POLARIS

Rudolf Eigenmann, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, eigenman@purdue.edu

1 Definition

Polaris is the name of a parallelizing compiler and research compiler infrastructure. Polaris performs source-to-source translation; programs written in the Fortran language are converted into restructured Fortran programs - typically annotated with directives that express parallelism. Polaris was created in the mid 1990s at the University of Illinois and was one of the most advanced freely available tools of its kind.

2 Introduction

Polaris aimed at pushing the forefront of automatic parallelization (see autoparallelization) and, at the same time, providing the research community with an infrastructure for exploring program analysis and transformation techniques. Polaris followed several earlier projects with similar goals, a key distinction being that the development of this new compiler was driven by real applications. While the benchmarks that prompted earlier parallelizing compiler research were typically small, challenging program kernels, the Polaris project was preceded and motivated by an effort to manually parallelize the most realistic application program suite available at that time – the Perfect Benchmarks [1]. This effort identified a number of beneficial transformation techniques [2], the automation of which constituted the Polaris project.

Polaris was able to improve the state of the art in automatic parallelization significantly. Earlier autoparallelizers were measured to parallelize to a significant degree only 2 of the 13 Perfect Benchmarks. By contrast, Polaris achieved success in six of them – increasing the parallelization success rate in this class of science and engineering applications from less than 20% to nearly 50%.

Polaris was originally developed at the University of Illinois from 1992 to 1995 [3], with significant extensions made at Purdue University and Texas A&M University, in later project phases. Among the examples and predecessors were several parallelizers developed at the University of Illinois and Rice University. An important contemporary was the Stanford SUIF project [4]. Among the more recent, related compiler infrastructures are Rose [5], LLVM [6] and Cetus [7].

The architecture of Polaris exhibits the classical structure of an autoparallelizer. A number of program analysis and transformation passes detect parallelism and map it to the target machine. The passes are supported by an internal program representation (IR) that represents the source code being transformed and offers a range of program manipulation functions.

3 Detecting Parallelism

Polaris includes the program analysis and transformation techniques that were found to be most important in a prior manual parallelization project [2]. At the core of any autoparallelizer is a data dependence detection mechanism. Data dependences prevent parallelism, making dependence-removing techniques essential parts of an autoparallelizer's arsenal. To this end, Polaris includes passes for data privatization, reduction recognition, and induction variable substitution. The compiler focuses on detecting fully parallel loops, which have independent iterations and can thus be executed simultaneously by multiple processors. For the basics of the following techniques, see "Autoparallelization".

Data-dependence test: Typical data dependence tests detect whether or not two accesses to a data array in two different loop iterations could reference the same array element. The detection works well where array subscripts are linear – of the form $a * i + b * j$, where a,b are integer constants and i,j are index variables of enclosing loops. The Polaris project developed new dependence analysis techniques that are able to detect

parallelism in the presence of symbolic and non-linear array subscript expressions. For example, in the above expression, if a is a variable, the subscript is considered symbolic; if the term i^2 appears, the subscript is non-linear. If the compiler cannot determine the value of a symbolic term, it cannot assume it is linear. Hence, symbolic and non-linear expressions are related. In real programs it is common for expressions, including array subscripts, to contain symbolic terms other than the loop indices. Through non-linear, symbolic data-dependence testing, Polaris was able to parallelize several important programs that previous compilers could not.

Privatization: Data privatization [8] is a key enabler of improved parallelism detection. A privatization pattern can be viewed as one where a variable, say t, is being used as a temporary storage during a loop iteration. The compiler recognizes this pattern in that t is first defined (written) before used (read) in the loop iteration. By giving each iteration a separate copy of the storage space for t, accesses to t in multiple iterations do not conflict. Polaris extended the basic technique so that it could detect entire arrays that could be privatized. For example, the following loop can be parallelized after privatizing the array *tmp*. (the notation *tmp(1:m)* means "the array elements from index 1 to m")

```
DO i=1,n
   tmp(1:m) = a(1:m)+b(1:m)
   c(1:m)   = tmp(1:m)+sqrt(tmp(1:m))
ENDDO
```

Privatization gives each processor a separate instance of *tmp*. Without this transformation, each loop iteration would write to and read from the same *tmp* variable, creating a data dependence and thus inhibiting parallelization.

Implementing array privatization is substantially more complex than the basic scalar privatization technique. The compiler must determine the sections of each array that are being defined and used in a loop. If each element of an array that is being used has previously been defined in the same loop iteration, the array is privatizable. More sophisticated analysis may privatize sections of arrays that fit this pattern. To do so, the compiler analysis must be able to combine array accesses into sections. As array subscript expressions may contain symbolic terms, these operations must be supported by advanced symbolic manipulation functions, which is one of Polaris' strengths.

Reduction recognition: This transformation is another important enabler of parallelization. Similar to the way Polaris extended the privatization technique from scalars to arrays, it extended reduction recognition [9]. The following loop shows an example *array reduction* (sometimes referred to as irregular or histogram reduction). Different loop iterations modify different elements of the *hist* array. The pattern of modification is not important; any two loop iterations may modify the same or different array elements.

```
DO i=1,n
   val = <some computation>
   hist(tab(i)) = hist(tab(i)) + val
ENDDO
```

Polaris recognizes array reductions by searching for loops that contain statements with the following pattern: An assignment statement has a right-hand-side expression that is the sum of the left-hand-side plus a term not involving the reduction array. A loop may have several such statements, but the reduction array must not be used in any other statement of the loop.

Because two iterations may access the same element, the compiler has to assume a possible dependence. Reduction parallelization takes advantage of the mathematical property that sum operations can be reordered (even under limited-precision computer arithmetic, reordering is usually valid, although not always.) A reduction loop can be executed in parallel like this: Each processor performs the sum operations over the assigned loop iteration space on a private copy of the original reduction array (*hist*, in the above example).

At the end of the loop, the local reduction arrays from all participating processors are summed into the original reduction array.

Reduction patterns are common in science and engineering applications. Array reduction parallelization was a key enabler of parallel performance in a number of important loops in the Perfect Benchmarks.

Induction variable substitution: Induction variable substitution eliminates dependences by replacing an arithmetic sequence by a closed-form computation. In the following loop, the induction variable *ind* forms a sequence.

```
ind = ind0
DO i=1,n
   ind = ind + k
   a(ind) = 0
ENDDO
```

In the basic form of an induction variable, the next value in the sequence is generated by adding a constant to the previous value. The term k could be an integer constant or a loop-invariant expression. The closed-form expression for the sequence is *ind0+i*k*. By substituting this expression into the array subscript, *a(ind0+i*k)*, the induction statement can be removed and the dependence on the previous value disappears.

Polaris extended this well-known transformation to a more general form, where k can be non-constant, such as another induction variable. For example, if the expression k is the loop index ($ind = ind + i$), the closed form becomes *ind0+i*(i+1)/2*.

The closed form of a generalized induction variable usually contains non-linear terms. Parallelization in the presence of such terms requires the application of non-linear data-dependence tests. Further complication arises when the induction variable is used after the loop. In this case, the compiler must compute and assign the last value. In the above example, it would insert the statement *ind = ind0+n*k* after the loop. Such *last-value assignment* will be correct if assignment to the induction variable is guaranteed in all iterations. Polaris makes use of symbolic program analysis techniques to make this guarantee where possible.

3.1 Advanced Program Analysis

Symbolic Analysis: In addition to powerful transformations, key to Polaris' performance is the availability of advanced symbolic analysis and manipulation utilities. For example, when performing generalized induction variable substitution, the compiler needs to evaluate sum expressions, such as $\sum_1^n j = n(n+1)/2$.

Polaris' symbolic range analysis technique is able to determine symbolic value ranges that program variables may assume during execution. The technique looks at assignment statements, loop statements, and condition statements to determine constraints on the value range of variables. After an assignment, the left-hand-side variable is known to have the newly given value or symbolic expression. Within a loop, the loop variable is guaranteed to be between the loop bounds. In the `then` clause of an `if` statement, the `if` condition is guaranteed to hold (and not hold in the `else` clause). For example, the analysis can determine that inside the following loop the inequality $1 \leq i \leq n$ holds and the loop accesses the array elements from position 2 to n+1.

```
DO i=1,n
  a(i+1)=0
ENDDO
```

Polaris created powerful tools for compilation passes to manipulate and reason with symbolic ranges, including comparison, intersection and union operations. All of the described parallelization techniques depend on the availability of symbolic analysis.

Interprocedural analysis: Because structuring a program into subroutines is an important software engineering principle, the ability of compilers to analyze programs across procedure calls is crucial. Advanced parallelizers, such as Polaris, attempt to find parallelism in outer loops. Larger parallel regions can better amortize the overheads associated with parallel execution, as will be discussed later. However, outer loops tend to encompass subroutine calls, making the application of the described techniques difficult. Furthermore, interprocedural analysis is important even for code sections that do not contain subroutine calls. For many optimization decisions, the compilation passes must collect information from across the program. For example, symbolic analysis will find the value ranges of program variables in the entire program and propagate them to the subroutines where needed.

The needs for interprocedural operation are different for each compiler technique. Creating specialized techniques for each optimization pass can be prohibitively expensive. Instead, Polaris includes a capability to expand subroutines inline. By default, small (by a configurable threshold) subroutines are expanded in place of their call statements. This capability obviates the need for specialized interprocedural analysis in most common cases. The drawback of subroutine inline expansion is code growth. Depending on the subroutine calling structure, a code expansion of an order of magnitude is possible. While Polaris has demonstrated that significant additional parallelism can be found this way, the needed compilation time can grow substantially. To address this issue, Polaris also included an interprocedural data access analysis framework [10] as a basis for unified interprocedural parallelism detection.

4 Mapping Parallel Computation to the Target Machine

Mapping parallel computation to the target machine has two objectives. First, the parallelism uncovered by an autoparallelizer may not necessarily fit the model of parallel execution by the eventual machine platform. Additional transformations of the parallel execution or the program's data space may be needed. Second, almost all program transformations incur overheads. Privatization uses additional storage; reduction parallelization adds computation (e.g., summing up local reduction arrays); induction variable substitution creates expressions of higher strength (e.g., addition is replaced by multiplication); starting/ending parallel loops incurs *fork/join* overheads (e.g., communicating to the participating processors what to do and warming up their cache). Advanced optimizing compilers make use of a performance model to estimate the overheads and to decide which transformations best to apply.

Scheduling parallel execution: Polaris detects loops that are dependence-free and marks them as fully parallel loops. To express parallel execution, it inserts OpenMP (openmp.org) directives, leaving the code generation up to the target platform's OpenMP compiler. A simple parallel loop in OpenMP looks like this.

```
!$OMP PARALLEL DO PRIVATE(t)
   DO i=1,n
      t = a(i)+b(i)
      c(i) = t + t*t
   ENDDO
```

The "OMP" directive expresses that the loop is to be scheduled for parallel execution and the data element t must be placed in private storage. By default, the OpenMP compiler assigns the loop iteration space to the available parallel threads (or cores) in chunks. For example, on an 8-core platform, the first core would usually execute the first $n/8$ iterations, etc. Polaris can influence this choice via OpenMP "Schedule" clauses. For example, if the compiler detects that the amount of computation per loop iteration is irregular, it may choose a "dynamic" schedule, which maps iterations to available processors at runtime.

To determine profitability of parallel execution, Polaris applies a simple test. It estimates the size of a loop by considering the number of statements and the number of iterations. If the size can be determined and is below a configurable threshold, the compiler leaves the loop in its original, serial form. While the creation of advanced performance models is an important research area, this simple test worked well in practice.

Data placement: The data model is of further importance. Polaris assumes that all non-private variables are shared. All processors simply refer to the data in the same way the original, serial code does. In the above example, all processors participating in the execution of the parallel loop see the arrays a, b and c in the same way and have direct access. The variable t is stored in processor-private memory. Depending on the architecture, private storage may be actual processor-local memory or it may simply be a processor-private slice of the global address space.

Current multicore architectures implement this shared-memory model in hardware and are thus suitable targets for Polaris. Another important class of parallel computers is not: Many high-performance computer platforms have a distributed memory model. Data need to be partitioned and distributed onto the different memories. Processors do not have direct access to data placed on other processors' memories; explicit communication must be inserted to read and write such data. Autoparallelizers, including Polaris, have not yet been successful in targeting this machine class. Explicit parallel programming by the software engineer is needed.

5 Internal Organization

Polaris is organized into a number of program analysis and transformation passes, which make use of the functionality for program manipulation offered by the abstract internal program representation (IR). The IR represents the Fortran source program in a way that corresponds to the original code closely. The *Syntax Tree* has a structure that reflects the program's subroutines, statements and expressions. The compiler passes see the IR in an abstract form – as a C++ class hierarchy; they perform all operations, such as finding properties of a statement or inserting an expression, via access functions.

The objects of the IR class hierarchy are organized in lists, which are traversable with several iteration methods. A typical compiler pass would begin by obtaining a list of Compilation Units (subroutines), traverse them to the desired subroutine and obtain a reference to the list of statements of that subroutine. Next, the statement list would be traversed to the desired point, where the statement's properties and expressions could be obtained. Various "filters" are available that let pass writers iterate over objects of a specific type, only. For example, for some loop analysis pass, it may be desirable to skip from loop to loop, ignoring the statements in between. them. Among the most advanced IR functions are those that allow a pass to traverse the IR until a certain statement or expression pattern is found.

A key design principle of Polaris was that these functions keep the IR consistent at all times. It would not be possible to insert a statement without properly connecting it to the surrounding scope or rename a variable without properly updating the symbol table. These bookkeeping operations are performed internally to the extent possible; they are not visible in the IR's access functions. This design offers the compiler researcher a convenient, high-level interface. It was one reason for Polaris' popularity as a compiler infrastructure.

The implementation of the compiler consists of approximately 300,000 lines of C++ code. 45% of the code represents the IR with its access and manipulation functions; 55% implements the compilation passes.

6 Uses of Polaris

The primary use of Polaris was as an autoparallelizer and compiler infrastructure in the research community. Many contributions to autoparallelization technology have been made using Polaris as an implementation and evaluation test bed. Polaris supported other applications as well. Its ability to perform source-to-source transformations made it a good platform for writing program instrumentation passes. A simple such pass might insert calls to a timing subroutine at the beginning and end of every loop, producing a tool that can create loop profiles. Other uses of Polaris included the creation of functionality that measures the maximum possible parallelism in a program, predicts the parallel performance of application programs, and creates profiles of detected dependences.

Eighteen years after it was first conceived, Polaris continues to be distributed to the research community. The last download was recorded in 2010, in the same month this article was written.

7 Challenges and Future Directions

Polaris was able to advance autoparallelization technology to the point where one in two science and engineering programs can be profitably executed in parallel on shared-memory machines. Non-numerical programs and distributed-memory computer architectures are not yet amenable to this technology and remains an elusive research goal. In pursuing this goal, the increasing complexity of the compiler is a challenge. Learning the underlying theory and realizing its implementation is highly time-consuming for both the researchers exploring new analysis and transformation passes and the engineers developing production-strength compiler products.

One of the severe limitations of Polaris – and autoparallelization in general – is the lack of information that can be gathered from a program at compile time. Both the detection of parallelism and the mapping to the target architecture depend on knowledge of information that may be available only from the program's input data set or the target platform. Therefore, many optimization decisions cannot be made at compile-time or can only be made by the compiler making guesses. Guesses are only legal where they do not affect the correctness of the transformed program. Therefore, compilers often must make *conservative assumptions*, which may limit the degree of optimization. Future compilers will increasingly need to merge with runtime techniques that gather information as the program executes and perform or tune optimizations dynamically. Polaris explored one aspect of this area with *runtime data-dependence techniques.* These techniques detect at runtime whether or not a loop is dependence-free and choose between serial and parallel execution [11].

8 BIBLIOGRAPHIC

References

[1] M. Berry et. al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *International Journal of Supercomputer Applications*, vol. 3, no. 3, pp. 5–40, 1989.

[2] Rudolf Eigenmann, Jay Hoeflinger, and David Padua, "On the Automatic Parallelization of the Perfect Benchmarks," *IEEE Trans. Parallel Distributed Syst.*, vol. 9, no. 1, pp. 5–23, Jan. 1998.

[3] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu, "Parallel programming with Polaris," *IEEE Computer*, vol. 29, no. 12, pp. 78–82, Dec. 1996.

[4] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, pp. 84–89, Dec. 1996.

[5] D. J. Quinlan et al., "Rose compiler project," http://www.rosecompiler.org/.

[6] Chris Lattner and Vikram Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*, Washington, DC, USA, 2004, p. 75, IEEE Computer Society.

[7] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[8] Peng Tu and David Padua, "Automatic array privatization," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science.*, Aug. 1993, vol. 768, pp. 500–521.

[9] Bill Pottenger and Rudolf Eigenmann, "Idiom Recognition in the Polaris Parallelizing Compiler," *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.

[10] Jay Hoeflinger, Yunheung Paek, and Kwang Yi, "Unified interprocedural parallelism detection," *International Journal of Parallel Programming,*, vol. 29, no. 2, pp. 185–215, 2001.

[11] Lawrence Rauchwerger and David Padua, "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, New York, NY, USA, 1995, pp. 218–232, ACM.