A Performance-based Approach to Automatic Parallelization

Lecture Notes for ECE 663
Advanced Optimizing Compilers

Draft

Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University

Chapter 1

Motivation and Introduction

Slide 2: Optimizing Compilers are in the Center of the (Software) Universe

Compilers are the translators between “human and machine”. This interface is one of the more challenging and important issues in all of computer science. Through programming languages, software engineers tell machines what do to. Today’s programming languages are still rather cryptic, borrowing a few words from the English language vocabulary. Perhaps, tomorrow’s languages will be more like natural languages. They may offer higher-level expressions, allowing problems to be specified, rather than coding the detailed problem solution algorithms. Translating these languages onto modern architectures with their low-level machine code languages is already a challenge. As every generation of computer architectures tends to grow in complexity, translating future programming languages onto future machines will be an even grander challenge.

In performing such translation, optimization is important. Basic translation is usually inefficient. Performance almost always matters. The are other optimization criteria, as well. Energy efficiency is of increasing importance and is critical for devices that are battery operated. Small code and memory size can be of great value for embedded system. In this course, we will focus mainly on performance.

Optimizations for parallel machines are the focus of this course. Today’s CPUs contain multiple cores. They are one of the most important classes of current parallel machines. This was not always so. Before the year 2000, mainstream computers used single-core processors. The present course was a specialty course for those interested in parallel computation – aiming at applications that needed to run very fast. Today, optimizations for parallel execution need to be understood by all software engineers.

Slide 3: Issues in Optimizing / Parallelizing Compilers

The goal of a compiler, as seen in this course, is to translate programs written in standard programming languages onto parallel machines. We will assume a machine model that is most common in today’s multicores: A shared-memory architecture, where all processors (or cores) see the same address space and, hence, reference the same program variables. Later in the course, we will discuss alternatives to this model.

Standard programming languages, such as C, C++, Java, and Fortran, do not provide explicit constructs for expressing parallel actions. The compiler needs to derive possible parallel actions from the sequential source code. Detecting such parallelism is a first major topic of this course. Not all detected parallelism can be executed efficiently, as creating and ending a parallel activity introduces overhead (sometimes called *fork/join cost*), which offsets the gain. Mapping parallelism efficiently onto parallel machines is a second main topic of the course. The third topic deals with the question of engineering a compiler. We will discuss issues of compiler infrastructure and putting together all the learned techniques. The next three slides outline these three course topics in more detail.

Some programming languages provide explicit constructs to express parallel activities. An example is the OpenMP directive language, which can be used to annotate C, C++, and Fortran programs with parallel activities. We will use OpenMP to illustrate some of the transformations. Programmers may apply these annotations explicitly, by hand. Doing so is called parallel programming and is not the primary subject of this course. However, it is helpful to know that the same transformations presented in this course are also applied by software engineers for parallel programming. Understanding how these transformations are automated can be useful for parallel programmers as well. Hence, this course has a second target audience and will make occasional comments about manually applying the transformations.

Slide 4: Detecting Parallelism

Most compiler optimizations have two parts: An analysis and a transformation part. The analysis gathers, from the source program, the information needed to decide applicability, correctness, and best flavors of the optimizations. The transformation performs the actual code modification.

A key analysis technique in a parallelizing compiler is data dependence analysis. It determines, which statements (or expressions) need to be kept in the same order as in the original program. Other statements can be rearranged for possible parallel execution.

Some dependences can be eliminated. Techniques to do so are among the most important transformations. They can significantly enhance parallelism.

Loops are the primary target of automatic parallelization. Loops express repeated execution patterns in a program and thus tend to contain the most time-consuming code. In regular programs, after dependence detection and elimination, many loops may be completely dependence free. Such loops can be easily translated onto parallel code, where each processor executes a portion or the loop's iteration space. Partial parallelism can also be extracted from loops that contain dependences. If the compiler manages to enforce the order of dependent statements, the rest can run concurrently.

A key limitation of all optimizing compilers is that some of the decisions about applicability and correctness of the transformations depend on program input data. For example, the upper bound, n, of a loop may be a variable read from the program's input. If n is too small, then parallelizing the loop may introduce more overhead than gain. Other input variables may be directly responsible for deciding on data dependences. Some runtime data dependence detection techniques are known that can make the decision during the program's execution.

As compilers introduce code that makes optimization decisions at runtime, they also introduce additional overhead. Almost all transformation techniques introduce some degree of overhead; this fact represents the Achilles heel of parallelizing compilers. Compilers do not have sufficient static (compile-time) knowledge to make smart optimization decisions, but making the decisions at runtime adds further overhead. Unfortunately, this fact has led to the current generation of optimizing compilers delivering inconsistent performance. Programmers may experience slowdown instead of speedup as a result of automatic parallelization.

Therefore, even in automatic parallelization scenarios, the programmer needs to be involved; the decision on which programs to parallelize automatically is up to the user. By setting command line options, the user can assist the compiler in making some of the optimization decision. However, this way, automatic parallelization becomes even more interactive.

Slide 5: Mapping Parallelism onto the Machine

Overhead considerations are one reason for the existence of techniques that map detected parallelism onto the target machine. For example, a loop that can be recognized by the compiler as having too few iterations and/or too few statements may best be left in its original, serial form.

The architectural diversity of computer platforms is another reason. Parallelism exists as many levels: There are multiple cores, multiple processors, and multiple distributed, possibly heterogeneous computers; parallelism exists at the instruction level; and there is vector parallelism. We will briefly discuss techniques for each of these situations.

When an architectural unit, to which a parallel activity was mapped, references data, the data must have been mapped into the unit's address space. For the primary class of shared memory architectures considered in this course, this is not an issue. However, distributed-memory and heterogeneous architectures include non-shared address spaces; thus, issues of data partitioning, data placements, and communication become relevant. Locality enhancement – reordering computation, so that instructions that use the same data get moved closer – fall into the same category.

Slide 6: Architecting a Compiler

From our basic compiler textbooks, we know that some compiler passes can be created automatically. Lexical analysis and parsing can be automated, starting from formal specifications, such as token definitions and grammars. The compiler optimizations discussed in this course belong to the category of semantic routines, which are difficult to automate. There have been attempts to create formal specifications for compiler optimizations, but they are not in wide use. Other than lexing and parsing, all passes of the Cetus [1] and Polaris [2] compilers, which have motivated this course, have been written by hand.

The compiler-internal program representation (IR) is an important consideration. This course will discuss source-to-source transformations. Compilers that work at a level close to the source program use IRs that reflect the syntax and semantics of the programming constructs. Both the Cetus and Polaris compilers use abstract representations (in the sense of data abstraction); all elements of the IR are accessed through interface functions, hiding the low-level representation from the pass writer. Another important aspect of the IR is the way analysis passes represent their results for use by other passes. The primary means for this in both Cetus and Polaris are statement annotations. Often, these annotations are associated with a loop (statement) expressing that certain variables can be privatized to the loop or that the loop can be executed in parallel.

This course will discuss implementation considerations of the presented analysis and transformation techniques. Among these implementation aspects are correctness, profitability, as well as information needed from other compiler passes and the interaction with these passes.

A compiler contains many passes. The compiler engineer cannot simply run all passes in sequence. Analysis passes generally need to execute before transformation passes. However transformations may invalidate some analysis results. Some passes may need to be run multiple times. The orchestration of compiler techniques – the question of when to apply which technique and to which program sections – is quite complex and may need runtime information.

We will consider the performance of individual techniques as we discuss them. The many techniques in a compiler interact significantly. Hence, the compiler engineer needs to perform a comprehensive evaluation in the context of a full compiler implementation. We will look at performance results of automatic parallelization on overall benchmark programs. We will also look at the performance contributions of individual techniques to the overall results. In fact, this course will present those techniques first that make the biggest performance difference.

Slide 7: Parallelizing Compiler Books and Survey Papers

Michael Wolfe is one of the earliest architects of a parallelizing compiler. His book is a classic and is closest to the terminology used in this course. Utpal Banerjee is an expert in data-dependence analysis. He has written several books on this topic. Wolfe's and Banerjee's work has roots at the University of Illinois, where David Kuck created important foundations for dependence-based parallelization. The work of Ken Kennedy at Rice University followed Kuck's approach closely; the terminology differs slightly, however. Zima followed these approaches from a European perspective, so does the book of Darte et. al.

Among the survey articles, the Encyclopedia entry on Parallelizing and Vectorizing Compilers is easy reading of many of the techniques presented in this course. The survey by Banerjee et. al. is a more extensive, but a bit older summary of the material. The survey by Bacon et. al. is also still a very useful summary of classical compiler optimizations and parallelization techniques.

Slide 8 : Course Approach

The list of compiler optimizations is very long. This course presents the analysis and transformation techniques that have proven to make a substantial performance difference in the evaluation of automatically parallelized programs.

In doing so, this course is kept a bit light on formal analysis techniques. We do not cover the topic of data dependence analysis in the depth that other courses may choose, nor will we cover such theories as unimodular transformations and polyhedral models. For compiler developers, knowledge of these techniques is clearly useful, but we leave it to the further reading of the advanced student.

Most of the techniques will be presented assuming an underlying shared-memory (shared-address space) machine. In a later chapter, we will discuss issues and compiler techniques that are found in other architectures as well.

The course will conclude with a discussion of compiler infrastructure issues, giving importance to the fact that individual techniques need to be understood in the context of an overall, well-engineered compiler.

Slide 9: The Heart of Automatic Parallelization: Dependence Testing

Loops represent the time-consuming kernels of many computations. This is especially true in scientific/engineering applications, where numerical algorithms operate on large data volumes. Imagine you could tell that all iterations of such a loop access disjoint data. This loop would be fully parallel. The compiler (usually with assistance of the runtime system) could divide the iteration space onto multiple processors, or processor cores. These processors can execute concurrently, producing the results for their partition of the iteration space in parallel.

Data dependence testing and automatic parallelization are most successful in science/engineering applications that perform numerical computations on regular array data structures. The Cetus compiler parallelizes programs written in C, whereas the (older) Polaris compiler parallelizes Fortran.

A typical parallelizable loop in C has the form

```
for (i=1; i<=n; i++) {
   a[i] = b[i] + c[i];
}
```

The Fortran equivalent of this loop would be

```
DO i=1,n
```

```
      a(i) = b(i) + c(i)
   ENDDO
```

Fortran is still being used in many science/engineering applications, even though we are seeing an increasing number of C (and even C++ and Java) applications. Because of the more concise form of the loop statement, we will often use Fortran notation in this course.

An important characteristic of the above parallelizable loops is that their iteration space is known before the loop begins. That is, the upper bound n is not modified in the loop. Such loops are often called DO-loops, referring to the Fortran loop construct, which only allows this type of loop. By contrast, loops in the C language may modify the loop termination condition – such as loop is referred to a while-loop. (There are techniques that can transform some while-loops into do-loops). Non-numerical programs, such as business applications, data processing codes, and compilers tend to have many while-loops, which makes them difficult to parallelize. They also have other properties that add to this difficulty: They tend to operate on relatively small data spaces and make use of recursion

Slide 10: Data Dependence Tests: Motivating Examples

The iterations in the previous loop example could be easily recognized as independent; the loop index appears as the single expression in all array subscripts. In the example on the left, this task is already a bit less trivial (although still fairly simple).

One can easily imagine that loop subscripts get even more complex and data dependence testing gets difficult. Advanced technology is needed.

While, in this course, we will almost exclusively deal with the question of loop parallelization, data dependences are also important for other compiler optimizations. Can the statements in the loop on the right be reordered? If there is no data dependence, reordering is legal.

We are aiming at creating an automatic data dependence test. The test needs to be correct under all circumstances. Precise definition of all involved terms is needed. We begin by defining the term data dependence: A data dependence exists between two data adjacent references if (i) both references access the same storage location and (ii) at least one reference is a write access.

If the first reference writes the storage location and the second reads, we call this a *flow* or RAW (read after write) dependence. Write after read (WAR) is an *anti* dependence, and write after write (WAW) is an *output* dependence.

Anti and output dependences are also called *storage-related* dependences. We will see that they can often be eliminated. Eliminating flow dependences is essentially impossible, as there is a true flow of information from the writer to the reader. A flow dependence is also sometimes called a true dependence. It takes an algorithmic change to make flow dependences disappear. We will discuss several such transformations.

We do not need to consider data dependences between non-adjacent references. Enforcing dependences of adjacent references leads to the proper ordering of non-adjacent data accesses.

Slide 11: Data Dependence Test Concepts

A few more concepts and definitions are useful and will help our later discussion.

The dependence distance is the difference in loop iteration numbers between the first reference (called the *source*) and the second reference (the *sink*) of the dependence. In a multiply nested loop, there is a distance with respect to every loop that encloses the two references. Hence the distance becomes a vector. In the following singly-nested loop, the dependence is 1. In the doubly nested loop, the dependence vector is (2,0).

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

```
DO i=1,n
   DO j-1,m
      c(i,j) = c(i-2,j) + d(i,j)
   ENDDO
ENDDO
```

Sometimes, all we need to know about a dependence distance is the sign (i.e., does the dependence go forward or backward in the iteration space). We call this the *dependence direction* and use the notation "<" for forward, ">" for backward, and "=" for within the same iteration. The above doubly nested loop has a direction vector of $(<, =)$. Note that the direction w.r.t the outermost loop can never be ">" (backward). This is because a dependence is defined w.r.t the original, sequential execution order of the loop, which goes forward by definition.

Loop-carried or *cross-iteration* dependences are those with "<" or ">" direction, whereas "=" dependences are called *non-loop-carried* or simply *equal* dependences.

Some researchers use alternative notations and terms. For dependence directions and vectors, $(+1, 0, -1)$ may be used instead of $(<, =, >)$. Loop-carried and non-loop-carried dependences may be called *loop-dependent* and *loop-independent* dependences, respectively.

Slide 12: Iteration Space Graph

Graphical representations are often useful to illustrate concepts. We will use iteration space graphs to discuss execution orders and dependences. Each iteration of the loop is shown as a point. The axes show the iteration numbers of each of the nested loops. This graph is particularly useful for doubly-nested loops.

Slide 13: Formulation of the Data-dependence Problem

Recall that we are focusing on array data. A storage location is uniquely identified by the array name and subscript. Two references to the same location must have the same array name and subscript. Mathematically, this means that the two subscripts are equal, under the constraints given by the iteration space; that is, both loop subscripts are between the loop lower and upper bounds (and also considering the stride, if it is greater than 1). In the example on the slide, this yields a system of equations and inequalities as follows:

$$\begin{aligned} 4 * i_1 &= 2 * i_2 + 1 \\ 1 &\leq i_1, i_2 \leq n \end{aligned}$$

Note, again, that the two iterations where the two subscript expressions equal may be different; hence, two different iteration variables, i_1 and i_2, are being used.

Remember our earlier definition of data dependence: Two references to the same location, at least one being a write. The compiler sets up such equations for each pair of references to the same array in a loop, where at least one is a write operation. From then on, it is a mathematical problem. If the system of equations and inequalities has a solution, then there is a data dependence.

Slide 14: The Problem is Harder Than it Looks

The above is a simple case, just to get us started. Real programs have multiply nested loops and multi-dimensional arrays. Extending our data dependence formulation to such cases is not too difficult, however. The real reason that this course will go on it this:

- The above system of equations and inequalities may look innocent, but is non-trivial to solve. The reason is that the loop variables are integer-valued. In fact, if i_1 and i_2 were real-valued, the system would be over-defined, having many solutions. Solution methods for integer equations are known, but their high complexity makes them unsuitable for use in a compiler. Many approximate data-dependence tests have therefore been developed; they try to provide fast solutions with as few false positives as possible.

- Real programs do not have enough loops that are dependence free. We have already mentioned that there are dependence-removing techniques and techniques that find partial parallelism when dependences remain.

- Even if there were enough dependence free loops, they cannot always be run efficiently on today's parallel machines. The overhead of starting and ending a parallel activity incurs overhead. Furthermore, there is no machine with the pure form of shared-address space architecture that we consider in this course. Some adapting transformations are needed.

- Last but not least, implementing parallelizing optimizations in a real compiler is non-trivial. The sheer number of compiler passes, their interaction, and the need to apply the techniques beneficially pose substantial challenges.

Chapter 2

Performance of Basic Automatic Program Parallelization

Slide 16: Two Decades of Parallelizing Compilers

From the beginning of this course, we want to keep in mind the performance of the introduced optimization techniques. This is important for two reasons. First, the number of parallelization techniques that have been proposed in the literature is very large. Fortunately there is a relatively small subset that is responsible for the bulk of the performance gain in real programs. Second, all transformation techniques have the potential to *degrade performance*. Applying them indiscriminately in a compiler may lead to speedup in some programs but degradation in others. The latter is unacceptable to users – they may tolerate the lack of gain from parallelization, but will object to a parallelizer that has the potential to make the performance worse than the original program. It may be this issue that is responsible for the fact that autoparallelizers are not common tools, today. By understanding both the performance benefits and the overheads of each technique, we will acquire the necessary knowledge to create more consistently performing tools.

We will begin our performance discussion by looking at a study of the performance of parallelization techniques at the beginning of the 1990s, done by William Blume. While this is an old study, it will help us learn simple techniques first, we will understand how parallelization techniques have evolved, and we will later see that the techniques that performed well in the 1990s are simple versions of those techniques that perform well today.

Slide 17: Overall Performance of Parallelizers in 1990

This slide shows the overall program speedup, versus serial execution, after translating the Perfect Benchmarks programs with two state-of-the-art parallelizers at that time: Kap and Vast. Both compilers are Fortran parallelizers; Kap was from Kuck&Associates (now belonging to Intel) and Vast from Pacific Sierra. The programs were run on an Alliant FX/8 machine, which had 8 processors, each with a 4-stage vector unit. While the theoretical maximum speedup would be 32, a speedup of 12 was very good. Two of the applications obtained significant speedups, four other applications showed signs of speedup, and six programs could not benefit from automatic parallelization. For the Kap compiler, the study also compared vectorization-only and concurrent-only (only engaging the 8 processors) with vector-concurrent (using both). In general, where parallelization (Concurrent) didn't work well, vectorization didn't perform well, either.

Slide 18: Performance of Individual Techniques

This slide shows the importance of the individual techniques. Each bar indicates the program speedup after disabling a single technique in the Kap compiler. The first bar shows the same performance as in the previous slide (vector-concurrent).

If a technique is important, we expect a significant drop in performance after disabling the technique. One technique stands out: *Scalar Expansion* (we will discuss the techniques next). Disabling it leads to a significant performance drop in four of the applications. *Reductions* are important as well; there is a significant effect in two and a minor effect in two more programs. *Induction variables* make a minor difference in two of the benchmarks. Stripmining has a positive effect in one program, but hurts performance in three others (performance increases after disabling the technique). *Recurrences* are switched off in the compiler, by default. Enabling them has a small negative effect in two of the applications. All other techniques have no or only an insignificant effect.

Let us keep these performance results in mind, as we discuss the individual techniques. They are listed on Slide 19.

Slide 20: Scalar Expansion

Scalar Expansion is the technique of the Kap compiler with the highest impact. In the example on the slide, t is used as a temporary variable in every loop iteration. If this loop were run in parallel in its original form, the temporary uses would conflict and overwrite each others' values. In data-dependence terms, all three types of dependences (flow, anti, and output) are present. The flow dependence is non-loop-carried; it does not prevent loop parallelization. The output dependence is loop-carried, but it is not strictly a dependence, as it does not happen between adjacent references. The anti-dependence is the one to worry about. It is loop-carried. It occurs, as a next iteration overwrites a value used by the previous iteration. The next iteration tries to re-use the storage location; hence the term storage-related dependence. Recall our earlier claim that storage-related dependences can often be eliminated.

One can easily see that t is used only locally, within one loop iteration. The value of t is not used after the iteration (except, perhaps the last value of iteration n, which we will discuss later). If the compiler can find a way to give each iteration a separate version of t, these conflicts will disappear.

Scalar Expansion transforms t into an array, with the loop variable being the array index. This way, each iteration has its own version of the temporary variable. The same effect can be achieved by declaring t *private* to the loop. Private variables will obtain a separate storage location for each loop iteration (in reality there is only a separate storage location for each processor, which suffices, because all iterations assigned to a processor execute serially).

Scalar expansion looks less efficient than privatization. Historically, scalar expansion existed before privatization, because the earliest parallelizers were actually vectorizers. As we will see later, vectorization requires expansion rather than privatization.

If the value of t is used after the loop, we call it *live out* of the loop. A *last-value assignment* is needed. The simple-most form of last-value assignment is an explicit statement, inserted after the loop: `t = c(n)`. More about last-value assignment will be discussed later.

Slide 21: Parallel Loop Syntax and Semantics in OpenMP

At this point, we insert a brief discussion of parallel loop syntax. The previous slide used informal notations `DO PARALLEL` and `PRIVATE` for expressing a parallel loop and a private variable, respectively. Sometimes we will use the OpenMP parallel directive standard to express parallelism. A simple example of a parallel OpenMP loop expressed in the C language is shown on the left. In C, OpenMP directives are inserted in the form of pragma statements, beginning with `#pragma omp`. Placed before a loop, the `parallel for` directive tells that the following loop is to be executed in parallel, with the loop's iteration space being split among the participating threads.

A more involved form is shown on the right and for a Fortran program. Fortran OpenMP directives begin with the keyword `!$OMP`. Private data can be declared as such with the `PRIVATE(...)` clause. In OpenMP, the two actions "begin parallel execution with multiple threads" and "split the loop iteration space among available threads" can be expressed separately. The first action is invoked with `OMP PARALLEL`; it creates a *parallel region*. The second action is created with `OMP DO`. This is called a *work-sharing construct* and is equivalent to `omp for` in C programs. Within parallel regions but outside of worksharing constructs all threads execute the same code. This allows loop "preambles" and "postambles" to be created. We will make use of this feature in the next transformation.

Slide 22: Reduction Parallelization

The parallelization of reduction operations was the second-most important transformation in the Kap parallelizer. In general, a reduction transforms an n-dimensional data structure into an (n-1)-dimensional result. The example on the slide shows a scalar reduction. It reduces the array a by summing it up. The difficult, loop-carried dependence in this code is a flow dependence. Recall our earlier claim that it is essentially impossible to eliminate flow dependences – an algorithmic change is necessary.

The idea for parallelization of this code is quite straightforward. Different processors can create partial sums of a concurrently; then the partial results can be combined. The code on the right shows how this idea can be coded in OpenMP. The core of the code, inside the work-sharing construct, is essentially the same as the source code, except that it uses a private variable s for keeping a partial sum on each thread. This partial sum is initialized to zero in the preamble and added to the original *sum* variable in the postamble. As each thread concurrently attempts to modify *sum*, a `OMP ATOMIC` construct is used; it makes sure, only one thread can modify *sum* at once.

Because reduction patterns are common, OpenMP has a special construct to express them. With the `REDUCTION` construct, the source code does not need to be modified; only OpenMP directives are inserted. The code transformed by the OpenMP compiler will look similar to the code on the right.

Consider the performance of this code. The preamble and postamble code adds overhead. The initialization of s in the preamble is done fully concurrently, but the postamble essentially serializes the update of *sum*. These overheads add to the loop's fork/join cost, which needs to be offset by the gain from parallel execution. In general, a reduction benefits from running in parallel if the reduction size (the bounds of the original loop) is large. It is also very important that the `OMP ATOMIC` construct be very efficient. A good implementation will enforce the indivisible update of the global memory location by inserting special instructions, such as *test-and-set*. An inefficient (albeit correct) implementation would use a *critical section*, which may consist of a library call with software-implemented synchronization).

Let us come back to the question of algorithmic change. Does the transformation we have applied represent an algorithmic change? The answer is *yes*. We have made use of the mathematical properties that sum operations are associative (subsets of the summation are done by different threads) and commutative (the threads may update the global *sum* variable in arbitrary order). The compiler by itself does not know that it is OK to build partial sums. Doing so would violate the order of the original sum operations. This order is represented by the flow dependence. Our math knowledge allowed us to break the flow dependence. What is true for addition, may not be true for other operations.

Researchers have pointed out that, while mathematical addition is commutative and associative, computer arithmetic is not. Due to the limited precision of the digital representation, the results of sums in different combinations differ. For most applications, available computer precision is sufficient and this is not an issue. However, some science/engineering applications may be sensitive. Compiler writers need to keep this in mind, as parallelization is most applicable in science/engineering programs. Parallelizing compilers often include an option with which the user can enable/disable the parallelization of reductions.

Addition is the most common reduction operation. Other operations include multiplication, taking the minimum, and taking the maximum. The transformations for these types of reductions are analogous to the one for addition.

Slide 23: Induction Variable Substitution

Induction variables represent sequences of values. Each element of the sequence is computed from the previous value – by a constant offset, in the simplest case. The use of the previous value creates a flow dependence. Again, an algorithmic change is needed to eliminate this dependence. The idea is to use a closed-form computation of the sequence. The slide illustrates such a case. After the substitution, the induction statement can be eliminated and the data dependence disappears with it.

Computing the closed form is more costly than computing the next element of the sequence. In the example, an addition is turned into a multiplication. As we will see later, more complex induction variables have an even more involved closed-form computation. What was true for reduction parallelization holds here as well: The increased overhead must be compensated by the performance increase from the parallel execution.

The lower cost of computing the induction sequence is a well-known fact among compiler writers. In fact, the strength-reduction technique of classical compilers performs the opposite transformation. Strength reduction aims to replace the multiplication – as it typically results from array address computation – by addition, which is an operation of lower "strength" and faster. The introduced data dependence does not matter in sequential machines. In parallel machines, however, there is a tradeoff between the gained parallelism and the added cost from substituted induction variables. We will come back to this tradeoff in the discussion of advanced induction variable substitution.

Slide 24: Forward Substitution

Forward substitution gathers knowledge about the values and relationship of variables and makes it available to the places needed – typically the analysis and transformation passes optimizing a given loop. Of all considers techniques in Kap, forward substitution is the only one that operates at the global program level (even though it does not perform interprocedural analysis). All other techniques operate on a particular loop or loop nest. Forward substitution captures the expressions assigned to variables and then substitutes the uses of these variables with the assigned expressions.

The example on the slide shows that such knowledge can be important for disproving data dependences. It is also important for deciding on profitability of parallelization. The evaluation of Kap shows little effect of forward substitution, however. In fact, there is a slight negative impact in one program (SPEC77). Like all techniques, forward substitution can cause overhead: Substituting an expression in place of a single variable adds execution cost. The technique can create long expressions. One common engineering compromise is to apply forward substitution only to integer expressions (which are the expressions that often need to be known for making optimization decisions) and not to real-valued expressions. Another method to keep the overhead low is to do the substitution on a "shadow IR". That is, the substituted expression is visible to compiler optimizations but will not be applied in the transformed program. The Kap parallelizer used this method.

Another source of overhead caused by forward substitution is that it may enable additional parallel loops, but these loops are not profitably parallel. This is an issue with most program analysis techniques and may explain the effect in SPEC77.

Forward substitution eliminates flow dependences. The substituted statement no longer depends on the original variable. Instead, the statement recomputes the value. The "algorithm change" that is being done is the replacement of the reuse of a value with the recomputation of that value.

Similar to the relationship between induction variable substitution and strength reduction, forward substitution relates to *common subexpression elimination (CSE)* – a classical compiler optimization. CSE recognizes redundant computation; it keeps the result of the computation in a temporary and uses that temporary instead of the repeated computation. In doing so, it introduces a flow dependence from the write to the read operation of the temporary. This is the opposite action of forward substitution, which may insert redundant computation, whereby eliminating dependences. Recall that the transformation does not always perform the actual substitution. When it does, it introduces the very overhead that CSE tried to eliminate.

Slide 25: Stripmining

Stripmining (or loop blocking) splits a loop into two nested loops. The iteration space of the original loop is partitioned so that, in the given example, the inner loop iterates over a *strip* and the outer loop steps from one strip to the next. Note that the last strip may be cut short, requiring the min function in the upper bound of the inner loop.

There are many variants of stripmining. A variant without a min function is this:

```
DO i=1,strip
   DO j=i,n,strip
      a(j) = b(j)
   ENDDO
ENDDO
```

The Kap compiler applied stripmining in order to employ the vector hardware as well as the parallel processors in singly-nested loops. The inner loop got vectorized (see the later discussion on vectorization) while the outer loop got parallelized. Because the length of the vector registers in the Alliant FX/8 architecture was 32, the strip was fixed at that number. In the FLO25 program, the combined vector-concurrent execution is beneficial – the performance drops after disabling stripmining. In TRFD, MDG, ADM, and SPEC77, however, performance increases when disabling stripmining. Many singly nested loops in these programs are too short, causing stripmining in add more overhead than benefit.

Slide 26: Loop Synchronization

So far we have discussed data-dependence analysis and three techniques that help eliminate dependences – scalar expansion (or privatization), reduction parallelization, and induction variable substitution. These three techniques were the most important ones in the Kap compiler. Together, they help create fully parallel loops, which turns out to be the most beneficial form of parallelism.

Researchers have proposed several techniques to deal with loops where dependences remain. One of these techniques synchronizes data dependences, such that statements that depend on others get delayed until the first statement has executed, while the rest of the loop runs in parallel. Such loops are often referred to as *doacross* loops, while fully parallel loops are called *doall* loops (owing to early parallel Fortran dialects that used these keywords for DO-loops that were either fully or partially parallel.)

The reference `a(j-1)` in the example on the slide reads, in any iteration j', the value produced by the reference `a(j)` in iteration $j' - 1$. Hence there is a loop-carried flow dependence. The inserted synchronization construct *wait()* delays the execution until a corresponding *post()* function has been executed. The positioning of these constructs in the code is so that *post()* is executed immediately after the dependence source and *wait()* right before the dependence sink. The function parameter is usually the current iteration for *post()* and the current iteration minus the dependence distance for *wait()*.

There is an important requirement for the scheduling order of the iterations – they must be started in the original, sequential order. If a later iteration started before an earlier one, and if these two iterations were dependent, the later iteration would incur an additional delay. There is even the possibility of deadlock. It occurs if (i) all iterations are in a wait() operation for iterations that have not yet started, or (ii) if the dependent iterations are scheduled on the same processor but not in sequential order (the latter is assuming that wait() does not relinquish the processor, which is the most efficient implementation of this operation.) A common iteration-to-processor assignment is *block scheduling*. It assigns to all processors a contiguous subset of the original loop iteration space. This schedule violates the above ordering requirement. Instead, *cyclic scheduling* is best, where the first of p processors executes iterations 1, then $p + 1$, then $2p + 1$, etc.; processor two executes iterations 2, $p + 2, 2p + 2$; and so on.

If all processors execute at the exact same speed and if the synchronization constructs are very efficient, the code on the slide executes fully parallel. The synchronization ensures that the data dependence is enforced, even if the processors differ in execution progress (which is almost always the case).

In the given example, the data dependence is *lexically forward* (from an earlier to a later statement in the order the program is written). If the dependence is lexically backward (but still forward in execution order, as it goes from an earlier to a later iteration), the synchronized parallel loop is only partially parallel – even with perfectly efficient synchronization. The extreme case would be a *post(i)* at the end of the loop iteration and a *wait(i-1)* at the beginning of the next iteration, which would completely serialize the loop. This would be caused by the first statement of a loop iteration reading a value produced by the last statement in the previous iteration.

The cost of the synchronization constructs is critical. The measured Alliant FX/8 machine had hardware synchronization capabilities – *post* and *wait* were translated directly into corresponding instructions, performing the synchronization (there was an additional parameter – the synchronization register number; eight such registers were available, allowing the synchronization of up to eight data dependences in a loop.) Today's multicores do not usually provide such instructions.

Doacross-style loop synchronization was an important research topic in early parallelizing compilers. The measurements in the Kap compiler show insignificant performance impact of these capabilities, however. This may explain why today's parallelizers do not apply this type of loop synchronization and why multicores do not include corresponding synchronization hardware. As the importance of parallel non-numerical programs increases, so does the number of programs that exhibit many data dependences. This fact may lead to renewed interest in such synchronization.

Slide 27: Recurrence Substitution

A recurrence is a recursively defined sequence. The induction variables discussed earlier are simple, first-order recurrences, where the recursive definition only uses the previous value in the sequence. A more general form of a linear recurrence of order k is this:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + ... + c_k a_{n-k}$$

A loop with dependences may resemble this equation. For example, the following loop computes a 3^{rd} order linear recurrence:

```
DO i=1,n
   a(n) = c1*a(n-1) + c2*a(n-2) + c3*a(n-3)
ENDDO
```

Even though this loop has three flow dependences with distance one, two, and three, respectively (which seems impossible to eliminate), researchers have developed algorithms to compute recurrences in parallel. The Kap compiler was able to recognize such patterns and substitute the loop with a call to the parallel recurrence solver, as shown on the slide.

We refer to the literature for details of parallel recurrence solvers [3, 4]. The slide illustrates the basic idea: In step one of the algorithm, the original loop of 40 iterations is executed in parallel on four threads (thread 0 through thread 3) as shown by the four loops of 10 iterations. In doing so, the threads make an error. Thread 1 (with iteration space 11–20) uses the original value of a(10), instead of the updated value computed by the recurrence. This error propagates as a constant through all values computed by thread 1; we call it $\Delta a(10)$. Thread 2 makes a similar error, using the original a(20) instead of the correct one. The error is $\Delta a(20)$ (the difference of the original a(20) and the value after step one,) plus $\Delta a(10)$ (thread 1's error). The error made by thread 3 is $\Delta a(10) + \Delta a(20) + \Delta a(30)$. In step two, these errors are computed. This is similar to computing a reduction operation. In step three, the values of the array a are adjusted by the computed error. This can be done in parallel.

Step one and three are fully parallel. Step two is partially parallel. If the recurrence length is much greater than the number of threads, then step 2 is negligible. The algorithm uses twice as many operations as the original loop, but is essentially fully parallel.

Unfortunately, the results of the Perfect Benchmarks do not confirm the speed advantage. In fact, by default, recurrence substitution is switched off. When switching it on, the performance drops by approximately 5% in two programs (FLO52 and ARC2D). The technique has no effect on the other programs.

There are two reasons for the lack of performance. Even though the benchmarks contain a number of loops that can be translated into parallel recurrences, the recurrence lengths are short. Furthermore, the compiler needed to add code to convert the original arrays into forms that can be passed as parameters to the recurrence solver (e.g., converting strided accesses into a contiguous block). These overheads offset the gains from parallel execution, in the given benchmarks.

Slide 28: Loop Interchange

This transformation swaps two loops of a nest. Here, we consider only perfectly nested loops. These are nests where both loops enclose all statements of the loop body. More advanced loop interchange is possible.

Loop interchange has two important effects: enhancing locality and increasing parallel granularity.

Enhancing locality: Consider the left loop on the slide. Fortran has *column-major order;* that is, increasing the left-most array index accesses the adjacent storage location (by contrast, C programs have row-major order.) Because the inner loop index, j, uses the right-most dimension, this code has non stride-1 reference behavior – consecutive statements jump in the referenced addresses. Loop interchange creates stride-1 behavior, which has two important locality advantages. The first is spatial locality. Adjacent array locations are likely placed in the same cache line. As the next statement references this location, the access is a cache hit (for non stride-1 behavior chances are that the cache line has been replaced by the time the computation accesses the adjacent array location). The second advantage is reduction or avoidance of false sharing. Consider the outer loop in the left code being parallelized by splitting the iteration space onto the available tasks. At the split points, two adjacent tasks write onto adjacent storage locations. If these locations happen to use the same cache line, false sharing results, causing substantial overhead. Placing the stride-1 loop within the same task avoids this overhead.

The scheduling (assignment of iterations to threads) of the outer, parallel loop is important. So far, we have considered *block scheduling*, where the original iteration space is partitioned into p parts (p being the number of tasks/processors). Scheduling can also be *cyclic*, where the first task executes iterations 1, 1+p, 1+2p,..., task 2 executes iterations 2, 2+p, 2+2p,... and so on. Cyclic scheduling exacerbates false sharing in the above example, as the threads write onto more adjacent storage locations. Scheduling can also be dynamic, where the *runtime system* distributes iterations to available threads. Dynamic scheduling can reduce load imbalance (which results from threads executing conditional code or processors proceeding at different speeds). From the discussion on false sharing it should be clear that it is advantageous for dynamic scheduling to hand out chunks of iterations, not individual ones.

The organization of the cache is an important consideration as well. Today's cache architectures include distributed caches. False sharing is an issue associated with these organizations, because multiple cache lines of the same storage may exist simultaneously. Higher-level caches (e.g., L3) are often shared organizations. In shared caches, false sharing does not happen. Also, locality is preserved across accesses by different threads.

Increasing parallel granularity: Creating and ending a parallel activity causes overhead (a.k.a fork-join cost). The larger a parallel activity is, the better is can offset that cost. In the code on the slide, the i-loop is parallel while the j-loop is not. In the left code, the fork-join cost is incurred exactly once, while the interchanged code will create and end a parallel loop m times (once for every iteration of the outer loop).

This effect of loop interchange can be critical, as small parallel loops may cause more overhead than benefit. Unfortunately, the two effects (locality and granularity) do not always go hand in hand. In fact, from the locality point of view, the right loop nest is preferred; from the granularity point of view, it's the left one. Detailed performance models would be needed to decide on the preferred form on a case-by-case basis. The performance models would need to formulate the cache behavior at all levels, the fork-join costs (both at the architecture and the software level) and the amount of code in each task. Current compilers do not include such complex models. Simple heuristics may take their place. On today's multicore architectures, locality considerations are often more important than those of granularity.

Slide 29: Effect of Loop Interchange

Loop interchange does not show significant effects on the Perfect Benchmarks parallelized with the Kap compiler and run on the Alliant FX/8 machine at the beginning of the 1990s. We attribute this fact to two architecture properties, both of which have changed significantly in the past two decades: (i) The ratio of memory versus cache access time was relatively low and (ii) the FX/8 architecture has a shared cache organization. Given these properties, the above mentioned heuristic for granularity versus locality would have been reversed. Granularity considerations were the overriding factor.

The figure on the slide shows that loop interchange *does* make a significant difference in the ARC2D code on a modern 4-code processor, measured two decades later. The figure shows the speedup of the four most time-consuming loops of the program. Two of the loops obtain superlinear speedups of close to 10. The explanation is that loop interchange had been applied in the process of parallelization. Note that the same transformation would be applicable to the original, sequential program as well. However, even though the benchmark has been "optimized" by hand, it did not have this transformation applied.

An interesting historical development is worth noting. Locality enhancement was indeed first applied in the process of automatic parallelization. While parallelizing compilers where largely a niche technology, it was discovered that these tools could actually speedup sequential programs. For a while, parallelizing compiler made more revenue being sold on single-processor systems (which was the dominant class of machines, at that time) than on parallel machines. Today, locality enhancement techniques are integral parts of most single-core compilers.

Slide 30: Execution Scheme for Parallel Loops – Architecture Support

So far, we have tacitly assumed that a parallel loop can somehow be executed efficiently on a multi-processor architecture. We conclude the chapter by considering how this is done.

The two-decade-old Alliant FX/8 architecture had instruction support for executing parallel loops. The Alliant Fortran compiler translated parallel do-loops into the instruction sequence shown on the slide. The `cdoall` instruction initiates the parallel execution of the eight processors, given the execution was sequential before. A parameter (register *D6*) contains the number of iterations. A reserved register, *D7*, acts as the iteration variable on each processor. There is fast hardware support for setting and advancing this register on each processor: Idle processors fetch the next available iteration until the loop is completed.

Such hardware was unique in the Alliant FX architecture. Today's multicores do not provide this level of support. Software solutions are needed. A common implementation is the *microtasking scheme.*

Slide 31: Execution Scheme for Parallel Loops – Microtasking Scheme

A parallel loop could be implemented using Unix or pthreads fork functions. The fork-join overhead would be too high, however. Unix or pthreads fork operations take in the oder of 10s to 100s of microseconds. By comparison, many loops that would like to be run in parallel execute in less than 10 microseconds. We have already discussed that high fork-join costs are a major impediment to efficient automatic parallelization. We need to develop a scheme that is highly efficient. *Microtasking* is such as scheme; it dates back to early IBM mainframes.

The basic idea of microtasking is simple: Create all threads at the beginning of the program; put them to sleep and wake them up when needed. Fast wake-up is key to performance.

The implementation of this scheme may make use of pthreads. The one-time cost of task creation is incurred at the beginning of the program, where it is not significant. Wake-up is often done using a busy-wait implementation. The next slide shows the compiler transformations as well as the involved functions of the runtime library.

Slide 32: Compiler Transformation and Runtime Function for the Microtasking Scheme

In the microtasking scheme, the compiler extracts the body of the parallel loop into a *loop-body subroutine* and replaces the original loop with a call to a runtime function, `loop_scheduler()`. The parameters of the loop scheduler function include the code to be executed (the loop-body subroutine), the loop variable and bounds $(i, 1, n)$, and the addresses of all shared variables accessed in the loop (variable b). The loop-body subroutine will be called in each participating task (helper task). The subroutine parameters include a task number (*mytask*), a partition of the iteration space (lb, ub), and the same list of shared variables that was passed to the loop scheduler function.

The loop scheduler function is the core of the runtime library. It is invoked on the master thread during the sequential execution of the program. It is responsible for invoking and coordinating the executions of the loop-body subroutine on all participating threads, and for properly ending the parallel activity. It does this by

1. partitioning the iteration space for the available threads,
2. writing control blocks in shared address space for each thread, informing them of the code to be executed (loop-body subroutine), the assigned loop iteration space partition (lb, ub), and shared variables (b);the control block also includes a flag indicating *sleep* or *awake*,
3. waking up the tasks, by setting their flag to *awake*,
4. executing a share of the iteration space on the main task by calling the loop-body subroutine,
5. waiting for all tasks to indicate completion by resetting their flags to *sleep*.

The helper tasks are created and put to sleep by the initialization function of the runtime library, which also sets up their control blocks. The helpers wait as long as their flag is set to sleep. On wakeup, they access their control block and call the indicated loop-body subroutine with the indicated parameters. After the loop-body subroutine returns, they reset the flag and put themselves back to sleep.

Multi-level parallelism: Common implementations of languages that provide parallel loops do not support nested parallelism. Most parallelizing compilers serialize a loop that is inside of an already parallel one. If a parallel loop is inside a subroutine, the compiler may not be able to detect this situation; the same subroutine containing a parallel loop may even be invoked from both a serial and a parallel section within the same program. In this case, the runtime library may check if the code is currently executing in parallel and, if so, execute the entire loop by the one thread that calls the `loop_scheduler()`.

If multi-level parallelism is supported, the runtime library needs to decide how many and which helper threads to engage at each level. This decision often needs runtime knowledge. At the outer fork operation, it may not be known if there is inner parallelism and, if so, how much resources are needed. Some researchers have proposed user annotations with which parallel loops can be told how many threads to use.

Chapter 3

Performance of Advanced Parallelization

Slide 34: Manual Improvements of the Perfect Benchmarks (1995)

Recall the overall results of the "Blume study". The measured Kap and Vast compilers – among the best industrial parallelizers in the 1990s – could achieve significant improvements in only 2 out of the 13 Perfect Benchmarks (ARC2D and FLO52) and minor improvements in a few more. The third column of the table measured the same performance. The fourth column shows the performance of the same programs on the Cedar machine [5], which had 32 processors (4 Alliant FX/8 machines with an additional global memory).

As in the "Blume study", the results do not list the 13th code, SPICE, of the benchmark suite. This code is a hopeless case for parallelization. SPICE is widely used in industry for device simulation. It is highly unstructured and does not operate on large, regular matrix data. Instead, it processes device lists, which are best created with dynamic data structures. However, as all Perfect Benchmarks, SPICE is written in Fortran 77, which does not support memory allocation. Fortran-77 programmers dealt with dynamic data structures by declaring large arrays and carving variable-size data structure out of them. SPICE is also the largest of the Perfect Benchmarks (including some 20,000 lines). It represents the class of data processing codes, which are not amenable to automatic parallelization.

An important question that remains after the "Blume study" is whether or not the "2 out of 10" success rate for automatic parallelization (on science/engineering applications) can be increased. By manually improving the Perfect Benchmarks, a follow-on project [6] tried to answer this very question. The project used transformations that could eventually be implemented in a compiler. No algorithmic changes were used. Columns five and six show the results of this study on the Alliant FX/8 and the Cedar machines, respectively. In all cases, the manual improvements achieved significant additional over the automatically parallelized code.

Some of the codes (e.g., BDNA, TRACK) still show low speedups. An interesting question is whether or not these numbers could be improved through additional optimization, such as algorithmic changes. A partial answer to this question was given by an effort that was made at the same time, with the goal of improving the Perfect Benchmarks through algorithmic modifications. The results did not exceed those shown in the table. Unfortunately, the effort was never published; it was important, nevertheless, as it indicated that automating the techniques used in this project [6] may create a parallelizer that can compete with hand parallelization. In the following, we will take a closer look at these techniques.

This project concluded that most of the techniques applied could be automated in a compiler. Among the exceptions are some techniques applied to MG3D (elimination of file I/O) and to QCD (parallelization of a random number generator). User knowledge helped decide that these transformations were valid.

Slide 35: Performance of Individual Techniques in Manually Improved Programs (1995)

The table on this slide again shows the Perfect Benchmarks (excluding SPICE). The experiments were run on the 32-processor Cedar machine.

Similar to the "Blume study", which switched off individual optimization techniques and measured the resulting performance drop, the table shows the reduction in speedup when *not* using the indicated technique. For example, when not privatizing arrays, the MDG code takes 21 times longer to execute. This is because the main loops in this code consume 98% of the execution time and need array privatization in order to be parallelized. The table shows that array privatization is by far the most important of the advanced techniques. It impacts all but one application (FLO52). Recall that scalar expansion (which is equivalent to scalar privatization) was found to be most important in the KAP compiler.

The second most important technique is the parallelization of complex reductions. These are reduction operations that use arrays as targets instead of scalars, as we will discuss in the next chapter. The same loops in MDG that need array privatization also need parallelization of complex reductions in order to run in parallel. Reduction parallelization significantly affects six of the programs.

The third technique is the substitution of generalized induction variables. The technique affects only two of the benchmarks; however, the impact is high. The two codes would include an 8-fold and 12-fold performance reduction without this transformation.

The fourth technique shown in the table is an advanced data-dependence test. Most data dependence analysis techniques work in the presence of affine subscripts. Affine expressions are linear combinations of the loop indices of enclosing loops, with the coefficients being integer constants. Non-affine expression include those that have coefficients that are variables (which is the case in OCEAN), nonlinear polynomials (TRFD) and array subscripts (TRFD) – called subscripted subscripts. The polynomials in TRFD were introduced by induction variable substitution. As we will see, the substitution of generalized induction variables introduces non-linear expressions; hence, advanced induction variable substitution needs non-linear data-dependence tests.

Slide 36: Overall Performance of the Cetus and ICC Compilers (2011)

The Polaris and the Cetus compiler were directly motivated by the project shown in the previous two slides. A new study, performed in 2011, re-evaluated the importance of the techniques in Cetus on a modern multi-core processor, an 8-core x86 platform in 2011. The programs used in this study are the NAS Parallel benchmarks (NPB) – the OpenMP versions of NPB were used; serial versions were derived by ignoring the OpenMP directives. The NAS benchmarks represent code sections extracted from real computational fluid dynamics codes. Hence, they represent the cores of important science/engineering applications. The programs are less complex than the Perfect Benchmarks; however, they include data sets suitable for today's machines, whereas the data sets of the Perfect Benchmarks have never been updated (the codes would run on today's machines in a fraction of to a few seconds).

This slide shows the overall results. It compares the Cetus performance with an important commercial parallelizer, which is part of Intel's ICC compiler, and with hand-parallelized versions of the programs. The best version of Cetus (Empirically Tuned) shows significantly better speedups than ICC. This indicates that not yet all of the advanced parallelization techniques developed in research compilers have been transferred to industrial products. In four of the programs, the best Cetus version matches, and in one of these cases outperforms, the hand-parallelized codes. We again find that, in roughly 50% of scientific/engineering codes, parallelizers succeed.

The figure also shows the performance of several "tuned" Cetus versions. Tuning in Cetus addresses the need of the parallelizer to move some of the optimization decisions to runtime. The "Cetus Untuned" variant eagerly parallelizes all possible loops. This leads to substantial overheads in about half of the codes. "Model-based Tuned" uses a simple compile-time performance estimator to decide whether or not the product of the number of statements and loop iterations exceed a threshold (it evaluates this formula at runtime if it is symbolic.) This test avoids the most severe overheads, such that the parallel code is at least no worse than the original serial program performance. "Empirically Tuned" makes use of an advanced tuning system that tries many combinations of optimizations, executes them, and picks the best. This is done in an offline manner, similar to profile-based optimization [7]. This form of dynamic decision support has a pronounced impact over "Model-based tuned" in three of the programs.

Slide 37: Performance of Individual Cetus Techniques (2011)

This figure measured individual techniques in Cetus. Similar to the earlier such study, one technique at a time is switched off and the resulting performance measured. Three classes of techniques are measured: Program analysis, parallelism enabling, and locality enhancing techniques.

The transformation technique with the highest impact is Privatization. We expected this, as the earlier studies have found privatization to be important as Well. Similarly, reduction parallelization is the second most transformation technique, affecting two of the programs.

Induction variable substitution also affects two programs significantly.

All analysis techniques are important. Symbolic analysis affects half of the programs. Inlining affects all programs, except those where parallelization makes no difference in the first place. The same holds for alias analysis.

The locality enhancement techniques show an insignificant difference. While we did not expect a big impact – after all, locality has already been enhanced in the the hand-parallelized NPB – the result is sill somewhat surprising, as the best versions (All On) were generated through empirical tuning, which finds the best combination of optimizations for the given architecture. This architecture is different than the one used when hand-parallelizing the benchmarks. As explained in [7], the two techniques, loop interchanging and tiling, are *substituting.* They have similar effects; when switching off one of them, the other takes over. Switching off both, drops the performance by 15% in SP.

Let us now compare these results with the earlier performance studies. The parallelism-enabling transformation techniques that were found important are still very important today; advanced versions thereof are key in today's parallelizers. They will be the first to implement for a software engineer developing an industrial parallelizer.

The analysis techniques tell a slightly different story. Forward substitution can be considered a simple version of symbolic analysis. While it had no significant impact in Kap, it is important now. The explanation is possibly that the technique needed to mature to a certain point, where it begins to make a difference. Another reason may be that the simpler optimization techniques in Kap did not need advanced analysis, whereas the more advanced techniques in Cetus do.

Inlining takes the place of inter-procedural analysis. Even though inlining was available in Kap, it was not included in the study. Indiscriminate inlining can have a negative impact on performance. Therefore Cetus subjects inlining to tuning as well; it is applied only where of benefit and, in this form, contributes significantly to the obtained performance. Alias analysis was not present in Kap, as it was a Fortran compiler; the Fortran language defines that subroutine parameters can be safely assumed as non-aliased by the compiler. This is not so in C programs. Not having alias information available would force the compiler to make too many conservative assumptions about data dependences, essentially serializing the programs.

Loop interchange was measured in the study of Kap, but did not make a difference. Because of the much higher processor-to-memory speed ratio today, we expected this technique, as well as tiling, to make a bigger difference. We have given possible explanations for the small measured impact, above. Further study is needed.

Chapter 4

Analysis and Transformation Techniques

We now proceed to a more in-depth discussion of the analysis and transformation techniques that are important constituents of a parallelizing compiler. This course can touch on only a small part of the large body of existing transformation techniques. The priority is given by the performance seen in the studies that we discussed. The given references to books and survey papers are important resources for learning about additional techniques and for further in-depth discussions of the presented techniques.

4.1 Dependence Analysis

Slide 39: Data-dependence Testing

Recall our earlier data-dependence test example. The loop contains a write and a read access to array a with the subscripts $4 * i$ and $2 * i + 1$, respectively. We had defined a system of equations, $4 * i_1 = 2 * 1_2$, and inequalities, $1 \leq i_1, i_2 \leq n$. The slide shows the general form for a singly-nested loop with *lower* and *upper* bounds and a one-dimensional array with two subscript functions f and g, respectively.

Real programs have multiply-nested loops that access multi-dimensional loops. The following two slides show the systems of equations and inequalities for such cases.

Slide 40: DDTests: doubly-nested loops

This slide shows a doubly nested loop with an access to a one-dimensional array. The array subscript function is now a linear combination of the loop indices. As in the previous example, the mathematical system has one equation, which equates two subscripts. However, there are two sets of inequalities – corresponding to each of the two loops.

Recall that, if the coefficients a, b, c are integer constants, the subscript functions are called *affine*. Most data dependence tests work well with affine expressions. Most array subscripts have simple, affine form. An early study [8] even found, that the majority of array subscripts contain a single loop variable, with the constant offset (c on the slide) being 1, 0, or 1.

More complex subscripts are seen occasionally. If they are present in time-consuming loops of an application, dealing with them is important. The expressions can be polynomials of the enclosing loop variables of degree two or higher, requiring a non-linear data-dependence test. If the coefficients (a and b) are variables with unknown values, many data-dependence tests also consider the subscript nonlinear. The same holds in the presence of "unknown" terms in the subscript expression. Such unknowns could be arrays (subscripted subscripts), non-basic mathematical operators, and function calls. Some dependence tests include methods to deal with these non-linear subscripts in various ways.

Symbolic program analysis techniques are important in this context, as they help gather knowledge about the values of the terms in an expression. For example, Forward Substitution is a basic form of symbolic analysis; the example on Slide 24 propagated critical knowledge about the value of variable m to the loop being tested for data dependences.

Slide 41: DDTests: even more complexity

The common case is that n nested loops contain accesses to an n-dimensional array; each loop traverses one of the loop dimensions. For example, a frequently seen form is a doubly nested loop operating on a 2-dimensional data space:

```
for (i=0; i<n; i++) {
   for (j=0; i<m; j++) {
      a[i,j] = ...
   }
}
```

The slide shows a doubly nested loop accessing a 2-dimensional array with the most general form of a linear expression. In this general form, all loop variables of the enclosing loops show up in every array dimension.

The mathematical system now has two equations – one for each array dimension – and two sets of inequalities – one for each enclosing loop. Notice, that such a system needs to be solved for every possible dependence in a given loop. Compilers enumerate all pairs of array references in a loop nest and solve the corresponding system of equations/inequalities to test for independence. In the example on the slide, there is one pair of references that needs testing – between the write and the read access to array X.

Recall that the potential self output dependence of the first statement does not need to be tested, because the two references are not adjacent. If the second statement did not exist, this test would be needed. Another consideration is the type of the dependence, anti or flow, between the write and read access. The mathematical system is the same for both flow and anti dependences; the type is given by the order of the two accesses. This order cannot easily be seen without doing the test; it results from the mathematical solution as well. We will see how "data dependence testing with direction vectors" can be done.

Slide 42: Data Dependence Tests – The Simple Case

Let us now turn to the solution of the mathematical system. Data dependence testing was the first technology developed for parallelizing compilers. There exist many excellent textbooks; we will keep our discussion brief.

Recall that the equations and inequalities are integer-valued. Solving such systems is more difficult than finding real-valued solutions, which are the subject of most basic math courses. However, there is a well known and exact mathematical solution method for solving integer-valued equations as well – often referred to as *integer programming*. Unfortunately, the complexity of the general form of these methods is very high, making them impractical for use in a parallelizer – the user of the compiler might see very long execution times for some programs. Whenever an problem solution is of high complexity, engineers develop approximate methods. The challenge for approximate data dependence tests is to be fast, but produce very few false positives. Note that false negatives are not permitted; they would lead to incorrect programs. The tests check for *independence*; if they cannot prove it, they assume that there is a dependence.

Slide 43: Performing the GCD Test

The GCD test can be extended to integer equations with more than two variables. If the GCD of all loop-variable coefficients evenly divides the constant, there is a solution, otherwise there isn't.

The old Euklid algorithm and the fact that gcd(a,b,c) = (gcd(a,gcd(b,c)), helps us engineer a solution.

The *GCD* test is a good starting point for learning about data dependence analysis. However, the test it is not very powerful. The integer equation $a * i_1 - b * i_2 = c$ has a solution if (and only if) the greatest common divider (gcd) of a and b evenly divides c. If the coefficients of the loop variables are 1 (remember, that is the common case), then the GCD is also 1; it will divide any (integer) number. The key to the solution in that case lies in the inequalities. The solution of the equation is only valid for a pair of i_1, i_2 that both lie within the loop bounds. This check is not always straightforward, motivating more advanced data dependence analysis technology.

Slide 44: Other Data Dependence Tests

There is a large variety of data dependence tests. This topic is very well covered in text books [9]. In this course, we will not go in as much depth. We will briefly consider the

- Banerjee-Wolfe test: it is often simply referred to as the Banerjee test. It is one of the most widely used state-of-the-art test. We will discuss the concepts of this test, leaving out the detailed mathematical solution. We will also look at the dependence test driver algorithms, which are important and non-trivial, but not covered well in the literature.

- The Power test: This test is more precise than the Banerjee test. It uses FourierMotzkin elimination (FME) for part of the test. FME is a precise mathematical algorithm for eliminating variables from a system of linear inequalities.

- Omega test: A precise integer programming test, using the exact mathematical formulation and solution of the data dependence problem for affine subscripts. The Omega test is engineered so that it is fast for the common forms of subscripts.

- Range Test: This test is able to deal with non-linear and symbolic subscripts.

Slide 45: The Banerjee(-Wolfe) Test

This is one of the most common tests implemented in parallelizers. It is an inexact test, sometimes called a real-valued test, as is does not look specifically for integer solutions. The test looks at the extreme values that the loop subscripts can assume, given the loop bounds. If the entire ranges between the extreme values of two accesses do not overlap, there is no possible dependence. Sometimes, this test is referred to as the extreme-value test.

For the simple example on the slide, the concept of the test can be seen without mathematical notation. The extreme values of a(j) are 1 and 100. The extreme values of a(j+200) are 201 and 200. The two ranges do not overlap; hence, there is no data dependence.

Slide 46: Mathematical Formulation of the Test Banerjees Inequalities

Let us look at the same situation a bit more mathematically. The dependence equation for the two accesses is $j_1 - j_2 = 200$. Consider the extreme values of the term $j_1 - j_2$. Substituting the loop's lower bound for the positive coefficient, $+1 * j_1$, and the lower bound for the negative coefficient, $-1 * j_2$, we obtain the minimum, -99. Correspondingly, the maximum is 99. It turns out that the entire range is outside of the right-hand side value of 200; this means, there is no dependence.

The slide also shows the more general form for a doubly nested loop and a single loop subscript. The extreme values are obtained for each pair of variables that represent a loop subscript. Note that the coefficients a, b and c are integer constants. The expressions on the slide are the ones to use for positive coefficients. For a loop variable with a negative coefficient, we would substitute the lower loop bound to obtain the maximum value, etc.

In the presence of multiple dimensions, the test is applied to each subscript independently. This works well for subscripts that are not *coupled*, i.e., each loop variable appears only in one of the subscripts. Recall that this is the common case. For coupled subscripts, the test usually fails; it assumes dependence even if there is none. A possible workaround is to *linearize* the subscript. The linearized form of an array access $a[f(i,j), g(i,j)]$, given a declaration $a[dim1, dim2]$ is $a[1, f(i,j) * dim2 + g(i,j)]$, assuming row-major order. Linearization is not a perfect solution, as it loses the information that the subscript stays within bounds, which can be important for the compiler to know; the term $f(i,j) * dim2 + g(i,j)$ exceeds the dimension bound $dim2$.

The extreme value comparisons are also referred to as Banerjee's inequalities.

Slide 47: Banerjee(-Wolfe) Test (continued)

Banerjee's inequalities, as presented so far, are not powerful enough. Consider the example on the slide. Let us test for the potential loop-carried flow-dependence between the write and the read access. One can easily see that the read access never consumes a value produced by the write access; hence there is no flow dependence. However, the ranges of the write and read access overlap and, thus, Banerjee's inequalities would indicate dependence.

The inequalities have not considered that a loop-carried flow dependence only exists if the read access happens in later iterations, *after* the write access. That is, given iteration j_1 of the write access and iteration j_2 of the read access, the condition $j_2 > j_1$ must hold. This condition expresses the dependence direction. Using Banerjee's inequalities with dependence direction information is what is usually called the Banerjee-Wolfe test. The next slide presents the basic idea.

Slide 48: Using Dependence Direction Information in the Banerjee(-Wolfe) Test

Let us again look at extreme values of the accesses, but now we consider j_1, representing any iteration at which the write access can occur. Given the condition $j_2 > j_1$, the minimum value of the read access' subscript is $j_1 + 6$; the maximum is 105. Now the test can determine that the read range $[j_1 + 6 : 105]$ is outside of the write access' subscript value of j_1 (or, expressed as a range: $[j_1 : j_1]$). There in no flow dependence with direction "$>$".

Of course, there *is* a dependence in this loop. It is an anti-dependence. The values of the array a will be overwritten, after they have been read. This means that the loop cannot be parallelized, as is. However, knowing that flow-dependences are hard to eliminate while anti-dependences are not, may help us find a parallel form of the loop. For example, if we rename a in the write access, the loop becomes parallel. Of course, subsequent accesses to a will need to be renamed properly as well.

We could also do a similar test for the "=" direction. There would be an additional equation, $j_1 = j_2$, instead of the inequality $j_2 > j_1$. The resulting dependence equation would be $j_1 = j_1 + 5$, which has no solution. Recall that "=" and "<" are the only possible directions for this loop; the same holds for all outermost loops. Both tests show independence, while the test without dependence direction information (usually denoted as a "*" direction) could not prove independence. This holds in general; testing for all possible directions separately is more precise than testing with "*".

Slide 49: Dependence Testing with Direction Vectors

We can also test multiply nested loops with this idea. Testing for a particular direction of a loop adds an equation ($i_1 = i_2$) or inequality ($i_1 < i_2$ or $i_1 > i_2$) to the dependence problem, at that loop level. All combinations of such direction vectors are possible. For example, independence in a triply-nested loop may be proven if we test for "=" at the outermost loop, "<" at the second loop, and ">" at the inner-most loop.

Remember that testing for all directions is more precise than testing for "*". To keep the complexity low, a good engineering solution is to test first with "*" direction at all loop levels. If this doesn't prove independence, we test for "=" and "<" at the outermost loop, with "*" at all inner loops. If both tests return dependence, we continue with $(=,=,*...), (=,<,*...), (<,=,*,...), (<,<,*,...), (<,>,*,...)$, and so on. This forms a tree of possible direction vectors.

Slide 50: Data Dependence Test Drivers

So far, we have considered testing for dependence of two given array accesses. Before the compiler can invoke the test on two array subscripts, it needs to do such steps as testing for eligible loops, gathering the necessary information about subscript expressions and enclosing loops, determining array aliases, collecting all array reference pairs that need to be tested, partitioning subscripts of multi-dimensional arrays, and traversing the direction-vector tree. The compiler may also select from among multiple available tests. If a dependence is found, the compiler needs to add the dependence type and direction (or distance) to a data dependence graph.

The highest-level data dependence driver in the Cetus compiler creates a data dependence graph for the entire program. It does so by finding all eligible loop nests, traversing them one by one to obtain their dependence graph, and adding the loop dependence graph to the overall program dependence graph.

Eligible loops are those whose for-statements have well-defined initialization, bound, and increment expressions; they do not contain function calls; and they do not contain goto statements nor premature exits. The compiler may apply loop normalization transformations and inline expansion to bring non-eligible loops into the desired form.

To obtain the dependence graph for a loop nest, the algorithm

- collects bound and stride information for each of the nested loops,
- collects all array access expressions in the loop nest, and then
- calls the next level of the dependence test driver, passing on the collected information, to create to actual loop dependence graph.

Slide 51: Data Dependence Test Drivers (continued)

The function *runDependenceTest(loop_info, accesses)* creates the dependence graph for the loop nest described by *loop_info*, containing the array references *accesses*. For every write reference in *accesses* it finds the set of all other, potentially conflicting references. To obtain this set, the function collects all references to the same array in *accesses* plus all references to arrays that are aliased to this array. Cetus' alias information has been generated through points-to analysis.

Next, for each access pair, the function

- determines the subset of the loop nest that encloses both accesses,
- calls the next level of the dependence test driver, passing the two accesses and the inclosing loop nest information, to obtain a set of direction vectors for which the two accesses have a dependence, and
- adds a dependence arc to the loop dependence graph for every direction vector in the returned set.

Slide 52: Data Dependence Test Drivers (continued)

The function *testAccessPair* takes as input two array accesses and information about the loop nests enclosing both accesses. It tests for data dependence between the two accesses and returns a set of direction vectors, representing these dependences. Note that there can be more than one direction vector, for which a dependence exists. Hence, the function return parameter is a set.

The array accesses may contain multiple dimensions. The dimensions must be partitioned into dimension pairs. For each partition, the function then applies the actual dependence test. If a partition's subscript expressions do not contain any loop variables (ZIV=zero index variables), then a simple test is applied. For subscripts with one or more index variables, the algorithm traverses the direction vector tree and calls the test, as discussed on slide 49.

Slide 53: Non-linear and Symbolic Dependence Testing

The tests we considered so far work well with affine subscripts. Remember, these are subscript expressions that are linear combinations of the index variables of their enclosing loops. There are several forms of non-linearity that can appear in array subscript expressions:

- non-constant coefficients of the loop variables have to be conservatively assumed as non-linear, if the compiler does not know they represent constant values. For example, in a term $a * i$, where i is a loop variable and a is an integer-type program variable, the value of a could have been assigned any expression in a previous program statement, including i (which would make the term i^2) and the result of a subroutine (which could make the term arbitrarily complex). There are techniques that can substitute program variables by constants, where possible; examples of such techniques are the classical constant propagation technique, forward substitution (discussed earlier) and symbolic range analysis (which we will discuss later). It is important that such techniques be applied in the compiler prior to data dependence testing.

- true non-linear expressions of the loop variables cannot be handled by most dependence tests. Array subscripts the are polynomials of degree higher than one are rare, but possible. Of the three Perfect Benchmarks on Slide 35 that contain non-affine subscripts, TRFD contains true non-linear subscript terms in important loops. They had been introduced by the substitution of generalized induction variables (which will be presented later).

- subscript expressions may also contain other terms that are unknown to the data dependence test algorithm. For example $a[i + sqrt(m)]$ has a term that most test could not handle. However, because the data dependence equations contain subtractions of two subscript functions, there is a chance that this term may cancel out. Therefore, under certain conditions (all variables in the unknown term must be loop-invariant and all function calls by be pure functional) the unknown terms may be brought into the data dependence equations. The test only needs to give up if the unknown term remains in the final form of the data dependence system.

The range test deals with all three types of non-linear expressions. The test is based on the Symbolic Range Propagation technique [10], which analyzes symbolic lower and upper bounds for every variable at every program statement and maintains a *range dictionary* that can be queried by compiler passes. The range test makes use of the range dictionary's comparison function when comparing expressions of the dependence equations; the function attempts to substitute terms of the expressions that need be compared as much as possible, so that they become comparable.

The range test can deal with subscripts that contain non-linear polynomials if they increase or decrease monotonically w.r.t., the loop variables. To prove monotonicity, the algorithm checks if the symbolic difference between two consecutive values is either always positive or always negative. Monotonicity implies that the extreme values of a subscript function are at the boundary values of the contained loop variables.

The third type of non-linearity, unknown terms, is automatically handled by the range test, as it expresses all dependence equations symbolically.

Slide 54: The Range Test

Building on the compiler's capabilities for symbolic representation, analysis, comparison, and monotonicity testing, the basic idea of the range test is quite simple: capture the range of array accesses, $r(j)$, made in any loop iteration, j, and test if $r(j)$ overlaps with $r(j+1)$.

The slide shows an example: we want to see if there is a loop-carried self output dependence in the outer loop with index i. The range of accesses in given loop iteration, i_x, is $[i_x * m + 1 : (I_x + 1) * m]$. The algorithm finds this range by substituting the lower and upper bound of the j-loop into the subscript expression, obtaining the minimum and maximum value, respectively, for the access range. Note, all these expression manipulations happen symbolically. The range accessed by the next iteration, $i_x + 1$, is $[(i_x + 1) * m + 1 : (i_x + 2) * m]$. Subtracting the upperbound of the first range from the lower bound of the second results in

$(i_x + 1) * m + 1 - (i_x + 1) * m$, which simplifies to 1. The two ranges do not overlap, and this is true for any iteration, i_x. Hence, the loop has no cross-iteration dependences.

Slide 55: Range Test (continued)

The range test builds on advanced symbolic manipulation capabilities. Let us look at these requirements in a bit more detail.

Given a n-nested loop with loop variables i_x, lower bounds L_x, and upper bounds U_x, $1 < x < n$. Let us find the range of accesses made by an array with subscript expression $f(i_1, ..., i_n)$ in loop l_k of the nest, $1 < k < n$. Assume that the subscript expression f is monotonic w.r.t. all loop variables. We can obtain the access range [min:max] as follows:

```
$min = f(i_1, ..., i_n)$
$max = f(i_1, ..., i_n)$
for x=n to k-1 (in steps of -1)
    if $f$ is monotonically increasing w.r.t. $i_x$
        substitute all occurrences of $i_x$ in max with $U_x$
        substitute all occurrences of $i_x$ in min with $L_x$
    else
        substitute all occurrences of $i_x$ in max with $L_x$
        substitute all occurrences of $i_x$ in min with $U_x$
```

To test monotonicity w.r.t. a loop variable i_k, we check if the difference $d = f(i_1, ..., i_k + 1, ..., i_n) - f(i_1, ..., i_k, ..., i_n)$ is alway (for all $i_k, L_k < i_k < U_k$) greater or always less than zero. Usually, all terms but those containing i_k will cancel out in this difference expression, making the comparison with zero easy.

If the subscript expression contains an unknown term, the comparison function of the range dictionary will try to substitute the term with equivalent terms or ranges, until a comparison becomes possible or no more substitution are found.

Slide 56: Handling Non-contiguous Ranges

The range test works well if every loop in a nest has a stride that is as large or larger than the total range accessed by the inner loops. The test fails, if an inner loop has a larger stride than an outer loop, as in the example on the slide with $n = 1$ and $m \geq u1$. By interchanging these two loops, the smaller-stride loop can be moved to the inside, giving the test a chance to succeed.

There are two issues to be resolved. First, the interchange is undesirable in the final code; it should only happen for the analysis. The range test does so by only logically permuting the loops. Second, interchange is subject to legality constraints, which are derived from dependence information. Since dependence information is not yet available, the range test uses interchange only in "obvious" cases: If all loops in a subset of a nest turn out to be free of loop-carried dependences, all possible permutations of this subset (including the original one) are legal.

Slide 57: Some Engineering Tasks and Questions for Dependence Test Pass Writers

The slide summarizes many of the engineering tasks and questions that need to be dealt with when creating a data dependence test. Students writing their first dependence analysis pass may want to deal with the simple case first: Create a test that finds independence of a single loop with a one-dimensional array access. The loop has canonical form, where both the lower bound and the stride are one and the upper bound is an integer constant. The slide lists a possible sequence of features that can be added to the test, to make it increasingly powerful.

The interaction of dependence analysis with other compiler passes is an important engineering question and involves the following issues:

- Handling loop-variant terms: The only loop-variant terms that are allowed to appear in subscripts during dependence testing are the loop variables of enclosing loops. The data dependence test relies on prior passes to either have substituted other loop-variant terms by expressions involving loop variables or to provide a knowledge base (e.g., the range dictionary of the Cetus compiler) of expressions that are equivalent to these terms.

- The interaction with privatization, reduction parallelization, and induction variable substitution is important. In Cetus, privatization happens before dependence analysis; private scalars do not need to be tested for dependences (for private arrays, see the discussion of array privatization.) There are other possible methods. Dependence analysis could go first and privatization could base its analysis on the presence of anti dependences.

  Similarly, in Cetus, reduction variables are flagged as such before dependence analysis, and their dependences are not tested. Alternatively, reductions could be identified from dependence information; the presence of a self, anti, and output dependence suggests a possible reduction statement.

  Induction variables are loop-variant terms and thus need to be eliminated or properly flagged during dependence analysis. Recall that induction variable substitution adds overhead. Loops that end up not being parallel are better run without induction variables substituted. Hence, it is preferred to mark them as such for dependence testing and substitute them in a later pass.

- Deciding on adequate data structures for representing data dependence information is important. The Cetus compiler keeps a data dependence graph for every loop nest in the program. The nodes of the graph are array accesses that are involved in data dependences. Edges represent the dependences between these accesses; they are labeled with the dependence direction vector and dependence type. The needs of other passes decide the appropriate form of the dependence graph. Most parallelization passes only need a statement-level graph. Some passes, such as the insertion of doacross synchronization, may require dependence distances to be represented, not just directions. Some passes may need inter-loop dependence information, which the Cetus compiler does not provide.

4.2 Parallelism Enabling Techniques

Slide 60: Advanced Privatization

Privatization is the technique with the highest impact in the Cetus compiler as well as in the early Kap compiler. The technique in Cetus is an advanced version of scalar privatization. It is able to privatize entire arrays or array sections. The same observation that we made for scalar temporaries, on Slide 20, holds for arrays: An array may be used as temporary storage during one loop iteration. The temporary use of different iterations would create conflicts when executing in parallel. By giving each iteration (or each processor/thread) a private version of the array, the conflicts are avoided.

The example on the right shows such a situation. The variable t now is an array. For brevity, the example uses the array notation 1:m, meaning “the entire subscript range from 1 to m”. In the OpenMP parallel directive language, arrays can be listed in a private clause the same way as scalars.

The strict temporary use of a private variable is important. A variable cannot be privatized if there is a potential use of any of its values in any future iteration. The compiler must be able to prove that all values of the temporary consumed in an iteration have been defined (written) in the same iteration, prior to the use. For scalars, this proof is relatively easy. If the first access to the variable in the loop iteration is a write reference, then the variable is privatizable. A small complication is the presence of IF statements. If the write access happens conditionally, then the variable is only privatizable, if the read accesses happens under the same condition. The general rule for correct privatization of a variable is that any read reference must be preceded by a write reference in the same loop iteration. A different way of expressing this rule is that no read reference must be *upward exposed* to the beginning of the loop iteration. The same rule applies to array elements. If it holds for all elements, the entire array is privatizable. This is the case in the example on the slide. Partial array privatization is possible, but is more difficult to implement. OpenMP supports only the privatization of entire arrays.

The demand on the compiler for privatizing arrays is much higher than for scalars. For scalar privatization, name-only analysis suffices. Symbolic subscript analysis is needed for capturing the sections of an array that can be privatized. The next slide shows an example where this is the case.

Slide 61: Array Privatization

The two examples show a privatizable array t that is written and read in multiple statements of the loop, some of which are executed conditionally. A combination of analysis techniques is needed to prove that t is privatizable.

The key technique is array definition-use analysis. It captures array sections that are written and read at each statement of a program. The technique relies on symbolic manipulation capabilities for comparing, intersecting, and combining array sections. The choice of representation for array sections is important. *Regular sections* are a common representation. They describe an array section as a pair of lower/upper bounds. The Cetus compiler represents an array section as a set of regular sections with symbolic bounds.

Conditional execution can be dealt with in various ways. Ideally, the array section representation allows for conditions to be expressed. This allows the privatization algorithm to capture definitions and used of arrays together with their execution condition; based on this information, the algorithm can determine precisely which uses cover which definitions in a loop. In the absence of a representation for conditions, the privatization algorithm may compare definitions and uses that execute within the same IF statement.

Array definition-use analysis may be imprecise. The representation will not always allow the exact capture of definitions, uses, and conditions. Section manipulations may add further imprecision, as insufficient information may be available about the relationship of two sections or the operation is too complex. Where imprecision arises, the compiler needs to make conservative estimates. Uses must be overestimated, while definitions must be underestimated; this is essential for correctness.

Slide 62: Array Privatization Algorithm

This slide shows a high-level algorithm for array privatization. The algorithm iterates from the innermost to the outermost loop in a nest. At each loop, it traverses all statements of the loop body (considering an inner loop one compound statement) and gathers their array definitions and uses. Definitions are added to the known definitions for the current loop. At each use, the algorithm checks if there is a covering definition. If so, the used array section is added to the list of privatizable variables for the current loop; otherwise the array section is added to the upward-exposed uses. At the end of the loop, the definitions and upward-exposed uses are aggregated; that is, subscript expressions that contain the current loop variable are replaced by a range. To create the lower and upper bounds of the range, the aggregation algorithm replaces the loop variable in the subscript expression with the loop lower and upper bound, respectively. The aggregated definitions and upward-exposed uses represent the loop's definitions and uses for the analysis of the next outer loop.

Besides the aggregation step, the non-trivial parts of the algorithm include the addition of a definition and the check for covering definitions. When adding a definition, the algorithm combines adjacent array ranges. The availability of symbolic analysis and manipulation capabilities are important. If adjacent subranges cannot be recognized as such, the represented definitions may get fragmented and imprecise. Since correctness dictates that the represented definitions must never include an array element that is not defined (a.k.a. *must* definitions), the algorithm may need to be conservative and delete a known definition. The choice of array section representation is important for this step. For example, if only a single regular section (i.e., lower/upper bound pair) is kept for representing a defined array section, then two non-adjacent sections cannot be privatized. Similarly, if the representation does not include conditionals, then the algorithm may need to ignore definitions inside if statements. The test of covering definitions checks if a defined array section exists that is equal or larger than the use seen at the current statement. Symbolic reasoning capabilities are important here as well. In addition, conditional array accesses need to be compared. If both the use and the covering definition are under a condition, then the use condition must imply the define condition (i.e., whenever the use condition is true, the define condition must be true.)

A step not shown in the high-level algorithm is the test for last-value assignment. If an array is used (read) after the loop in which it is privatized, the compiler must make sure that the last assigned value is transfered to the original variable. To detect the need for last-value assignment, the compiler performs liveness analysis. Last-value assignment does not happen automatically; the storage used by the private variable is different from that of the original variable and becomes undefined at the end of the loop. The OpenMP parallel directive language offers a special keyword *lastprivate* to indicate private variables that need last-value assignment. If the parallelizer's output language is OpenMP, it may simply us this keyword instead of *private*. There are several methods to make explicit last-value assignments. An explicit assignment statement may be added after the loop. The statement must be guarded by a condition that the loop has at least one iteration, unless the compiler can prove this condition to hold true. Alternatively, the compiler may use the original variable in the last loop iteration, as in the following example.

```
                              PARALLEL DO i=1,n
                              PRIVATE tp
                                IF (i=n) THEN
DO i=1,n                          t = a[i]+b[i]
  t = a[i]+b[i]                   c[i] = t+sqrt(t)
  c[i] = t+sqrt(t)    =>        ELSE
ENDDO                             tp = a[i]+b[i]
print t                           c[i] = tp+sqrt(tp)
                                ENDIF
                              ENDDO
                              print t
```

In the following loop, it is not possible for the compiler to determine the iteration at which the last assignment is made. If the compiler cannot generate the proper last-value assignment, the variable cannot be privatized.

```
DO i=1,n
  IF (c[i]=0) THEN
    t = a[i]+b[i]
    c[i] = t+sqrt(t)
  ENDIF
ENDDO
print t
```

The interaction of privatization with dependence analysis needs careful consideration. In the Cetus compiler, privatization is done first. Private variables do not create data dependences and hence dependence analysis can ignore them. An important exception is for an array that is both privatizable and independent, as in the following example.

```
DO i=1.n
   a(i) = ...
    ... = a(i)
ENDDO
```

The element $a(i)$ would be recognized as private by our definition (write before read). However, the accesses to a are also dependence free. Privatization would not make sense in this case. The Cetus compiler handles the case by analyzing private arrays for dependences as well; if no dependences are found, the array is removed from the private list.

Slide 63: Some Engineering Tasks and Questions for Privatization Pass Writers

This slide suggest a path for the student writing a privatization pass. It begins with a simple scalar privatizer and adds successively more advanced capabilities.

Slide 65: Reduction Parallelization

Earlier, we have discussed the parallelization of scalar reductions. These are reductions of the form shown on this slide. The most common form is a scalar variable summing up the elements of an array. By exploiting the mathematical properties of associativity and commutativity, one can rewrite this summation algorithm into a form that collects partial sums of subranges of the array and then adds them up.

If OpenMP is available as a target language, all the compiler needs to do is to recognize such reduction variables and list them in a *reduction* clause on the OpenMP parallel loop directive. An actual parallel transformation would use a private variable for storing the partial sums and add them to the original variable using an OpenMP *atomic* directive. We refer to this form of parallel reduction as *privatized reduction implementations.* Another implementation makes use of *expansion*: Instead of a scalar, an array, indexed by the current processor/thread, holds the partial sums. These partial results are then added to the original reduction variable in the form of a loop that is a reduction itself, but with fewer iterations than the original reduction. One requirement for reductions to perform well is that the reduction length is much greater than the size of the reduction array, $n >> num\_proc$.

Slide 66: Parallelizing Array Reductions

Instead of the scalar reduction variable, there can also be an array expression. If the subscript expression is loop variant, we call this an *array reduction.* Other names are irregular reductions or histogram reductions.

The parallel implementation is similar to that of a scalar reduction. The partial sum variable now is an array of the same size as the reduction variable (the compiler may narrow this to the range of the subscript expression). There are differences in the initialization and the final addition of the partial sums. The initialization simply clears all array elements. In the privatized reduction implementation, the final addition combines the partial results using an *atomic* directive; in the expanded reduction implementation, it makes use of a fully parallel loop.

Array reductions have limited support in OpenMP. OpenMP 3.0 for Fortran supports them.

The following is an alternative implementation of parallel reductions, which we call *synchronized* reduction implementation. It makes use of an OpenMP atomic directive to protect the updates to the reduction variable. The efficiency of this implementation depends on the cost of the atomic operation. Note that this synchronization is executed in every iteration, whereas the above transformation essentially creates a fully parallel loop. One advantage of the synchronized reduction implementation is that no code change is needed, other than inserting the atomic directives.

```
#pragma omp parallel for
for (i=1; i<n; i++){
    ...
#pragma omp atomic
    s = s + a[i]
    ...
#pragma omp atomic
    sa[tab[i]] = sa[tab[i]] * expr;
    ...
 ENDDO
```

All variants of parallel reduction implementations make use of the mathematical commutativity of the involved operations. It is important to keep in mind that computer floating-point arithmetic is *not* strictly commutative, due to the limited precision. The transformed code will produce answers that are different from the original one and will vary from run to run. In almost all cases, this is not a problem, as the imprecision occurs within the least-significant digits. Some computation may be vulnerable to such imprecision, however.

One example is a numerical solver that operates near a data singularity. Compiler writers usually push the solution to this problem onto the user – by providing a command line flag for enabling or disabling parallel reduction transformations.

Slide 67: Recognizing Reductions

There are two criteria for recognizing reductions that can be translated into the shown parallel form.

1. The loop may contain one or more statements of the form in statements s1 through s3 in the code example below. The reduction variable can be a scalar (such as s) or an array element (such as $sa[c[i]]$). The reduction variable is modified by a commutative and associative operator, such as $+$, $*$, min, and max. Different reduction statements for the same variable must use the same operator.

2. The reduction variables must not appear in any non-reduction statement within the enclosing loop, nor in the modifying expressions (a[i], expr1, expr2).

```
      DO i=1,n
         ...
s1:      s = s + a[i]
         ...
s2:      sa[c[i]] = sa[c[i]] * expr1
         ...
s3:      sa[d[i]] = sa[d[i]] * expr2
      ENDDO
```

Slide 68: Reduction Recognition Algorithm

The slide shows an algorithm for recognizing and annotating additive reductions, by the described two criteria. The algorithm traverses all statements of the given loop, L. At assignment statements, it checks criterion 1 by first subtracting the left-hand-side (LHS) from the right-hand side (RHS) expression; if the remainder does not contain the LHS, the statement is a reduction statement and satisfies criterion 1. The algorithm adds the LHS to the set of reduction expressions (scalar or array references). It also records all variables referenced in the remainder of the RHS, for later checking criterion 2. For non-assignment statements, all referenced variables are recorded for checking criterion 2.

To test criterion 2, the algorithm traverses all potential reduction variables and verifies that they do not appear in any of the recorded references. If this criterion is satisfied, an annotation for a sum reduction is generated in the IR. The algorithm creates different annotations for scalar and array reductions. Array reductions are recognized by a reduction expression consisting of an array reference with a loop-variant subscript.

Slide 69: Reduction Compiler Passes

There are two passes in the Cetus compiler that deal with parallel reductions, the reduction recognition and the reduction transformation pass. The recognition algorithm is placed between induction variable recognition and privatization. The generated annotations will be looked at by the parallelization pass, which decides on the loops that can be correctly executed in parallel (this decision also involves private annotations and data dependence information, but not yet profitability considerations).

The reduction transformation pass turns parallel loops containing reduction variables into the final, parallelized form. It follows the profitability test, which decides on serial execution of those parallel loops where overheads seem to outweigh the benefits of parallel execution. The reduction transformation pass uses OpenMP reduction clauses to express scalar reductions and transforms array reductions explicitly, as shown on Slide 66.

Slide 70: Performance Considerations for Reduction Parallelization

Parallel loops that contain reductions execute substantially more code than their original, serial versions. In general, the overhead is excessive if the reduction size (n) is small. For large reductions, the added code for initialization and final sum of the partial results tend to be insignificant.

False sharing is an important performance consideration for reductions. It may occur in expanded reductions, if multiple processors use adjacent array elements of the temporary reduction array. Expanded reductions exhibit more parallelism in the sum-up operation. Hence there may be a tradeoff. False sharing can be avoided by placing the additional dimension of the partial reduction array (the processor/thread dimension) away from the stride-1 position (add right-most in Fortran, left-most position in C).

If the reduction operation touches only few elements of the reduction array (the number of distinct elements of the subscript expression is much less than its range), we call this a *sparse reduction*. The overhead of sparse reductions can be high. Alternative reduction parallelization methods may need to be developed that compress the elements that are not touched. The use of a synchronized reduction implementation is another option.

Slide 71: Induction Variable Substitution

Recall the simple form of the induction variable substitution technique, which was available in the Kap compiler. Like reduction statements, induction statements have a loop-carried flow dependence; it takes a new algorithm to get rid of this dependence. This new algorithm is a closed-form computation of the value, replacing the induction-based computation. The closed-form computation gets substituted at each use of the induction variable. If the induction variable is used multiple times within the same loop iteration, a slightly better implementation is to perform the closed-form computation once, assign it to a private variable, and use this variable for the substitution. After the substitution, the induction statement can be eliminated, which removes the flow dependence.

The simple form of an induction statement uses a constant (or loop-invariant) increment. The example uses the constant 2, which leads to the closed-form expression $k + 2 * i$. The next slide shows more complex forms of induction variables, called *generalized induction variables, GIVs.*

Slide 72: Generalized Induction Variables

Generalized induction variables, or GIVs, lead to closed-form computations that are non-linear expressions of the enclosing loop variables. The non-linear expression is due to a non-constant increment of the induction statement. The slide shows several examples. In the first example, the increment is the loop variable j. The corresponding closed-form is $k + (j^2 + j)/2$. Note that loop variables are induction variables themselves. The second example shows a case of *coupled induction variables.* The first one, $ind1$, has a simple form and is used as the increment of the second induction variable, $ind2$. This leads to a similar closed-form expression as in the first example. The third GIV example contains a triangular loop nest. Even though the induction variable has a constant increment and is of the simple form in the inner loop, the increment added in each iteration of the outer loop is non-constant.

All these examples lead to non-linear closed-form expressions. Since most data dependence tests handle linear expressions only, the loops with the substituted induction variables could not be parallelized. The range test is needed for successful parallelization. The following workaround for this problem is possible when subscripts contain the induction variable only: The fact that induction variables represent monotonic sequences implied that subscrips in different iterations assume distinct values. This method requires the induction variable recognition pass to annotate induction expressions as such, and the dependence pass to consider these annotations. The two passes need to be integrated more tightly.

Slide 73: Recognizing GIVs

There are several methods for recognizing generalized induction variables.

- The compiler can find induction statements in a loop that match the form $iv = iv + expr$ or $iv = iv * expr$. The induction variable iv must be an integer scalar, and $expr$ must be either a loop-invariant expression or another induction variable. If GIVs depend on other GIVs, there must not be a cyclic relationship. There may be multiple induction statements in a loop with the same variable; the increment expressions may differ, but the operator (+ or *) must be the same. This form of recognition is used in the Cetus compiler.

- A more formal way of recognizing induction variables makes use of abstract interpretation [11]. The key idea is this: The algorithm symbolically executes several loop iterations and detects the increment incurred by scalar variables. If there is a constant increment, the variable is a linear induction variable. If the differences between the increments are constant, there is a second-order induction variable, and so on. This method is able to detect induction variables, even if they do not appear in the form of the induction statements we have considered so far. For example, a sequence $k = m + 1; m = k$; would get recognized.

- A Static Single Assignment (SSA) program representation can be used to detect the dependences and cyclic statement relationships that indicate induction variables [12]. From this information, the presence of induction variables can be detected. The proposed method also recognizes occasionally seen variants of induction variables, such as *wrap-around* and *flip-flop* variables.

Slide 74: GIV Algorithm

After induction variables have been recognized, the following algorithm finds the closed form of a given induction variable and performs the substitution [13]. The algorithm is applied for each of multiple induction variables. For coupled induction variables, it is first applied for the one with no dependence on other induction variables.

Consider a multiply-nested loop that contains multiple induction statements of the variable *iv* at several loop levels. As the algorithm traverses the loop nest, it considers three types of statements: Induction statements of the form $iv = iv + expr$ (type I), loop compound statements (type L), and statements using induction variables (type U). There is a closed-form computation and a substitution part of the algorithm.

The main driver performs the closed-form computation by calling $FindIncrement$, passing the outermost loop, L_0, as a parameter. FindIncrement returns the total increment incurred by *iv* in the loop nest. Next, it calls *Replace* to substitute the uses of *iv*. If the value of the induction variable is used after the loop nest, it also inserts a statement that adds the total increment to the original value. (If the compiler cannot prove that the loops in the nest execute at least one iteration, it also needs to enclose this statement with an IF-clause testing this condition – a.k.a. *zero-trip* test.)

Function *FindIncrement* traverses all type I and L statements. It finds the increment added to *iv* from the beginning of the loop iteration, by every induction statement. The values are summed symbolically. To obtain the increment added by L-statements, it recursively calls $FindIncrement$. The current value of the increment, at the end of statement s_i, is stored in $inc\_after[s_i]$. These values will be used by the substitution algorithm. After all statements of the current loop have been processed, the algorithm computes two closed-form expressions. The first one computes the increment added to *iv* from the beginning of the current loop L to the beginning of the j^{th} iteration; this expression is stored in *inc_into_loop*[L]. The second closed form computes the total increment for the current loop; this value is returned to the caller of $FindIncrement$.

The substitution algorithm visits the statements of all three types. The key step is the one generating the symbolic expression representing the value of the induction variable at a given program statement. That expression is the sum of three terms, (i) the value of *iv* at the beginning of the loop (this value is passed into *Replace* as a parameter), (ii) the increment from the beginning of the loop to the beginning of the current loop iteration (this expression is stored in *inc_into_loop*), and (iii) the increment from the beginning of the loop iteration to the current statement (this value is stored in inc_after). This expression is kept in val, which is initialized at the beginning of the algorithm and updated at every statement of type L and I. Before the update at L-statements, the algorithm recurses into the inner loop, passing the current val as the initial value for this loop. The actual substitution happens when visiting statements of type U; all occurrences of iv in the statement are replaced by the expression val.

Slide 76: Loop Skewing

The loop skewing technique was originally introduced for the specific example on the next slide, where it implements a scheme called the *wavefront technique* [14]. Here, we are using the term more generally. Loop skewing changes the axes of an original iteration space away from their perpendicular directions to new directions that are more favorable for parallel execution, given the data dependence directions.

The example on this slide shows a loop nest with dependences in diagonal direction. The outer loop cannot be parallelized. Loop interchange cannot change this fact (the inner loop could be run in parallel, but our goal here is to find outer parallelism.)

The key idea is to pack all iterations that are dependent on each other – a shaded region – into an inner, serial loop. Different regions are independent of each other and can be executed in parallel. The new, outer iteration space enumerates the independent regions; the inner iteration space goes diagonally, from bottom left to top right in the original iteration space. To build the new iteration spaces, min and max functions are often needed. They introduce overhead. For reasonably large iteration spaces, the advantage of outer loop parallelism will likely offset this overhead.

Slide 77: Loop Skewing for the Wavefront Method

In the previous example, the inner loop could be parallelized when serializing the outer. In the example on this slide, this is not possible. The idea of wavefront execution extracts parallelism, nevertheless.

Consider the loop's iteration space. A wavefront execution proceeds in (serial) steps. Each step executes concurrently all those iterations that have either no incoming dependences or depend on iterations that have already executed in previous steps. Step 1 has only one iteration, (i,j)=(2,2); it has no incoming dependences. Step 2 has iterations (3,2) and (2,3). They only depend on the already executed iteration of Step 1. Step 3 includes iterations (4,2),(3,3),(2,4); after Step 2 has completed, all these iterations can execute in parallel. Further steps proceed accordingly. The set of iterations in one step are called a *wavefront.* The wavefronts execute serially; within wavefronts, all iterations execute in parallel.

The loop nest can be skewed, so that the steps represent the new outer loop and the wavefronts the inner. The slide shows the transformed loop.

The pattern to which loop skewing is applicable represents a stencil operation in finite element computations. The transformation has been found applicable in benchmarks such as the LU code of the NPB Suite [15]. It has significantly lesser impact than the transformations discussed to far, however. The same holds for many of the other transformations that have been proposed for parallelization. They may be applicable in rare cases, where they can make a performance difference. However, the difficulty of implementing the techniques is not only in understanding and developing the transformation algorithms and their interactions with other compiler passes. The added difficulty is in detecting the source code sections that can benefit from the transformations. As we have discussed previously, this challenge is formidable. It takes the creation of a performance model that can predict the speed difference between original and transformed code. Such models are highly complex and depend on information that is often unavailable at compile time (e.g., the program's input data). Overly eager parallelization may *degrade* the performance of the "optimized" program.

On the bright side, the presented techniques have proven to be a sufficient set of parallelism detection and enabling capabilities for creating a compiler that can match the performance of the best existing parallelizer. Given this set, we conclude the section on parallelism enabling techniques.

4.3 Techniques for Multiprocessors: Mapping Parallelism to Shared-memory Machines

The transformations described in this section help implement detected parallelism on architectures that have a shared address space. As earlier classifications of compiler techniques, this category is useful but not strict. For example, loop distribution can also play a role in enabling parallelism detection and extracting parallelism in the presence of dependences, as we will see.

Slide 79: Loop Fusion and Distribution

Loop Fusion merges two adjacent loops into one. This is easy when the iteration spaces match, as in the code shown on the slide. In other cases, loop iteration spaces can be aligned to match, by shifting the bounds and/or by loop peeling (extracting the first or last few iterations into a separate loop). Loop fusion removes fork/join overhead. It can also affect data locality, as it forces corresponding iterations of the original two loops to execute on the same processors; if the two iterations reference the same data, the same thread will access the data, which we call *data affinity.*

Loop fusion changes the order of execution. Before fusion, all iterations of the first loop are executed first, followed by the second loop. Fusion causes iteration k of the second loop to execute before iteration $k + 1$ of the first loop, for any k. All possible dependences of the original loops go from the first to the second loop; that is, they are lexically forward. The legality rule is that all dependences in the fused loop must go lexically forward as well.

Loop distribution is the reverse transformation of fusion. It splits a loop into two. The legality rules correspond to those for loop fusion: All dependences must be lexically forward. For loops that do not have dependence cycles (a statement depends, directly or indirectly, on itself) the statements can be reordered to achieve lexically forward dependences. Hence, the real barrier to loop distribution are dependence cycles.

For parallel execution, distribution creates a barrier between the two split loops. This can be used to enforce the synchronization needed on Slide 26, instead of post/wait synchronization. This may be the only choice on architectures that do not support fine-grained iteration synchronization. Later, we will see that loop distribution is important for implementing vectorization as well.

Slide 80: Loop Distribution Enables Other Techniques

Loop distribution can also enable other transformations. In the example on the slide, loop distribution creates a perfectly-nested loop, which can then be interchanged to achieve stride-1 reference behavior. Clearly, the benefit of this effect will have to be greater than the cost of the additional fork/join.

The Kap compiler, discussed earlier, used loop distribution extensively. Given a non-perfectly nested loop, it created as many distributed loops as necessary, so as to create all perfect nests. It then parallelized and interchanged these nests independently, followed by fusing the loops back together where feasible. The Cetus compiler does not apply this aggressive form of loop distribution.

Furthermore, if only one of the two loops ends up being parallel, distribution can have the effect of enabling/extracting partial parallelism – the serial portion is split from the parallel portion of a loop. In general, the serial portion includes those statements that are involved in dependence cycles. The other statements can be reordered, so that all dependences are lexically forward. These statements can then be distributed into separate parallel loops.

Slide 81: Enforcing Data Dependence

A correctly executing parallel program must enforce the data dependences present in the original program. There are two essential rules, which must be observed for all dependence arcs of a data-dependence graph.

1. recall the general form of a direction vector that describes a loop-carried dependence, $(=, ..., <, *, ...)$. The vector begins with zero or more =-directions, followed by an initial $<$ element, followed by zero or more directions or any kind). We refer to this as a *legal* dependence vector. The loops corresponding to the leading $=$ directions can be marked as parallel w.r.t. this dependence; the initial $<$ direction carrying loop must be executed serially; the remaining loops can be marked for parallel execution, also; the initial $<$ direction *covers* the dependence for these loops ($<$ direction means that the dependence exists *only* in a forward direction; serializing the loop runs only

one of its iterations during the parallel execution of the inner loop, where there is no dependence).

This rule needs to be applied to all dependence arcs. We must consider two separate arcs for any dependence that exists under two different directions. At the end, loops that have never been marked as "to be executed serially" become parallelizable loops.

2. Transformations that change dependence vectors (e.g., loop interchange) must never result in an illegal dependence vector. In particular, an interchange that would create a vector, whose initial non-= element is a > direction is not legal.

Recall that data dependences are defined w.r.t. an order of execution. In our context of automatic parallelization, this order is the original sequential execution of the program. If the source program were already parallel, the dependence analysis would be different. For example, a loop that the programmer has marked as parallel does not have a defined ordering among iterations. There is no dependence between iterations that read or write the same storage location. Instead, there is a race condition. Unless the program is erroneous, the programmer has implicitly stated that the race condition is acceptable. Whether or not race conditions are acceptable, depends on the high-level algorithm. An autoparallelizer, without further knowledge, cannot answer this question. It strictly needs to enforce dependences. Hence, the compiler-parallelized version of an originally sequential program cannot have race conditions.

Slide 82: Loop Interchange

Earlier, we have discussed the benefits of loop interchange. Here, we consider the legality of the transformation. Loop interchange alters the access order of iterations (and their data references). The transformation is illegal if the sink of a dependence would end up executing before the source. Dependence vectors are the tools needed to check legality. The rule is quite simple: loop interchange switches the two elements of the direction vector that correspond to the interchanged loops; after the switch, the dependence vector must still be a legal vector, by our above definition.

The compiler needs to check this rule for every dependence within in the two candidate loops to be interchanged. The following three cases show the correctness of the rule for a single dependence.

Case 1: Switching two loops in a nest that do not carry the dependence, is always legal, as the switch does not affect the order of dependence source and sink. The corresponding dependence vector is $(..., =, ..., =, ...)$; it looks the same after interchange.

Case 2: Let the outermost loop that carries the dependence be the *dominant loop.* A key observation is that the dominant loop determines the order of dependence source and sink. Any reordering of inner loops do not change this fact. Hence, any interchange within the dominant loop is legal. This rule preserves legal dependence vectors: The dominant loop has a $<$ direction; inner loops can have any directions.

Case 3: Let us now consider interchanging the dominant loop with the next inner or next outer loop. Three pairs of directions are possible. (i) Changing $(<, =)$ and $(=, <)$ preserves the order of source and sink. (ii) The dependence vector after interchanging $(<, <)$ loops remains the same. The dominant loop will change. However, since both loops carry the dependence in the same direction, the order of dependence source and sink is preserved. (iii) Switching $(<, >)$ to $(>, <)$ is illegal. It would reverse the order or source and sink. It also would create an illegal direction vector, as the dominant loop must have a $<$ direction.

Interchanging the dominant loop with non-adjacent inner loops can be thought of as bringing the inner loop into the next innermost position (Case 2) and then testing for Case 3. Similarly, applying Case 1 and Case 3.i covers interchanging the dominant and any outer loops. Hence the three cases cover all possibilities. In all scenarios, the only illegal case is moving a $>$-direction loop from inside to the outside of the dominant loop.

The above rules for loop interchanging apply to sequential loops. When interchanging parallel loops, a relevant question is how parallelism attributes propagate from the old to the new positions of the loops. We again consider the above cases. Loops with two $=$ directions are parallel; after interchanging this remains true. When interchanging the dominant loop with an adjacent $=$ loop, the dominant loop, remains serial. All loops inside of the dominant loop are parallel and remain that way after interchanging among them. A $(<, <)$ nest, where the first is the dominant loop, is serial in the outer position and parallel in the inner position.

Slide 83: Loop Coalescing

Loop coalescing (a.k.a. loop collapsing) merges two nested loops into one. The transformation increases the number of iterations, which may improve load balance. For example, consider the execution of a loop with 10 outer and 100 inner iterations on an 8-core processor. Parallelizing only the outer loop would leave 2 processors idle. Parallelizing only the inner loop would incur the fork/join overhead 10 times. Coalescing the nest into one loop with 1000 iterations avoids both issues.

The number of loop iterations often depends on a program input parameter. In this case, the compiler cannot make a good decision on which one of the two loops to parallelize. Coalescing them into one obviates the need for this decision.

There is a cost of loop coalescing, however. The values of the original loop variables may need to be recovered from the new loop variable, as in the example on the slide. The cost of the additional code must be less than the gain from the transformation.

Loop coalescing is often applicable in the presence of two fully parallel, nested loops. In this case, the transformation is always legal. Coalescing a serial with a parallel loop does not benefit parallelism, as the resulting loop would be serial.

Slide 84: Loop Blocking/Tiling

We have earlier introduced loop blocking as a synonym to stripmining. The literature also uses the term as “stripmining followed by loop interchange”, which has the ability to enhance locality in loop nests that have temporal reuse. The slide shows such a loop nest. Iteration j' uses the array elements $A(1 : n, j')$ and iteration $j' + 1$ reuses the same data. Given a large enough cache, the second accesses to these elements are cache hits. Between the first and the second access to any array element, there are $3 * n$ other accesses ($2 * n$ reads and n writes). For large n, even an ideal cache will have replaced the line holding the first reference, before its reuse. The idea of loop blocking/tiling is to reduce the number of accesses between use and reuse to an amount that fits in cache. This works as follows.

The transformation is a combination of stripmining and loop interchange. The inner loop is first stripmined into *blocks*, as shown on the slide. The outer loop produced by the stripmining transformation is then interchanged with the original outer loop. Now, the number of references made between use and reuse are $3 * block$ instead of $3 * n$. The value of *block* is chosen so that $3 * block \leq cache\_size$.

Blocking is as important in sequential programs as in parallel programs. On parallel architectures, the outermost loop can execute in parallel. The available cache size per thread/core is important. For distributed caches, the cache size is used directly in the above formula. For shared caches, the effective cache size for each of the p cores is $cache\_size/p$. The most benefit from locality enhancement is at the cache level that has the highest hit/miss performance ratio. This is usually the highest cache level, which is often a shared among the cores.

Slide 85: Loop Blocking/Tiling (continued)

The readability of blocked code is low – there is the additional loop generated by stripmining, and the loop bounds look complex. Using OpenMP syntax, the same transformation can be expressed by adding only directives.

Understanding OpenMP semantics is needed to see how this works. The *omp parallel* directive starts the parallel execution, with all threads executing the same come. All threads will execute the j-loop in its full iteration space, 1 to n. Within this loop, the i-loop is work-shared. Its iteration space is split onto the available p threads. At the end of the inner loop, the threads do not need to wait for other threads to reach the same point; they can continue immediately with the next iteration of the j-loop. The *nowait* clause achieves that effect.

The figure on the slide corresponds to a situation with four threads and a block size equal to n/p. This will not be the general case. the block size will likely need to be different. The *schedule* clause on the *omp do* workshare directive can change the chunk of i-iterations assigned to a thread, which determines the block size.

Slide 86: Choosing the Block Size

Let us generalize the formula for the block size that we developed earlier. Consider two inner loop of the transformed code. Its iteration space is the size of *block*. There is reuse in the enclosing, j-loop. Use and reuse happen at an iteration distance of d. Before and after the statement that makes these accesses there are *r1* and *r2* data references, respectively. The total number of data references between the use and the reuse are $(r1 + r2 + 3) * d * block$ (counting one reference for the left hand side of the reuse statement). Assuming an ideal cache (full associativity and LRU replacement policy), this expression must be no greater than the cache size. Hence, the maximum block size is $cache\_size/(r1 + r2 + 3)$. IF we we want to evaluate the formula in units of bytes, we also need to multiply the denominator by the the size of a data reference – typically the width of a floatingpoint variable.

For caches that are shared by *num_cores* cores, the effective cache size is smaller: $block < cachesize/(r1 + r2 + 3) * d * num\_cores$.

Tiling: The transformation can be applied in two or more dimensions of an iterations space, given a loop nest that exhibits reuse in multiple dimensions. The blocking transformation can be applied in all these dimensions, which is usually referred to as tiling. For an in-depth discussion of tiling in the context of parallelism, see [16].

Slide 87: Multi-level Parallelism from Single Loops

Stripmining was used in the Kap compiler to employ both parallel processors and vector execution within the processors. A single loop can be split into two; the outer loop will be executed on the parallel processors and the inner loop gets vectorized. The same transformation can be applied twice to extract 3-level parallelism. This was done in an early hierarchical shared-memory parallel machine [17], shown in the slide.

The study of individual techniques in the Kap compiler showed that stripmining can help as well as hurt. When transforming for multi-level parallelism, the performance consideration is even more important. Only loops with a sufficiently large iteration space and workload can beneficially be transformed in this way.

4.4 Advanced Program Analysis

It is essential for the compiler to have advanced knowledge about the program in order to make good optimization decisions. Such knowledge includes the structure of the program, execution orders of statements and data references, the values variables can assume, variables that may be allocated in the same or close storage, the relationships between statements that produce and consume values, etc.

Program analysis techniques extract such knowledge from the source program. Analysis techniques do not transform the program, unless they are implemented in combination with transformations. Their results are represented in various forms, so that later compiler passes can consume them. The form of the representations can range from simple statement annotations (e.g., "this is a reduction statement") to complex data structures (e.g., the data dependence graph).

Parallelizing compilers use many of the basic analysis techniques found in classical compilers. Examples are constant propagation, control dependence analysis, definition-use analysis, general data flow analysis, and abstract interpretation. We refer the reader to the literature on fundamental compiler techniques for their introduction [18, 19]; here, we will only briefly mention their role in parallelization.

We have already discussed constant propagation and the slightly more advanced variant of forward substitution, as basic analysis techniques. Classical implementations [19] make use of data flow analysis, which is a fundamental method underlying many compiler analyses. Data flow analysis operates on the control flow graph (CFG) of a program. The CFG is equally fundamental; it is usually created at the very beginning of the compilation process, as part of constructing the IR. The CGF implicitly expresses control dependences – the relationship of a given statement s with the conditional expression(s) that decide whether or not s executes. The techniques discussed so far did not need explicit control dependence information. This is due to the fact that the techniques operate on well-structured program constructs, such as do-loops. The control dependences of the statements in a nest of do-loops are clear; their dependence sources are the termination conditions of the enclosing loops. No explicit control dependence analysis is needed. Similar is true for the analysis of structured if-statements.

Some compiler techniques need to know the statement that defines a given variable value or the statements that use a given value. Within loops, data dependence information expresses these relationships. Across loops, compilers may create *def-use chains* (or the reverse, *use-def chains*) to express the same information. There is also a form of the program representation that expresses definition-use relationships: The *Static Single Assignment* (SSA) form contains only one assignment to each variable. A program's SSA form is generated by essentially renaming repeated definitions. To express the value of a variable after a control flow merge point, SSA introduces a new construct, called ϕ-function. The representation has several advantages for compiler passes; one drawback is reduced readability of the produced code, which is significant for source-to-source translators.

The following slides present some of the more advanced analysis techniques. We start by discussing issues of inter-procedural analysis, followed by alias and shape analysis. We conclude the section on analysis techniques by taking a look at advanced symbolic analyses, including range analysis and array section analysis.

Slide 90: Interprocedural Analysis

All techniques discussed so far operate intra-procedurally – within a subroutine. For example, the data dependence test driver on slide 50 checked for eligible loops; one condition for eligibility was that loops do not contain subroutine calls. Obviously, subroutines are an important program structuring concept for software engineers. Compiler techniques that work only intra-procedurally will make suboptimal decisions.

Ideally, both analysis and transformation techniques can work inter-procedurally. In practice, performing global program analysis followed by local transformations is fairly effective. Furthermore, of the analysis techniques, those that gather and propagate knowledge about variable values are most important to perform inter-procedurally. For example, the Polaris compiler, which matched the manually obtained performance for many of the programs shown on Slide 34, performed only inter-procedural expression propagation; all other analyses and transformations were applied locally.

In the following, we present two inter-procedural analysis techniques. They illustrate a small subset of global analyses. A big challenge for compiler engineers is that the inter-procedural implementation of most techniques are rather different. There does not exist a successful framework, into which an intra-procedural technique could be inserted to make it inter-procedural. Adding to this challenge is that subroutines typically have several callers and several invocations. The parameters may be different for every instance of the subroutine, producing potentially different analysis results.

Subroutine inline expansion has the potential to resolve many of the above problems. Because the resulting code appears as one procedure, all techniques can operate in their intra-procedural form. Furthermore, because subroutines get inlined at each of their call sites, multiple instances of the code will be generated and can be optimized separately. The drawbacks of subroutine inlining are the code increase (which can be significant; up to $10\times$ has been observed in some benchmarks) and its consequences. Increased compilation time and increased instruction cache misses are among the consequences. In some cases, a reduction in the detected degree of parallelism has also been observed. Inlining produced more complex array subscripts than in the original program. As a result, the compiler could no longer parallelize certain loops, even though it could do so in the original program.

Slide 91: Interprocedural Constant Propagation

The slide shows a motivational example of inter-procedural constant propagation. The loop in subroutine B can be parallelized if the compiler knows that $m > 100$. This relationship is made true by subroutine A. Propagating $j = 150$ from A to B (where it becomes $m = 150$) will supply the necessary knowledge to the dependence test when analyzing the loop.

Slide 92: An Algorithm for Interprocedural Constant Propagation

The inter-procedural algorithm for constant propagation includes an intra-procedural and an inter-procedural part. The intra-procedural part computes *jumpfunctions* for each parameter of each subroutine called in the program. $J_{sub,p}$ indicates the jumpfunction for the p^{th} parameter of subroutine *sub*. The representation of the jumpfunctions is important. The example on the slide assumes a hypothetical compiler with a chosen representation of $P + const$, where P is a formal parameter of the caller and *const* is a numerical constant.

The algorithm computes the jumpfunctions through forward substitution from the beginning of the caller. Subroutine Y has one parameter; its jumpfunction is the caller parameter c. Subroutine Z has five parameters. The first and second parameter are expressions of the caller parameters a and b, respectively. The third parameter is unknown because subroutine Y may have modified it in any way (having advanced *maymod* information of the subroutines will help the compiler be less conservative). An unknown value is represented by the symbol $\perp$, pronounced *bottom*. The fourth parameter is a constant value 10. The fifth parameter is unknown, because the $b * 2$ cannot be expressed by the representation $P + const$ that our hypothetical compiler has chosen.

Slide 93: Constant Propagation Algorithm: Interprocedural part

The inter-procedural part of the algorithm includes three initialization steps. Step four is the iterative, main section of the algorithm. Each formal parameter has as many jumpfunctions as there are call sites to its subroutine. After step two, a formal parameter has a $\perp$ value, if either one of its jump functions is $\perp$ or if its jump functions produce different values.

The iterative step is a worklist algorithm. At each step, it checks if the jump functions of a given formal parameter produce different values. If they do, the parameter is marked as $\perp$, otherwise it is set to that value. The formal parameter is put back on the worklist if its value has changed. Once no more values change (i.e., the worklist is empty), the non-$\perp$ values are the final, inter-procedurally propagated constants.

Slide 95: Interprocedural Data-Dependence Analysis: Motivation-sal Examples

Interprocedural data-dependence analysis is one of the most complex analysis techniques. The slide shows several examples of loops where data dependences cross subroutine boundaries. The loops cannot be parallelized without knowledge of these dependences.

In the first example, the only array reference is in the called subroutine. However, the compiler gathers loop information in the caller, hence interprocedural operation is necessary.

The second and third examples involve data references to the same array in the caller and callee. Among the issues involved are understanding variable renaming (arrays have different names in caller and callee), array accesses and their subscripts, reshaping of variables (arrays may have different dimensions in caller and callee) and side-effects of called subroutines.

Slide 96: Interprocedural Data-Dependence Analysis: Overall Strategy

Several overall driver strategies for interprocedural data dependence analysis have been proposed. We have already discussed subroutine inline expansion. Another idea is to move the enclosing loops *into* the callee subroutines, so that the entire loop nest is inside one function. This method avoids some of the disadvantages of inlining.

The methods we will discuss further is to collect array access information from the called subroutine(s) and use this information in the analysis of the caller.

Slide 97: Interprocedural Data-Dependence Analysis: Representing Array Access Information

The representation of information has an impact on the power of analyses and transformations. We have encountered that fact several times already. It is holds when gathering access information about called subroutines. Among the representations are simple *low:high* pairs, representing an access range, and *low:high:stride* triples, which include stride information, allowing for the representation of certain non-contiguous accesses. Sets of such ranges make it further possible to represent access ranges that include gaps; they are often needed to represent accesses to the same array by different statements.

It is important to recall that data dependence analysis always *must* show a dependence if there is one, but it *may* show a dependence if there is none. Therefore, if exact representation is not possible, the compiler must overestimate. An access range of [10:20:2] would be correctly represented by [10:20] for the purpose of data dependence analysis, but not vice versa.

Exact representations would need to fully capture the subscript functions, the enclosing loop ranges, and conditions under which the references are being made. For large subroutines this information can be large; summarization is needed.

The representation of multiple subscripts is also important. Separate representation is one option; it will support data dependence tests on separate subscripts. For coupled subscripts, linearization is an option. For example, for an array declared as $a[10, 20]$ in the C language, the linearized access of the references $a[i, j]$ would be $a[1, i * 20 + j]$. Linearization is also useful when arrays are reshaped, discussed in the next slide. A drawback of linearization is that the compiler loses the information that the subscript function stays within bounds. For example, in the above original array reference, the compiler can derive the constraint $j < 20$ from the fact that the language disallows out-of-bounds array indexing. This constraint can no longer by derived from the linearized representation; in fact, the representation *does* violate the original bounds of the second array dimension.

Slide 98: Interprocedural Data-Dependence Analysis: Reshaping Arrays

The array dimensions of formal and actual subroutine parameters may not agree. The most common case is that a single row (in Fortran) or column (in C) of a two-dimensional array is passed into the subroutine. This dimensional shift must be considered, in addition to the name change, when collecting interprocedural array reference information. Some languages support other reshapings, as long as the overall arrays size of the original array is not exceeded. For example, Fortran would allow an array declared as $B(10, 10)$ in the caller to be passed onto an array declared as $W(5, 20), X(50), Y(7, 9)$, or $Z(5, 4, 3)$.

The subscript functions of oddly reshaped arrays can be converted accurately using div and mod functions. While this would not be difficult for the compiler, it most likely causes the data dependence test to give up. By linearizing them, oddly reshaped arrays can be turned into arrays with subscript functions that can be compared. The Fortran statement

`EQUIVALENCE(B,Z)` expresses that the two arrays are mapped to the same storage location. It can be used to express the relationship of the two arrays when performing inline expansion (even though this may not help dependence analysis).

Slide 99: Symbolic Analysis

A major distinction of the Cetus and the Polaris compilers over other parallelizers is their ability to manipulate expressions that contain symbolic terms as well as to gather information and reason about these terms.

Symbolic Manipulation

Among the expression manipulation techniques are those that normalize and simplify expressions. For example:

$1 + 2 * a + 4 - a \Rightarrow 5 + a$ (folding)
$a * (b + c) \Rightarrow a * b + a * c$ (distribution)
$(a * 2)/(8 * c) \Rightarrow a/(4 * c)$ (division)

Compiler transformations tend to introduce expressions that may contain multiple constants in different positions, multiple variables with the same name, or subexpressions in parentheses, etc. Such expressions make the output of a source-to-source translator hard to read; they are inefficient to compute at runtime; and they may confound later compiler passes. Polaris and Cetus have frequent calls to the function *simplify(expr)*, which brings transformed expressions back into a normalized form. It enables subsequent analysis and transformation steps to assume that the expressions are in normalized form, which can significantly reduce the number of cases a pass needs to consider.

Expression comparison is a frequent function called in a compiler. As most expressions include one or more variable names, symbolic comparison capabilities are essential. Even if the values of the variables are unknown, there is a chance that symbolic terms may cancel out in the comparison. Where they do not, symbolic range analysis becomes important.

Of further importance are functions that perform symbolic arithmetic. A simple form is the addition of two symbolic expressions $e1$ and $e2$. The addition function would create the symbolic form $e1 + e2$, followed by calling *simplify()* on the result. An advanced symbolic arithmetic function is the computation of $\sum_i^n$, which is useful in induction variable substitution, for example.

Symbolic Range Analysis

Symbolic range propagation [20] gathers and propagates information about the values variables can assume. It also provides access functions to this information, such as obtaining the value range, comparing expressions, and substituting variables with equivalent expressions.

Symbolic Range analysis gathers the information from three sources: assignment statements, conditionals of if-statements, and loop bound expressions. An assignment statement creates a direct equivalence of the left-hand-side variable with the assigned expression. Within an if-statement of the form `if (a<b)` {...} the compiler knows that $a < b$ holds. Within a loop of the form `for (i=1; i<n; i++)` {...} the compiler knows that i assumes values within the range $[1 : n - 1]$. This information is propagated using *abstract interpretation (AI)* [21]. Essentially, this technique symbolically executes the program, but keeps only the information that is of interest plus as much information as needed to execute the program in the desired accuracy. AI of straight-line code is simple; the compiler gathers the assigned expressions in their symbolic form and manipulates them as defined by the assignment statements, using symbolic arithmetic capabilities. Loops pose a challenge for AI, as the number of loop iterations usually is a symbolic expression and thus unknown at compile time. AI executes one or a few iterations and then estimates the effect of the entire loop.

Symbolic range analysis keeps the values and value ranges that variables can assume at a given point in the program in a data structure called *range dictionary.* Pass writers can directly query the symbolic bound of a variable or can compare two symbolic expressions using the constraints given by the set of value ranges.

Slide 100: Symbolic Range Analysis Example

The following example shows a function with the derived variable values and ranges.

```
int foo(int k) {
        [ ]
  int i, j;
        [ ]
  double a;
        [ ]
  for ( i=0; i<10; ++i ) {
        [0<=i<=9]
    a=(0.5*i);
  }
        [i=10]
  j=(i+k);
        [i=10, j=(i+k)]
  return j;
}
```

Slide 101: Alias Analysis

Programming languages offer many ways of naming storage. What is convenience for the programmer can create challenges for the compiler. One of these challenges arises from the feature of the languages that the same storage location can be accessed through multiple names.

When performing compiler analysis and transformations, the compiler must consider the effect of and on *all* aliased variables. This is a basic requirement for all compiler techniques. Even basic register allocation must consider that a write to a variable affects the registers associated with any of the variable's aliases. Alias analysis is therefore a topic of classical compiler text books. We limit the discussion to the motivation of the subject and its impact on parallelizing compilers.

The most direct way of using aliases is the explicit use of two or more names for the same variable. In Fortran, the equivalence statement, `equivalence (a,b)` allocates the two variables, a and b, in the same memory addresses. The *union* construct in C has a similar effect. More subtle ways of aliasing are through parameter passing. The writer of a subroutine *SubX(y,z)* introduces y and z as two different parameters. However, some user may call *SubX(p,p)*, passing twice the same variable. If the parameters are passed by reference, y and z end up being aliased. (Parameter passing by-reference is the default in Fortran; it is supported in C++ (& notation); C only supports parameter passing by value.) Similar holds when passing global variables as subroutine parameters. This may be bad software engineering practice, but the compiler must consider the possibility.

Slide 102: Pointer Alias Analysis

Even more subtle aliasing occurs through pointer references. If two pointers, p and q, hold the address of the same variable, their dereferenced form, $*p$ and $*q$, are aliases of the same memory location. The use of pointers is very common in C programs. Because the C language does not support parameter passing by reference, programmers tend to pass pointers to data structures. It is very important for C optimizing compilers to understand pointer aliases. Because a pointer that was passed as a subroutine parameter could potentially represent any program variable, compiler optimizations would have to be extremely conservative in the absence of alias analysis.

Parallelizing compilers are most successful in science/engineering applications. The common case for these applications is to use explicitly named array data structures. Existing pointer analysis techniques are usually successful in detecting the needed alias relationships in these programs. Programs that contain dynamic, pointer-linked data structures pose additional challenges. The nodes of a dynamic data structure do not have explicit names. Their "names" are only implicitly given by a pointer referencing a node. Such data structures can be looked at as graphs. Alias analysis for graphs is often called *shape analysis*, referring to the structure of the graph – linked lists, rings, or trees, for example. Shape analysis is the first step in understanding alias information in graphs. Alias relationships can then be formulated from the graph type. For example, *children of a node in a tree are not aliased.*

Slide 103: Is Alias Analysis in Compilers Important?

Fortran used to be the dominant language in science/engineering applications. Today, we see an increasing number of such applications written in C, C++, some even in Java. While the latest Fortran standard, Fortran 2008, supports dynamic data structures, many "legacy" science/engineering applications still exist in Fortran77. This language did not support dynamic storage allocation and pointers. Explicit equivalencing, as in *equivalence (a,b)*, caused one of the few alias problems Fortran77 compilers had to worry about. Even though subroutine parameters are passed by reference, there is an important Fortran language rule that made the compiler's life easier: *Subroutine parameters that are aliased must not be written* – it is the programmers' responsibility to make sure of this. This rule essentially allows compilers to ignore aliasing by parameter passing. The rule makes a crucial difference for compilers between transforming C and Fortran programs. The absence of a corresponding rule in C, makes alias analysis necessary. This is true even in C programs that do not use pointers other than for passing subroutine parameters. We have mentioned above that alias analysis is important for classical compiler optimizations already. For parallelizing compilers of C programs, the absence of alias information would force the compiler to be so conservative that the detection of parallelism were essentially inhibited.

Adding to the challenge of pointer analysis in C programs is the language's support for dynamic memory allocation and – most difficult for compilers – pointer arithmetic and type casting. The latter opens the door for pointers to point "anywhere", as even the most sophisticated analysis may not have sufficient knowledge at compile time to tell accesses to different storage locations apart.

Compilers may offer command line options, through which the user provides certain guarantees to the compiler. An important option that is supported by some compiler is "assume Fortran aliasing rules", telling C translators that subroutine parameters can be assumed as not aliased. This option is especially useful in Fortran programs that have been converted to C.

4.5 Dynamic Decision Support

Slide 105: Achilles' Heel of Compilers

Even the most sophisticated compilers are limited in their power because they do not have enough information to make best optimization decisions. These decisions entail what optimizations to apply to what program sections and what flavor of the optimization to choose, where such options are available. Making these decisions correctly is very important, as we have seen in the many discussions of the techniques' performance and overheads. All techniques, with very few exceptions, have the potential to *reduce* the performance of the "optimized" program. This is because they introduce overheads by transforming the code or by providing advanced analysis information, which, in turn may enable transformations causing overheads. Since the early generations of optimizing compilers, the phenomenon has been observed that a newer, more powerful compiler sometimes led to reduced performance.

This phenomenon is not reserved for parallelizing compilers. Users who have experimented with even basic command line flags have realized that "-O2" sometimes outperforms "-O3", and "-O5" may even break some programs.

The primary reason for insufficient information for good optimization decisions is the missing program input data. The size of a program's data set is usually given in an input file. Yet, it often ends up, directly or indirectly in a loop bound, where is has a decisive impact on the profitability of executing the loop in parallel. The architecture's cache size is important to know when tiling a loop. For optimizations that affect the placement of data in memory, the memory layout is of further importance to know.

A secondary reason for dynamic decision support is that, even if compile-time information were available, the needed performance models for deciding which combination of optimization techniques is the best for a given code section is excessively complex. Today, even the most sophisticated models, only capture a limited number of performance and overhead factors. The fact that there is interactions among optimization techniques and among the optimized code sections further increases the complexity of performance modeling.

The effect of these compiler challenges for the compiler writer is that the profitability of an optimization is difficult to determine. Heuristics are usually the best compile-time choices.

For the user, the same challenges mean inconsistent performance behavior of the compiler. Some compiler engineers limit the potential damage optimizations can do, by being overly conservative; the compiler may refuse to apply a certain optimization, even when directed explicitly by a command line option. Another effect users have to deal with is the inflation of command-line option. The number of these options reflect the problem discussed here in a direct way. By giving the user command line flags, the compiler writer passes the responsibility of making good optimization decisions onto the user. Users may need to "play" with these options extensively to obtain the best possible performance.

Slide 106: Multi-version Code

An early attempt of parallelizers to overcome some of these issues was the introduction of two-version loops. The compiler generates a serial and a parallel version and inserts code that chooses at runtime between the two. In the example shown on the slide, the decision tests a dependence criterion. Another criterion could deal with an unknown loop iteration or statement count; a simple heuristic may choose the serial loop, if the product of the number of iterations and statements is less than a threshold.

If there is more than a single, binary optimization decision to make, the compiler can generate multi-version code. There are some obvious disadvantages. The readability of the translated code can decrease and the generated code size can increase dramatically if the compiler generates many multi-version loops. Not all optimizations are amenable to multi-version code generation. With a large number of optimizations, this approach becomes infeasible, as the number of versions and their combinations would become excessive.

Slide 107: Profiling

Profile-based optimization has been known for some time. The compilation process takes two phases. The first compiler phase simply instruments the code, so that it gathers profile information as is runs. The second compilation phase uses the profile for optimization decisions. Profiling was initially used for branch prediction in code-generating optimizers. The profile gathered branch frequencies, which was then used for instruction scheduling. Recently, compilers have added information that is gathered in the profile, enabling an increased number of optimization decisions.

A profile run should be made with a data set that is representative of those used in the later, production use of the code. The data used for profiling is often called the *training data*, as opposed to the *production* or *reference data*. The sensitivity of the profile-based optimization to the representativeness of the training data is critical. This seemed less of a problem in early branch prediction; however, more advanced optimizations are more vulnerable to differences between training and production data. It can still happen that profile-based optimization leads to suboptimal programs performance.

Note that the availability of profile information does not remove the challenge for the compiler writer to create advanced performance models. This represents a second reason why profile-based optimization may perform suboptimally in some programs.

Slide 108: Autotuning – Empirical Tuning

The ultimate judge of the right optimization decisions is the runtime performance of the program. Empirical tuning tries many optimization variants at runtime and picks the best. This method is sometimes referred to as autotuning; it is an active research area, where terminology is subject to change.

The figure shows a generic scheme for empirical tuning. The available optimizations represent a *search space*, each point representing a particular program variant that may be explored. The overall driver of the process (sometimes called the tuning engine) picks one point at a time, calls the *version generator*, which is essentially the compiler. Next, it executes the generated code and measures its performance. This process repeats until the driver decides that it has found a suitable variant.

The big advantages of empirical tuning are that this process works without performance models and that the actual program execution time is used to judge performance. A big disadvantage is the involved tuning time. This tuning time is large and increasing for several reasons. First, the number of compiler options increases, as discussed earlier. Second, many optimizations interact; an optimization may have a positive effect, but, after applying another transformation, the same technique may degrade program performance. Third, compilers want to make optimization decisions not only at the whole-program level, but for individual loops or program sections. This requirement further increases the already large search space. It also challenges compiler writers into developing methods for specifying optimizations on individual code sections. For example, command line options could request the application of optimizations to certains loops.

The effect of empirical tuning was presented on Slide 36. Without dynamic decision making, the Cetus compiler often degrades performance. Heuristics ("Model-based" on Slide 36) and Profile-based can often remedy this situation and at least deliver non-degrading performance. Empirical tuning leads to additional performance gains. In half of the benchmarks, the resulting speed matches or exceeds the performance of the hand-parallelized code.

4.6 Techniques for Vector Machines

Slide 110: Vector Instructions

A vector instruction performs n operations, such as add or multiply, on n operands "simultaneously". For example, *vadd va,vb,vc,32* adds the two vector registers *va* and *vb* and puts the result in *vc*. The architecture needs to provide the support for input and output operands of the vector instruction. This can be the provision of vector registers or the ability of memory accesses to stream into and out of the operation. The latter was done in early supercomputers. There were up to three paths to/from memory – two for the input operands and one output stream. In today's processors, vector operations are usually performed on vector registers. There can be specialized vector registers. Also, regular registers can be used for vectors of small numbers. For example, a 64-bit register can be used to hold eight 8-bit numbers. Adding two such registers, for example, takes a specialized add instruction (one that suppresses the carry bit across the seven boundaries between the numbers). Such instructions are often referred to as multi-media extensions (MMX), because they are useful for certain graphics applications, among others.

The arithmetic units that perform the vector operation can be n-wide ALU's (again, this can be a modified wide operation, such as a 64-bit adder with suppressed carry bits). In early supercomputers, where arithmetic operations took many cycles and stages, vector speedup was achieved by pipelining the vectors through the stages of one (or a few) ALUs.

Slide 111: Applications of Vector Operations

Vector operations are regular in nature and apply to a large data space. This property is found in science/engineering applications, which contain loops with large iteration counts, no internal control flow (no if-statements) and regular accesses to array data. These are the same properties that enable the automatic detection and exploitation of parallelism.

The availability of this type or large, regular parallel operations was very important for the early supercomputers. They gained their speed from pipeline parallelism. The vector pipelines could contain many stages. The peak speed usually was obtained by the *chained* operation of a multiply and add vector instruction, where the multiply-pipeline fed into the add-pipeline. A pipeline length of 10 was not unusual. Hence the startup and completion of the vector operation was costly. Long vector operations were needed for reaching peak-speeds on the overall application.

The pipeline startup is not an issue on today's MMX instructions. Many basic image processing and rendering applications are amenable to such operations, as they can make use of large arrays of small (e.g., 8-bit) integer data representations.

Slide 112: Basic Vector Transformation

A single loop with a single, dependence-free array-type statement can be directly translated into vector operation. In high-level languages, the *triplet* notation is often used to express such operations. *A(1:n)=B(1:n)+C(1:n)* assigns the element-wise sum of B and C to A, for the entire range from 1 to n. Note that this notation is also used in some languages, such as modern Fortran, to express array operations, without implying that the operation needs to be implemented through vector instructions.

If the loop includes two statements, each statement is translated into a vector operation. Note that this translation changes the execution order. All elements of the first vector statement are computed first before proceeding to the second statement. The common semantics of the vector statements are so that the entire right-hand side is evaluated first, before assigning the left-hand side.

Slide 113: Distribution and Vectorization

The transformation that reorders the computation as needed for vector operation represents a loop distribution step, as shown on the slide. Loop distribution is subject to data-dependence constraints, as we have seen earlier. The transformation can be legally applied if the dependences are lexically forward. Hence, multiple statements in a loop can be vectorized if they do not contain lexically backward dependences.

Statements in a loop that do not contain dependence cycles can be reordered, so that all dependences are lexically forward. Detecting dependence cycles is an important step in vectorizing compilers. Cycle-free dependence carrying loops can be vectorized, whereas direct parallelization is not possible. However, loop distribution could enable the same degree of parallelism.

The overheads of vectorization and parallelization are different. The measurements of the *vector* and *concurrent* capabilities of the Kap compiler (Slide 17) have shown that vectorization-only performs better in some programs than parallelization-only, and vice versa. Applying the combined transformation is often best, but now always. Understanding the overheads and benefits of the transformations is crucial, here as well.

Slide 114: Vectorization Needs Expansion

The analysis of privatizable data is as important for vectorization as for parallelization. However the implementation is different. Data needs to be expanded, rather than simply declared private, as shown on the slide. This is needed because of the reordering of the computation. All temporary values of the privatizable data (scalar or array) need to be kept for later use. This need would be the same if the loop were distributed before parallelization.

So, in effect, it is the distribution transformation that causes the need for expansion instead of privatization.

In the example shown on the slide, the need for expansion can easily be seen when expressing the vector statements in triplet notation. The array T needs to hold all the values computed in the first statement. The entire $T(1 : n)$ is then used in the vector operation of the second statement.

Early compilers for parallel machines mention only the scalar expansion technique, including the Kap compiler discussed in this course. Privatization was developed later. The reason for this is simply that some of the early, powerful parallel machines were vector computers.

Slide 115: Conditional Vectorization

Vectorization is most efficient in regular, straight-line code. An if-statement cannot be directly translated into vector code. However, there is a vector conversion that enables conditional vectorization. The $WHERE$ statement is the vector equivalent of the IF construct. It expresses that an operation is to be applied to any element for which the condition evaluates to true.

The implementation of the $WHERE$ statement is as follows. In a first step, a boolean vector – a *mask* – is created that reflects the true/false conditions for the entire vector. In a second step the specified operation is executed as a vector instruction; the instruction is performed for all elements of the vector, but the mask is used to enable or disable storing the result in the target register.

Conditional vectorization performs unnecessary operations – for every element whose condition is false. For conditions that are true at most positions, this overhead is small. If the condition is rarely true, the overhead can be large.

Slide 116: Stripmining for Vectorization

Stripmining is useful and needed in vectorizing compilers in two different ways. In the actual vector code generator, long vector operations need to be split into the lengths supported by the instructions and vector registers. In the given example, the inner loop would be turned into vector instructions of length 32 and the outer loop encloses these instructions.

Stripmining is also useful for dividing singly-nested loops into two loops for parallelization and vectorization. In this case, the outer loop would be executed in parallel, while the inner is vectorized. This form of stripmining was applied in the Kap compiler measured on Slide 18.

4.7 Compiling for Heterogeneous Architectures

Slide 118: Why Heterogeneous Architectures?

Heterogeneity in computer architectures has two reasons, performance and energy. Programs that cannot be parallelized need a powerful uniprocessor. By contrast, highly parallel code runs best on a large number of simpler cores. This is because the return in performance for increasingly complex architectures diminishes; if an application can be parallelized, it is more beneficial to use the architectural real estate for creating a second core rather than enhancing the first core's capabilities. An architecture that is capable of efficiently running both serial and parallel programs would consist of one powerful processor and a large number of simpler cores.

A similar heterogeneity argument can be made for special code patterns that are executed at best performance on special-purpose hardware. Many different, specialized cores could be integrated into a general-purpose platform that performs well on diverse applications.

From the energy point of view, similar arguments hold as well. Even if sufficient numbers of transistors were available to create many powerful cores, the energy dissipation would become infeasible. Creating more of the simpler cores is the alternative.

Slide 119: Examples of Accelerators

The combination of a CPU with a co-processor or *accelerator*, represents a heterogeneous architecture. Many such accelerators exist today. Math co-processors are among the earliest examples. Network processors, which perform incoming and outgoing communication, and graphics processing units, which operate the video output, are also common in today's platforms. There are processors that can handle encryption and encoding/decoding of video streams. IBM's Cell processor was an early accelerator for high-performance computing. It was used in the first supercomputer that exceeded one petaflop/second [22]. Graphics processors were developed into general-purpose GPUs by Nvidia. GPGPUs are currently one of the most widely used platforms for research in heterogeneous computing. FPGAs are flexible components that can be programmed to perform a number of functions "in hardware". Among the newer developments is also Intel's MIC (many integrated cores), which is a many-core architecture that can also be used as a standalone platform.

Slide 120: Accelerator Architecture

The question of what is the best accelerator architecture is a topic of current research. One characteristic that distinguishes these architectures from CPUs is that caches and individual cores are much smaller, allowing more cores to be placed, as illustrated on the slide.

The slide also shows the architecture of a GPGPU, as seen by the compiler or programmer using the CUDA programming language. The address space is separate from the CPU, requiring data to be transferred before and after a code section executing on the GPGPU device (referred to as a *kernel call*). There is a complex memory hierarchy, consisting of global, shared, texture, and constant memory, as well as registers. There is a large number of cores (e.g., 512), which execute in SIMD style. The architecture uses multithreading, which is able to switch from one block of threads to another while waiting on data. Of great importance are *coalesced* data accesses. That is, adjacent threads need to access adjacent memory locations, for best performance. This allows the architecture to transfer a block of data from memory and access at once by the threads executing in SIMD lock-step. This and the multithreading mechanism are key to these devices' performance.

Slide 121: Compiler Optimizations for GPGPUs

The following three categories of optimizations are important in GPGPUs [23]. The first category optimizes GPU global memory accesses. It includes the transformations *Parallel Loop Swap, Loop Collapsing,* and *Matrix Transpose.* The second category exploits GPU on-chip memories. It finds the most suitable locations in the complex memory hierarchy for data placement, depending on access type. The third category optimizes CPU-GPU data movement by analyzing data that needs to be copied and by finding the best program point at which to perform the transfer.

Slide 122: Parallel Loop-Swap Transformation

Parallel Loop swap interchanges loops, moves their parallel status, and changes the scheduling, so that adjacent threads access adjacent memory locations – creating coalesced accesses. Note that this transformation aims for the opposite memory access behavior of what is desirable in today's multicores. In common multicores, the same thread must access adjacent memory locations in order to achieve spatial locality and avoid false sharing.

Slide 123: Loop Collapsing Transformation

Loop Collapsing also creates coalesced accesses. It is used for the pattern of irregular memory accesses found in some computational algorithms. In this pattern, an outer loop strides in irregular distances through an array, while the inner loop touches all elements within these distances. By collapsing the two loops, the accesses touch all array elements contiguously.

Slide 124: Matrix-Transpose Transformation

Matrix transpose has the same effect on array accesses and spatial locality as loop interchange. It switches the declared dimensions of an array. Transposition of data structures is rarely applied in classical optimizing compilers, because it affects all parts of a program, requiring global program analysis and optimization. The situation is different in kernel functions for off-loaded computation. The transposition can be done during the data transfer between GPU and CPU; hence, it becomes a local optimization.

Slide 125: Techniques to Exploit GPU On-chip Memories

The second category of optimizations deals with the placement of data in the complex memory hierarchy of GPGPUs. The table shows the data access attributes and memory placement/caching strategies used in [23]. This method uses simple placement heuristics. Optimal placement of data in complex accelerator memory structures is still a topic of active research.

Slide 126: Techniques to Optimize Data Movement between CPU and GPU

The third category of optimizations deals with the transfer of data between CPU and GPU. Data that is no longer live after a GPU kernel execution does not need to be copied back to the CPU. Similarly, data that already resides on the GPU, does not need to be transferred there when needed again for the execution of the next kernel. Finding the best transfer point is of further importance. The copy operation can be hoisted to an earlier point in the program, so that the data transfer gets overlapped with computation. This amounts to data prefetching. The common drawback of prefetch operations apply: The operation reserves space in the GPU global memory, which reduces the space available for the current computation.

Slide 127: GPGPU Performance Relative to CPU

This slides shows overall performance gains from optimizing and executing some of the NAS Parallel Benchmarks on GPGPUs. The reference point on the slide is a single core execution. Multi-core execution would reduce the shown, relative speedups.

Among the important results are the *unoptimized* performance, which shows the speed of programs translated for GPU execution by offloading all eligible parallel loops to the GPU and copying all needed data before and after these kernels. *All opts* shows the performance after applying the transformations described on the previous slides. Various tuning options have been applied, for the details of which the reader is referred to [23]. A most important comparison point is the hand-optimized CUDA performance, which was obtained by the benchmark versions that were manually transformed into efficient GPGPU code using the CUDA programming interface.

Slide 128: Importance of Individual Optimizations

This slides summarizes the importance of individual optimizations for GPGPUs on the NAS and Rodinia benchmarks. For details, the reader is referred to [24]. The figure shows the performance drops resulting from switching off individual optimizations in the best-performing version of the program. A large performance drop indicates high importance of the technique. The summary shows that a large number of optimizations, including the transformations presented previously and a range of CUDA options, influence performance. The figures also show that the performance varies significantly across the benchmarks.

4.8 Techniques Specific to Distributed-memory Machines

Slide 130: Execution Scheme on a Distributed-Memory Machine

The common execution model on distributed-memory systems is referred to as SPMD, or single-program multiple data model. A key difference from the fork-join model is that SPMD execution starts and ends as a parallel program. By contrast, a program in the fork-join execution model starts and ends as a sequential thread, spawning parallel threads when needed. There is no fork instruction at the beginning of an SPMD program. It is usually the operating system or runtime library that invokes a given program on multiple processors. The multiple processors execute the same program, which is often referred to as a *node program*. In order to perform different actions, the node programs use their processor (or process, thread, node) number (in MPI it is called *rank*) to select that action. The slide shows an example. The *node_id* is used to select a part of the original loop iteration space for execution on each node.

Since the address space is distributed, data now needs to be partitioned and placed onto the various memories. Furthermore, if a processors wants to access data placed on a different node, the data needs to first be transferred. This is usually done through explicit message passing. The most commonly used message passing library is MPI [25]. An important question is also how to synchronize processors; for example, how to realize the barrier that usually terminates a parallel loop. Synchronization is generally performed as part of the data communication.

Slide 131: Data Placement

The most common data placement scheme is to assign a partition of the data to each involved memory. This way, each data element has a single *owner*. Most programs use static ownership. That is, the owner remains the same during the course of the program.

Alternatively, data can be replicated on multiple memories. This will benefit multiple readers of the same data, but will increase the difficulty of maintaining data consistency (i.e., multiple copies of the data having the same value).

Slide 132: Data Distribution Schemes

Data can be partitioned in ways similar to the iteration partitioning schemes discussed earlier. The most common scheme is *block partitioning*, where each node takes a contiguous block of the data structure. *Cyclic partitioning* of an array assigns to the first of n nodes the array elements $1, n+1, 2n+1, ...$, to the second node the elements $2, n+2, 2n+2, ...$, etc. *Block cyclic* is a mix of the two schemes, where the first node takes the first b elements (b being some block size), the second through the n^{th} node take b elements each, and this scheme repeats until all elements are distributed.

The above three data distribution schemes are regular. There is also a scheme that supports irregular distributions. *Indexed distribution* uses and index array that indicates the node numbers on which array elements (or blocks of elements) are placed.

Data distribution is commonly done manually. The High-Performance Fortran language, which was widely used in the 1990s, offered data distribution directives to the programmers. The job of HPF [26] compilers was to generate messages from these directives. HPF essentially provided a shared-address space model to the programmer, which the compilers translated into node programs for execution of distributed-memory machines.

Automatic data distribution was the topic of several research efforts [27, 28, 29], but did not get adopted in commercial compilers. A big challenge for automatic data distribution is that it needs to perform a global optimization, finding the best partitioning across potentially many loops and subroutines. To date there is no commercial compiler that fully automatically translates a pure shared-address space program into code for distributed-memory architectures. We will discuss research efforts in this direction in a later slide.

Slide 133: Message Generation for Single-Owner Placement

The slide shows a simple example of a loop and data that are distributed in a way that minimizes communication. The partitions of both the iteration space and the data have the same lower and upper bounds, lu and ub. Hence, the accesses $A(i)$ and $B(i)$ are local accesses; however, $A(i-1)$ needs to be communication from node x to node $x+1$. Each processor sends the element $A(ub)$ to the next processor and receives the element $A(lb-1)$ from the previous processor.

Languages such as HPF and related research papers have extensively explored techniques for generating such code from sequential programs that are annotated with data distribution directives.

Slide 134: Owner Computes Scheme

A scheme used by most compilers that implemented HPF and related languages was called *owner computes*. Each statement of a loop (or other constructs) are executed by the processor that owns the data element being written. Conceptually, the owner computes scheme can be implemented by guarding each statement with an if-statement that tests if the left-hand side element of the statement is placed on the local memory, as shown in the slide.

Under the owner computes scheme, data writes are local operations. However, reads may need to be received from other processors. In the conceptual design, before a receive operation precedes each statement, obtaining needed data that reside on remote processors. Each processor also needs to determine what other processors needs data from its local memory and send it to them.

For simple statements with regular data accesses, the described method is not difficult to apply by a compiler. The first statement $A(i) = B(i) + B(i-m)$ is such a case. For irregular accesses, the involved compiler decisions are complex and, for efficient code generation, may need information that is unavailable at compile time.

Clearly, the conceptual scheme shown in this example is inefficient. Compiler optimizations are needed to generate quality code. The next slides summarizes these optimizations.

Slide 135: Compiler Optimizations for the Raw Owner Computes Scheme

Many techniques help improve the code generated by the raw owner computes scheme. For regular accesses, the if-conditions guarding the statements can be evaluated at compile-time and, as a result, the iteration spaces of the loops can be reduced. In may such cases, the iteration spaces of the resulting loops may end up matching the local partition of the data being written.

Aggregating communication is another important optimization. It is more efficient to send multiple data elements combined in a single message than multiple messages. In the conceptual example, each loop iteration sends and receives data needed by the current statement. It is beneficial to move the communication of the aggregated data for all n statements before the loop.

There is a tradeoff however. Aggregation may requires the producer of the first element to wait until the last element of the combined message has been produced. As a result, there will be less potential overlap of communication and computation.

Overlap of computation and communication can also be increased by moving send operations to earlier program points. The earliest possible point would be after the statement that produces the data value. This optimization is similar to data prefetch; it reduces the effective message latency.

Slide 136: Message Generation for Virtual Data Replication

The owner-computes scheme results in write operations being local data accesses. Only data read references need to be received from the owner nodes. The opposite is the case in a scheme that virtually replicates data [30, 31]. In this scheme, parallel sections (usually parallel loops) operate only on local data; after each parallel section, the written data is conceptually broadcast to all nodes. An important compiler optimization is the reduction of the broadcast to the communication of only the data that is needed by each processor in the future. The advantages of this scheme include the fully parallel sections with local reads and writes. Also, the computation of message sets seems easier for the compiler than with given data distributions. The disadvantages include limited data scalability; due to replication, a program cannot easily process a larger data set with increasing numbers of processors. Also, replication may result in increased write operations, even though collective communication functions perform such operations efficiently.

Bibliography

[1] 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.

[2] 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.

[3] SC Chen, D.J. Kuck, and A.H. Sameh, "Practical parallel band triangular system solvers," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 270–277, 1978.

[4] D.D. Gajski, "An algorithm for solving linear recurrence systems on parallel and pipelined machines," *Computers, IEEE Transactions on*, vol. 100, no. 3, pp. 190–206, 1981.

[5] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U.M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner, "The Cedar System and an Initial Performance Study," *Proceedings of the 20th Int'l. Symposium on Computer Architecture, San Diego, CA*, pp. 213–224, May 16-19, 1993.

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

[7] Dheya Mustafa, Auranzeb, and Rudolf Eigenmann, "Performance analysis and tuning of automatically parallelized openmp applications," in *Proc. of the International Workshop on OpenMP, IWOMP*, 2011, pp. 150–164.

[8] Z. Shen, Z. Li, and P.E.N.C.H. YEW, "An empirical study on array subscripts and data dependencies," in *1989 International Conference on Parallel Processing, University Park, PA*, 1989.

[9] U.K. Banerjee, *Dependence analysis for supercomputing*, Kluwer Academic Publishers, 1988.

[10] William Blume and Rudolf Eigenmann, "Symbolic range propagation," in *the 9th International Parallel Processing Symposium*, 1995, pp. 357–363.

[11] M.R. Haghighat and C.D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 4, pp. 477–518, 1996.

[12] M. Wolfe, "Beyond induction variables," in *ACM SIGPLAN Notices*. ACM, 1992, vol. 27, pp. 162–174.

[13] Bill Pottenger and Rudolf Eigenmann, "Idiom Recognition in the Polaris Parallelizing Compiler," in *The 9th ACM International Conference on Supercomputing (ICS'95)*. 1995, pp. 444–448, ACM Press.

[14] M. Wolfe, "Loops skewing: The wavefront method revisited," *International Journal of Parallel Programming*, vol. 15, no. 4, pp. 279–293, 1986.

[15] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, et al., "The nas parallel benchmarks summary and preliminary results," in *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*. IEEE, 1991, pp. 158–165.

[16] Zhelong Pan, Brian Armstrong, Hansang Bae, and Rudolf Eigenmann, "On the interaction of tiling and automatic parallelization," in *First International Workshop on OpenMP*, 2005, pp. 24–35.

[17] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua, "Cedar Fortran and its Restructuring Compiler," in *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau D. Gelernter, T. Gross and D. Padua, Eds. 1991, pp. 1–23, MIT Press.

[18] K. Kennedy and J.R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann Publishers Inc., 2001.

[19] Samuel P. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*, Morgan&Claypool Publishers, 2012.

[20] William Blume and Rudolf Eigenmann, "Symbolic Range Propagation," *Proceedings of the 9th International Parallel Processing Symposium*, pp. 357–363, Apr. 1995.

[21] 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 the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[22] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho, "Entering the petaflop era: the architecture and performance of roadrunner," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 1.

[23] Seyong Lee and Rudolf Eigenmann, "Openmpc: Extended openmp for efficient programming and tuning on gpus," *International Journal of Computational Science and Engineering*, vol. 7, no. 1, 2012.

[24] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann, "Effects of compiler optimizations in openmp to cuda translation," in *Proc. of the International Workshop on OpenMP, IWOMP*, 2012.

[25] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface, seconde édition*, the MIT Press, 1999.

[26] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steel Jr., and M. E. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.

[27] M. Gupta and P. Banerjee, "Paradigm: A compiler for automatic data distribution on multicomputers," in *Proceedings of the 7th international conference on Supercomputing*. ACM, 1993, pp. 87–96.

[28] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.Y. Wang, W.M. Ching, and T. Ngo, "An hpf compiler for the ibm sp2," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*. IEEE, 1995, pp. 71–71.

[29] J. Garcia, E. Ayguade, and J. Labarta, "A novel approach towards automatic data distribution," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*. IEEE, 1995, pp. 78–78.

[30] Ayon Basumallik and Rudolf Eigenmann, "Towards automatic translation of openmp to mpi," in *Proc. of the International Conference on Supercomputing, ICS'05*, 2005, pp. 189–198.

[31] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff, "A hybrid approach of openmp for clusters," in *Proceedings of the 17th ACM symposium on Principles and practice of parallel programming*, 2012.