Performance Analysis of Symbolic Analysis Techniques for Parallelizing Compilers *

Hansang Bae and Rudolf Eigenmann

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

Abstract. Understanding symbolic expressions is an important capability of advanced program analysis techniques. Many current compiler techniques assume that coefficients of program expressions, such as array subscripts and loop bounds, are integer constants. Advanced symbolic handling capabilities could make these techniques amenable to real application programs. Symbolic analysis is also likely to play an important role in supporting higher–level programming languages and optimizations. For example, entire algorithms may be recognized and replaced by better variants. In pursuit of this goal, we have measured the degree to which symbolic analysis techniques affect the behavior of current parallelizing compilers. We have chosen the Polaris parallelizing compiler and studied the techniques such as range analysis – which is the core symbolic analysis in the compiler – expression propagation, and symbolic expression manipulation. To measure the effect of a technique, we disabled it individually, and compared the performance of the resulting program with the original, fully-optimized program. We found that symbolic expression manipulation is important for most programs. Expression propagation and range analysis is important in few programs only, however they can affect these programs significantly. We also found that in all but one programs, a simpler form of range analysis – control range analysis – is sufficient.

1 Introduction

Automatic program parallelization has been studied and developed intensely in the last two decades, especially in an effort to automatically detect parallelism present in numerical applications. As a result, advanced analysis and transformation techniques exist today, which can optimize many programs to a degree close to that of manual parallelization. The ability of a compiler to manipulate and understand symbolic expressions is an important quality of this technology [1]. For instance, the accuracy of data dependence tests, array privatization, dead code elimination, and the detection of zero-trip loops increases if the techniques have knowledge of the value ranges assumed by certain variables.

* This material is based upon work supported by the National Science Foundation under Grant No. 9974976-EIA and 0103582-EIA.

Several research groups have developed symbolic analysis capabilities to make the most of program analysis techniques implemented in their compilers [1, 3–8]. The Polaris parallelizing compiler [2] has incorporated advanced symbolic analysis techniques in order to effectively detect privatizable arrays, to determine whether a certain loop is a zero-trip loop for induction variable substitution, and to solve data dependence problems that involve symbolic loop bounds and array subscripts. Range Propagation [3] is the basis for this functionality. It can determine the value ranges that symbolic expressions in the program can assume. Polaris' symbolic nonlinear data dependence test – the Range Test [4] – makes use of Range Propagation. The Range Test is the main advanced data dependence test in the Polaris compiler. Expression propagation is another important technique, which can eliminate symbolic terms by substituting them with known values. Furthermore, symbolic expression simplification is essential in several passes in the Polaris compiler.

The major impediment in adopting certain symbolic analysis techniques is their relatively high cost [5, 9]. For example, in one of our experiments, the Polaris compiler exhausted the available memory space that kept the range information of that program. Whether or not this cost is worth expending is not known, as the techniques' effectiveness in the context of an advanced optimizing compiler and with contemporary application programs has not been studied. These facts motivated our effort to quantify the gains of symbolic analysis techniques. We present the results as follows. Section 2 outlines the analysis techniques we measured. Section 3 describes our experimental methods and metrics. Section 4 discusses the results in detail. Section 5 reviews related work, and Section 6 presents our conclusion.

2 Symbolic Analysis Techniques in Polaris

We categorize the studied techniques into three groups. First, Range Propagation and the Range Test are the most important techniques that deal with symbolic terms in array subscripts. Second, Expression Propagation is a conventional technique to transform symbolic expressions into more analyzable form. Third, Symbolic Expression Simplification was included, because it provides essential functionality that several Polaris passes make use of.

2.1 Range Propagation and Range Test

The Range Analysis technique determines the value ranges assumed by variables at each point of a program. It does this by performing *abstract interpretation* [10] along the control and data flow paths. The results are kept in a *range dictionary* [3], which maps from variables to their ranges. Polaris supports two levels of range dictionaries. The *control range dictionary* collects information by inspecting control statements, such as `IF` statements and `DO` statements. The *abstract interpretation (AI) range dictionary* subsumes the control range dictionary and

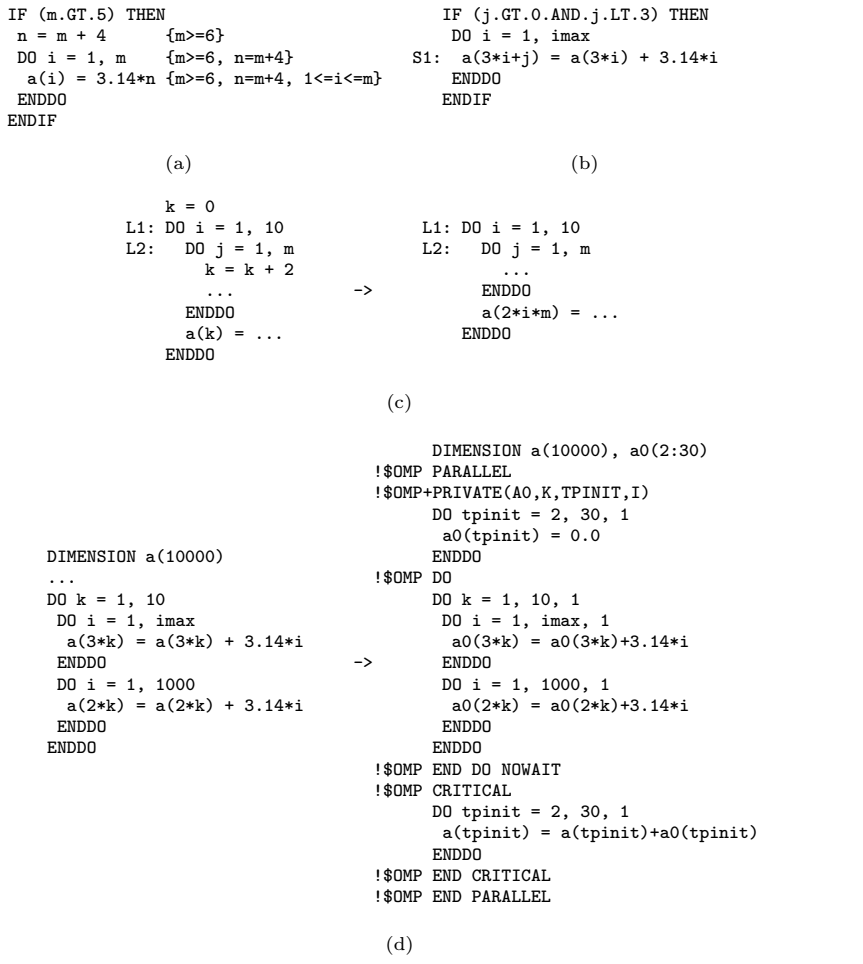

Fig. 1. The Range Propagation technique and its applications. (a) Contents of the range dictionary. (b) Loop that can be parallelized with the Range Test. (c) Induction variable substitution. (d) Reduction transformation

collects additional information from all assignment statements. Figure 1(a) shows an example code and the contents of the range dictionary.

One objective of this study is to determine the effectiveness of the range dictionary. The Polaris compiler currently uses range dictionary information to detect zero-trip loops in the induction variable substitution pass, to determine array sections referenced by array accesses in the reduction parallelization pass, and to compare symbolic expressions in the Range Test. Figure 1(c) illustrates

the case of induction variable substitution. Let us suppose the range dictionary for the shown code section keeps the range for the variable `m`, which is also the upper bound of the loop `L2`. If it is possible to prove that `m` is greater than or equal to one thanks to the range dictionary, the loop `L2` is not a zero-trip loop, and induction variable substitution can be applied safely in the array `a`, as shown in the right-hand side code. Otherwise, the compiler keeps the original code or generates multi-version loops.

The Polaris compiler is able to recognize array reductions and translate them into parallel form [11]. *Privatized Reductions* is one possible translation variant, shown in Figure 1(d). The range information for `k` was used to determine the accessed region of array `a`. This information is then used as the dimension of the private copy `a0`, and for the bounds of the preamble and postamble loop of the parallel reduction operation. If the range information were not available, Polaris would use the declared dimension of the array `a` instead, which may be too large, causing overhead in the preamble and the postamble.

The most important application of range analysis is the Range Test. The test performs many comparisons between symbolic expressions in order to analyze array subscripts. The comparison procedure determines the arithmetic relationship of two expressions by examining the difference of the two expressions. If the difference contains a symbolic expression, the range information of that expression is searched in the range dictionary. Figure 1(b) shows a simple example. Since the possible value of `j` is either one or two, the Range Test can determine that there is no loop-carried dependence in the statement `S1`.

2.2 Expression Propagation

By propagating the expression assigned to a variable to the variable's use sites, symbolic expressions can deliver more accurate information. Polaris can propagate constant integer, constant logical, and symbolic expressions within a procedure and across procedures. Real-valued expressions can also be propagated, but this option is switched off by default.

Before analyzing individual subroutines, Polaris performs interprocedural expression propagation, during which assignments to propagated expressions are inserted at the top of each procedure. Then, these expressions are propagated to possible call sites. This process iterates until no new expressions are discovered. Subroutine cloning is performed during this process, if the same subroutine is called with two different expressions.

Intraprocedural expression propagation is performed on each subroutine after the induction variable substitution pass. This Polaris pass introduces variables for which propagation can be important, as our measurements will show. However, other than this effect, intraprocedural expression propagation is essentially subsumed by intraprocedural and range propagation. As mentioned earlier, this technique can also propagate real-valued expressions and array expressions. We will include this option in our measurements as well.

Table 1. Benchmark suite

Code	# Lines	Serial time	Code	# Lines	Serial time
APSI	7361	57.8	TURB3D	2100	183.6
HYDRO2D	4292	80.3	WAVE5	7764	96.3
MGRID	484	54.5	ARC2D	4650	55.1
SU2COR	2332	55.0	MDG	1430	27.1
SWIM	429	84.7	TRFD	580	126.0
TOMCATV	190	96.1	MG	1460	57.4

2.3 Symbolic Expression Simplification

The simplification of symbolic expressions is important, as compiler–manipulated expressions tend to increase in complexity, making them difficult to analyze. Nearly all Polaris passes make use of expression simplifier functions. For instance, the procedure performing symbolic expression comparison assumes that the two expressions are reduced to their simplest form. Polaris provides the following three simplifying techniques for symbolic expressions:

- Combine: `A+4*A -> 5*A`
- Distribute: `A*(3+B) -> 3*A+A*B`
- Divide: `3*A/A -> 3`

3 Experimental Methodology

The fully-optimized programs serve as the baseline of our measurements. Starting from these programs, we disabled each compiler technique individually. For techniques that contain several levels of optimization, we took measurements with increasing levels. We compared the resulting, transformed code with the base code to examine the difference in the number of parallelized loops, and we measured the overall program performance of the parallel programs. In order to understand the performance impact of the techniques on the programs before parallelization, we also measured the performance of the transformed programs executed sequentially (i.e., without OpenMP translation).

3.1 Benchmark Suite

Table 1 shows our benchmark suite. We selected scientific/engineering programs written in Fortran77 from the floating point benchmarks in SPEC CPU95, from the Perfect Benchmarks and from the serial version of NAS Parallel Benchmarks. We chose the SPEC CPU95 over CPU2000 codes because Polaris requires Fortran77 input. All SPEC CPU2000 Fortran77 programs are present in the SPEC CPU95 suite. Since they are essentially the same codes, we expect that the results in terms of the number of parallelized loops would be the same for the SPEC CPU2000 codes. However, the speedup of the parallel programs is expected to be

higher for the SPEC CPU2000 compared to the CPU95 codes, due to the larger input data sizes. Several Perfect Benchmarks with their original input data sets execute in only a few seconds on today's machines. Therefore we have increased the problem sizes of `ARC2D` and `TRFD`. Among the four NAS benchmarks written in Fortran77, we found that symbolic analysis makes a difference only in MD. We included this code in our figures.

3.2 Set-up and Metrics

The Polaris compiler outputs parallel programs written in OpenMP, which we compiled using the Forte compiler to generate code for Sun workstations. We used a four-processor shared-memory Sun E-450 system for our experiments. The following shows all settings for this experiment. The compiler flag "-stackvar" allocates all local variables on the stack, and "-mt" is needed for multithreaded code.

- CPU: 480 MHz UltraSparc II
- Number of processors: 4
- Memory: 4GB
- Operating System: SunOS 5.8
- Compiler: Forte 6.1
- Compiler flags: -fast -stackvar -mt -openmp

We use the overall program speedup – serial execution time of the original code divided by parallel execution time of the transformed code – as a metric for presenting the performance of the programs, and also briefly describe the features of the transformed codes, such as the number of parallelized loops to explain the quality of the transformed codes.

4 Results and Analysis

4.1 Impact on Sequential Execution of Transformed Programs

In a first experiment, we ran all transformed benchmarks without compiling OpenMP directives and compared it with the serial programs. In this way, we could observe the effect of each technique on the performance before parallel code generation. Such effects are due to (1) direct changes of the source code by the techniques and (2) affected restructuring transformations. Our expression propagation techniques are implemented such that they perform direct substitutions of the source code. Also, the expression simplification techniques affect the programs directly. Among the restructuring transformations that are affected by symbolic analysis are induction variable substitution and reduction transformations. Understanding these performance effects is important because they represent degradation that is not a result of the parallel execution or lack thereof.

We found that expression propagation can introduce overhead. For example, expression propagation made a statement longer than a hundred lines in `APSI`,

Table 2. Number of parallel loops with and without range analysis. RT stands for Range Test. The figures in each colum describe: "total (lost outer-level)" parallel loops. "=" means the code is identical with the base code, The "No AIRD" row shows the results with no AI range dictionary used; in "No RD" both the AI and the control range dictionary are switched off; The last row "No RT" serves as a reference showing the effect of disabling the Range Test

Code	APSI	HYDRO2D	MGRID	SU2COR	SWIM	TOMCATV	TURB3D	WAVE5	ARC2D	MDG	TRFD	MG
Base	141	92	10	54	16	6	24	185	126	20	5	35
No AIRD	141(2)	=	=	=	=	=	=	185(2)	=	20	5	=
No RD	139(10)	92(3)	=	=	=	=	24	181	=	19(2)	11(2)	35(7)
No RT	122(7)	89	8	44	7	=	22	160(7)	125	13(1)	8(2)	24(2)

disabling code generation by the backend compiler. TURB3D ran 144% longer because of the overhead from expression propagation. SU2COR increased by 9%. All other programs showed no more than 5% overhead after program transformation.

We found that the expression simplification technique is necessary for many benchmarks. For instance, TRFD ran 240% longer after disabling the combining functionality for simplification. That means that, without this technique, the restructured, serially executed program would run so inefficiently that it would offset much of the gain from parallel execution.

Another interesting situation is the reduction transformation. Sometimes, this transformation is expensive because of inefficient preamble and postamble, as shown in Figure 1(d). This happend after certain compiler techniques were disabled in our experiments, resulting in insufficient information for data dependence analysis. We deal with that situation further in the following subsection.

In order to observe the effect of each compiler technique on the parallel program performance, we ran all benchmarks on four processors. We describe the result of the experiments category by category in the following subsections.

4.2 Range Analysis

Table 2 shows the characteristics of the transformed codes, with enabled/disabled techniques that relate to range analysis. The figures in each column represent the total number of parallel loops in the program. The numbers in parentheses explain how many outer level parallel loops were lost. For example, without the range dictionary, Polaris found 92 parallel loops in HYDRO2D. Although the base code and the code with disabled range dictionary have the same number of parallel loops, three outer–level loops could no longer be found parallel.

As expected, many programs benefit from the Range Test, since it is the only advanced data-dependence test used in Polaris. (Polaris also includes an optional Omega test. The performance when using this test instead of the Range Test is essentially the same as the "No RD" version.) More importantly, the table shows that the computation of full range information using the AI range dictionary is necessary only for APSI and WAVE5. This means that the relatively inexpensive

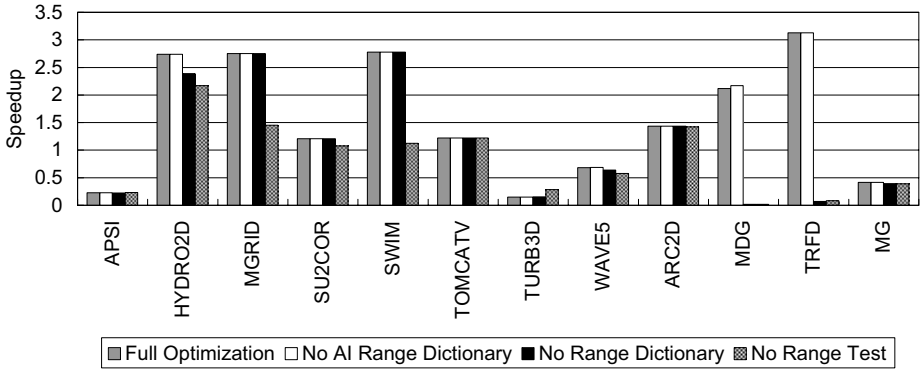

Fig. 2. Program performance with and without Range Analysis

control range dictionary is sufficient for Polaris to analyze the other codes. The table further shows that half of the benchmark codes do not need any range information – they can still be analyzed as accurately as the fully-optimized code. Although many subscripts contain symbolic terms, these terms cancel out in comparison operations. For example, the two expressions `i+m-3` and `i+m+5` can be compared by the test without the need of range information.

Figure 2 shows the program performance on four processors. Three programs `APSI`, `WAVE5`, and `TURB3D` achieved speedup less than one. This is due to the fact that we have used the "eager parallelization scheme" in Polaris. That is, the compiler is conservative in its profitability analysis – it avoids the parallelization of small parallel loops only if there is a provable disadvantage. This is appropriate for our study, where we are interested in the compiler's ability or inability to detect parallelism. In addition, `TURB3D` is included without advanced interprocedural analysis, which could improve the performance of the code significantly, but was not yet available in our version of Polaris.

In terms of program performance, four benchmarks, `HYDRO2D`, `WAVE5`, `MDG` and `TRFD`, benefit from the range dictionary. The code section in Figure 3 shows the case that needs range analysis in `HYDRO2D`. This loop accounts for 5% of the serial execution time, and is the main factor of the performance difference. After induction variable substitution, the array `tst` contains symbolic subscripts `i-mq+j*mq`. The range information for the variable `mq (mq>=1)` enables the Range Test to compare expressions such as `j*mq` and `1-mq+(j-1)*mq`, allowing the compiler to disprove the output dependence on `tst`. The perfomance gain after disabling the Range Test in `TURB3D` comes from not parallelizing small loops.

The phenomenal performance loss in `MDG` comes from an inefficient reduction transformation. Remarkably, `INTERF_do1000` and `POTENG_do2000`, the most time-consuming loops in `MDG`, were parallelized even without the Range Test. However, a small other loop was transformed very inefficiently without Range Analysis. The loop looks like a reduction operation, but is a fully parallel loop. This fact can be detected by the Range Test with range dictionary information.

Table 3. Cost of range anlaysis. (a) Percent compilation time of range analysis. The analysis time spans from 0.63 seconds to 153 seconds with full range dictionary, and from 0.38 seconds to 21 seconds only with control range dictionary. (b) Normalized memory requirement (1=No range analysis). The memory requirement spans from 33 Megabytes to 266 Megabytes with full range dictionary, and from 33 Megabytes to 125 Megabytes only with control range dictionary

(a)

Code	APSI	HYDRO2D	MGRID	SU2COR	SWIM	TOMCATV	TURB3D	WAVE5	ARC2D	MDG	TRFD	MG
Full	27.0	16.8	6.8	5.9	14.9	9.9	7.0	9.4	15.6	13.4	17.8	8.4
Control	3.1	9.5	1.6	2.0	6.6	8.6	2.7	3.6	4.7	4.3	7.1	2.4

(b)

Full	2.13	1	1	1.06	1	1	1	1.05	1.07	1.12	1	1.15
Control	1	1	1	1.01	1	1	1	1	1	1.02	1	1.02

However, without range information, the loop ends up being transformed as an array reduction, which is highly inefficient due to large pre/postambles in this case. The graph indicates the control range dictionary is sufficient to detect the explicit parallelism in the loop. In TRFD, Polaris was unable to parallelize the two most time-consuming loops without any range dictionary. Instead, many small inner loops were parallelized, causing significant performance degradation.

The results in this section indicate that Range Analysis is performance-critical for a small set of benchmarks only. Figure 4 shows that the range information affects the compiler, nevertheless. It presents the failure rate of expression comparison during the Range Test. It clearly shows that Polaris is able to make better decisions, thanks to range information, in all but one program. In MDG, the number of comparisions is less without the Range Dictionary. The lack of range information made several monotoniciy tests fail during the Range Test, making further comparisons useless. MG has more comparison failures with full range analysis because the effort to get more accurate value ranges ended up generating less useful information. For example, the AI range dictionary gives LT

```
k = 0
DO j = 1,nq
  DO i = 1,mq
    ...
    k =  k + 1
    tst(k) =  DMIN1(tcz, tcr)
  ENDDO
ENDDO
```

->

```
DO j = 1,nq
  DO i = 1,mq
    ...
    tst(i-mq+j*mq) = DMIN1(tcz, tcr)
  ENDDO
ENDDO
```

Fig. 3. Need for range dictionary in HYDRO2D

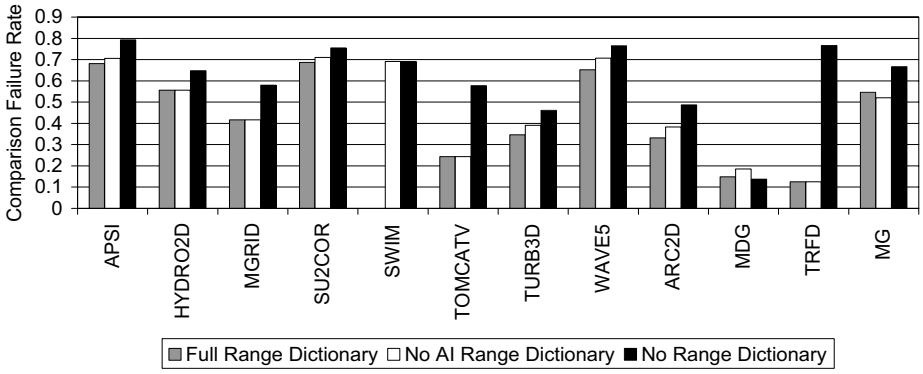

Fig. 4. Expression comparison failure rate

Table 4. Number of parallel loops with and without expression propagation. "'*" means outer-level parallelism. Expression propagation was not applied to the base code for WAVE5

Code	APSI	HYDRO2D	MGRID	SU2COR	SWIM	TOMCATV	TURB3D	ARC2D	MDG	TRFD	MG
Base	141	92	10	54	16	6	24	126	20	5	35
No IntraEP	141	=	=	54	=	=	24	=	20	6(1)	=
No InterEP	141(*1)	92	10	54	16	=	25	126	17	8(2)	35(2)
No EP	141(5,*1)	92	10	54	16	=	25	126	17	8(2)	35(2)

`>= MAX(1,1+LB)` for a certain code section, where `LB` is unavailable at compile-time, and the compiler cannot compare `LT` with `1+LB`. On the other hand, the control range dictionary gives `LT >= 1+LB` in the same situation, and the above comparison can be done easily.

4.3 Expression Propagation

The effects of expression propagation on the benchmark codes are presented in Table 4. Interprocedural propagation helped Polaris find more parallel loops in `MDG` and `TRFD`. Moreover, additional intraprocedural expression propagation was essential in finding outer-level parallelism in `TRFD`. The reason is that the potential dependence caused by a variable introduced by induction variable substitution was disproved by propagating information to the use site of that variable. For the benchmark `APSI`, interprocedural expression propagation provided information that helped the compiler recognize that an outermost loop had only one iteration and serialize this loop. This explains why Table 4 shows more outer-level parallelism *without* that technique.

In `MDG` and `TRFD`, expression propagation improved performance, as presented in Figure 5. The most time-consuming loops `POTENG_do2000` in `MDG` and `OLDA_do100` and `OLDA_do300` in `TRFD` were not parallelized without interprocedural expression propagation and even without additional intraprocedural expression propagation for the case of `TRFD`. However, expression propagation does

not make much difference in terms of execution time for the SPEC benchmarks. For some benchmarks, the actual substitution of the propagated expression incurred a slight overhead because of the increased strength of the operation. Using `NINT((160*xljet)/xlr)` instead of `npjet` a huge number of times is an example of the potential disadvantage of expression propagation.

The OpenMP translation of Polaris is responsible for the odd behavior of `ARC2D` when interprocedural expression propagation was disabled. The code section in Figure 6(a) accounts for the performance difference. The left-hand-side code was generated with interprocedural expression propagation, whereas the right-hand-side code was generated without this technique. Both outermost loops are parallel. Polaris detected the original array `work` as private. Expression propagation helps Polaris determine that only an array subrange of size `jmax`, where `jmax` is definied in the subroutine, is actually needed as private. Since OpenMP does not support partial arrays to be declared private, Polaris chooses to allocate from the heap a smaller array and use array expansion rather than privatization. Unfortunately, this turns out to perform less than privatizing the full array in this case.

However, without expression propagation, there were four loop nests where the outer loop was no longer parallel, setting off the performance gain in Figure 6(a). Those four loops have similar shapes. Figure 6(b) shows one of them. With expression propagation, the compiler could substitute `jx` with `ju-j` and remove the assignment to `jx`, making it possible to do loop-blocking utilizing OpenMP directives. The subroutine containing this loop is called 400 times, and the value of `ju-jl` is 288, making the number of fork-joins in the right-hand side loop 114799 more than that of the left-hand side loop. `MG` speeds up only with intraprocedural expression propagation. In other configurations, the parallelizer performs reduction transformation that introduces huge overhead in preambles/postambles.

We have also measured the effectiveness of more specific options of expression propagation. These are the propagation of array expressions and propagation of

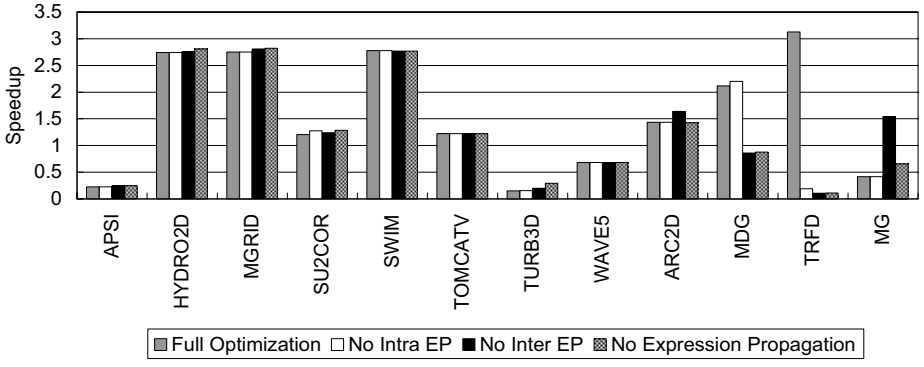

Fig. 5. Program performance with and without expression propagation

```
SUBROUTINE filery(jdim, ...)
...
jdim = jmax
...
ALLOCATE (work0(1:jdim, ...))
DO n = 1, 4
  DO k = 1, kmax-1
    DO j = jlow, jup
      work0(j,k,1,my_cpu_id) = ...
...
```

```
SUBROUTINE filery(jdim, ...)
...
!$OMP+PRIVATE(..., work)
      ...
     DO n = 1, 4
        DO k = kbegin, kup
          DO j = jlow, jup
            work(j,k,1) = ...
       ...
```

(a)

```
!$OMP PARALLEL
      DO j = 2, ju-jl, 1
!$OMP DO
       DO k = 2, ku, 1
        f(ju+(-j), k) = ...
        ... = f(2+ju+(-j), k)
       ENDDO
!$OMP END DO NOWAIT
      ENDDO
!$OMP END PARALLEL
```

```
      DO j = 2, ju-jl, 1
       jx = ju+(-jl)
!$OMP PARALLEL
!$OMP DO
       DO k = kl, ku, 1
        f(jx, k) = ...
        ... = f(2+jx, k)
       ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
      ENDDO
```

(b)

Fig. 6. Code sections of ARC2D with and without expression propagation. (a) Privatization in subroutine FILERY. (b) Loop blocking

real expressions. The base code was generated with array expression propagation and without real expression propagation. In general, the effects of these techniques are negligible compared to other techniques examined in this section. However, we have seen code examples where propagation generated very long expressions, which is undesirable.

4.4 Expression Simplifier

In general, the symbolic expression simplifier turned out to be important. This is because nearly all passes implemented in Polaris use that functionality. For instance, the Range Test assumes the two expressions to be compared are in their simplest form before the comparison. Table 5 shows the effects of this technique on the benchmark codes. The combining capability plays a more important role in helping the Range Test than the other two capabilities, because it performs the actual simplification in the expression manipulation. On the other hand, canceling common factors in the denominator and the numerator does not happen frequently, so the effect was negligible, except for `TRFD`.

Figure 8 shows the overall performance of each program without the expression simplifier. As we have seen in this section, `TRFD` again shows an extreme

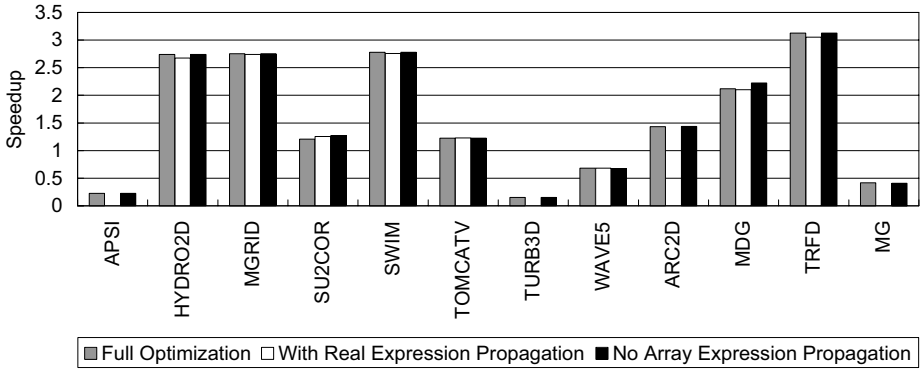

Fig. 7. Program performance with and without expression propagation

Table 5. Number of parallel loops with and without expression simplifier. SU2COR could not be parallelized without combining functionality

Code	APSI	HYDRO2D	MGRID	SU2COR	SWIM	TOMCATV	TURB3D	WAVE5	ARC2D	MDG	TRFD	MG
Base	141	92	10	54	16	6	24	185	126	20	5	35
No Divide	=	92	=	54	=	=	24	185	=	=	6(1)	=
No Distribute	130(15)	92(3)	8	48	=	6	23	177(7)	125	15(2)	11(2)	31(8)
No Combine	121(11)	89	8	N/A	7	6	22	156(11)	129(1)	13(2)	9(2)	24(4)

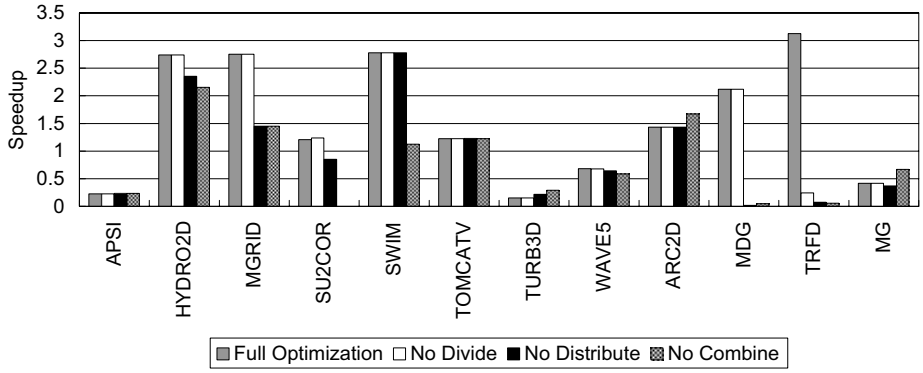

Fig. 8. Program performance with and without expression simplifier

performance loss. For TURB3D, disabling the simplifier resulted in serializing small loops with a performance gain.

An important additional consideration is the expression simplifier's impact on memory usage during program analysis. Polaris could not fully analyze SU2COR with expression simplification turned off because it exhausted its swap space (2 GB).

As in the case without expression propagation, ARC2D shows odd behavior without the combining capability. It turned out that the code section responsible for this behavior runs faster with inner-level parallelism, which is not true in general. Without the combining capability, the compiler could only find inner-level parallelism, which happened to perform better because of insufficient number of iterlations at outer level.

5 Related Work

We have found no comprehensive studies that measure the impact of symbolic analysis techniques, as presented in this study. However, a number of projects have developed compile-time symbolic analysis techniques similar to those considered in this paper. Haghighat and Polychronopoulos proposed a methodology for the discovery of certain program properties that are essential in the effective detection and efficient exploitation of parallelism [6]. The authors' methodology was implemented as a symbolic analysis framework for the Parafrase-2 parallelizing compiler. Induction variable substitution, dead-code elimination, symbolic data dependence test, and program performance prediction were suggested as possible analysis techniques that could benefit from the symbolic analysis framework. The authors also showed their analyzer performed well, especially in detecting complex induction variables such as induction variables in conditional statements.

Fahringer proposed symbolic analysis techniques to be used as part of a parallelizing compiler and a performance estimator for optimization of parallel programs [7]. He suggested an algorithm for computing lower and upper bounds of symbolic expressions based on a set of constraints, to be used in comparing symbolic expressions, simplifying systems of constraints, examining non-linear array subscript expressions for data dependences, and optimizing communications. Another functionality he suggested is the capability of estimating the number of integer solutions to a system of constraints, which can be used to support detection of zero-trip loops, elimination of dead code, and performance prediction of parallel programs. His techniques were implemented for the Vienna Fortran Compilation System (VFCS), a High-Performance-Fortran-style parallelizing compiler. His technique for comparing symbolic expressions enabled hoisting communication out of loop nests in FTRVMT, a dominant loop in OCEAN.

There were also efforts to analyze symbolic expressions across procedure boundaries. Havlak constructed interprocedural symbolic anaylsis mechanisms as an infrastructure for the Parascope compilation system [8]. His analysis is based on the program representation in Thinned-gated single-assignment (TGSA) form

– an extended form of Static single-assignment (SSA), and operates interprocedurally by relating a call graph to each value graph for a procedure. During the analysis, information such as passed values, returned values, array subscripts and bounds, loop bounds, and predicates are considered. The effectiveness of his technique was measured only by comparing dependence graphs.

Recent research deals with symbolic analysis by formulating it as a system of constraints. Pugh and Wonnacott's research on nonlinear array dependence analysis [12, 14] suggested a method of obtaining certain conditions under which a dependence exists. Value-based dependence analysis, which delivers more accurate information, was also described by a set of constraints, and the author suggested uninterpreted function symbols as a way of representing non-affine terms that could exist during dependence analysis. Rugina and Rinard's work on symbolic bound analysis [15] proposed a scheme to compute symbolic bounds for each pointer and array index variable at each program point, and to compute a set of symbolic regions that a procedure accesses. They reduced the systems of constraints to linear programs to obtain symbolic bounds.

6 Conclusion

We have measured the impact of symbolic analysis techniques, specifically range propagation, expression propagation, and symbolic expression simplification. Using several SPEC CPU95, Perfect, and NAS benchmarks we have analyzed the techniques' ability to help recognize parallelism and improve program performance.

We have found that all techniques make a significant difference in at least one of the programs. Expression simplification is important for most programs, while full range propagation does not affect the program performance substantially. Somewhat unexpected, interprocedural expression propagation was only relevant for two of the programs. More complex programs are affected more significantly. Symbolic analysis effects performance the most in the Perfect the least in the NAS benchmarks. This suggests that the techniques will impact full-scale applications more significantly than measured in our experiments.

In our analysis we found that secondary effects of symbolic analysis techniques can make a performance difference. For example, in several program sections, the techniques helped recognize parallel loops that were too small to improve performance and introduced overhead instead. Improved performance estimation capabilities could help remedy these situations and would thus be important complements of advanced program analysis techniques.

We found that, in all programs, the simpler form of range propagation, which derives information from control statements only, was sufficient. This finding is significant, as it will allow compiler developers to implement advanced optimization techniques that rely on symbolic analysis without high compile-time expenses. It holds in particular for the Range Test, which is able to analyze data dependences in the presence of nonlinear and symbolic subscripts.

References

1. W. Blume and R. Eigenmann. An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages 233–238, August 1994.
2. W. Blume, R. Doallo, R. Eigenmann, J. G. J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, pages 78–82, December 1996.
3. W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, Santa Barbara, CA, April 1995.
4. W. Blume and R. Eigenmann. Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1180–1194, December 1998.
5. P. Tu and D. A. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 414–423, 1995.
6. M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.
7. T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *The Journal of Supercomputing*, 12(3):227–252, May 1998.
8. P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994.
9. W. Blume and R. Eigenmann. Demand-driven symbolic range propagation. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, Columbus, OH, 1995.
10. P. Cousot and R. 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.
11. W. M. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, pages 444–448, 1995.
12. W. Pugh and D. Wonnacott. Nonlinear Array Dependence Analysis. In *Proceedings of 3rd Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, November 1994.
13. V. Aslot and R. Eigenmann. Performance characteristics of the spec omp2001 benchmarks. In *Proceedings of the 3rd European Workshop on OpenMP (EWOMP'2001)*, Barcelona, Spain, September 2001.
14. W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.
15. R. Rugina and M. 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.