Scalar vs. Parallel Optimizations

Michael Wolfe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-010

July, 1990

Scalar vs. Parallel Optimizations

Michael Wolfe
Oregon Graduate Institute of Science and Technology
19600 NW von Neumann Drive
Beaverton, OR 97006

mwolfe@cse.ogi.edu
503-690-1153

Abstract

In the realm of high performance and parallel computers, defining the order in which to perform program optimizations and the interaction between optimizations is a difficult task. The compiler must perform some tradeoff between different optimizations, in particular trading traditional scalar optimizations against new parallel optimizations. Since the potential performance benefit of parallel optimizations can be an order of magnitude higher, performing scalar pessimizations (inverting scalar optimizations) to enable parallel optimizations can be the right overall decision. It is difficult to treat parallel and scalar optimizations with a single framework, since they deal with different machine abstractions. We also show that no fixed ordering of optimizations will find the optimal transformed program; the ordering of optimizations must depend both on the target machine and on the program being compiled.

Keywords: compiler optimization, loop optimization, loop parallelization

1. Prologue

Optimizing compilers are important for producing efficient code for high-level languages. Translators for high level languages typically perform several classical scalar optimizations, such as code floating, strength reduction, common subexpression removal, and so on. Most applications of classical optimizations arise from the semantic gap between the high level language and the machine instruction set. For instance, where a language supports arrays and loops, the instruction set might only support address arithmetic, register indirect loads and conditional branches. A simple translation of each array reference and loop to its direct instruction

set equivalent would be too inefficient. Recognizing the common addressing arithmetic among multiple array references, floating partial computations out of loops and changing high cost operations (multiply) into lower cost operations (add or increment) can generate code almost as efficient as hand written machine code, at a lower cost to the programmer and with fewer errors. Some computer systems attempt to fill the semantic gap with hardware, for instance by directly supporting multidimensional array indexing; experience seems to have shown that a more effective solution is to use less expensive hardware and optimizing compilers. Scalar optimizations typically produce speed improvements in the range of 2-3, by reducing the total number of instructions in the program, the cost of those instructions, and the frequency of execution of some instructions.

With high performance computers, such as parallel or vector computers with deep memory hierarchies, there is again a semantic gap between the program and the hardware of the computer system; some approaches to filling this gap are to change the language to more closely model the computer system hardware, such as adding parallel extensions to the programming language. Many users have found that the critical element in extracting maximum performance is to efficiently use the memory hierarchy, such as effective use of cache memory. Some of these users resort to manually restructuring their programs to take advantage of this (supposedly transparent) architectural feature. As with scalar optimizations, these manual techniques have spawned a number of automatic parallel optimizations. These optimizations are in some sense more important than scalar optimizations, since the potential benefit from a parallel optimization is an order of magnitude greater than for scalar optimization, by improving memory hierarchy utilization, reducing the frequency of interprocessor communication, or increasing the total parallelism in the program. Since these computers are often used in applications in which the computation is concentrated in nested loops, many parallel optimizations focus on loops, such as loop distribution, loop fusion, loop interchanging, strip mining, supernode partitioning, and so on.

A recent paper by Whitfield and Soffa described an approach for ordering optimizations in a compiler [WhS90]; they attempt to treat scalar and parallel optimizations in a uniform manner, and show that some optimizations work at cross purposes. In particular, some optimizations prevent (disable) application of other optimizations. From this observation they offer a framework from which to deduce an ordering in which the various optimizations should be performed.

Since the potential benefit of parallel optimizations is much greater, we focus here on the parallel optimizations. Moreover, we show how parallel optimizations can be enhanced by performing the inverse of some scalar optimizations, an option not previously considered. We also show that a single ordering of parallel optimizations cannot be chosen that will give optimal results all the time.

In particular, the scalar optimizations considered here are:

- CSE - Common Subexpression Elimination
- IFU - IF Unswitching (floating IFs)
- ICM - Invariant Code Motion
- RED - Strength Reduction

The term "parallel optimization" reflects that the machines for which these transformations are of most interest are parallel or vector computers; nevertheless they can be used to enhance the performance of many high performance scalar computers also. The parallel optimizations considered are:

- DIS - Loop Distribution
- FUS - Loop Fusion
- INX - Loop Interchanging
- PAR - Loop Parallelization

Whitfield and Soffa showed that invariant code motion can disable INX and FUS in some instances; here we intend to show that ICM^{-1} can enable INX. We will also show (for instance) that RED applied in certain circumstances can disable PAR, CSE can disable DIS, IFU can disable INX and FUS, and that the application of the inverse of these scalar optimizations can sometimes enhance the application of parallel optimizations.

2. Optimizations and Equivalent Programs

An optimization O is a function F_O that takes a program $\mathbf{P}_0$ and generates a new (equivalent) program $\mathbf{P}_1$, such that according to some cost metric C, the cost is reduced.

$$\mathbf{P}_1 = F_O(\mathbf{P}_0)\,, \quad \text{such that} \quad C(\mathbf{P}_0) > C(\mathbf{P}_1)$$

Typically, optimizing compilers use the estimated or predicted number of instructions executed as the cost metric; other metrics may be code size, amount of parallelism, usage of higher level of memory hierarchy, and so on.

Two correct programs are equivalent (in the context of this paper) when the two programs exhibit the same observable behaviour. For instance, the programs:

```
A := 5;          A := 2+3;
B := A + A;      B := 2*A;
print B;         print B;
```

would be considered to be equivalent, since they would both print the number "10". The observable behaviour is generally taken to mean that certain well-defined operations that communicate with the environment of the program (such as input/output statements, or messages to/from other programs) will occur in the same order and with the same content. We do not mean equivalence in the sense of [HPR88], where the behaviour of a program is taken to mean that the same operations are performed on the same data. We also focus on correct programs; we allow a compiler to optimize a program that terminates abnormally to change the mode in which it terminates, thus avoiding many complications [CaF89]. Some applications can use more aggressive optimizations if the order of externally visible events does not matter, such as if reading and writing two different files can be reordered (it *can* matter if the write is a prompt for the data for the read). Also, some parallel compilers allow optimizing a loop such as:

```
for I := 1 to N do
  S = S + A(I)
endfor;
```

by executing the iterations of the loop in a different order, such as backwards. Mathematically, the sum may be accumulated in any order since addition is commutative. However, any numerical analyst will realize that the answer with floating point computer arithmetic can be sensitive to the order of accumulation. In general, we assume that optimizations will only reorder floating point calculations when the roundoff characteristics will not be affected, and that input/output (or any intentional external visible events) will not be reordered or modified.

We assume the scalars in the program have been converted to Static Single Assignment (SSA) form [AWZ88, CFR89]. In some sense, this simplifies the program and eliminates the need for an extra precondition for loop fusion involving upward-exposed uses of scalar variables; using the SSA form, simple dependence tests are sufficient.

3. Brief Review of Dependence

Since dependence conditions are used to test for when a parallel optimization is legal, here we briefly review definitions of dependence [BCK79, FOW87]. In these definitions, a statement refers to a element of the intermediate language of the compiler, which may be a subexpression,

a statement or a basic block. A flow dependence relation $S_v \delta S_w$ occurs when statement S_v defines a variable and statement S_w uses or might use that definition. An anti-dependence relation $S_v \bar{\delta} S_w$ occurs when S_v uses a variable that is subsequently redefined by S_w. An output dependence relation $S_v \delta^o S_w$ occurs when S_v defines a variable that is subsequently redefined by S_w. A control dependence relation $S_v \delta^c S_w$ occurs when S_v is a control statement (conditional branch or loop header) that determines whether or not statement S_w will execute. Usually we are not concerned with the kind of dependence, so we will just say that S_w is dependent on S_v, written $S_v \delta^* S_w$.

In loops we are also concerned with the relative iterations of the statements involved in the dependence, and we distinguish three cases [PaW86]. If S_v is enclosed in one loop, let S_v[i] refer to the execution of S_v during iteration i of the loop. If S_v[i] δ^* S_w[j], then depending on the sign on i-j we have three possible dependence directions. If i-j<0, then we have a forward dependence, written S_v[i] $\delta^{(<)^*}$ S_w[j]; if i-j>0, we have a backward dependence direction, written S_v[i] $\delta^{(>)^*}$ S_w[j]; finally, if i-j=0, we have an equal dependence direction, written S_v[i] $\delta^{(=)^*}$ S_w[j]. In normal sequential loops, a backward direction would correspond to a dependence flowing backwards in time, and so cannot occur. In nested loops, a dependence relation has a direction for each loop, giving rise to a direction vector; in this case, an inner loop can have a backwards direction. Also, when we compute directions for adjacent loops to test for loop fusion, backwards directions can occur (see below). For dependence relations within a loop or nested loop, the outermost loop with a forward direction is said to *carry* that dependence relation. When we apply a parallel optimization to a particular loop, we ignore any dependence relation carried by an outer loop.

4. Enabling Parallel Optimizations

Since our premise is that parallel optimizations have the most potential benefit, we want the compiler to have the most flexibility in applying them possible. We do not consider exactly why a particular transformation *should* be performed; the decision as to whether or not to perform a transformation is a very important part of the optimization process, and will be considered in a subsequent section. Here we describe the parallel optimizations and show how they are affected by particular scalar optimizations.

Loop Distribution. DIS is legal as long as any cycle of dependence relations is not broken across the distributed loop(s); some reordering of the code may be necessary [BCK79]. For

instance, the following loop:

```
      for I = 1 to N do
S1:     A(I) = B(I) + 1
S2:     B(I+1) = B(I)*C(I) + D(I)
      endfor
```

has the dependence relations

$S_2\ \delta_{(<)}\ S_1$ $\qquad$ $S_2\ \delta_{(<)}\ S_2$

The loop can be distributed, as long as the statements are reordered; note the single-statement dependence cycle involving S_2:

```
      for I = 1 to N do
S2:     B(I+1) = B(I)*C(I) + D(I)
      endfor
      for I = 1 to N do
S1:     A(I) = B(I) + 1
      endfor
```

Loop distribution is in some sense the essence of automatic vectorization, and is sometimes explicitly implemented for partial loop vectorization.

Common subexpression elimination (CSE) can disable DIS. Take the loop:

```
      for I = 1 to N do
S1:     A(I) = A(I)*(B(I) + 1)
S2:     B(I+1) = B(I) + 1
      endfor
```

the subexpression `B(I)+1` is common to both statements. If it were pulled out and assigned to a temporary variable, as in:

```
      for I = 1 to N do
Sc:     T = B(I) + 1
S1:     A(I) = A(I)*T
S2:     B(I+1) = T
      endfor
```

then the loop can no longer be distributed. One solution is to "expand" the temporary variable T into an array; this makes sense in a vector environment, where the objects being manipulated are vectors anyway:

```
      for I = 1 to N do
Sc:     T(I) = B(I) + 1
S1:     A(I) = A(I)*T(I)
S2:     B(I+1) = T(I)
      endfor
```

In other environments, many of the common subexpressions are the array references anyway, so assigning these to temporary arrays does not address the problem, and increases the memory

bandwidth requirement. In fact, inserting redundant computation, through global forward substitution, can enable distribution. This is essentially the inverse of CSE.

Loop Fusion. FUS is legal when the loop limits match, and when there is no dependence relation with a (>) direction [Lov77, WoB87]. For instance, the only dependence relation in the program:

```
      for I = 1 to N do
S1:     A(I) = B(I) + 1
      endfor
      for I = 1 to N do
S2:     B(I+1) = C(I) + D(I)
      endfor
```

is $S_1\ \bar{\delta}_{(>)}\ S_2$; this means that there is a dependence $S_1[i]\ \bar{\delta}\ S_2[j]$ where $i > j$. Loop fusion, applied to this program, would "turn the dependence around", resulting in a flow dependence $S_2\ \delta_{(<)}\ S_1$. Opportunities for loop fusion are relatively rare in normal programs, but in some languages that treat arrays or vectors in a single statement, loop fusion will become an important optimization.

IF unswitching (IFU) [AlC72] can disable FUS. Take the program:

```
      for I = 1 to N do
S1:     A(I) = A(I)*(C(I) + 1)
      endfor
      for I = 1 to N do
S2:     if( X <> 0 ) then
S3:       B(I+1) = X*(B(I) + 1)
        endif
      endfor
```

As it stands, the loops can be fused; if IFU is applied to the second loop, however, the loops in the result program cannot be fused:

```
      for I = 1 to N do
S1:     A(I) = A(I)*(C(I) + 1)
      endfor
S2:   if( X <> 0 ) then
        for I = 1 to N do
S3:       B(I+1) = X*(B(I) + 1)
        endfor
      endif
```

Moreover, if we define IF Sinking as IFU^{-1}, then sinking IFs into loops can enable loop fusion.

IFU is related to general ICM, although it is applied to a control construct. Whitfield and Soffa [WhS90] claim ICM can disable FUS, by making the loops nonadjacent; if we assume the compiler works from a dependence structure rather than a lexical structure, ICM will not

disable FUS unless there is a dependence chain from the first loop through the invariant code to the second loop. Usually this will mean a (>) dependence direction in the original loop, except in the case where the dependence is from the first and only the first iteration of the first loop, as in:

```
     for I = 1 to N do
S1:    A(I) = A(I)*(C(I) + 1)
     endfor
     for I = 1 to N do
S2:    B(I+1) = (A(1)+1)*D(I)
     endfor
```

Here, the dependence relation is $S_1\ \delta_{(\leq)}\ S_2$, and the original loops can be fused; after ICM, the loops can no longer be made adjacent and so cannot be fused:

```
     for I = 1 to N do
S1:    A(I) = A(I)*(C(I) + 1)
     endfor
     T = A(1) + 1
     for I = 1 to N do
S2:    B(I+1) = T*D(I)
     endfor
```

Loop Interchanging. Simple INT is legal when the loops are tightly nested and there is no dependence relation with a (<,>) direction [AlK84]. Interchanging of non-tightly nested loops has also been studied elsewhere; it requires the loop limits to match (square or triangular) and has more complex dependence conditions, and so not be considered here. Loop interchanging is commonly used to expose additional parallelism in programs and to optimize the memory reference patterns of arrays in loops.

IFU can disable INX. In a program such as:

```
       for J = 1 to N do
         for I = 1 to M-1 do
S1:        A(I,J) = A(I,J)*(C(I,J) + 1)
S2:        if( X(I) <> 0 ) then
S3:          B(I+1,J) = X*(B(I,J) + 1)
           endif
         endfor
       endfor
```

the only dependence relation is $S_3\ \delta_{(=,<)}\ S_3$, which does not prevent interchanging. Applying IFU however produces the program:

```
     for J = 1 to N do
S2:    if( X(I) <> 0 ) then
         for I = 1 to M-1 do
S1:        A(I,J) = A(I,J)*(C(I,J) + 1)
S3:        B(I+1,J) = X*(B(I,J) + 1)
         endfor
       else
         for I = 1 to M-1 do
S1:        A(I,J) = A(I,J)*(C(I,J) + 1)
         endfor
       endif
     endfor
```

Since the loops are not tightly nested, they cannot be interchanged. Of course, if the original loops were interchanged, then IFU could not be applied.

ICM can also disable INX for exactly the same reasons as IFU. Applying the ICM^{-1} and IFU^{-1} can enable INX by making non-tightly nested loops be tightly nested.

Loop Parallelization. PAR is legal when a loop does not carry any dependence relation [ACK86]. RED applied in its simplest form can disable PAR by inserting dependence cycles for induction variables. In the program:

```
     for I = 1 to N do
S1:    J = 4*I
S2:    B(J) = C(I)
S3:    B(J+1) = D(I)
     endfor
```

there are no loop carried dependence relations. Standard compiler analysis would recognize J as an induction variable, and RED would replace it by:

```
     J = 0
     for I = 1 to N do
S1:    J = J + 4
S2:    B(J) = C(I)
S3:    B(J+1) = D(I)
     endfor
```

This translation inserts a dependence cycle for J, preventing effective parallelization. Of course, the same standard compiler analysis would recognize this variable J as an induction variable, and could do the inverse substitution, using RED^{-1}. This example may seem a bit trite, but it exposes the importance of recognizing the interaction of standard scalar optimizations and parallel optimizations.

5. Analysis vs. Synthesis

For the most part, compiler optimizations for high performance computers can be divided into two disjoint parts: the analysis phase and the synthesis phase. The analysis phase discovers facts about the program but does not necessarily perform any modifications or code improvements. Examples of the types of analysis necessary before optimization are conversion to Static Single Assignment (SSA) form [AWZ88, CFR89], induction variable recognition and data dependence analysis. Even constant propagation is simply recognition of what variable names have constant values at certain points in the program.

One obvious exception to this rule is dead code elimination. Eliminating dead code can only improve other optimizations, by removing any possible constraints that would prevent their application.

The synthesis phase uses the knowledge collected by the analysis to choose which transformations to perform and in which order. Even for scalar computers this process cannot be guaranteed to generate optimal code, due to the interaction of optimizations such as register assignment, instruction selection and scheduling. For parallel computers the problem is even harder. Let us return briefly to a previous example, where the optimizer must choose between interchanging (INX) and unswitching (IFU):

```
      for J = 1 to N do
        for I = 1 to M-1 do
S1:       A(I,J) = A(I,J)*(C(I,J) + 1)
S2:       if( X(I) <> 0 ) then
S3:         B(I+1,J) = X*(B(I,J) + 1)
          endif
        endfor
      endfor
```

For a conventional scalar computer, the obvious choice is to perform IFU; this moves the conditional test outside of the innermost loop. If the arrays are declared of size M×N (assuming normal row-major array storage), the stride of the references to the arrays in the inner loop will be N (the distance in memory words between successive accesses to the same array); if N is very large (larger than the page size), this could cause virtual memory thrashing [AKL81]. However, most compilers proceed without considering this effect.

For a vector computer, the choice is no longer so obvious. The inner loop here cannot be vectorized, due to the dependence self-cycle $S_3\ \delta_{(=,<)}\ S_3$. Also, vector computers are much more sensitive to the stride of memory references. Some strides (typically multiples of powers of two) generate poor interleaved memory performance, while a stride of one is always safe (and sometimes optimal) since it can take advantage of long cache lines. Applying IFU generates two

inner loops, one of which can be fully vectorized and the other of which can only be partially vectorized; both of these loops have large memory strides. Applying INX generates one vectorizable inner loop, with stride-1 memory references, but with a conditional. A third option is to first apply DIS:

```
        for J = 1 to N do
          for I = 1 to M-1 do
S1:         A(I,J) = A(I,J)*(C(I,J) + 1)
          endfor
        endfor
        for J = 1 to N do
          for I = 1 to M-1 do
S2:         if( X(I) <> 0 ) then
S3:           B(I+1,J) = X*(B(I,J) + 1)
            endif
          endfor
        endfor
```

then to optimize each statement individually, using IFU or INX as appropriate; this could be important if the array references in the first statement were transposed, for instance.

There are several ways for an optimizer to choose which parallel transformation to perform. The first method chooses an ordering for the transformations; this is essentially the method that the Parafrase translator uses, allowing the user to choose the ordering of transformations [KKP81,KSC84]. This is also the method suggested by [WhS90]. When applying a transformation, the optimizer only needs to decide whether this transformation will (1) improve the code or (2) enable (or disable) some subsequent optimization from improving the code.

The second method performs an exhaustive search of all possible transformations and combinations of transformations of the program. In order to be reasonable, the catalogue of transformations which are searched this way must be relatively small. For instance, using a catalog of one transformation, it might be feasible to compare all possible loop orderings from loop interchanging. In certain applications, exhaustive search may be reasonable, such as when writing a subroutine library.

The third method iteratively chooses which transformation to perform next, based upon a model of the target machine and knowledge of what effects each transformation will have upon the program. This method has all the potential of the exhaustive search without the cost, and allows transformations to be applied in different orders to different programs. The problem is that this method requires an oracle to decide which transformations to perform in which order on each program. In many cases the decision is obvious, but it is not clear how detailed the machine model must be in order to always generate optimal or reasonable code.

We can compare these three methods by looking at the skeleton program:

```
        for I = ...
          for J = ...
S1:         A(I,J) ...
          endfor
          for K = ...
            for L = ...
S2:           B(I,K,L) ...
            endfor
          endfor
        endfor
```

Suppose that for some target machine, it is optimal to perform the following transformations: (INX(L,K)->FUS(J,L)->INX(J,I)), generating the program:

```
        for J = ...
          for I = ...
S1:         A(I,J) ...
            for K = ...
S2:           B(I,K,J) ...
            endfor
          endfor
        endfor
```

The first method, in which the transformation ordering is bound at compiler-generation time, cannot generate this program. Actually, the ordering suggested in [WhS90] already has one cycle (INX->ICM->INX, potentially allowing for more), but the addition of cycles quickly begins to look like our third method. The second method would generate (or attempt to generate) many different versions of the program, including (FUS(J,K)->INX(J,I)), (DIS(I)->INX(I,J)->INX(I,K)->INX(I,L)), and so on. Each of the versions, including all intermediate versions, would be evaluated, and the best version would be used as the final generated code. The third method, using a smart oracle for choosing the transformations, would perform the three transformations in the order required, without attempting any other optimizations.

The problem with any compile-time evaluation technique is that many of the parameters are not known at compile time. Factors such as relative loop limits, presence or absence of a data dependence relation, or even memory strides may all be unknown. Some current compilers already generate multiple versions of a loop or program and choose which version to execute at run time depending on actual parameters [BDH87].

6. Other Work

The most relevant other work is described in the paper [WhS90], which used a different set of optimizations. They studied the scalar transformations constant propagation and dead code elimination as well as ICM. As mentioned, we feel that constant propagation belongs more to the analysis phase of the compiler than the synthesis phase. We agree that dead code elimination should precede other transformations; though we expect its application to user code to be minimal, in this sense we agree that there are some transformations that can be ordered. A compiler might also perform dead code elimination after other scalar optimizations.

The parallel transformations they studied were strip mining and loop peeling, as well as INX and FUS. Strip mining [Lov77] has two uses for parallel compilers: it can divide a loop into fixed-sized chunks, as for vector register machines, and the outer strip and inner element loops can be interchanged with other loops, to change memory reference locality and improve memory hierarchy utilization [AKL81, GJG88]. Simple strip mining is more of a code generation issue than an optimization. When strip mining is combined with interchanging, the iteration space is effectively divided into "supernodes" [IrT88]; this effect depends more on the interchanging than the strip mining, so we have chosen to focus on interchanging here.

Loop peeling (peeling off the first or last one iteration of a loop) can be used to break a data dependence chain that involves only a single iteration. Loop peeling is really just a restricted application of general index set splitting [Ban79], which can be used for many reasons. To properly optimize index set splitting, the split point must be computed, such as the desired loop limits (e.g. for loop fusion) or the cross over point of some data dependence condition.

In this paper, we chose to inspect scalar transformations that have interesting inverses. Forward substitution can be the inverse of CSE, while code sinking (into loops) can be the inverse of IFU and ICM. In the special case of induction variables, RED also has a simple inverse. The point we wanted to make is that the potential benefit from parallel optimizations can outweigh the potential benefit from scalar optimizations to the point to where it can be necessary to explicitly invert some optimizations, or to perform scalar pessimizations, in order to generate overall optimal parallel code. As it turned out, the parallel optimizations we chose have simple inverses also.

There has been a great deal of work on compiler optimizations for parallel computers. We chose a small set of four transformations to illustrate the problems that can arise when trying to interleave parallel and scalar optimizations. Other transformations that are also relevant are loop skewing [Wol86], loop collapsing and loop coalescing [Pol87], loop reversal, index set

splitting, loop unrolling [DoH79], scalarization, alignment and replication [ACK87] and so on. Note that the goal here is not so much automatic detection of parallelism from sequential programs, but automatic generation of optimal (or close to optimal) code from a machine-independent sequential or parallel program.

7. Summary

Many machine independent scalar optimizations are possible because many conventional scalar computers and programming languages share a common "von-Neumann" model of computation. Common subexpression elimination and code floating are generally beneficial regardless of the target machine architecture. In this framework, it is feasible to build a retargetable optimizing compiler, where only a few back-end machine-specific optimizations and the final code generator need to be rewritten for a new machine.

In the realm of high performance and parallel computers, this is a much more difficult task. The gap between the programming language model of computation and the machine instruction set is somewhat larger, and efficient execution of machine-independent programs depends more heavily on compiler optimizations. The tradeoff between different optimizations (in particular, between scalar and parallel optimizations) becomes important. It is still feasible to have a retargetable optimizing compiler framework. The analysis phase of the compiler as well as many of the particular optimizations would be shared, although many transformations would have to be parameterized so they could be customized to each application; this is no harder than parameterizing a register allocation routine for different numbers of registers. We briefly described a framework in which the parallel optimizations can be controlled by an oracle based on a model of the machine and the characteristics of the particular program being compiled. After parallel optimization is complete, the program can be treated as a set of sequential tasks to which normal scalar optimization can be applied.

We wish to emphasize two key points. First, it is difficult and perhaps unreasonable to treat parallel and scalar optimizations in the same framework, since they deal with different machine abstractions. Second, the order in which parallel optimizations should be applied to a program depends upon both the target machine architecture and the particular program being compiled; an attempt to reduce this complexity by finding a single fixed optimization ordering must sacrifice some execution performance.

References

[AKL81] W. A. Abu-Sufah, D. J. Kuck and D. H. Lawrie, On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations, *IEEE Trans. on Computers C-30*, 5 (May 1981), 341-356.

[AlC72] F. E. Allen and J. Cocke, A Catalogue of Optimizing Transformations, in *Design and Optimization of Compilers*, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972, 1-30.

[AlK84] J. R. Allen and K. Kennedy, Automatic Loop Interchange, in *Proc. of the SIGPLAN 84 Symposium on Compiler Construction*, New York, June 1984, 233-246.

[ACK86] R. Allen, D. Callahan and K. Kennedy, , Automatic Decomposition of Scientific Programs for Parallel Execution, Rice Univ., Nov. 5, 1986.

[ACK87] R. Allen, D. Callahan and K. Kennedy, Automatic Decomposition of Scientific Programs for Parallel Execution, in *Conf. Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, ACM Press, New York, 1987, 63-76.

[AWZ88] B. Alpern, M. N. Wegman and F. K. Zadeck, Detecting Equality of Variables in Programs, in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, January 1988, 1-11.

[BCK79] U. Banerjee, S. Chen, D. J. Kuck and R. A. Towle, Time and Parallel Processor Bounds for Fortran-Like Loops, *IEEE Trans. on Computers C-28*, 9 (September 1979), 660-670.

[Ban79] U. Banerjee, *Speedup of Ordinary Programs*, PhD Thesis, Univ. of Illinois, October 1979. (UMI 80-08967).

[BDH87] M. Byler, J. Davies, C. Huson, B. Leasure and M. Wolfe, Multiple Version Loops, in *Proc. of the 1987 International Conf. on Parallel Processing*, S. K. Sahni (ed.), Penn State Press, University Park, PA, 1987, 312-318. August 17-21, 1987.

[CaF89] R. Cartwright and M. Felleisen, The Semantics of Program Dependence, in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, ACM, New York, June 1989, 13-27. June 21-23 1989, Portland OR.

[CFR89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and K. Zadeck, An Efficient Method of Computing Static Single Assignment Form, in *Conf. Record of the 16th Annual ACM Symp. on Principles of Programming Languages*, ACM Press, New York, January 11-13 1989, 25-35.

[DoH79] J. J. Dongarra and A. R. Hinds, Unrolling Loops in Fortran, *Software Practice and Experience 9*(1979), 219-229.

[FOW87] J. Ferrante, K. J. Ottenstein and J. D. Warren, The Program Dependence Graph and its use in Optimization, *ACM Trans. on Programming Languages and Systems 9*, 3 (July 1987), 319-349.

[GJG88] D. Gannon, W. Jalby and K. Gallivan, Strategies for Cache and Local Memory Management by Global Program Transformation, *J. Parallel and Distributed Computing 5*, 5 (October 1988), 587-616, Academic Press.

[HPR88] S. Horwitz, J. Prins and T. Reps, On the Adequacy of Program Dependence Graphs for Representing Programs, in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, January 1988, 146-157.

[IrT88] F. Irigoin and R. Triolet, Supernode Partitioning, in *Conf. Record of the 15th Annual ACM Symp. on Principles of Programming Languages*, ACM Press, New York, 1988, 319-329.

[KKP81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe, Dependence Graphs and Compiler Optimizations, in *Conf. Record of the 8th ACM Symp. on the Principles of Programming Languages*, Williamsburg, VA, 1981, 207-218.

[KSC84] D. J. Kuck, A. H. Sameh, R. Cytron, A. V. Veidenbaum, C. D. Polychronopoulos, G. Lee, T. McDaniel, B. R. Leasure, C. Beckman, J. R. B. Davies and C. P. Kruskal, The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance, in *Proc. of the 1984 International Conference on Parallel*

Processing, R. M. Keller (ed.), IEEE Computer Society Press, Los Angeles, August 21-24, 1984, 129-138.

[Lov77] D. Loveman, Program Improvement by Source-to-Source Transformation, *J. of the ACM 20*, 1 (January 1977), 121-145.

[PaW86] D. A. Padua and M. Wolfe, Advanced Compiler Optimizations for Supercomputers, *Comm. of the ACM 29*, 12 (December 1986), 1184-1201.

[Pol87] C. D. Polychronopoulos, Loop Coalescing: A Compiler Transformation for Parallel Machines, in *Proc. of the 1987 International Conf. on Parallel Processing*, S. K. Sahni (ed.), Penn State Press, University Park, PA, 1987, 235-242. August 17-21, 1987.

[WhS90] D. Whitfield and M. L. Soffa, An Approach to Ordering Optimizing Transformations, in *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, March 1990, 137-146.

[Wol86] M. Wolfe, Loop Skewing: The Wavefront Method Revisited, *Intl J. Parallel Programming 15*, 4 (August 1986), 279-294.

[WoB87] M. Wolfe and U. Banerjee, Data Dependence and Its Application to Parallel Processing, *Intl Journal of Parallel Programming 16*, 2 (April 1987), 137-178.

This work was supported in part by NSF Grant CCR-8906909 and DARPA Grant MDA972-88-J-1004.