     ...
      ENDDO
      ...
```

Fig. 18 Call site of MATMLT in Fig. 5 after annotation-based inlining

3.3.3 Reverse-Inlining

After automatic parallelization, the reverse-inlining step is performed to reverse the annotation-based inlining transformation while keeping the OpenMP directives inserted by the Polaris compiler intact. This step is necessary to ensure correctness of program optimization.

Specifically, while the user-supplied annotations are expected to carry equivalent side effects and dependence constraints as the real subroutine implementation, they are typically not semantically equivalent to the original implementation, due to simplification of internal implementation details and the use of the special-purpose summary operations, e.g., the *unknown* and *unique* operators. Therefore, the inlined annotations must be reversed back to an equivalent subroutine invocation to guarantee the correctness of the optimized code.

The reverse inlining transformation is applied to all the tagged code segments created by the earlier annotation-based inlining transformation. For each tagged fragment, it first finds the corresponding subroutine annotations and then proceeds to compute an instantiation value for each formal parameter of the subroutine. Specifically, when using these parameter values to instantiate the subroutine annotations, the resulting code must be equivalent to the tagged code segment. Currently we apply a pattern matching algorithm to compare the tagged code segment and the inlining annotations node by node, while allowing variable substitution, expression reordering, and OpenMP directives inside the tagged segment. Since the only optimization performed by Polaris is the insertion of OpenMP directives, which are simply ignored in the pattern matching process, a set of appropriate parameter values are guaranteed to be found for each tagged code segment. These values are then used as actual parameters to generate a subroutine invocation so that each tagged code segment is replaced with its original function call.

Figure 19 shows the resulting code after applying reverse inlining to the parallelized loop in Fig. 17. Note that while the single optimization applied by Polaris is the insertion of OpenMP directives, the compiler does perform several normalization transformations, e.g., reordering of statements, induction variable substitution, and constant propagation, to the tagged code segments. As a result our reverse inlining transformation cannot simply replace them with the original subroutine calls. Our pattern matching algorithm is tolerant of minor modifications to the inlined annotations and can automatically extract the correct actual parameters in subroutine invocation in spite of the normalization transformations.

```
      ...
!$OMP PARALLEL
!$OMP+DEFAULT(SHARED)
!$OMP DO
       DO KS=1, 15, 1
       IF(KS.GT.1) THEN
       CALL MATMLT(PP(1,1,KS-1),PHIT(1,1),TM1(1,1),4,4,4)
       ENDIF
       ...
       ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
      ...
```

Fig. 19 Call site of MATMLT in Fig. 17 after reverse-inlining

3.4 Correctness, Efficiency, and Generality

The correctness of our enhanced inlining approach depends on the soundness of the user-supplied annotations. Specifically, if the annotations accurately summarize the side-effects and dependence constraints of the subroutines, the automatic parallelization optimization is guaranteed to be safe. Currently, such consistency is not automatically verified, and we use runtime testers to check and verify the correctness of our optimized code. Future work will develop techniques to automatically verify the soundness of user-supplied annotations and to automatically generate inlining annotations when possible.

The compile-time overhead of applying annotation-based inlining is similar to that of conventional inlining except that since user-supplied annotations are expected to be much shorter than detailed implementations, the cost of applying annotation-based inlining is lower, and the inlined annotations are expected to be much easier to analyze by the compiler.

Since all the inlined annotations are reversed back to using the original subroutine invocations, the modularity of the original program is not affected. The cost of applying reverse inlining to each tagged code segment is proportional to the size of the corresponding subroutine annotations, with constant cost associated with tolerating local modifications of the inlined annotations.

Our enhanced inlining approach can potentially enable general-purpose compilers to better utilize domain-specific knowledge from developers in supporting more effective interprocedural optimization of large-scale applications. So far we have used this approach to support only automatic parallelization via OpenMP. When applying pattern-matching to reverse inlined code segments back to appropriate subroutine calls, our reverse inlining transformation can tolerate local modifications to the inlined code such as reordering of expressions, induction variable substitutions, and insertion of OpenMP directives. However, to extend our approach to similarly support other optimizations such as loop blocking and unrolling, it may become much more challenging to reverse the inlined code segments back to appropriate subroutine calls after dramatic modifications to the tagged segments. Therefore, to apply this approach more extensively in a general-purpose compiler, a more systematic approach needs to be developed to compute correct instantiation parameters after dramatic modifications to the inlined annotations, which is a subject of future work.

4 Experimental Results

To evaluate the effectiveness of our enhanced inlining approach when used to enable more aggressive automatic parallelization of loops, we selected 12 applications from the *PERFECT* benchmark suite [27], summarized in Table 1. For each benchmark, we applied both conventional inlining and our enhanced inlining combined with automatic loop parallelization by the Polaris compiler. To measure the amount of parallelism enabled by and the degree of code explosion resulted from inlining, we counted the number of loops being parallelized after

Table 1 Summary of the perfect benchmarks

Applications	Descriptions
ADM	Pseudospectral air pollution simulation
ARC2D	Two-dimensional fluid solver of Euler equations
FLO52Q	Transonic inviscid flow past an airfoil
OCEAN	Two dimensional ocean simulation
BDNA	Molecular dynamic package for the simulation of nucleic acids
MDG	Molecular dynamics for the simulation of liquid water
QCD	Quantum chromodynamics
TRFD	A kernal simulating a two-electron integral transformation
DYFESM	Structural dynamics benchmark (finite element)
MG3D	Depth migration code
TRACK	Missile tracking

optimization and the line numbers of the resulting source code. We then use two multi-core machines, an Intel Macintosh running MacOS 10.5 with two quad-core 3GHz Intel processors (32KB L1 cache per core) and an AMD Opteron running Linux with two dual-core 3GHz AMD Opteron processors (128KB L1 cache per core), to measure the performance of the parallelized code. All benchmarks are compiled using *gfortran* 4.2.1 on the Intel Mac and *iFort* 11.1 on the AMD Opteron, using the -O3 optimization flag.

4.1 Enhancing Automatic Loop Parallelization

Table 2 compares the number of automatically parallelized loops by the Polaris compiler using three different inlining configurations: disable inlining of all subroutines (i.e., *no inlining*); inline implementations of subroutines with less than 150 lines of source code via conventional inlining (the default inlining strategy adopted by Polaris); and annotation-based inlining, where our enhanced inlining approach is used. For each configuration, we counted the number of automatically parallelized loops (#par-loops) and the overall code size (the number of source code lines with all comments removed) after optimization by Polaris. Note that when conventional inlining is applied, Polaris could fail to parallelize some loops which were parallelizable when no inlining is applied, as discussed in Section 2.1. These loops are categorized as #par-loss in Table 2. Inlining may also enable additional loops being parallelized beyond those parallelized using no-inlining, these loops are categorized as #par-extra in Table 2. If parallelized, each loop in the original benchmark is counted only once, even when inlining has made multiple copies of the original loop and all copies are subsequently parallelized.

From Table 2, inlining (including both conventional inlining and annotation-based inlining) is able to improve the effectiveness of automatic parallelization for 6 out of the 12 PERFECT benchmarks. For the other benchmarks, Polaris was not

Table 2 Automatically parallelized loops using different inlining strategies

Applications	Total # of loops	No Inlining		Conventional Inlining				Annotation based Inlining			
		# par- loops	code size	# par- loops	#par- extra	#par- loss	code size	# par- loops	#par- extra	#par- loss	code size
ADM	268	179	8261	153	3	29	8703	190	11	0	8380
ACAN	147	30	36,774	30	0	0	36,774	30	0	0	36,797
QCD2	157	102	3498	102	0	0	4795	110	8	0	3574
MDG	52	37	1962	41	6	2	2459	43	6	0	2109
TRACK	87	54	3107	54	0	0	3465	55	1	0	3121
BDNA	219	129	6713	128	1	2	6371	134	5	0	6722
OCEAN	133	106	8607	106	0	0	8691	106	0	0	8631
DYFESM	197	133	4713	134	2	1	5674	139	6	0	4772
MG3D	150	51	22,878	51	0	0	30,100	51	0	0	22,918
ARC2D	208	182	5185	139	0	43	4931	182	0	0	5290
FLO52Q	175	149	3547	149	0	0	3562	149	0	0	3551
TRFD	38	27	1703	14	0	13	636	27	0	0	1710
Totals	1831	1179	106,948	1101	12	90	116,161	1216	37	0	107,575

Code size is computed as the number of source code lines with all comments removed able to identify additional parallelism from loops which contain subroutine calls. Note that we have manually annotated a subset of subroutines from the PERFECT benchmarks based on careful inspection of their implementations. It may be possible to parallelize more loops by annotating additional subroutines. Future work will investigate more systematic application of our enhanced inlining approach by automatically generating annotations. We have verified the correctness of all the automatically parallelized loops via both manual inspection and runtime testing of the parallelized code.

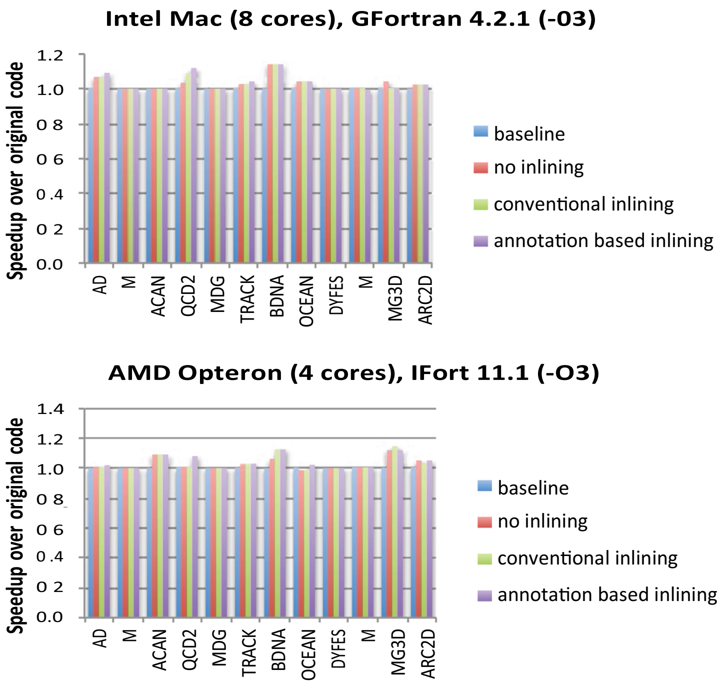
When combined with annotation-based inlining, the Polaris compiler is able to identify 37 additional parallelizable loops in different PERFECT benchmarks when compared with no-inlining. Most of these loops invoke complex subroutines which in turn invoke other routines, and the complexity of their implementations would overwhelm most state-of-the-art program analysis techniques. Since our annotation-based inlining allows developers to intervene with their application-specific knowledge, we are able to summarize the intended semantics of these subroutines to enable more effective parallelization of their surrounding loops. Examples of such annotations are illustrated in Section 3.2. When applying annotation-based inlining, the code explosion problem is avoided entirely, as the reverse inlining step has restored all the original subroutine invocations (the small increase in code size is mostly due to the extra OpenMP directives inserted to parallelize loops).

In contrast, conventional inlining enabled Polaris to parallelize only a small subset (12 out of 37) of the extra parallel loops identified by annotation-based inlining. Additionally, after conventional inlining, Polaris can no longer parallelize 90 loops which were categorized as parallelizable when no inlining is performed, due to issues discussed in Section 2.1.

After conventional inlining, the code size increased by about 10% even when only small subroutines were inlined. The increase is likely significantly higher when more extensive inlining is applied via the conventional approach.

4.2 Performance of Optimized Code

Figure 20 presents the runtime speedups achieved by the automatically parallelized benchmarks when using different inlining configurations. Note that a majority of the PERFECT benchmarks do not benefit from loop parallelization due to their small input data size, which is a known issue for these benchmarks [25]. To avoid degradation of performance by excessive parallelization of loops, we used empirical performance tuning to disable a selected set of loops from being parallelized if their parallelization incurs a slowdown of the overall execution time. As demonstrated in Figure 20, at most 10% performance improvement is achieved by automatic loop parallelization combined with different inlining configurations. Annotation-based inlining is able to achieve the best performance for two benchmarks (ADM and MDG) and has achieved similar performance as other inlining configurations for the other benchmarks.



* Baseline: performance of the original benchmarks with no optimization;

Fig. 20 Performance of automatically parallelized code

5 Related Work

Inlining is a widely adopted technique which can be used by compilers to erase procedural boundaries and apply optimizations to larger regions of code [1, 4, 30, 38]. Ayers et. all [4] shown that aggressive inlining and cloning based on profiled information can dramatically improve the effectiveness of a large number of back-end optimizations. However, when excessively applied, inlining can cause the well-known code-explosion problem [13] and could degrade the effectiveness of many compiler optimizations as the input code becomes overwhelmingly large and complex. Previous research has developed a variety of heuristics, including temperature heuristics [36], demand-driven online transformation [32], inlining trials [14, 38], and interprocedural flow analysis [3, 23], to selectively apply inlining so that performance benefits can be gained without incurring serious problems [2]. Existing research has also explored automatic tuning of different inlining heuristics [10], e.g., through machine-learning [40, 41]. Other directions include combining inlining with hot code outlining [37], context-sensitive trace inlining [39], region-based compilation [21, 24], and whole program optimization by merging and re-dividing code regions [31].

This paper proposes an annotation-based inlining approach to overcome the negative impact of conventional inlining both in terms of code explosion and in terms of unexpected complications. Our use of developer-supplied annotations to summarize the semantics of opaque subroutines is similar to the semantic inlining approach by Wu et al. [34, 35], which allows their compiler to treat user-defined abstractions as primitive types in Java, and the semantics-preserving inlining work by Stucki et. al [42], which uses inlining as a meta-programming tool for developers. The Broadway [18], DyC [16], and Orio [43] compilers used annotation languages to guide domain-specific optimizations, dynamic compilation of C code, and empirical performance tuning of scientific workload. Annotations and semantic specifications have also been used to specify dynamic properties of lower-level implementations in program verification [11, 15] and to drive various compile-time and runtime optimization decisions [44, 45]. We focus on using annotations to substitute for implementations of subroutines in inlining to enable more aggressive automatic parallelization.

Optimizing compilers have a long history of supporting automatic parallelization of user applications [8, 12, 19, 20, 26, 29, 33, 46]. This paper focuses on using inlining to enable interprocedural parallelization without resorting to expensive inter-procedural program analysis techniques [5, 22, 28, 47]. Our work is orthogonal to and can be integrated with existing other compiler frameworks for automatic loop parallelization besides the Polaris compiler.

6 Conclusions and Future Work

This paper presents a study which exposes some serious limitations of conventional inlining when using the Polaris compiler [7] to parallelize a collection of Fortran applications from the PERFECT benchmark suite [6]. We then present a new annotation-based inlining approach to overcome the lim- itations. Our experimental results show that the new approach can eliminate most of the negative impact of conventional inlining while significantly enhancing the effectiveness of automatic loop parallelization across procedural boundaries. Our future work will develop techniques to automatically derive necessary annotations and to verify the safety of manually supplied annotations.

Funding Funding was provided by National Science Foundation (Grant Number CCF-1910488).

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architec- tures. Morgan Kaufmann, San Francisco (2001)
2. Arnold, M., Fink, S., Sarkar, V., Sweeney, P.F.: A comparative study of static and profile-based heuristics for inlining. SIGPLAN Not. **35**(7), 52–64 (2000)
3. Ashley, J. M.: The effectiveness of flow analysis for inlining. In: In Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, pp 99–111 (1997).
4. Ayers, A., Schooler, R., Gottlieb, R.: Aggressive inlining. SIGPLAN Not. **32**(5), 134–145 (1997)
5. Balasundaram, V., Kennedy, K.: A technique for summarizing data access and its use in parallelism enhancing transformations. SIGPLAN Not. **24**(7), 41–53 (1989)

6. Blume, W., Eigenmann, R.: Performance analysis of parallelizing compilers on the perfect benchmarks programs. IEEE Trans. Parallel Distrib. Syst. **3**, 643–656 (1992)
7. Blume, W., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, W., Rauchwerger, L., Tu, P., Weatherford, S.: Polaris: Improving the effectiveness of parallelizing compilers. In: In Languages and Compilers for Parallel Computing, pp. 141–154. Springer-Verlag (1994).
8. Bondhugula, U., Hartono, A., Ramanujam, J.: A practical automatic polyhedral parallelizer and locality optimizer. In: In PLDI 08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (2008).
9. Calder, B., Grunwald, D.: Reducing indirect function call overhead in c++ programs. In: POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 397–408, ACM, New York, NY, USA, (1994).
10. Cavazos, J., OBoyle, M. F. P.: Automatic tuning of inlining heuristics. In: In ACM/IEEE Conference on Supercomputing, p. 14 (2005).
11. Chaki, S., Clarke, E., Groce, A.: Modular verification of software components in c. Trans. Softw. Eng. **1**(8), 388–402 (2004).
12. Chang, Y.-S., Lee, H.-J., Park, D.-S., Lee, I.-Y.: Interprocedural transformations for extracting maximum parallelism. In: ADVIS '02: Proceedings of the Second International Conference on Advances in Information Systems, pp. 415–424, Springer-Verlag, London, UK, (2002).
13. Cooper, K.D., Hall, M.W., Torczon, L.: Unexpected side effects of inline substitution: a case study. ACM Lett. Program. Lang. Syst. **1**(1), 22–32 (1992)
14. Dean, J., Chambers, C.: Towards better inlining decisions using inlining trials. In: In Proceedings of the 1994 ACM Conference on LISP and Functional Programming, pp. 273–282 (1994).
15. Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B.,Stata, R.: Extended static checking for java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02, pp. 234–245, ACM, New York, NY, USA (2002).
16. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: Dyc: an expressive annotation-directed dynamic compiler for c. Theor. Comput. Sci. **248**(1–2), 147–199 (2000)
17. Grout, J. R.: Inline expansion for the polaris research compiler. Technical Report, (1995).
18. Guyer, S.Z., Lin, C.: Broadway: A compiler for exploiting the domain-specific semantics of software libraries. Proc. IEEE Special Issue Prog. Gener. Optim. Adapt. **93**(2), 342–357 (2005)
19. Hall, M., Kennedy, K., Kinley, K. S. M.: Interprocedural transformations for parallel code generation. In: In Proceedings of Supercomputing '91, pp. 424–434 (1991).
20. Hall, M. W., Mellor-Crummey, J. M., Carle, A., Rodrguez, R. G.: Fiat: A framework for interprocedural analysis and transformation. In: In Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, pp. 522–545. Springer-Verlag (1995).
21. Hank, R. E., Mei, W., Hwu, W., Rau, B. R.: Region-based compilation: An introduction and motivation. In: In Proceedings of the 28th Annual International Symposium on Microarchitecture, pp. 158–168 (1995).
22. Hoeflinger, J.P., Paek, Y., Yi, K.: Unified interprocedural parallelism detection. Int. J. Parallel Program. **29**(2), 185–215 (2001)
23. Jagannathan, S., Wright, A.: Flow-directed inlining. In: In Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 193–205 (1996).
24. Liu, Y., Zhaoqing, Y. L., Qiao, R., Ching Ju, R. D.: A region- based compilation infrastructure. In: Proceefings of the 7th Workshop on Interaction between Compilers and Computer Architectures, pp. 75–84. IEEE Computer Society Press (2003).
25. Psarris, K., Kyriakopoulos, K.: The impact of data dependence analysis on compilation and program parallelization. In: ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing, pp. 205–214, ACM, New York, NY, USA (2003).
26. Ramaswamy, S., Sapatnekar, S., Banerjee, P.: A framework for exploiting data and functional parallelism on distributed memory multicomputers. IEEE Trans. Parallel Distrib. Syst. **8**(11), 1098–1116 (1997). https://doi.org/10.1109/71.642945
27. Saavedra, R.H., Smith, A.J.: Analysis of benchmark characteristics and benchmark performance prediction. ACM Trans. Comput. Syst. **14**, 344–384 (1996)
28. Seidl, H., Steffen, B.: Constraint-based inter-procedural analysis of parallel programs. Nordic J. of Computing **7**(4), 375–400 (2000)
29. Shenoy, U.N., Srikant, Y.N., Bhatkar, V.P.: An automatic parallelization framework for multicomputers. Comput. Lang. **20**(3), 135–150 (1994)

30. Shirako, J., Nagasawa, K., Ishizaka, K., Obata, M., Kasahara, H.: Selective inline expansion for improvement of multi grain parallelism. In: Parallel and Distributed Computing and Networks, pp. 476–482 (2004).
31. Triantafyllis, S., Bridges, M. J., Raman, E., Ottoni, G., August, D. I.: A framework for unrestricted whole-program optimization. In: In ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, pp. 61–71 (2006).
32. Waddell, O., Dybig, R. K.: Fast and effective procedure inlining. In: SAS '97: Proceedings of the 4th International Symposium on Static Analysis, pp. 35–52, Springer-Verlag, London, UK (1997).
33. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co. Inc, Boston, MA, USA (1995)
34. Wu, P., Midkiff, S. P., Moreira, J. E., Gupta, M.: Improving Java performance through semantic inlining. In: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing (1999).
35. Wu, P., Moreira, J. E., Midkiff, S. P., Gupta, M., Padua, D. A.: Semantic inlining - the compiler support for java in technical computing. In: PPSC (1999).
36. Zhao, P., Amaral, J. N.: To inline or not to inline? enhanced inlining decisions. In: In Workshop on Languages and Compilers for Parallel Computing (LCPC), pp. 405–419 (2003).
37. Zhao, P., Amaral, J.N.: Ablego: a function outlining and partial inlining framework. Softw. Pract. Exper. **37**(5), 465–491 (2007)
38. Aleksandar, P., et al.: An optimization-driven incremental inline substitution algorithm for just-in-time compilers.In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE (2019).
39. Häubl, C., Wimmer, C., Mössenböck, H.: Context-sensitive Trace Inlining for Java. Comput. Lang. Syst. Struct. **39**(4), 123–141 (2013). https://doi.org/10.1016/j.cl.2013.04.002
40. Cammarota, R., Nicolau, A., Veidenbaum, A.V., Kejariwal, A., Donato, D., Madhugiri, M.: On the determination of inlining vectors for program optimization. In: Proceedings of the 22nd International Conference on Compiler Construction (CC'13). Springer-Verlag, Berlin, Heidelberg, pp. 164–183. https://doi.org/10.1007/978-3-642-37051-9_9
41. Simon, D., Cavazos, J., Wimmer, C., Kulkarni, S.: Automatic construction of inlining heuristics using machine learning. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13). IEEE Computer Society, Washington, DC, pp. 1–12. https://doi.org/10.1109/CGO.2013.6495004
42. Stucki, N., Biboudis, A., Doeraene, S., Odersky, M.: Semantics-preserving inlining for metaprogramming. In: Proceedings of the 11th ACM SIGPLAN International Symposium on Scala (SCALA 2020). ACM, New York, NY, USA, pp. 14–24. https://doi.org/10.1145/3426426.3428486
43. Norris, H. B., Sadayappan, P.: Annotation-based empirical performance tuning using Orio, In: 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, pp. 1-11 (2009). https://doi.org/10.1109/IPDPS.2009.5161004
44. Papadopoulos, I., Thomas, N., Fidel, A., Amato, N. M., Rauchwerger, L.: STAPL-RTS: An application driven runtime system. In: Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15). ACM, New York, NY, USA, pp. 425–434. https://doi.org/10.1145/2751205.2751233
45. Yi, Q., Wang, Q., Cui, H.: Specializing compiler optimizations through programmable composition for dense matrix computations. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47). IEEE Computer Society, USA, pp. 596–608. https://doi.org/10.1109/MICRO.2014.14
46. Nguyen, T., Cicotti, P., Bylaska, E., Quinlan, D., Baden, S.B.: Bamboo: translating MPI applications to a latency-tolerant, data-driven form. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Washington, DC, USA, Article 39, pp. 1–11.
47. Ali, K., Lhoták, O.: Application-only call graph construction. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2012). Lecture Notes in Computer Science, Vol 7313. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-31057-7_30