4

Performance, Portability and Implicit Parallelism

FRANÇOIS IRIGOIN

4.1 INTRODUCTION

Performance and portability, across sequential and parallel processors, cannot be economically achieved without tools, restructurers or compilers. These tools have been extensively studied for more than ten years, for parallel and sequential programming languages, and new techniques have been regularly added. For instance, complex loop tilings and array privatization were added to the program transformation list to cope with non-uniform memory architectures and to improve automatic parallelization. Compilation techniques have been developed to handle the exchange of data between processors in distributed memory machines. More interprocedural analyses have been implemented in prototypes and products.

Meanwhile, the complexity of sequential and parallel processors has relentlessly increased because integration technology makes it possible to add more and more mechanisms in a single-chip processor. These mechanisms, deep pipelines, multiple functional units, integrated caches,... increase the peak performance of processors. But peak performance is reached by fewer and fewer codes, because too many parameters are overlooked by programmers and not dealt with by compilers.

The PIPS prototype[1] is a source-to-source compiler which was designed to explore the impact of interprocedural analysis on the compilation and automatic parallelization of scientific programs. Its implementation was started in 1988. During the past five years, different sub-projects have been started to target different architectures,

[1] Paralléliseur Interprocédurale de Programmes Scientifiques. This project was funded by DRET. CNRS and ESPRIT provided additional funding.

Book title and editor name

with shared or distributed memory, but the main emphasis has been put on shared-memory multiprocessors like the traditional Cray machines because they are dominant in commercial and academic scientific computing.

The PIPS prototype is now mature enough to carry out experiments on medium size codes of a few thousand lines such as the PerfectClub benchmarks. The first results are presented here. They are simultaneously encouraging because old problems are solved and disturbing because the advances made do not produce improvement over commercial tools. There are two reasons to explain this state of affairs. First, each time a research project is started to explore new territories, unexpected problems crop up. This is a positive aspect of failure. Secondly, the implementation of a program restructurer like PIPS requires many pieces, which are all necessary to obtain experimental results but whose implementation is not always research work. It was not possible to obtain the same level of quality everywhere and the only remedies foreseen are greater collaboration with other teams and integration with other tools.

This chapter is organized in four sections. First, in Section 4.2, the costs involved in high performance computing are surveyed to show why automatic tools are necessary to reduce them, why these tools should cope with an old language, Fortran 77, and which problems these tools have to solve for the applications and architectures at hand. Then, innovative analyses and transformations developed in the PIPS project are presented in Sections 4.3 and 4.4. Finally, Section 4.5, is dedicated to experiments. Positive results are briefly presented but more space is left to the stumbling blocks that were encountered. They are of interest to users, who should know what should be avoided with present tools, and to researchers, who may find ways to eliminate them.

4.2 PERFORMANCE, PORTABILITY AND COST

Performance, either meaning execution of a large amount of computations in a limited period of time (absolute performance) or high utilization of all machine resources by one task (relative performance or efficiency), is hard to achieve and requires programming time and programmer expertise. Expertise is necessary to understand the characteristics of the machine and the application, and time is necessary to find the best ways to exploit them by transforming and specializing the code.

Portability across machines and operating systems also requires programming time and programmer expertise. It may be easier to achieve with scientific applications than with interactive software but it always requires code generalization. Portability is attained by ignoring the target architectures as much as possible.

Parallel processors make these two goals harder to achieve. On one hand, parallelism makes efficient performance harder to achieve than sequential performance because many more factors may slow down the computation. On the other hand, parallelism has not been really exploited long enough to design successful standards and for manufacturers to apply them.

This means that not many users can pay the associated cost and consequently that tools are welcome to reduce this cost. An analysis of the cost/benefit structure, of the application potential for automatic tuning and of the critical machine features show where and how automatic and semi-automatic tools can be useful.

4.2.1 Cost/Benefit Analysis

Industrial users have to evaluate how a performance improvement, in execution time or problem size, would affect their business. Academic users have a similar problem with their research results. Four kinds of answers are usually given to such questions:

- performance does not seem to matter very much but competitors are investing in parallel machines; let's do the same so as not to be outperformed if it works in the future;
- performance does not matter unless it can be improved by at least an order of magnitude; for instance, better results can be obtained with 3-D modeling instead of 2-D modeling;
- performance does not matter at all; workstations are fast enough; they may be slower than larger machines but results are returned as quickly because they are not shared;
- performance does matter because some speed improvement would make a piece of hardware useless.

Obviously, there is not much room for expensive machines and expensive programming practice. Which factors could improve the return and the return-on-investment time, assuming that faster runs, i.e. performance, can save money by reducing the wallclock time or the computer resource usage? Mainly the number of runs. It depends on the application generality, which includes portability and functionality, and the application life expectancy, which introduces a new cost, maintenance cost.

High performance parallel machines would be used more often if:

1. programming time could be decreased by using parallel programming languages, by improving compilers, by using standard high-performance libraries and by exploiting parallel programming environments;
2. the life expectancy of applications could be increased by using standard languages and system calls;
3. the full generality of applications could be preserved by optimizations;
4. the maintenance cost was no higher for parallel applications than for sequential ones.

The case for new programming languages is not strong because many attempts during the past 20 years have not led to much improvement. Besides, a new language may not succeed and put the application life expectancy in jeopardy. Also new compilers for new languages usually do not generate efficient code. Functional languages, which were heralded as providing parallelism for free, are neither standard nor efficient, although some progress has been made in compilers [Can92]. New versions of imperative languages like Fortran 90 and High Performance Fortran are not yet supported by efficient compilers. Fortran 77 and C are thus likely to remain in use, even for new projects.

Users should be reminded that pointers in C make some common optimizations, e.g. loop invariant code motion, almost impossible to apply safely. The Fortran 77 standard also encourages interprocedural optimizations by prohibiting wild side effects between actual parameters. Finally execution times are easy to derive from Fortran 77 statements. Let's consider the simple assignment:

```
A = B
```

Its Fortran 77 semantics guarantee a small execution time, because `A` and `B` must be scalar variables. The Fortran 90 semantics allow array assignment and the execution time is unknown but potentially large. The HPF semantics allow array distribution and may induce a matrix transposition across the machine network.

Standard high-performance libraries are a possibility. They reduce development, optimization and maintenance costs, but they cannot fulfill every need.

Interactive tools can be used to assist programmers in transforming programs for performance but automatic tools, preprocessors and compilers have a better potential because:

- they reduce the programmer time needed since the tool can run unattended,
- they maintain the application generality since tools preserve program semantics,
- they lower maintenance cost since a unique version of the program can be used as a reference.

4.2.2 Classes of Applications

Scientific applications which cannot be run on standard machines always require a large amount of floating point computation. However, these elementary computations depend on each other in different ways. Broadly speaking, fine-grained parallelism corresponds to parallelism between elementary operations like a multiplication. Medium/large grained parallelism corresponds to parallelism between loop iterations. The exact granularity depends on the loop body: a few simple statements, another loop, a procedure call,... Large-grained parallelism is sometimes found in main programs but the number of parallel tasks is usually quite small.

However, large computation sets are not the only characteristics of high performance applications. Huge data sets and I/O transfers can also be major performance problems. Some applications spend more than half of their residency time performing I/Os, usually because intermediate results have to be stored on disk. But some applications, e.g. in weather forecasts, also require a lot of genuine I/Os, to read input data and to permanently save output data.

It is obvious that compute-bound, memory-bound and I/O-bound applications do not need the same kind of analyses and optimizations and that fine-grained and large-grained parallelism do not offer the same flexibility to program restructurers. It is also obvious that complex applications are made of more than one program, and that each program is made of many phases with different profiles.

4.2.3 Classes of Machines

Machines, like applications, challenge the compilers in different ways. Broadly speaking, five classes can be distinguished because different programming languages and compilation techniques were developed for each of them.

The first kind of machines are the pure SIMD machines like the CM-2 from Thinking Machines and the Maspar machines. They are very good for applications with a huge amount of fine-grained parallelism and excellent data locality. Special I/O subsystems provide very high bandwidths.

The second kind of machines includes the most recent machines like the CM-5 from Thinking Machines, the Paragon from Intel, the CS-2 from Meiko, the KSR-1 from KSR and the SP-1 from IBM. Future machines, like the Cray MPP0, belong to the same class. These machines have a physically distributed memory, which may be logically hidden from the programmer, at least as long as performance is not an issue. These machines do not have a very fast synchronization system and they are at their best exploiting medium- to large-grained parallelism. The total amount of memory may be very large. Fast I/O subsystems are neither always available nor standardized. Specific languages were developed for such machines and lead to High Performance Fortran. Theoretically, only one level of medium/large grained parallelism is needed but data transfers between memory are expensive and imply some memory locality if performance is at stake.

The third kind of machines are the vector multiprocessors like the main line of Cray and of Japanese manufacturers (NEC, Fujitsu,...). They require some medium/large grained parallelism to keep the processors busy, and some fine-grained parallelism for the vector functional units. Their global shared memory subsystem is usually so good that data locality is not an issue. For huge data sets, secondary memory may be used to provide very high bandwidth and to avoid non-genuine I/Os. However, two levels of parallelism are required to fully utilize the processors and the functional units. The I/O subsystem is very powerful.

The fourth kind of machines are multiprocessors with shared memory built with standard elementary processors like MIPS (SGI) and SPARC (SparcCenters from SMCC). Only one level of medium/large-grained parallelism is needed to use the machine. However the total amount of memory and the memory hierarchy may be bottlenecks for applications without enough data locality. Furthermore, advanced processors require some kind of very fine-grained parallelism.

The last kind of machines is the simplest one, the SISD workstations. They do not provide huge memories or fast disks but their processors are becoming more and more powerful. Attention must be paid to the memory hierarchy and, more and more, some low level parallelism is necessary to efficiently exploit their superscalar, superpipelined RISC processors.

The sources of inefficiency a compiler should try to address can be summarized as:

1. non-saturated resources:
   - functional units (vector or scalar)
   - pipeline slots (so called *pipeline bubbles*)
   - processors (load balancing)
   - co-processors (overlap of computations and communications, overlap of control activities and computations, overlap of I/Os and computations)
2. data movements, which are useless from a user point of view:
   - cache fill
   - TLB loads
   - communications between processors
   - pseudo-I/O to load from/store on disks
3. control activities, which also are useless from a user point of view:

- process activations/deletions
- process synchronizations
- critical sections

4. redundant operations:

- loop invariant code
- dead code

Different attempts have been made to address these issues at the language or architecture levels. For instance, the Parallel Computing Forum developed the specification of PCF Fortran to let the programmer minimize control overhead while the High Performance Fortran Forum focused on programmer control over data locality. At the hardware level, multithreaded architectures are considered to improve the average resource usage at the expense of data locality, and co-processors are added to unload communication and control activities from the computational processor.

Potential causes of inefficiency are listed in table 4.1. Plus signs are used to show how critical each factor is for each class of architecture. It is obvious that machines in Classes 4 and 5, built with the newest superpipelined superscalar microprocessors, are dependent on many more factors and in a more critical way than older machines. Recent machines in Class 2 have warning signs everywhere and are going to run non-optimized applications with a low or very low efficiency. They will not provide the relative stability users are accustomed to with machines in Class 3. However, an application running efficiently on a Class-2 machine is likely to run efficiently[2] on any machine from any class.

Table 4.1 Classes of Machines

Type		1	2	3	4	5
Examples		CM-200 MP-2	CM-5 CS-2 MPP-0 Paragon	C90	SparcCenter SGI	Alpha Pentium SPARC RS6000
parallelism	functional units	0	+	+	++	++
	pipeline slots	0	+	0	++	++
	processors	+++	++	+	+	0
	co-processors	0	+	0	0	0
locality	cache	0	++	0	++	+++
	TLB	0	+	0	+	++
	local memory	+++	+++	0	0	0
	data on disk	+	+	+	++	+++
control	process	0	+	+	+	0
	synchronization	0	++	+	+	0
	critical section	0	++	+	+	0
persistence	disk I/Os	+	+	+	+++	+++

[2] But not optimally. For instance, multidimensional arrays are efficient on distributed memory machines but may require linearization for optimal performance on a Cray machine.

4.2.4 Achieving Portability and Performance for Parallel Processors

An optimizing compiler has to define a schedule such that tasks of the same nature are executed in parallel (e.g. array operations) and that tasks of different natures are executed simultaneously (e.g. overlap of communications and computations) and that the amount of non-productive tasks (e.g. control or data movement) is minimum.

This scheduling problem requires information about the number of tasks, their execution times and their dependencies. Much research activity has been directed towards dependence analysis at the expense of the other two kind of analyses. The PIPS project [Iri91] was started in 1988 with a strong emphasis on interprocedural analyses to provide as much information as possible to the scheduler. These analyses are presented in Section 4.3.

Many program transformations have been defined to enhance the potential parallelism by removing some dependence arcs (e.g. scalar expansion), to adapt the task size to the machine resources (e.g. loop interchange for vector machines) and to reduce useless operations (e.g. tiling to reduce exchanges between cache and memory). It is very difficult to decide which program transformations are useful for a given piece of code and a given architecture. Most transformations simultaneously have positive and negative effects. For instance, scalar expansion increases the need for memory space and traffic. Loop interchange may degrade the cache hit ratio. Tiling may increase the control and address computation overheads.

The PIPS parallelizer is not as advanced in choosing program transformations as it is on the analysis side. Many classical transformations like scalar privatization, loop unrolling (full or partial), strip-mining, loop interchange, the hyperplane method, loop tiling and reduction detection have been implemented as well as communication generation for distributed memory machines. More transformations like array privatization [Fea88] or loop coalescing should be added soon. But, although some progress has been made to reduce the combinatorial explosion of transformation combinations by unifying as many transformations as possible [Sar92] [Wol92], there is no good algorithm to choose the right combination of transformations from quantitative descriptions of a piece of code and of a machine. The information gathered by PIPS analyses remains largely unused.

However, experiments have shown that simple partial evaluation, based on constant propagation, was a very useful transformation. There are no drawbacks. The code is usually streamlined. Expressions and control graphs are simplified. The execution time is reduced. The potential for optimizations is increased. Examples are given in Section 4.4.

4.3 ANALYSES

Many analyses are performed interprocedurally, on a module basis. Once the abstract syntax trees are available, memory read and write effects are first computed for each statement and for each module. These memory effects are used to derive relations between assignments and references[3] and an affine approximation of statement effects on the values of integer scalar variables. This approximation is called a *transformer*.

These transformers are then applied to the initial state of the program to derive pieces of information about the states reached before each statement is executed. Each piece is called a *precondition*. These preconditions are used to improve the accuracy of the dependence systems when the use-def chains are converted into a dependence graph. They are used to refine the crude atomic memory effects into regions, to reduce the unknowns in static execution estimations, to perform partial evaluation and dead-code elimination, to eliminate dependencies and to evaluate the amount of parallelism.

4.3.1 Symbolic Analysis

Many key variables are functions of the problem size and are unknown at compile time. Their values usually change from one run to the next according to the content of input data files. However, relations between these variables may be invariant from run to run. The symbolic analysis[4] implemented in PIPS [Iri92] was designed to capture these relations and to extend them to relations between values taken by variables at different program points. This analysis includes constant propagation and handles Fortran non-recursive procedure calls in a natural way. An affine framework is used to compute and propagate affine equalities and inequalities across a whole program. Summary information is preserved for each module to avoid multiple evaluations.

As explained above, this analysis is made up of two phases. First, statements and procedures are abstracted as affine relations between their input and output states. These relations are called *transformers*. For instance, the subroutine `absval`:

```
subroutine absval(n, m)
if(n.lt.0) then
   m = -n
else
   m = n
endif
end
```

is abstracted by the following transformer

```
t(m) {n<=m, 0<=m+n}
```

which means that `absval` defines its parameter `m` and uses `n`. The new value of `m` is greater than `n` and `-n`.

Then, these relations are used to propagate the program or the procedure initial state down the call graph. The initial program state may be partially defined because static variables may be initialized by `DATA` statements. The initial state of each procedure is the convex hull [Sch86] of all its call site states. This lets PIPS analyze each procedure only once, at the expense of some accuracy. Procedures can be cloned [Hal91] if more precision is useful.

[3] These relations are known as *use-def* chains. *Def-use* and *def-def* chains are also used.
[4] This analysis used to be called *semantics* analysis after [Hal78]. This was confusing since semantic analysis has another meaning in the compiler realm.

The relation attached to each statement s is called *precondition* because it is true for all the possible states reached by the program before executing s. Like transformers, preconditions are not predicates over a state but relations between the program initial state and the current state. The preconditions obtained for a small program calling `absval` are given below. Variable `n` is read from disk and its value has thus been modified but is unknown when `absval` is called. On return, `m` has also been modified but a relation between `n` and `m` is known. When `n` is assigned again, some information is preserved about `m`, namely that it is positive.

```
      program main

c p() {}
      read (unit=5,fmt=*) n
c p(n) {}
      call absval(n, m)
c p(m,n) {n<=m, 0<=n+m}
      n = 3
c p(m,n) {0<=2m, n==3}
      write (unit=6,fmt=*) m
      end
```

User information can be propagated over the whole program if it can be expressed in an affine form. The safest and cleanest way to add it is to add the equivalent tests in the main procedure when input parameters are known.

4.3.2 Static Execution Time Estimation

It is well known that a precise estimation of program execution time is not computable in general. Approximations have to be made to overcome this fundamental limitation, linked to the halting problem. However, many scientific algorithms have execution times which are polynomials of the problem size. For instance, the theoretical execution time of matrix product $A(m, n) = B(m, l) \times C(l, n)$ is polynomial on l, m and n.

The actual coefficients depend on the target machine and on the compiler used. The automatic estimator in PIPS [Ber90] [Zho92] assumes that elementary instructions have fixed execution times and that elementary constructs like DO loop headers and multidimensional array accesses also have fixed execution times[5]. It also is possible to count floating point operations by setting their execution times to 1 and the other ones to 0. Accurate predictions were obtained for SUN SPARC-1 and SPARC-2 workstations, and for different optimization levels as long as system calls and I/Os were avoided.

The practical execution time may be much more complex. Cache effects may introduce periodic perturbations which are not accounted for. We expect polynomials to be accurate enough for compilation purposes.

Approximations are necessary to cope with tests, DO loops with unknown ranges, WHILE loops and any construct that induces non-polynomial execution times, e.g. the

[5] These times decrease with higher compiler optimization levels. Different tables can be defined for the same target machine but for different compiler options.

loop nest of Fast Fourier Transform. They are obtained by adding pseudo-variables, which appear in the time expression but not in the program and which can have any value. These pseudo-variables may be non-polynomial functions of existing variables. Logarithms, for instance, are implicitly introduced when a FFT is analyzed.

Local variables must also be eliminated and replaced by pseudo-variables when interprocedural estimations are required. Time estimations are more useful when fewer extra-variables are introduced. Preconditions are used to evaluate as many expressions as possible and to replace local variables by parameters. For instance, subroutine `prod` once uses local variable `imm` as a loop bound:

```
      subroutine prod(im,jm,km,q,t,b)
      ...

      imm=im-1
      do 1 k=1,km
         do 2 j=1,jm
            do i=1,imm
               ...
```

But `im-1` may be substited for `im to obtain \verb`prod+'s complexity as a function of input parameters `im`, `jm`, `km`.

```
511*im.jm.km - 343*jm.km + 5*km + 4
```

The actual coefficients are irrelevant here since the code is not fully reproduced. A unit cost was assumed for every elementary instruction.

Symbolic estimations of the execution time could be used to schedule parallel loops on multiprocessors. Iterations may be allocated in different ways to keep the control overhead low or to improve the load imbalance. The adjoint-convolution example used in [Fly92] could be scheduled with varying chunk sizes since each parallel iteration i has a different execution time $14 * n^2 - 14 * i + 18$.

```
c                                      7*n^4 + 11*n^2 + 3 (summary)
      subroutine adjoint(a,b,c,n)
      real a(n*n), b(n*n), c(n*n)

c                                      7*n^4 + 11*n^2 + 3 (do)
      do i = 1, n*n
c                                      14*n^2 - 14*i + 18 (do)
         do k = i, n*n
c                                                      14 (stmt)
            a(i) = a(i)+x*b(k)*c(k-i)
         enddo
      enddo
      end
```

Different program versions can be quickly compared [Ber90]. Since reference `a(i)` does not depend on the loop index `k`, it can be scalarized and the execution time is reduced. This shows in the complexity estimation:

```
c                                        6*n^4 + 16*n^2 + 3 (summary)
      subroutine adjoint2(a,b,c,n)
```

The improvement would be greater if scalar loads and stores in inner loops were taken into account with a zero execution time because scalars can be register allocated, but a unit time is presently used for all instructions.

Since part of the inner expression is loop invariant, it might be interesting to precompute it if the memory space and the extra memory accesses are acceptable. Once again, with the simple model used, the estimated sequential execution time is reduced.

```
c                                       5*n^4 + 21*n^2 + 6 (summary)
      subroutine adjoint3(a,b,c,n)
```

Parallel execution times could also be estimated in the same framework. Experiments with a BBN TC2000 showed that a parallel loop execution time depends in a simple way on the number of processors, an activation time, an inter-iteration overhead and the loop body execution time [Zho93]. More experiments should be carried out before a model is implemented in PIPS.

Static detection of the most computationally expensive parts of an application would also be possible with such an analysis if statements were not labeled with their single instance execution time but with their cumulated execution time in a whole program.

4.3.3 Dependence testing

Testing dependence between two references is a three-step operation. First, information about these references must be gathered to build a system of equalities and inequalities, as accurately as possible. The minimum system contains equations made of subscript expressions for each array dimension. PIPS adds preconditions [Tri86] [Pol89] [Iri91] [Iri92] to restrict the possible memory states[6] and regions [Tri86] [Iri92] to refine references to many array elements, e.g. references in procedure calls.

Secondly, the resulting system is likely not to be affine. It also may explicitly refer to the memory state if some references are indirect as in `A(B(I))`. As explained in [Lic88], the feasability of such systems is undecidable. Each equality and inequality has to be affine or to be made affine before a general procedure can be used. In PIPS, non-affine information simply is discarded.

Finally, the set of dependencies between iterations must be defined in a finite way. The total independence, the empty set, is easy to represent. Sets of uniform, i.e. constant, dependencies always are finite. Sets of non-uniform dependencies have to be approximated in a more or less precise manner, depending on the kind of transformations that will be later applied to the code. Many approximations have been introduced but dependence direction vectors [Wol89], which preserve only the signs of the dependence vector components, are the most common technique. But less precise approximation such as dependence levels [All83] [All87] and more precise ones such as dependence cones [Iri88] have been introduced. PIPS uses dependence levels for its parallelization algorithm and dependence cones for global loop nest transformations.

[6] As a side effect, unreachable DO loops are always parallel.

These set approximations are computed by eliminating program variables which are not relevant in the dependence space. The dependence system must be projected along the program variables onto the dependence space. With simple dependence approximations that have a finite set of dependence sets (empty/non-empty, levels and dependence direction vectors), multiple feasability tests can be employed instead of projection algorithms.

This third step, projection or feasability, is often seen as the whole dependence test. Numerous algorithms have been published [Ban88], with different trade-offs between accuracy and speed. The dependence test used in PIPS is a multi-stage projection algorithm[7], which uses Fourier-Motzkin pair-wise elimination only to deal with inequalities. It is theoretically not as accurate as the Omega test [Pug92] but its practical polynomial execution time and space and its practical accuracy were found satisfactory [Yan93].

4.4 TRANSFORMATIONS

Program transformations for code optimization are numerous and most of them are now described in books [Wol89] [Zim90] [Ban93]. For isolated loops, the number of iterations is used to decide if loop unrolling, strip mining or parallelization is best. No dynamic test is generated when loop bounds are unknown.

For loop nests, many transformations, like generalized loop interchange, the hyperplane method and tiling [Iri88], have been implemented in PIPS but they are not applied in a satisfactory way. An affine quality function, depending on loop ranges and spatial and temporal localities, is used to select, when possible, a parallel and a vector loops. However, some key transformations, e.g. loop coalescing, are not taken into consideration, and the weights associated to each parameter are difficult to set for a given architecture. A more complex selection procedure should be used as in [McK92].

Besides loop parallelization and optimization, two unexpected techniques turned out to be very useful on real applications. The first transformation is partial evaluation and the second one is code specialization.

4.4.1 Partial Evaluation

Partial evaluation is a well-known code optimization for functional and object-oriented languages. It is also used by compilers for imperative languages to simplify expressions.

For instance, function `sdot` is used many times by an ONERA application to perform matrix computation because modularity was strictly enforced during program coding. Function `sdot` is as general as possible and no optimizations were performed by hand.

```
function sdot(n,x,incx,y,incy)
real*4 x(1),y(1),sdot
sdot=0.0
ix=1
iy=1
```

[7] This algorithm is very different from [Tri86] because its projecting ability is directly used. In [Tri86], Fourier-Motzkin was used to prove feasability for every possible dependence direction vector.

```
      do 10 i=1,n
         sdot=sdot+x(ix)*y(iy)
         ix=ix+incx
         iy=iy+incy
 10   continue
      end
```

However, its interprocedural precondition reveals that all call sites have the same constants as actual arguments for `n`, `incx` and `incy`.

```
p() {49==n, 1==incy, 1==incx}
```

Function `sdot` is thus partially evaluated with these constants and its intraprocedural preconditions are computed again. The induction variables `ix` and `iy` are recognized and they could be eliminated to produce an optimized version of `sdot`.

```
      function sdot(n,x,incx,y,incy)
      real*4 x(1),y(1),sdot

      sdot = 0.0
      ix = 1
      iy = 1
      do i = 1, 49
c                       p(i,ix,iy) {1<=i, i<=49, iy==i, ix==i}
         sdot = sdot+x(ix)*y(iy)
         ix = ix+1
         iy = iy+1
      enddo
      end
```

Partial evaluation is useful to increase the number of constant loop bounds, to transform non-affine expressions into affine ones by substituting symbolic constant coefficients, and to make dead code highly visible.

4.4.2 Code Specialization

Procedure cloning [Coo83] is a technique used to improve optimization opportunities without encurring as much increase in code size and complexity as with procedure expansion. It can be automatically applied for a large class of information gathered at call sites [Hal91].

For instance, procedure `makeprf` in astrophysics program `dyna`[8] has only one explicit parameter, `mode`, which can take two values, 0 and 1. It is an obvious candidate for cloning. The following precondition is obtained for clone 1:

```
p() {1==kconvec, 0==keye, 2==keyp, 0==keynuc, 0==nivun, 1==kcontin,
     1==mode}
```

[8] Many procedures were modified by hand to partially replace `GO TO` by structured tests. They also were automatically modified by `stf`, the Toolpack [Pol86] control restructurer.

This module precondition is transformed into a much more complex precondition when the body of the key loop is reached:

```
p(entry,goto46,i,j,jj,jpoint,k,keybum,keylum,keytau,kkey,kod,kstate,
  kviscos,nigzar,np,np2,npp1)
      {2<=j, j<=npp1, 1==kviscos, 1==kstate, 1==keylum, 1==entry,
       0==entry#init, 1==mode, 1==kcontin, 0==nivun, 0==keynuc,
       2==keyp, 0==keye, 1==kconvec, 1==keytau}
```

Applying partial evaluation of integer expressions to the loop body reduces the number of tests from 8 to 1. After dead code elimination, the body of the key loop is no longer complex and loop peeling eliminates the remaining test on integer control parameters.

The same kind of simplifications were requested by users who wanted to study large general applications under some limiting constraints like `mshext==0` or `nsoup!=0`. It also happens without any cloning or request with PerfectClub benchmark `ocean`. Many integer control parameters are assigned constant values in initialization routine `datast` and dead code consequently appears in the main program.

When frequent recompilations are allowed, it would be worthwhile to move key problem parameters from input data file into include files.

Also, a simple user directive could be sufficient to clone a procedure for such and such parameter values and to add tests around call sites to take advantage of the cloned versions. For instance a general convolution routine could have a specialized and optimized version for 3×3 kernels at no programming cost.

4.5 CASE STUDY

Successive versions of PIPS have been used on real applications since 1991. The first release of PIPS, in 1991, was tested with benchmarks from ONERA, the French Office for Research in Aeronautics. For the second one, in 1993, new benchmarks from CEA, the French Atomic Energy Research Center, were added. In the meantime, we also used the PerfectClub benchmarks and a few industrial and academic codes. Interestingly enough, these later codes had characteristics not found in the PerfectClub as far as data structures and I/Os are concerned. Unfortunately, they usually cannot be obtained without a non-disclosure agreement.

First, summary results are presented for the French benchmarks and for the PerfectClub benchmarks. Then small cases which were found in full applications and which tripped up PIPS are studied in detail to explore new research directions.

4.5.1 French Benchmarks

PIPS outputs were compared with Cray FPP listings to see how an advanced research tool could compare to a commercial tool. In spite of its interprocedural analyses, PIPS proved unable to find as many parallel loops as FPP and proved worse at exploiting them. Many reasons explain this shortcoming. First of all, interprocedural parallelism, which was a potential strong point of PIPS, cannot be exploited without array expansion or privatization [Iri91]. The computation of must/may regions was shown to be an interesting approach to solve this problem [Iri92], but it has not yet been implemented.

Secondly, PIPS exploits the affine framework as much as possible but it does not use other techniques like pattern-matching to handle non-affine cases that appear when arrays are linearized.

Thirdly, PIPS does not use loop coalescing and array linearization while FPP is able to do both at the same time when there is an opportunity. This does not increase parallelism but improves its usefulness.

Fourthly, PIPS does not recognized a large set of recurrences. The recurrence detection algorithm is based on abstract interpretation and pattern-matching [Jou89] and is implemented in CommonLisp which limits its applicability to very small programs. This implementation demonstrates our ability to exchange data structures between C, which is used for most parts of PIPS, and CommonLisp, but does not produce any experimental results.

Finally, PIPS does not manage to use the information gathered about loop ranges to make smart choices when loop nests can be restructured. Some progress has been made elsewhere [Kob89] [Sar92] [Wol92] [McK92] but this is still an open problem.

These shortcomings have to be compared to PIPS goals, which were mainly interprocedural analyses. The parallelization process is made of many steps and a commercial product like FPP put the same emphasis on each of them since the global result is not better than the weakest step. From this point of view, PIPS obtains results close to those of a commercial product and is quite remarquable as a research prototype. Better results will be hard to obtain in an academic environment, unless different teams manage to share their best components.

4.5.2 PerfectClub Benchmark

The PerfectClub benchmark [Ber89] is a set of programs which has been contributed and selected by a committee representing companies and universities. These programs contain the main algorithms used in scientific computing. They are used to evaluate machines and compilers on real applications and have been extensively studied [Eig91] [Pet92] [Pet93].

The PerfectClub benchmarks were used to evaluate PIPS dependence tests [Yan93]. The comparisons were made at the dependence arc level and not at the loop level. Many measurements were made to understand why the Fourier-Motzkin pair-wise elimination had a practical polynomial behavior and if it were accurate enough. Other measurements were made to see if loop bounds and *intra-procedural* preconditions were a decisive advantage. On average, loop bounds increase the number of independent arcs found by a mere 3.5 % (between 0 to 170 %) and preconditions suppress 3 % (between 0 and 26 %) more arcs. These average values are disappointing but they do not mean much. For some programs loop bounds or preconditions are necessary to reach good results[9]. For other ones, they have no impact.

Preliminary results with *inter-procedural* preconditions and regions are much more impressive but experiments have unfortunately not yet been completed. Due to PIPS implementation, results are much improved if a control restructurer is run first. Spe-

[9] Percentages of up to 300 % have been observed in non-PerfectClub benchmarks

cific results include that inequality $10 \leq morb \leq 40$ which is necessary to effectively parallelize procedure `olda` in `mdg` [Eig91] is automatically and interprocedurally found. Also, preconditions make forward substitution useless to parallelize loop 100 in procedure `filtr4` of `track`.

The PerfectClub benchmarks were also used to calibrate PIPS with respect to other research projects: DELTA [Pad89], PFC [Gof91], PTRAN [Sar91] and SUIF [May92]. Figures were initially hard to relate because each group measures different things. PTRAN measurements were at the loop level whereas DELTA used arcs labeled with dependence direction vectors. Numbers of independent arcs could be obtained with PFC, PIPS and SUIF. However subtle differences between systems make useful comparisons extremely difficult, even for elementary numbers like the number of array reference pairs that are tested. This first attempt at comparing systems was mainly successful in pinpointing bugs in PIPS. More interesting comparisons should be based on detailed comparisons of dependence graphs and not on summary numbers like the number of independent arcs found.

4.5.3 Fourier Transform

Each step of a Fourier transform uses a different grouping of array elements. These two dimensional enumerations, loop `m` and `i` in the example, are linearized to access a one-dimensional array `t`.

```
      mmax=2
2     if (nn.gt.mmax) then
        istep = 2 * mmax
        ...
        do m = 1, mmax, 2
          do i = m, nn, istep
            j =i + mmax
            ...       = ... t(j) ... t(j+1)
            ...       = ... t(j+1) ... t(j)
            t(j)   = t(i) ...
            t(j+1) = t(i+1) ...
            t(i)   = t(i) ...
            t(i+1) = t(i+1) ...
          enddo
            ...
        enddo
        mmax=istep
      go to 2
      endif
```

The independence between references `t(i)` and `t(j)` can be proven with the information carried by the precondition of the inner loop body, after the `j`'s definition:

```
p(i,istep,j,m)
    {1+mmax<=nn, 1<=m, m<=mmax, m<=i, i<=nn, j==mmax+i, istep==2mmax}
```

Let i and i' be the values of `i` for two different iterations, and dm and di the differences in iteration count for loops `m` and `i`. i' and i are linked by the equation:

$$i' = i + 2 \times dm + istep \times di$$

The assumed dependence between the two references implies that $j' = i$ and allows its rewriting as:

$$0 = mmax + 2 \times dm + istep \times di$$

since $j' = i' + mmax$. Preconditions are also used to substitute $istep$ and to bound $2 \times dm$ by $mmax - 1$:

$$0 = mmax + 2 \times dm + 2 \times mmax \times di \tag{4.1}$$

$$1 \leq mmax + 2 \times dm \leq 2 \times mmax - 1 \tag{4.2}$$

The division of equation 4.1 by $2 \times mmax$ gives equation 4.3 as quotient, and equation 4.4 as remainder:

$$0 = di \tag{4.3}$$

$$0 = mmax + 2 \times dm \tag{4.4}$$

The remainders of the left and right handsides can be identified because of constraint 4.2 which also makes equation 4.4 not feasible. This proves that there are no dependencies between references `t(i)` and `t(j)`, even within an iteration.

This shows that the dependence test cannot be purely affine, although affine preconditions carry all information necessary to prove independence. However, the technique used in this example can be automatized to rewrite affine equations with symbolic coefficients as two affine equations with numerical coefficients, when one of the symbolic coefficients can be used to divide the initial equation and when inequalities between this coefficient and parts of the equation show that the quotients and the remainders of the left and right hand side can be identified.

4.5.4 Array Linearization

To obtain long vectors needed to utilize vector processors without cache, multi-dimensional arrays often are linearized. However, the linearization is not necessarily exploited in all loops and linearized addresses may be computed as in this piece of code:

```
do k=1,nkp
   do j=1,njp
      do i=1,nip
         L000 = (k-1)*nipnjp + (j-1)*nip + i
         L100 = L000 + 1
         L110 = L100 + nip
         L010 = L000 + nip
         L001 = L000 + nipnjp
         ...
```

The `Lnnn` variables are used to access each array element and some of its physical neighbors. Independence cannot be proven unless the non-affine assignment to `l000` is handled and the non-affine relationship between `nip`, `njp` and `nipnjp` ($nip \times njp = nipnjp$) is retained across procedure boundaries. PIPS affine framework fails on both accounts.

Portability might be obtained by avoiding explicit linearization and letting the compiler make the transformation. However, boundary conditions may have to be checked and new arrays like `l_to_i` have to be added.

```
do lijk = 1, nipnjpnkp
   i = l_to_i(lijk)
   j = l_to_j(lijk)
   k = l_to_k(lijk)
   ...
```

Their declarations may tremendously increase the memory requirements since they have the same dimensions as the problem. Their use in the loop body increases each iteration execution time. Their initializations have to be amortized not only over a large number of iterations but also over a large number of loops since they are as large as other arrays.

Additional mask arrays may have to be introduced when the physical space is not exactly defined as a product of intervals. Some specific geometric conditions may be encoded in one- or two-dimensional arrays used as loop bounds. These arrays should be re-encoded as a three-dimensional mask array to nullify the loop body effects for out-of-bound iterations.

To summarize, on one hand, reverse-engineering linearized arrays and loops is beyond our current technology. Commercial tools can only retrieve the simplest ones by pattern-matching. On the other hand, linearization often implies adding auxiliary arrays, initializations of and references to these arrays. A cost/benefit analysis has to be performed at the program level and strongly depends on the target architecture and on each loop nest structure. This seems to be well beyond our tool capabilities, in terms of static execution time estimation and program transformation.

4.5.5 Privatization vs Expansion

Scalar privatization was chosen against expansion for PIPS because it has no negative byproduct. After privatization, memory use and memory traffic are not increased during a sequential execution and they are increased linearly with the number of processors during a parallel run. After expansion, more memory is used, regardless of the target machine nature and the number of processors.

However scalar expansion is potentially more powerful as can be seen in the next example. The `i` loop is sequential regardless of `f`'s privatization. However, `f` can be expanded or forward substituted. Then the loop can be distributed and each individual loop is parallel.

```
do i = 2, n
   f = ...
   t(i) = t(i) + f
```

```
   t(i-1) = t(i-1) + f
enddo
```

It is difficult to select the best solution. Array expansion increases memory traffic. In the initial code, scalar `f` can be stored in a register. In the parallel code, each element of array `f` is stored once and loaded twice.

```
doall i = 2, n
   f(i) = ...
enddo
doall i = 2, n
   t(i) = t(i) + f(i) + f(i-1)
enddo
```

Forward substitution, which might be difficult to apply when several lines are used to compute `f`, leads to a parallel loop but increases the computation time since `f` has to be evaluated twice as many times. Experiments [Eig91] have shown that forward substitution was damaging most of the time.

Other optimizations are possible. For instance, it might be advantageous on a RISC processor to keep the last two values of `f` in two registers and to software pipeline the loop.

```
f1 = 0.
do i = 2, n, 2
   f2 = ...
   t(i-1) = t(i-1) + f1 + f2
   f1 = ...
   t(i) = t(i) + f1 + f2
enddo
```

On a parallel processor, the best solution might be to introduce a critical section to update `t(i)` and `t(i-1)`. No extra memory space or computation is required but some control overhead is added for each iteration. This overhead may be reduced when the number of elementary processors is much smaller than the iteration number `n` and iterations are allocated to processors by strip-mining[10]. The first and last iterations may be peeled off the inner sequential loop and only two critical sections per processor are needed.

Many more versions of the initial loop can be derived by combining program transformations to exploit the architectural features of a given machine. Choosing a particular version in this large set according to machine parameters is hard, even for experienced programmers, although the initial code is very simple. This explains why more progress is necessary in selecting code transformations.

4.5.6 Induction variables

Annotations instead of program transformation are used in PIPS whenever possible to make outputs more user-friendly and to avoid negative effects of some transformations

[10] To keep the code as short as possible `P` is assumed to exactly divide `n-1`.

as long as they have not been proved beneficial. For instance, loops are not normalized and induction variables are not substituted by their closed forms before dependence testing. Their relationships are carried by the preconditions.

This works, as long as only one loop is involved. In

```
n = 0
do i = 1, nbc
   do j = 1, nbs
      n = n + 1
      v(n) = ...
   enddo
enddo
```

variable `n` is used to access a linearized array, `v`. The closed form of `n` is `nbs*(i-1)+j` which is generally not affine since `nbs` usually is a size parameter read from a file.

Assuming that such preconditions were computed, PIPS dependence test would have to be upgraded to handle them [Lic88]. Other techniques, based on pattern-matching or monotonicity, may be less robust but are easier to implement.

4.5.7 Overflow

PIPS implementation is also limited by its use of 32-bit integer to analyze programs using the same set of integers. In subroutine `lmult` from program `qcd` (PerfectClub benchmarks), constants are very close to the maximum positive value of $2^{31} - 1$. Furthermore, as can be seen in the excerpt below, the transformation of Fortran expressions into affine expressions introduces overflows. The conditional assignment of `k` becomes `k = k+2**31`. Smaller values, like `ipow16` and `ipow15`, quickly generate overflows when linear combinations of equalities and inequalities are computed. In the short term, the introduction of 64-bit machines is going to help but the precision problem should be explicitly dealt with in algorithms.

```
      subroutine lmult(i0,i1,j0,j1,k0,k1)
c
      parameter (ipow30=2**30,ipow16=2**16,ipow15=2**15)
c
      k = k0*j0
      if(k.lt.0) k = (k+ipow30)+ipow30
      ...
```

4.5.8 Future work

The evaluation of PIPS on real applications shows that, although a lot of progress has been made in program analysis and transformation, many problems remain open in automatic program restructuring. To capitalize on these advances without increasing the combinatorial aspect of parallelization [Eig91], it is tempting to recast the compilation process of imperative languages as a two-step problem. First the imperative program is transformed into a functional program dealing with values. Then values and computations are mapped on memory locations and processors, and tasks are scheduled. This promising approach has been shown to work on small pieces of code [Dar93] [Fea92].

A more evolutionary approach, favored by the PIPS team, requires smaller advances in different areas.

4.6 CONCLUSION

To keep high performance affordable by industry, its cost per floating point operation must go down. To simultaneously obtain performance, portability and maintenability for parallel processors, automatic tools are necessary. They should be able to convert the unique maintained version of a code into specialized versions for each target architecture.

The increase in computer architecture complexity has kept this ultimate goal as remote as it was 10 years ago in spite of many advances in tools. No magic bullet can be expected from ambitious research tools like PIPS. Efforts should be made in three areas to improve the present situation: user training, interaction between users and tools, and cooperation among research teams.

User training in architectures and compilers would improve the average code quality. Machine specific optimizations should be avoided. Input parameters should be tested. Memory allocation should be improved. Procedures should be restructured to isolate initialization phases from iteration phases, to break large subroutines and to inline small ones. This work has an immediate payoff for maintenance and it makes future parallelizations or optimizations easier and safer for programmers as well as tools. This work can benefit from interactive programming environments, as long as algorithmic changes are not necessary.

Code optimization cannot be dealt with by users or by tools only. Some cooperation is necessary but fully interactive tools are too wasteful of user time. Semi-automatic ones are necessary to let the user forget about details that are properly dealt with by tools. These tools should be able to use global information about the program and to remember transformations applied by users so as to decouple optimization and maintenance. Directives have been used for years by the industry. The current directives could be upgraded and partly standardized to let users specify conditional program transformations, e.g. tiling, inlining or unrolling, without producing unmaintainable codes.

Finally, research projects would provide more experimental results at a lesser cost if some cooperation were possible at a relatively low cost[11]. These results would also be more reliable because comparisons could be made. In spite of years of pressure applied by French funding agencies, PIPS does not share intermediate data structures with any other project, French or not. Research is too dynamic and research prototypes too unreliable to be so closely coupled. However, ASCII files can easily be exchanged and, to some extent, checked by human beings. The Parascope and PFC projects at Rice University have exchanged dependence graphs via such files [Cas91]. Most results obtained by PIPS are mappings from procedure names, statements or references to some kind of information. They could easily be exchanged as soon as a numbering

[11] And if experimental work was considered important by program committees!

scheme for statements and references is agreed on. Such a scheme has already been defined at Rice.

Acknowledgements

Examples were contributed by many individuals. Special thanks are due to Dominique Michaud who studied tool failures by parallelizing a whole industrial application by hand and who shared his experience with the PIPS team. Josef Stein showed us interesting points in parallelizing an astrophysical code, `dynadp`. Kathryn McKinley discovered problems in geophysical codes. Corinne Ancourt, helped by Benoit de Dinechin, compared FPP and PIPS results on CEA and ONERA benchmarks.

PIPS development involved many people. Corinne Ancourt developed and maintains the basic linear algebra routines. Bruno Baron implemented the partial evaluator. Pierre Berthomier and Lei Zhou designed and developed the static execution time estimator. Pierre Jouvelot maintains NewGen, the software engineering tool used to manage PIPS data structures. He also provided valuable feedback on earlier drafts of the chapter. Alexis Platonoff implemented the region computation. Yi-Qing Yang was responsible for the dependence test.

REFERENCES

[All83] John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformations.* PhD dissertation, Rice University, Department of Computer Science, April 1983.

[All87] Randy Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic Publisher, Norwell, Massachusetts, 1993.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[Ber89] Michael Berry and al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. CSRD Report 827, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, Illinois, May 1989.

[Ber90] Pierre Berthomier. Static Comparison of Different Program Versions. CAII Report 130, Ecole des Mines de Paris, Centre de Recherche en Informatique, Fontainebleau, France, September 1990.

[Can92] David Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[Cas91] Joel Castellanos and Paul Havlak. PSERVE Dependence Graph File Formats. TR90-134, Rice University, Department of Computer Science, Houston, Texas. July 1991.

[Coo83] Keith D. Cooper. *Interprocedural Data Flow Analysis in a Programming Environment.* PhD dissertation, Rice University, Department of Computer Science, April 1983.

[Dar93] Alain Darte. *Techniques de parallélisation automatique de nids de boucle.* PhD dissertation, Ecole Normale Supérieure de Lyon, Laboratoire LIP-IMAG, April 1993.

[Eig91] Rudolf Eigenmann and William Blume. An Effectiveness Study of Parallelizing Compiler Techniques. In Dr. Herbert D. Schwetman, editor, *Proc. 1991 International Conference on Parallel Processing*, pages II-17–II-25, Saint-Charles, August 1991. The IEEE Computer Society, CRC Press, Boca Raton.

[Fea88] Paul Feautrier. Array Expansion. In *1988 International Conference on Supercomputing*, pages 429–441, Saint-Malo, July 1988. ACM, ACM Press, New York.

[Fea92] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[Fly92] Susan Flynn Hummel, Edith Schonberg and Lawrence E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, 1992.

[Gof91] Gina Goff, Ken Kennedy and Chau-Wen Tseng. Practical Dependence Testing. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Canada, June 1991. ACM, ACM Press, New York.

[Hal78] Nicolas Halbwachs and Patrick Cousot. Automatic Discovery of Linear Restraints Among Variables. In *Fifth ACM Annual Symposium on Principles of Programming Languages*, pages 84–97, Tuczon, Arizona, January 1978. ACM, ACM Press, New York.

[Hal91] Mary W. Hall. *Managing Interprocedural Opimization.* PhD dissertation, Rice University, Department of Computer Science, April 1991.

[Iri88] François Irigoin and Rémi Triolet. Supernode partitioning. In *Fifteenth Annual Symposium on Principles of Programming Languages*, pages 319–329, San Diego, January 1988. ACM, ACM Press, New York.

[Iri91] François Irigoin, Pierre Jouvelot and Rémi Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *1991 International Conference on Supercomputing*, pages 244-251, Cologne, June 1991. ACM, ACM Press, New York.

[Iri92] François Irigoin. Interprocedural Analyses for Programming Environment. In Jack J. Dongara and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 333–350, Saint-Hilaire-du-Touvet, September 1992. NSF-CNRS, North-Holland, Amsterdam.

[Jou89] Pierre Jouvelot and Babak Dehbonei. A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions. In *ACM 1989 International Conference on Supercomputing*, pages 186–194, Crete, Greece, June 1989. ACM, New York.

[Kob89] Brian D. Koblenz and William B. Noyce. Constraint Based Vectorization. In *ACM 1989 International Conference on Supercomputing*, pages 195–204, Crete, Greece, June 1989. ACM, New York.

[Lic88] André Lichnewsky and François Thomasset. Introducing Symbolic Problem Solving Techniques in the Dependence Testing Phases of a Vectorizer. In *1988 International Conference on Supercomputing*, pages 396–406, Saint-Malo, July 1988. ACM, ACM Press, New York.

[May92] Dror E. Maydan. *Accurate Analysis of Array References.* PhD dissertation, Stanford University, Department of Computer Science, September 1992.

[McK92] Kathryn McKinley. *Automatic and Interactive Parallelization.* PhD dissertation, Rice University, Department of Computer Science, April 1992.

[Pad89] David A. Padua. The Delta Program Manipulation System - Preliminary Design. CSRD Report 808, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, Illinois, June 1989.

[Pet92] Paul Petersen and David Padua. Machine-Independent Evaluation of Parallelizing Compilers. CSRD Report 1173, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, Illinois, January 1992.

[Pet93] Paul M. Petersen. *Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques.* PhD dissertation, University of Illinois at Urbana-

Champaign, Center for Supercomputing Research and Development, January 1993.

[Pol86] A. A. Pollicini, editor. *Proc. Using Toolpack Software Tools (Ispra-Course)*, Ispra, November 1986. The ISPRA Courses on Information Science, Kluwer Academic Publishers, Dordrecht.

[Pol89] Constantine D. Polychronopoulos and al. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In Emily C. Plachy and Peter M. Kogge, editors, *Proc. 1989 International Conference on Parallel Processing*, pages II-39–II-48, Saint-Charles, August 1989. Penn State University, The Pennsylvania State University Press, University Park.

[Pug92] William Pugh. A Pratical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[Sar91] Vievk Sarkar. *PTRAN – the IBM parallel translation system*, in Boleslaw K. Szymanski, editor, *Parallel Functional Programming Languages and Compilers*, pages 309–391. ACM Press, Addison-Wesley, Reading, Massachusetts, 1991.

[Sar92] Vivek Sarkar and Radhika Thekkath. A General Framework for Iteration-Reordering Loop Transformations. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 175–187, San Francisco, June 1992. ACM, ACM Press, New York.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.

[Tri86] Rémi Triolet, François Irigoin and Paul Feautrier. Direct Parallelization of CALL statements. In *ACM SIGPLAN'92 Symposium on Compiler Construction*, pages 176–185, Palo-Alto, July 1986. ACM, ACM Press, New York.

[Wol92] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD dissertation, Stanford University, Department of Computer Science, August 1992.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Pitman Publishing, Cambridge, Massachusetts, 1989.

[Yan93] Yi-Qing Yang. *Tests des dépendances et transformations de programme.* PhD dissertation, Ecole des Mines de Paris, Centre de Recherche en Informatique, in preparation, 1993.

[Zho92] Lei Zhou. Complexity Estimation in the PIPS Parallel Programming Environment. In Luc Bougé and al., editors, *Proc. Second Joint International Conference on Vector and Parallel Processing*, pages 845–846, Lyon, France, September 1992. Springer-Verlag, Berlin.

[Zho93] Lei Zhou. First Experience on Shared-Memory Multi-processor MIMD Machine BBN TC2000. CRI Report E176, Ecole des Mines de Paris, Centre de Recherche en Informatique, Fontainebleau, France, September 1993.

[Zim90] Hans Zima and Barbara Chapman. *Supercompilers for for Parallel and Vector Computers.* ACM Press, Addison-Wesley Publishing, Reading, Massachusetts, 1990.