Prototyping Fortran-90 Compilers for Massively Parallel Machines

Marina Chen and James Cowie

YALEU/DCS/RR-881
December 1991

Prototyping Fortran-90 Compilers for Massively Parallel Machines

Marina Chen James Cowie *

Department of Computer Science
Yale University

Abstract

Massively parallel architectures, and the languages used to program them, are among both the most difficult and the most rapidly-changing subjects for compilation. The field's complexity and mutability have created a demand for new compiler prototyping technologies which allow novel styles of compilation and optimization to be tested in a reasonable amount of time.

Using formal specification techniques, we have produced a data-parallel Fortran-90 compiler for Thinking Machines' Connection Machine/2. The prototype produces code from initial benchmarks demonstrating sustained performance superior to hand-coded *Lisp and competitive with Thinking Machines' CM Fortran compiler. This paper presents some new compiler specification techniques necessary to construct such a prototyping compiler for massively parallel machines that is both competitive in performance and easily retargetable. Issues involving the production of a Fortran 90 compiler prototype for the recently announced Connection Machine/5 are also discussed.

Corresponding Author
James Cowie (cowie-james@cs.yale.edu) (203) 432-1243
Yale University
Department of Computer Science
51 Prospect St.
New Haven, CT 06520

*Support for this work is provided by the Defense Advanced Research Project Agency, monitored by the Army Directorate of Contracting, under contract DABT 63-91-C-0031.

Contents

1 Introduction

Existing compilers for massively parallel machines are generally designed using traditional methods, combining generation from specification for a few subsets of the problem (typically the syntactic analysis) with hand-coded solutions for most others. One such compiler is Thinking Machines' CM Fortran [1], which targets the Connection Machine/2, a massively parallel, SIMD supercomputer. CMF was awarded a 1990 Gordon Bell Prize honorable mention for compiled code performance.

While such compilers unarguably achieve production quality, they are difficult and expensive to construct, maintain, and extend as a result of their problem-specific, highly-tuned, hand-coded nature. Complexity often precludes full generality, in the sense that a high-performance compiler must often make tradeoffs which allow some problems to run extremely well at the expense of others. The developers of TMC's convolution compiler [3], for example, had to start again at the bottom, expressing critical operations in microcode, because the sort of fine-grain processing users perform using stencils cannot be expressed within the narrowly drawn CMF machine model. For similar reasons, the CMF compiler in its current form cannot be used for developing scientific library functions for the CM/2; these critical routines must be developed by hand at great expense.

By contrast, traditional systems for generating compilers drew on the the flexibility and power of formal specification to build and maintain systems which were truly machine-independent. They remained unacceptable for production use, however, due to the poor performance of the code they produced.

The MESS system [5] introduced "separable semantic specification" as an alternative to denotational semantics approaches, using a two-level semantic description to effectively separate model detail from language-specific properties. Efficiently implementing the primitive semantic operators which make up the target model alleviated the performance problems associated with generated compilers. Such systems, however, were still limited in practice to sequential-execution, uniprocessor implementations. They lacked action operators to express data parallelism as well as additional model refinements to target parallel systems.

Other research has shown the importance of a unified intermediate representation in constructing optimizing compilers. The SUIF language of Tjiang, Wolf, *et al.* [7] successfully preserved high-level information for low-level transformations, and began to show how multiple levels of optimization concerns must be coordinated. SUIF was not used in the context of formal compiler specification, however. Pingali *et al* use abstract interpretation techniques in their intermediate representation [6], and use a true imperative store and explicit formulation of loops. They are primarily concerned, however, with dataflow analysis, and do not have an abstraction which unifies data and control parallelism, nor a treatment of back-end concerns.

In this paper we propose a methodology for compiler design which brings together lessons learned from previous compilers for massively parallel machines with techniques from formal specification. Using a semantic algebra with operators for concurrency and a shape abstraction for computation over fields of data, we show how to model such machines in a working compiler specification. We also map the structure of such a specified compiler, using Thinking Machines' CM/2 and the recently announced CM/5 as sample targets.

Finally, we present initial results from our Fortran-90-Y prototype compiler for the CM/2. After six months of development, the prototype implementation generates code exhibiting sustained performance superior to that of hand-coded *Lisp and competitive with Thinking Machines' CM Fortran compiler. We argue that competitive performance, coupled with rapid prototyping and target flexibility, have become persuasive arguments for developing compilers for massively parallel machines using formal specification techniques.

2 Problem Overview

To clarify our motivations, we introduce brief overviews of the data-parallel features of Fortran 90 and the slicewise programming model for the Connection Machine/2. We then break down the problems to be solved in a prototype compiler for that machine, and describe the overall prototyping and optimization strategies which need to be developed as a result.

2.1 Source Language: Data Parallelism in Fortran 90

The Fortran 90 language [2] contains many extensions to the old Fortran 77 standard which are of interest for data parallelism. Specifically, the ability to refer to and compute directly with whole arrays or arbitrary subsections give the programmer the ability to express program parallelism which, under the old standard, would have been stated only implicitly or through the use of source-level annotation. The wide selection of intrinsic procedures for examining, reshaping, or performing reductions over Fortran 90 arrays serve to replace many common transformations which programmers were previously forced to approximate with serial syntax.

For example, the Fortran 77 code

```
       INTEGER K(128,64), L(128)
       DO 10 I=1,128
        L(I) = 6
        DO 20 J=1,64
          K(I,J) = 2 * K(I,J) + 5
20      CONTINUE
10     CONTINUE
```

could be replaced by the Fortran 90 assignments

```
       L = 6
       K = 2*K+5
```

Similarly, the loop

```
       DO 30 I=32,64
        L(I) = L(I+64)
        DO 40 J=1,64
         K(I,J) = K(I,J)**2
40      CONTINUE
30     CONTINUE
```

can be rewritten in Fortran 90 as

```
       L(32:64) = L(96:128)
       K(32:64,:) = K(32:64,:)**2
```

In some sense, then, our compiler has high-level parallelism pre-packaged for it by the user's choice of Fortran-90 constructs. We can therefore concentrate on matching the source's parallelism to that of the target, and avoid the tangle of extracting parallelism from serial code.

2.2 Machine Model: Slicewise CM/2 Programming

The Connection Machine Model CM/2 consists of up to 2,048 Slicewise Processing Elements (*nodes* or *PEs*), each consisting of 32 bit-serial processors coupled with one Weitek WTL3164 64-bit floating-point ALU, as shown in Figure 1. The PEs are connected by a 12-dimensional boolean hypercube with two wires along each dimension. The bit-serial nature of the processor set is hidden from the programmer, who sees a conventional bit-parallel, word-serial interface to memory and to the Weitek FPU.

The programming language designed by the CM Fortran group for this PE abstraction is PEAC (Processing Element Assembly Code). PEAC allows the Weitek chip to be programmed as a four-wide vector processor;

it also allows accesses to CM memory to be overlapped with arithmetic operations, and supports the Weitek chained multiply-add instruction.

Each slicewise processor synchronously executes instructions issued from the CM sequencer. Because of the restrictions imposed by the SIMD programming model, and because four-wide vector processing is the default, code written for these nodes cannot use conditional control flow in the usual sense. Instead, the programmer must use masked moves to simulate conditional assignment.

Efficiently compiling Fortran-90 code to the PE abstraction starts with recognizing computations which are completely grid-local and which may involve grid coordinates. These computations can be performed purely over locally available data in each processor without respect to the actual layout of data. This allows their efficient implementation as a *virtual subgrid loop* over the local block of data allotted by the CM runtime system to each slicewise PE.

Unlike computation, interprocessor communication under the slicewise model is in general no faster than in the previous (fieldwise) programming model. When compilation to the canonical PEAC format is not possible due to dependencies, the front end must generate calls to the CM runtime system to perform communication. If the dependencies are regular, grid communications suffice; if they are not, general communications via the CM router result. Many special-purpose communications routines have been efficiently implemented in microcode, however, and can be substantially faster than the worst-case router alternative.

2.3 Main Compiler Issues

The main tension in CM code arises between communication and computation. Specifically, given a section of user code, we want to perform decompositions and transformations to minimize interprocessor communication. Simultaneously, we want to maximize in-processor performance through efficient vectorization. There are, therefore, multiple levels of program optimization to consider.

The relatively high cost of data movement presents us first with high-level challenges of data layout and communication. At the same time, codes which are naturally computation-dominated, with good locality and potential for vectorization, will benefit most from treatment of processor-level concerns.

These two optimization levels are not entirely orthogonal. High-level compilation strategies which group computations with similar layout, minimize conversion between rival layouts, and unroll serial loops all help to maximize the PEAC elemental codeblock size. This in turn allows more efficient low-level coding techniques.

We believe that node processor complexity, which continues to track advances in microprocessor technology, will continue to increase this pressure for multilevel optimization. Just as the CM/1 bit-serial processor gave way conceptually to the Weitek-augmented abstraction of the CM/2, the CM/2 PE will in due course be replaced by the CM/5's SPARC node, augmented with optional vector hardware. For massively parallel architectures in general, increases in node sophistication will dramatically increase the need for compilers which can orchestrate strategies for cooperative optimization at many levels of target abstraction, from high- to low-level.

These, then, are the primary concerns which a realistic prototype must satisfy:

- The scope of the project is, in fact, a prototype, intended to serve as a testbed for research into optimization strategies. Therefore, minimizing development time is critical.
- We therefore require a framework for compilation which will help us to apply minimal effort to compiler scaffolding (front end data, serial code, and bookkeeping concerns) while optimizing the compiler's critical paths — selecting appropriate forms of communication, efficient register allocation and effective vectorization in the node code.
- The final goal is to produce compiled object code for the CM whose performance is competitive with traditional compilation techniques for massively parallel architectures, for a fraction of the development cost.

These goals translate into a metric which we will refer to while developing compiler specifications. Specifically, in the early target-independent phase we want to

- lay out data to processors to reflect the structure of the computations to be performed,
- recognize regular and well-supported communication patterns in the source code, and
- transform the user's code to minimize domain conflicts between blocks of computations over similarly aligned data.

These restructurings are aided by target reference information fed from the back end. They take the form of source-level program transformations over our intermediate representation.

Machine-level optimizations, by contrast, will take place in the target-specific compilation phase, during the interpretation of programs in the intermediate representation to native node processor code. These include efficient use of registers to minimize memory traffic and superscalar instruction issue.

This specification structure is shown in Figure 2.

3 Native Intermediate Language

In the Fortran-90-Y compiler we bring formal specification techniques to bear on the dual challenges of performance and development time, using an abstract semantic algebra with special operators for describing serial and parallel iteration to formulate an appropriate machine model.

These operators are collected in the central notation of the Fortran-90-Y compiler specification, Native Intermediate Language (NIR). The label "native" refers to the fact that this language serves as the common source notation for each component of the prototype compiler after the initial semantic lowering phase. NIR serves as the basis for target- independent program transformation as well as for specific compilation to the SPARC control processor, the CM/2 slicewise node processor, and the CM/2 as a whole. The overall structure of the compiler, and the role of the intermediate language, is shown in Figure 2. Because of NIR's central role in the specification process, the following sections give a description of its use.

We begin with a very brief overview of the theoretical framework underlying NIR. We then present extensions to NIR which allow us to model and match opportunities for parallelism in the source and target.

3.1 Theoretical Foundation

We define a series of semantic domains, classically **Type, Declaration, Imperative,** and **Value**, and create a set of operators which model program actions within each domain. Together, the domains cover all dynamic program behaviors, and productions of the algebra are equivalent to programs for this abstract machine. Generating a compiler for a specific machine target then reduces to the problem of compiling constructs of this semantic algebra to native code.

An exact description of techniques for constructing semantic algebras to model target machine behavior is beyond the scope of this paper. The traditional core domains of NIR are described in the Appendix; the following sections build on that core by introducing a new shape domain for modelling data parallelism.

The individual semantic domains can be pictured as forming the facets of a gem (Figure 3). Within each domain there are primitive operators lying within each facet, as well as bridging operators which wrap fragments from other domains into a new one. As we will show, the new domain of shapes gives us data parallelism by augmenting the core domains with a new facet, orthogonal to the others, but connected by new bridging operators along each edge.

3.2 Modelling Iteration: Shapes

Using only the core NIR operators, we could model imperative languages implemented serially, such as Fortran 77, Pascal, or Algol. A complete implementation of our operator set for a given target environment would comprise a complete, working compiler. However, for the data-parallel extensions of Fortran 90 we require a richer set of primitive operators than those already defined.

User code which iterates over data or time, as well as target hardware which exhibits both parallelism and sequential restrictions, can be thought of as forms of serial or parallel iteration over abstract spaces. We represent such spaces in NIR using ***shapes***, a class of primitive semantic operators which model iteration. The basis for a shape in the current work is simply a Cartesian product space, as suggested in Figure 4, although future work may include tree, hypercube, or butterfly domains as shape primitives, as suggested by previous research into domains and data fields [4].

To integrate these shape operators into non-iterative NIR, we add bridge operators to each of the other semantic domains. The new domains and operators to describe shapes are summarized in figure 6.

- **Types.** To model array types, we add the type operator `dfield:S*T`, which defines a new type whose shape is `S` and whose elements are each of type `T`. `T` may itself be a `dfield`, which is one interpretation of the shape cross-product.

- **Declarations.** Because the existing declaration operators allow identifiers to be bound to any expressible type, array declarations just take the form `DECL(i,dfield(S,T))`.

- **Values.** We add a new value-producing operator, `AVAR(i,F)`, which references storage bound to identifier `i` through field action `F`. Field actions, which represent an overrestricted form of shapes, specialize the declared shape with one of `subscript(S)`,`everywhere`, and `local_under(S,d)`. These represent shapewise subscripting, universal selection, and construction of a local coordinate matrix, respectively. This allows us to construct array references which are explicitly parallel, as demonstrated in Figure 7.

- **Imperative actions.** Finally, we model serial and parallel iterations over spatial and temporal domains with the operator `DO(S,I)`, which carries out the action `I` at each point in the shape action `S`. Whether the iterations of the modelled loop are executed serially or in parallel depends entirely on the definition of `S`. `I` may itself be another `DO`-construct, giving us another way to inductively define loops over Cartesian product spaces. We can then begin to define such standard transformations as loop interchange and loop fusion using `DO`.

  Moving data between data of like shape is handled normally as part of the general data motion operator `MOVE`, where the left- and right-hand types are `dfields` with matching shapes and elemental types. In general, `MOVE[(True,(src,tgt))]` is equivalent to `DO(s,MOVE[(True,(src locally, tgt locally))])`, where `s` is the common shape of `src` and `tgt`.

The Fortran 90 code mentioned earlier,

```
INTEGER K(128,64), L(128)
L = 6
K = 2*K+5
```

might be expressed in shape-bearing NIR as shown in figure 8. Note that the `everywhere` operator is used for the array variable subscript to specialize different shapes, with the meaning specified by context. `Everywhere` thus allows us to express parallelism in data movement decoupled from the specific shape associated with the array variables.

3.3 Compiling NIR

NIR serves as a common representation for serial and parallel constructs, independent of the target machine. The last stage of compilation, therefore, reduces an explicitly parallel NIR program to the target machine language, for some specific target. The parallelism of our potential targets (MIMD and SIMD architectures with hypercube or fat-tree connectivity) can be represented using the shape notation as well in order to facilitate this final compilation.

The details of this compilation stage cannot be presented here due to space restrictions. Briefly, however, on a MIMD hypercube we might model the set of processors as a particular shape, transform the parallel arrays used in the user's NIR program to carry out block or cyclic layout of array elements to that processor shape, and generate host and node code based on the resulting partition. On the Connection Machine, we currently leave the exact partitioning up to the runtime system, and generate host and SIMD node code based on purely local computation over the user's shapes, laid out blockwise to the CM processing elements. The parallel computation over each block is simulated in-processor by a *virtual subgrid loop*, each of whose iterations carries no dependencies.

The following sections describe one implementation of this final compilation process, the Fortran-90-Y prototype compiler, as depicted in figure 2.

4 Fortran-90-Y: Target-Independent Phase

The Fortran-90-Y prototype compiler consists of five layers: the front end, the semantic lowering stage, the intermediate optimization stage, the slicewise CM/2 PA compiler, and a pair of sibling PA compilers for the Sun SPARC front end and the CM/2 slicewise processing element. Each phase is specified in Standard ML of New Jersey.

The first phase translates the user's syntactic forms (represented internally by ASTs) to constructs of the semantic algebra (valid NIR programs). The intermediate representation is then optimized to block computations over like shapes, before it is passed to the target-specific NIR compiler.

4.1 Front End Semantic Lowering

The semantic lowering stage consumes ASTs produced by syntactic analysis and performs pattern matching using a set of semantic equations. Each complete procedural unit or main program compiles to a single imperative action which has been typechecked and shapechecked. Static shapechecking is an analogous operation to static typechecking, but over the shape domain. This step satisfies assertions that in all direct computations between arrays, the shapes of interacting arrays agree.

There are five semantic equations, one for each of the semantic domains — declarations, types, values, imperatives, and shapes. Each is defined piecewise as a mapping from specific syntactic forms to NIR fragments.

These equations are formulated as an exercise in pattern-matching, and are not reproduced here for sake of brevity. They simply filter out the static semantics of the source language (in our case, Fortran 90), and express the residual as a valid NIR program without attempts at optimization or other transformation. The result is target-independent and can then be passed to the NIR optimization phase or to the target NIR compiler in the back end.

4.2 NIR Transformations

The NIR optimization stage performs source-to-source transformations over NIR code. This framework supports program transformation within each semantic domain, propagating the effects of transformations through the program by way of NIR's bridging operators, where domains meet. The object is to produce programs in which computations over like shapes are blocked as much as possible, forming computation phases sometimes punctuated by communication. These shape-based program transformations allow us to express standard compilation techniques such as various loop transformations, as well as specific goals like efficient layout to processors and use of supported communication patterns.

One such transform passes over the source program, analyzing the shapes of data and the operations over them. It generates an execution partition in which each phase either carries out a single computational action over data with a common shape and alignment, or expresses a single communication of data from one shape/alignment to another. Then it attempts to rearrange these phases so as to maximize the length of the blocks of aligned computation between successive communications. Successive loops over common, aligned domains appear in NIR as `DO`- or `MOVE`-constructs with common shapes, and as such are easily recognized and their actions composed sequentially — the shape equivalent of loop fusion. A sample transformation is shown in Figure 9.

Additional shape transformations may be performed to enable this blocking. By generating mask code, the compiler pads computations over array subsections to full-array operations, increasing the pool of sibling computations which could be implemented in the same computation block. When multiple array subsections can be shown to be disjoint, as in a `WHERE/ELSEWHERE` construct, the logical mask which is generated can be reused and the computations blocked together.

In the transformations shown in Figure 10, the odd-domain assigment to `c` is moved above the other, like-shape computations, and the two masked assignments are grouped together because their masks are disjoint and manage to cover the common shape `S`.

In each of these examples, discovering and exploiting the data movement along common shapes, as well as performing transformations to increase the number of common-shape computations, can result in a potentially substantial performance improvement. This is due to reduced CM procedure call overhead and improved register allocation during longer computation bursts.

5 Fortran-90-Y: Target-Specific Phase

5.1 CM2/NIR Compiler

The problem of compiling a valid NIR program into code for the CM/2 is broken down into a hierarchy of NIR compilers for different levels of target abstraction. The top-level abstraction (CM2/NIR) models the CM/2 host and nodes together as a single machine, and then partitions input NIR programs into NIR subprograms for each half.

The source NIR program has been restructured by the optimization phase to consist of blocked computation and communication phases. The CM2/NIR compiler just cuts out the computation phases and patches the remaining program to include appropriate NIR calling code. Each computation phase will be compiled as a single node procedure, and the remainder will become supporting host code.

The CM2/NIR compiler then calls the PE/NIR compiler for each excised computation block and the FE/NIR compiler over the remainder program. If the "role" of the top-level CM2/NIR compiler was the division of labor between the host and nodes, the role of these sibling compilers is the actual reduction of NIR to native code.

5.2 CM2/FE and CM2/PE NIR Compilers

The FE/NIR compiler translates the NIR remainder program into SPARC assembly code plus runtime system library calls. `DO`- and `MOVE`-constructs over serial shapes become explicit iteration. References to front-end data, along with CM data used in a front-end context all become front-end code. Declarative NIR constructs become memory allocations, with their home determined by usage. Certain primitive function calls which represent communication intrinsics are replaced by calls to their CM runtime library implementations. For each computation block being executed remotely, the compiler inserts calling code to push PEAC procedure arguments over the IFIFO to the processors.

This partitioning into host and node code is shown in Figure 11. The sample program is represented graphically; "A" nodes are pure-computation iterations over some shape A, "B" nodes over some shape B. Communications between different shapes take place on the graph edges. Where control dependencies allow, iterations over like shapes have been fused into single nodes before being cut out and passed to the component compilers for the host and nodes.

The FE and PE compilers in the current implementation represent the point where true specification ends. They compile NIR fragments to native code directly, using hand-coded algorithms to achieve competitive performance. The formalism of NIR is still present, however, insofar as these implementations are organized internally along the facets shown in figure 3. Pushing the level of specification down into the FE and PE compilers is an ongoing process, guided by this facetwise organization.

Having these separate compilers for the host and nodes allows effort to be expended intelligently to bring the prototype up to speed. During execution, the node processor and runtime libraries' speeds are the limiting factor for performance; the SPARC front end just has to keep up, feeding PEAC code and data through the sequencer and calling runtime communication routines. As problem size increases, therefore, front end time comprises a negligible fraction of the overall execution profile. In our prototype, SPARC performance questions could be postponed, so the current front-end semantic implementation uses a simple memory-to-memory load/store model with little attention to effective register use or delay slot filling. Allowing the host and node implementations to be fully separated gives us the freedom to filter out and focus on the performance-critical parts of the prototype compiler instead.

The prototype CM/PE node compiler is carefully tuned for optimizing the loop over local data in each processor, the process known as ***virtual subgrid looping***. The role of the processing elements is limited in this programming model to purely local, pointwise computations, allowing us to restrict the set of NIR programs which the CM/PE compiler accepts. Each computation burst (PEAC routine) is derived from computations over like shapes in the source program. CM/PE therefore only needs to process procedures whose body is a single loop containing a sequence of (optionally masked) moves from the local points of source arrays to the corresponding points in the target. This control structure is shown in the pseudocode compilation of Figure 10.

Furthermore, because such a virtual subgrid loop with purely local references can be represented graphically as one basic block with a single back-edge, register allocation can be optimized. Vector registers tend to be the limiting resource, so spill code is generated where necessary, although it need not occur at the exact spill site. We overlap the resulting memory accesses with computation where possible to minimize lost cycles, since a single vector spill-restore pair costs 18 cycles – roughly equivalent to three single-precision floating point vector operations. PEAC's support for load chaining also allows one in-memory operand to be substituted for a register operand, which helps reduce register pressure. Multiply-add sequences are converted to chained multiply-adds wherever possible.

After the host and node compilers have reduced their NIR source programs to SPARC assembly code and PEAC, their output is then compiled and linked normally, and can be executed on any CM/2 equipped with Weitek 64-bit FPUs.

5.3 Flexibility Issues

5.3.1 CM5/NIR Compiler

The CM/5 NIR compiler retains the majority of its structure and, therefore, its specification from the CM/2 version. The CM/5 presents more target levels due to the increased complexity of the processing node. In the new model a single NIR program will be split three ways rather than two; one part will go to the control processor, as before; a second part will be executed on the SPARC node processor, and a third part will carry out floating point vector operations on the CM/5 vector datapaths. This structure is represented in the second back-end diagram of Figure 2.

The host subcompiler remains relatively unchanged from the CM/2 implementation, but the node subcompiler partitions its input into subprograms for the SPARC and the four vector pipelines, instead of performing direct compilation. Porting effort is thus concentrated on taking advantage of the additional powers of the processing node. Most importantly, the new compiler can still take advantage of the machine-independent blocking and vectorizing NIR transformations defined in the front end.

5.3.2 Other Computation Models

There are, in practice, no reason why the compiler should adhere to a single, restrictive programming model at the expense of flexibility. For example, many codes would benefit from the ability to break the CM/2's virtual processor runtime model, restricted to pointwise locality and subgrid looping. A more flexible model would allow the compiler to pipeline communication and computation, or perform general neighborhood computations directly, using the full register set to store intermediate results and performing physical communications as required.

Implementing other programming models would only require the specification of new FE and PE compilers, and adjustments to the top-level CM2/NIR compiler to use them. The NIR source transformation stage might also benefit from extra modules to provide services from the runtime system previously taken for granted, such as explicit data layout.

6 Experimental Results

The basic specification methods were tested in the Fortran-90-Y compiler prototype, which we regard as "realistic" by the following standards:

Development time. The current work began in July 1991, with research into the peculiarities of the slicewise CM programming model, the Fortran-90 language, and the CM Fortran compiler. Implementation of the prototype compiler actually began in mid-August. The compiler began generating code for simple test programs by late October, and was compiling benchmark code by November.

Direction of Effort. By modeling the CM with a unified phase (CM2/NIR), we recognized that the CM taken as a whole is still a useful abstract target for compilation. But by dividing the labor of compilation between these host and node compilers, imposing restrictions on the constructs each accepts, and optimizing the compiler's critical path first, we were able to develop a competitive system for an interesting subset of the source language in under six months.

Performance. The initial benchmark was an updated Fortran-90 version of a dusty deck code to implement a meteorological model, the "shallow-water equations," or SWE. It has good locality, consisting of a series of circular shifts interspersed with blocks of local computation, and so represents an ideal problem for a SIMD, data-parallel machine like the CM/2.

A hand-coded *Lisp version of SWE running under fieldwise mode peaked at 1.89 gigaflops. The slicewise CM Fortran compiler (v1.1) reached an extrapolated 2.79 gigaflops. The prototype Fortran-90-Y compiler, after the first six months of development, produced code which attained a competitive untuned peak rate of 2.99 gigaflops.

This reasonable level of performance is attributable to a number of factors. First, the use of shape analysis and program transformation to recognize and group computations over elemental blocks into computation groups of maximal length means that the PEAC subroutine calling time and the overhead of receiving pointers and data from the front-end FIFO is amortized over more floating point computations, in longer virtual subgrid loops. Furthermore, wherever possible, loads and stores of data have been chained with the first or last use of a live variable, respectively, or overlapped with unrelated computations. Finally, lifetime analysis allows optimal register assignment within the body of the virtual subgrid loop, with minimal spill traffic, and spill/restore code may move up- or downstream from the actual spill site, as overlapping permits.

7 Conclusions

In this extended abstract we have presented a brief overview of a new approach to parallelizing compiler design incorporating concepts from both formal specification and high-performance computing. The original motivation of this work was to demonstrate that developing scalable, portable, competitive compiler technology for massively parallel computing environments was feasible, in terms of reasonable development time and competitive performance. The achievements of the current Fortran-90-Y prototype implementation suggest that these methodologies are generally applicable to the challenge of developing compilers for complex parallel targets at reasonable cost.

8 Acknowledgements

The authors thank Young-Il Choo for his advice regarding earlier versions of NIR, and Yu Hu for his many comments and technical advice. Also, thanks to Woody Lichtenstein, Bob Millstein, and Gary Sabot of Thinking Machines for their help with CM Fortran, CM/RT, and the slicewise programming model. Finally, special appreciation is due Lennart Johnsson, also of Thinking Machines, for his help in formulating the problems to be addressed and information on all aspects of the CM.

References

[1] *CM Fortran Reference Manual*, 1991.

[2] *Fortran 90 Standard*, 1991.

[3] M. Bromley, S. Heller, T. McNerney, and G. Steele Jr. Fortran at ten gigaflops: The connection machine convolution compiler. *ACM SIGPLAN '91 Conference on Programming Language Design and Engineering*, 26(6):145–156, 1991.

[4] Marina Chen, Young-il Choo, and Jingke Li. Theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers*, chapter 7. ACM Press and Addison-Wesley, 1991.

[5] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1988.

[6] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 67–78, 1991.

[7] S. Tjiang, M. Wolf, M. Lam, K. Piper, and J. Hennessy. Integrating scalar optimizations and parallelism. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, pages c1–c16, 1991.

Appendix: NIR Core Domains

This material, which is purely supplemental, describes the four core semantic domains of the Native Intermediate Language.

Types, Declarations. The set of NIR type operators models all of the "machine-level" types we want to represent in our semantic algebra. Currently these are defined in a somewhat *ad hoc* fashion, including integers, logicals, and single- and double-precision floating point numbers.

The declarative operators allow identifiers to be bound to types and, optionally, to provide some initial values for them. These operators by themselves do not define scoping rules; because scoping is a static program behavior which bridges the domains of declaration and control, we achieve scoping using the imperative bridge operator `WITH_DECL(d,I)`, which executes an imperative action `I` in a context in which the declaration `d` is visible.

As an example, the Fortran-90 code

```
double precision m,n
```

could be represented by the NIT fragment

```
DECLSET[DECL('m',float_64),DECL('n',float_64)]
```

Values. The value-producing operators represent program actions which compute values, either by referencing the (implicit) store, by reference to constants, by function calls, or by computation parameterized by other value-producers.

The Fortran computation

```
a*b+sin(c)
```

might be translated by the NIR fragment

```
BINARY(Plus,SVAR 'a', BINARY(Mul, SVAR 'b', UNARY(Sin, SVAR 'c')))
```

Control and Store. The imperative operators model most remaining program behaviors. The `SEQUENTIALLY[i1,i2]` operator, for example, composes the two sub-actions `i1` and `i2` for sequential execution. Similar operators are provided for manipulating the store (`MOVE`), the declarative scope (`WITH_DECL, PROCEDURE`), and for some forms of looping (`WHILE, UNTIL`).

For example,

```
a = cos(b)
b = b + a
```

might become

```
(SEQUENTIALLY
[
 MOVE[(True, (UNARY(Cos, SVAR 'b'), SVAR 'a'))],
 MOVE[(True, (BINARY(Plus, SVAR 'b', SVAR 'a'), SVAR 'b'))]
])
```

A partial listing of the NIR operators can be seen in figure 5.

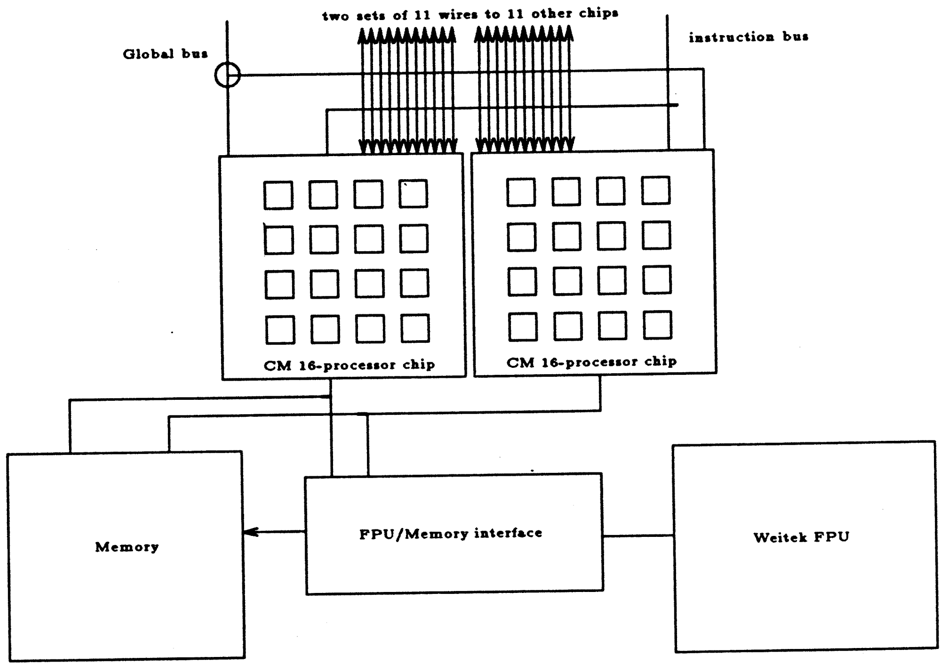

Figure 1: CM Slicewise Processing Element

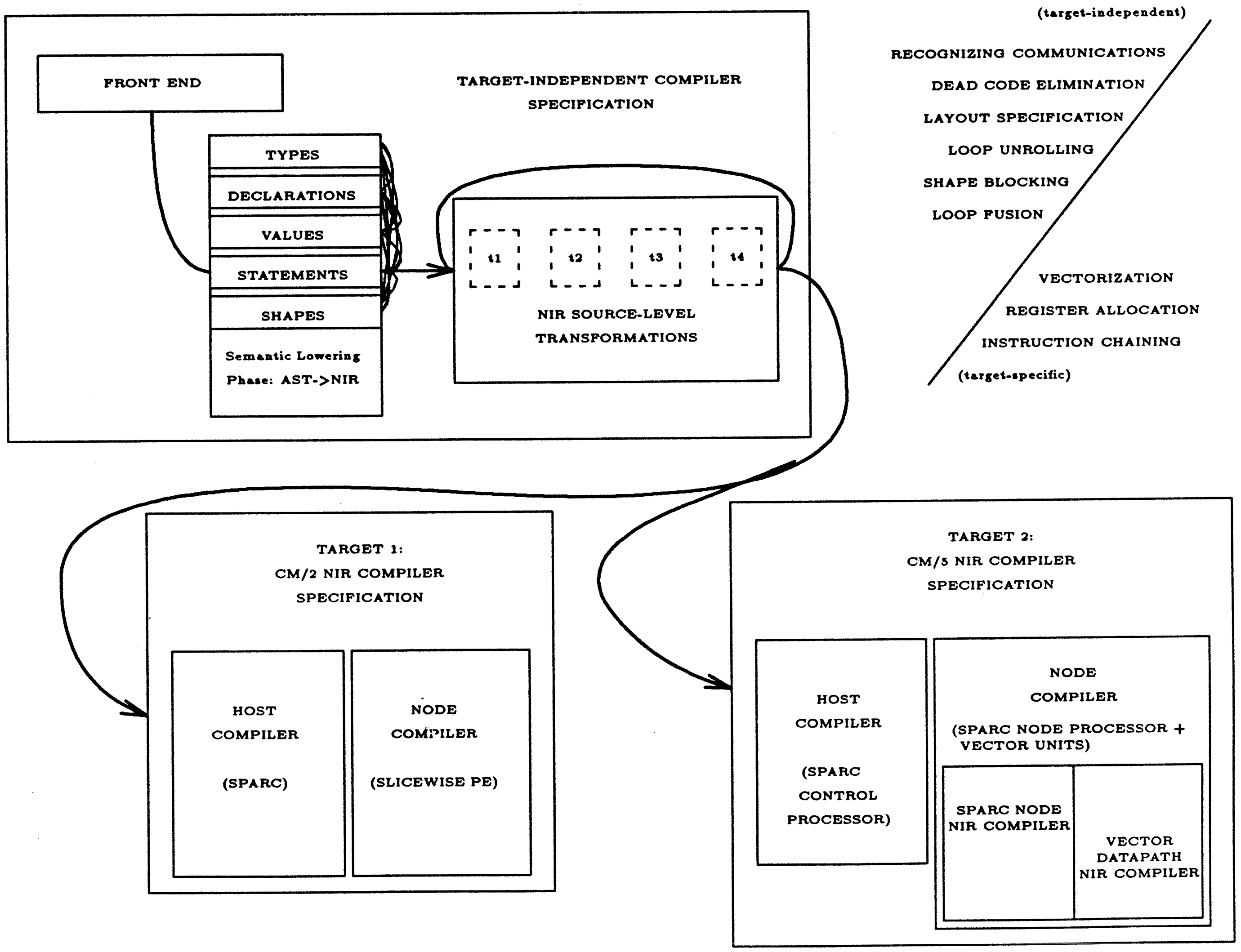
FRONT END
TARGET-INDEPENDENT COMPILER SPECIFICATION
TYPES
DECLARATIONS
VALUES
STATEMENTS
SHAPES
Semantic Lowering Phase: AST->NIR
t1
t2
t3
t4
NIR SOURCE-LEVEL TRANSFORMATIONS
(target-independent)
RECOGNIZING COMMUNICATIONS
DEAD CODE ELIMINATION
LAYOUT SPECIFICATION
LOOP UNROLLING
SHAPE BLOCKING
LOOP FUSION
VECTORIZATION
REGISTER ALLOCATION
INSTRUCTION CHAINING
(target-specific)
TARGET 1: CM/2 NIR COMPILER SPECIFICATION
HOST COMPILER (SPARC)
NODE COMPILER (SLICEWISE PE)
TARGET 2: CM/5 NIR COMPILER SPECIFICATION
HOST COMPILER (SPARC CONTROL PROCESSOR)
NODE COMPILER (SPARC NODE PROCESSOR + VECTOR UNITS)
SPARC NODE NIR COMPILER
VECTOR DATAPATH NIR COMPILER

Figure 2: Fortran-90-Y specification structure

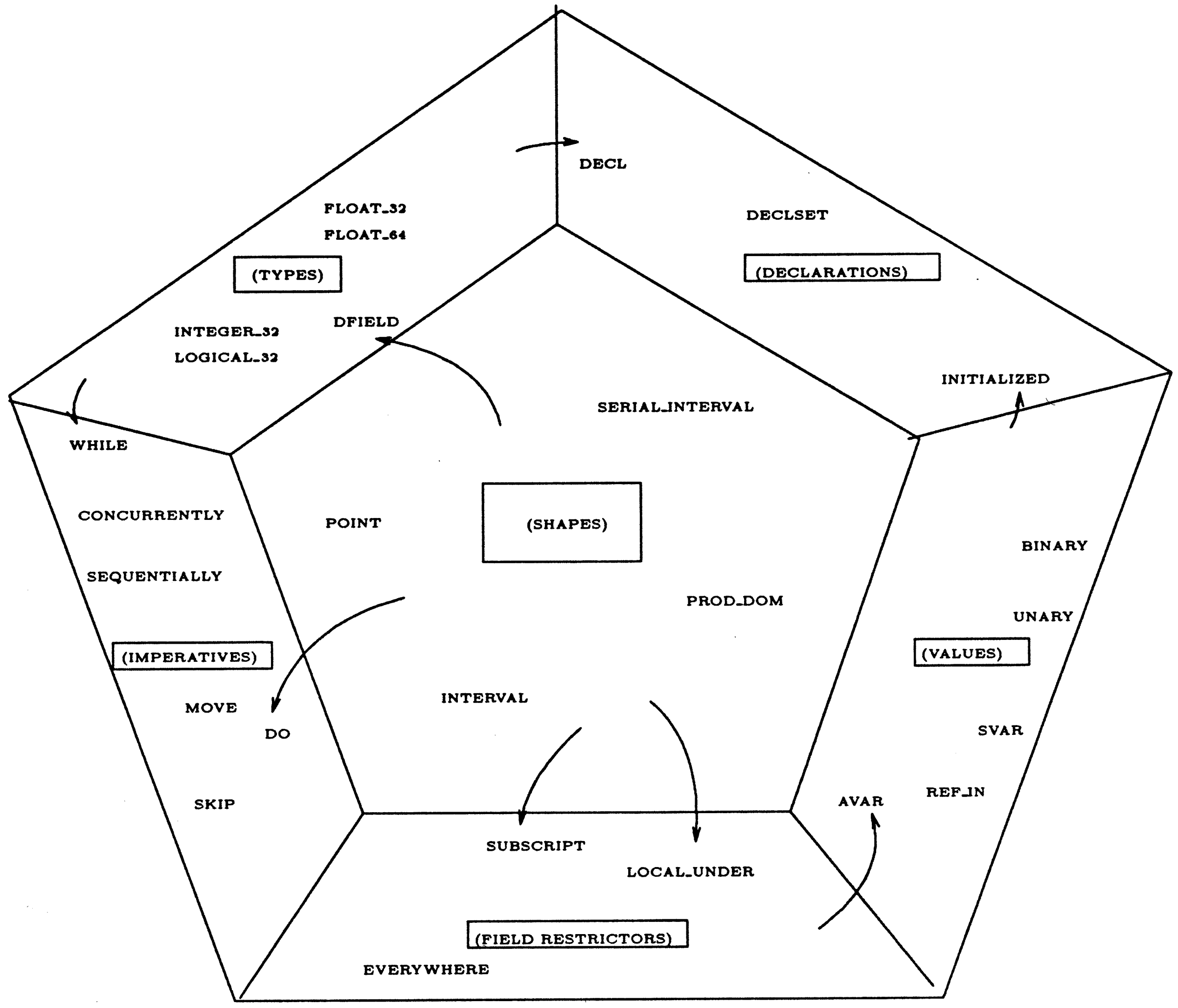

Figure 3: NIR facets and bridges

```
  LOOP(action,point X) => action(X)

| LOOP(action,interval(point min..point max)) =>
    SEQUENTIALLY
      [LOOP(action,point min);
       LOOP(action, interval(point succ min..point max))]

| LOOP(action, product space [dim1]) => LOOP(action,dim1)

| LOOP(action, product space [dim1,dim2,..]) =>
    LOOP(LOOP(action, product space [dim2,..]), dim1)
```

Figure 4: Inductive modelling of serial loops defined over shapes

TYPE DOMAIN (T)		
integer_32	T	32-bit integer
logical_32	T	32-bit logical
float_32	T	single-precision floating point
float_64	T	double-precision floating point
...		
DECLARATION DOMAIN (D)		
DECL	id * T -> D	simple declaration
DECLSET	D list -> D	multiple declarations
INITIALIZED	id * T * V -> D	declaration plus initial value
...		
VALUE DOMAIN (V)		
BINARY	binop*V*V -> V	binary computation
UNARY	monop*V -> V	unary computation
SVAR	id -> V	scalar variable
SCALAR	T*s_rep -> V	scalar constant
FUNCALL	id*(T*V)list -> V	function call
REF_IN	V	receives call-by-reference parameter
COPY_IN	V	receives call-by-value parameter
...		
IMPERATIVE DOMAIN (I)		
PROGRAM	I->I	top-level program action
SEQUENTIALLY	I list ->I	sequential composition
CONCURRENTLY	I list ->I	concurrent composition
MOVE	(V*(V*V))list -> I	move multiple under mask
IFTHENELSE	(V*I*I) -> I	classical if-then-else
WHILE	(V*I) -> I	classical while-construct
REF_OUT	V->I	passes call-by-reference parameter
COPY_OUT	V->I	passes call-by-value parameter
WITH_DECL	D*I -> I	execute in extended environment
SKIP	I	defines (SEQUENTIALLY nil)
...		

Figure 5: NIR Core Domains and Operators

SHAPE DOMAIN (S)		
point	`int -> S`	single point
interval	`S*S -> S`	parallel vector shape
serial_interval	`S*S -> S`	serial vector shape
prod_dom	`S list -> S`	shape cross-product
...		
TYPE DOMAIN (T)		
dfield	`S*T->T`	a field of elements of the given type
VALUE DOMAIN (V)		
AVAR	`id*F -> V`	array variable
...		
FIELD RESTRICTOR DOMAIN (F)		
subscript	`S -> F`	subscripting action
everywhere	`F`	unrestricted shape access
local_under	`S*int -> F`	coordinate shape constructor
...		
IMPERATIVE DOMAIN (I)		
DO	`S*I -> I`	execute over the given shape
...		

Figure 6: NIR augmentations to describe shapes

```
    INTEGER, ARRAY(32,32) :: A
    FORALL (i=1:32, j=1:32) A(i,j) = i+j
```

could be expressed in NIR using a single move written using the parallel array notation as

```
(WITH_DOMAIN('alpha',
 prod_dom(interval(point 1,point 32),interval(point 1,point 32)),

  (WITH_DECL
   (DECL('a',
    dfield{shape=domain alpha,element=integer_32})),

     MOVE(True,
      (BINARY(Plus,
         local_under(domain 'alpha',1),local_under(domain 'alpha',2)),
       AVAR('a', everywhere))))))
```

Figure 7: Example: Parallel Array Notation

```
WITH_DOMAIN(('alpha',interval(point 1,point 128)),
 WITH_DOMAIN(('beta',prod_dom[domain 'alpha',interval(point 1,point 64)]),
  WITH_DECL(DECLSET[DECL ('k',dfield{shape=domain 'beta',
                                     element=integer_32}),
                   DECL ('l',dfield{shape=domain 'alpha',
                                     element=integer_32})],
 SEQUENTIALLY
 [
   ...

   MOVE[(True, (SCALAR(integer_32,'6'), AVAR('l',everywhere))),
        (True, (BINARY(Plus,
                 BINARY(Mul, SCALAR(integer_32,'2'), AVAR('k',everywhere)),
                 SCALAR(integer_32,'5')),
                AVAR('k', everywhere)))],
   ...
])))
```

Figure 8: NIR Excerpt: Shape-Parameterized Parallel Computation

```
      integer, array(64,64) :: A,B
      integer, array(64) :: C
      do 10 i=1,64
forall j=1:64 A(i,j) = B(i,j) + j
10    continue
      do 20 i=1,64
        C(i) = A(i,i)
B(i,:) = A(i,:)
20    continue
```

could be naively expressed in three MOVEs:

```
(WITH_DOMAIN('gamma', interval(point 1, point 64),
  WITH_DOMAIN('beta',  serial_interval(point 1, point 64),
   WITH_DOMAIN('alpha', prod_dom(domain beta,domain gamma),
WITH_DECL(DECLSET
 [DECL('a',domain 'alpha'),DECL('b',domain 'alpha'),DECL('c',domain 'beta')],
SEQUENTIALLY
[ MOVE[(True, (BINARY(Plus, AVAR('b',everywhere),
                            local_under(domain 'alpha',2)),
              AVAR('a',everywhere)))],
  DO(domain 'beta',
    MOVE[(True, AVAR('a',subscript
                  (prod_dom[local_under(domain 'beta',1),
                            local_under(domain 'beta',1)])),
               AVAR('c',subscript[local_under(domain 'beta',1)]))]),
  MOVE[(True, (AVAR('a',everywhere), AVAR('b',everywhere)))]])))))
```

Notice that the first and last MOVE take place over the domain alpha, while the second uses domain beta, and there are no dependencies between the second and third MOVEs. We can move the like-domain MOVEs together, and compose them within the scope of the common domain, so that they will become one computation block on the CM.

```
SEQUENTIALLY
[ MOVE[(True, (BINARY(Plus, AVAR('b',everywhere),
                            local_under(domain 'alpha',2)),
              AVAR('a',everywhere))),
       (True, (AVAR('a',everywhere), AVAR('b',everywhere)))],
  DO(domain 'beta',
    MOVE[(True, AVAR('a',subscript
                  (prod_dom[local_under(domain 'beta',1),
                            local_under(domain 'beta',1)])),
              AVAR('c',subscript[local_under(domain 'beta',1)]))])]
```

Figure 9: Domain Blocking Transformation

```
integer, array(32,32) :: A,B
integer, array(32) :: C
...
A = N
B(1:32:2,:) = A(1:32:2,:)
C = N+1
B(2:32:2,:) = 5*A(2:32:2,:)
```

becomes

```
MOVE[(True, (SVAR 'n',AVAR('a',everywhere))),
     (BINARY(Equals,BINARY(Mod,AVAR('b',local_under(S,1))
                               SCALAR(integer_32,'2')),
                 SCALAR(integer_32,'0')),
        (AVAR('a',everywhere),
         AVAR('b',everywhere))),
     (True, (BINARY(Add,SVAR 'n',SCALAR(integer_32,'1')),
             AVAR('c',everywhere))),
     (BINARY(NotEquals,BINARY(Mod,AVAR('b',local_under(S,1))
                               SCALAR(integer_32,'2')),
                 SCALAR(integer_32,'0')),
        (BINARY(Mul,SCALAR(integer_32,'5'),AVAR('a',everywhere)),
         AVAR('b',everywhere)))]
```

Where S is the 2D (32,32) shape common to A and B. This would be better blocked as

```
SEQUENTIALLY
[       MOVE[(True, (BINARY(Add,SVAR 'n',SCALAR(integer_32,'1')),
                AVAR('c',everywhere)))],
        MOVE[(True, (SVAR 'n',AVAR('a',everywhere))),
             (BINARY(Equals,BINARY(Mod,AVAR('b',local_under(S,1))
                                       SCALAR(integer_32,'2')),
                         SCALAR(integer_32,'0')),
                (AVAR('a',everywhere),
                 AVAR('b',everywhere))),
             (BINARY(NotEquals,BINARY(Mod,AVAR('b',local_under(S,1))
                                       SCALAR(integer_32,'2')),
                         SCALAR(integer_32,'0')),
                (BINARY(Mul,SCALAR(integer_32,'5'),AVAR('a',everywhere)),
                 AVAR('b',everywhere)))]]
```

because while the assignment to array C is along a one-dimensional array, the other moves take place in parallel over shape S, and dependencies allow the code movement. This fragment could be compiled into two PEAC routines; the host program would consist of the code needed to allocate CM storage for A,B, and C, and to call the two initializing routines. The second computation block might be encoded in PEAC pseudo-code as

```
Receive pointers to the local subgrids of A and B.
Receive a pointer to the local coordinate 1 subgrid.
Receive the virtual subgrid size V.
While V>0
    Move immediate N to A.
    Compute the mask (0 mod 2) over the coordinate subgrid.
    Compute the value 5*A into a temporary register.
    Move (mask?A:5*A) into B.; Decrement V.
```

Figure 10: Blocking With Parallel Masked Assignment

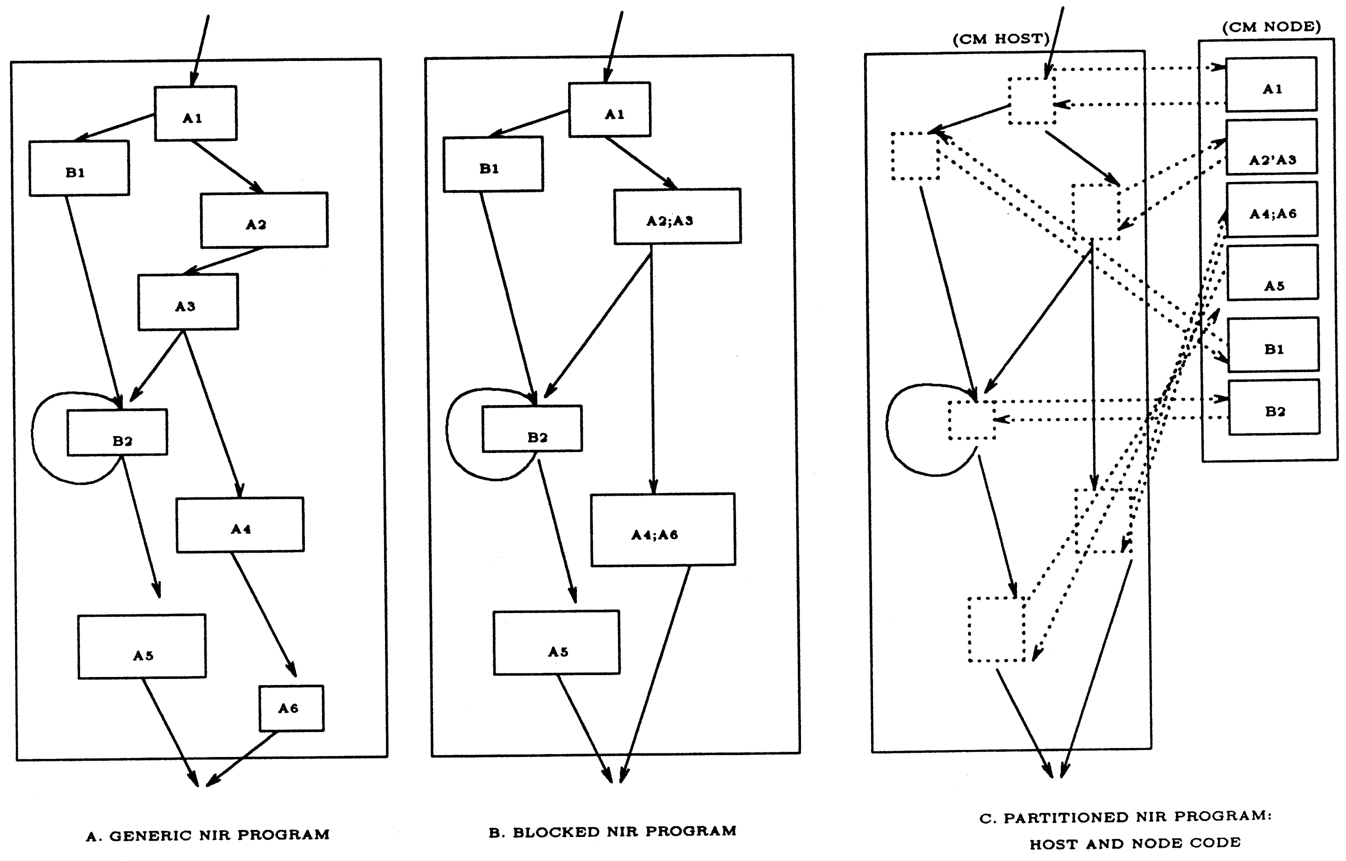

Figure 11: Naive, Blocked, and Partitioned NIR Program

SWE EXCERPT, IN FORTRAN 90 AND CM/PE NIR:

```
z=(fsdx*(v-CSHIFT(v,DIM=1,SHIFT=-1))
1     -fsdy*(u-CSHIFT(u,DIM=2,SHIFT=-1)))
2     /(p_temp+CSHIFT(p_temp,DIM=2,SHIFT=-1))
```

```
MOVE[(True,
  (BINARY(Div,
    BINARY(Sub,
      BINARY(Mul,SVAR"fsdx",
        BINARY(Sub,SVAR"v",SVAR"tmp0")),
      BINARY(Mul,SVAR"fsdy",
        BINARY(Sub,SVAR"u",SVAR"tmp1"))),
    BINARY(Add, SVAR "p_temp", SVAR "tmp2")),
   SVAR "z"))]
```

NAIVE PEAC ENCODING:

```
Pk51vsl_:
        flodv   [aP7+0]1++ aV3
        flodv   [aP4+0]1++ aV2
        fsubv   aV3 aV2 aV1
        fmulv   aS28 aV1 aV3
        flodv   [aP8+0]1++ aV4
        flodv   [aP3+0]1++ aV2
        fsubv   aV4 aV2 aV2
        fmulv   aS29 aV2 aV4
        fsubv   aV3 aV4 aV1
        flodv   [aP5+0]1++ aV2
        flodv   [aP2+0]1++ aV3
        faddv   aV2 aV3 aV3
        fdivv   aV1 aV3 aV3
        fstrv   aV3 [aP6+0]1++
        jnz     aC2 Pk51vsl_
```

OPTIMIZED PEAC ENCODING:

```
Pk51vsl_:
        flodv   [aP7+0]1++ aV3
        fsubv   aV3 [aP4+0]1++ aV1
        fmulv   aS28 aV1 aV3,    flodv   [aP8+0]1++ aV4
        fsubv   aV4 [aP3+0]1++ aV2
        fmulv   aS29 aV2 aV4
        fsubv   aV3 aV4 aV1,     flodv   [aP5+0]1++ aV2
        faddv   aV2 [aP2+0]1++ aV3
        fdivv   aV1 aV3 aV3
        fstrv   aV3 [aP6+0]1++
        jnz     aC2 Pk51vsl_
```

Figure 12: SWE Excerpt: Fortran-90-Y to CM/PE NIR