Prototyping Fortran-90 Compilers for Massively Parallel Machines

*Marina Chen & James Cowie**
Department of Computer Science, Yale University
chen-marina@cs.yale.edu cowie-james@cs.yale.edu

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. This has created a demand for new compiler prototyping technologies that allow novel styl 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 subset compiler for Thinking Machines' Connection Machine/2 and Connection Machine/5. The prototype produces code from initial Fortran-90 benchmarks demonstrating sustained performance superior to hand-coded *Lisp and competitive with Thinking Machines' CM Fortran compiler. This paper presents some new specification techniques necessary to construct competitive, easily retargetable prototype compilers.*

1 Introduction

Existing compilers for massively parallel machines have generally been constructed 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.

*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.

ACM SIGPLAN '92 PLDI-6/92/CA
© 1992 ACM 0-89791-476-7/92/0006/0094...$1.50

While CMF and similar compilers unarguably achieve production quality, they are difficult and expensive to construct, maintain, and extend as a result of their target-specific, highly-tuned, hand-coded nature. By contrast, traditional systems for generating compilers from denotational semantics drew on the flexibility and power of formal specification to build and maintain systems that were truly machine-independent. They remain unacceptable for production use, however, due to the generally poor performance of the code they produce, and their size and speed.

The MESS system [5] introduced "separable semantic specification" as an alternative to denotational semantics approaches. MESS used a two-level semantic specification to separate high-level language properties from model detail. This enforced separation helped alleviate the performance problems associated with compilers generated from formal descriptions. Such systems, however, were still limited in practice to sequential-execution, uniprocessor implementations of simple imperative languages. MESS-generated compilers had no systematic framework for program transformation, focusing instead on simple two-pass translation. They lacked the necessary semantic features to express parallelism, and thus did not attempt to model or 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 preserves high-level information for low-level transformations, and begins to show how multiple levels of optimization concerns must be coordinated, but was not used in the context of formal compiler specification. 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 that unifies data and control parallelism, nor a treatment of back-end concerns.

In this paper we propose a methodology for compiler design that brings together lessons learned from previous compilers for massively parallel machines with techniques from formal specification. Like the MESS system, our prototype compilers define a semantic algebra as their intermediate format, and use production rule sets to define the translation process from the user's syntactic forms to constructs of this algebra. Adding a framework for intermediate program transformation, support for multilayered target systems, and a shape abstraction for data parallel computation, we then show how to model massively parallel machines in a working compiler specification.

Finally, we present initial results from our prototype Fortran-90-Y compiler for the CM/2 and CM/5. After eight months of development, the prototype implementation generates code for multiple platforms exhibiting sustained performance superior to that of hand-coded *Lisp and competitive with Thinking Machines' CM Fortran compiler. Furthermore, the prototype's space and time requirements are comparable to those of native language tools for the CM. 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 that 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 that are of interest for data parallelism. Specifically, the ability to refer to and compute directly with whole arrays and arbitrary subsections give the programmer the ability to express program parallelism that would previously 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 that 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
```

To some extent, 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 in slicewise mode [1] 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. The PEs are connected by a 11-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. Each instruction processes a vector of four 32- or 64-bit elements – "slices" through local memory. PEAC code is only

[1] as opposed to the more traditional fieldwise, or Paris mode.

used to express purely elemental program segments; that is, under the normal programming model, there can be only pointwise dependencies between data that are aligned to the same geometry and used in the same PEAC routine. When an instruction sequence is applied serially to each of the (potentially very many) elements local to each processor memory, the result is a *virtual subgrid loop* parameterized by a set of strided data pointers and a subgrid loop count.

For computations that are not purely elemental, but have some degree of locality or regularity, the CM runtime system provides a large set of special-purpose uniform communication primitives, efficiently implemented in microcode. When dependencies in user code exceed the restrictions of both the elemental and the special-purpose uniform styles, however, general router communications result, at significant performance expense.

For example, the Fortran 90 code fragment

```
real, array(n,n) :: I
I(:,:) = 6.0
```

might be compiled as a single PEAC node routine storing the 32-bit scalar value `6.0` to each local element serially, in batches of four. On the other hand, the Fortran 90 assignment

```
I(:,2:n) = I(:,1:n-1)
```

might be implemented by the CM runtime system call `CMRT_cshift`, and the negative-strided array section assignment

```
I(:,n:1:-2) = I(:,1:n:2)
```

would be implemented via the router as general point-to-point communication from `I` to `I`.

2.3 Main Compiler Issues

Clearly, the main tension in CM code arises between communication and computation. Specifically, given a section of user code, an effective compiler must perform decompositions and transformations to minimize interprocessor communication. Simultaneously, the code generation process should maximize in-processor performance through efficient vectorization. There are, therefore, multiple levels of program optimization to consider.

The relatively high cost of data movement forces the compiler to address the high-level challenges of data layout and communication. Simultaneously, those codes that are naturally computation-dominated, easily vectorized due to elemental locality, will benefit most from treatment of processor-level concerns.

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

Node processor complexity, which continues to track advances in microprocessor technology, will continue to increase this pressure for multilevel optimization. As the CM/1 bit-serial processor gave way conceptually to the Weitek-augmented abstraction of the CM/2, so too the CM/2 PE will eventually 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 that can orchestrate strategies for cooperative optimization at many levels of target abstraction, from high- to low-level.

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

- Because the project is primarily a prototyping system, intended to serve as a testbed for research into optimization strategies, minimizing development time is critical, as is support for staged development. Formal specification techniques help support "attack strategies" for extremely rapid compiler development.
- The problem therefore requires a framework for compilation that will help 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 of the node code.
- The final goal is to produce compiled object code whose performance is competitive with traditional compilation techniques for massively parallel architectures, and, in many cases, with equivalent hand-coded solutions, for a fraction of the total development cost. Simultaneously, generated compilers must run in reasonable space and time on available workstations, generally within an order of magnitude of hand-coded equivalents, if they are to be considered "realistic" for production use.

3 Theoretical Basis

In the Fortran-90-Y compiler we use formal specification techniques to address the dual challenges of performance and development time, using an abstract semantic algebra with special operators for describing se-

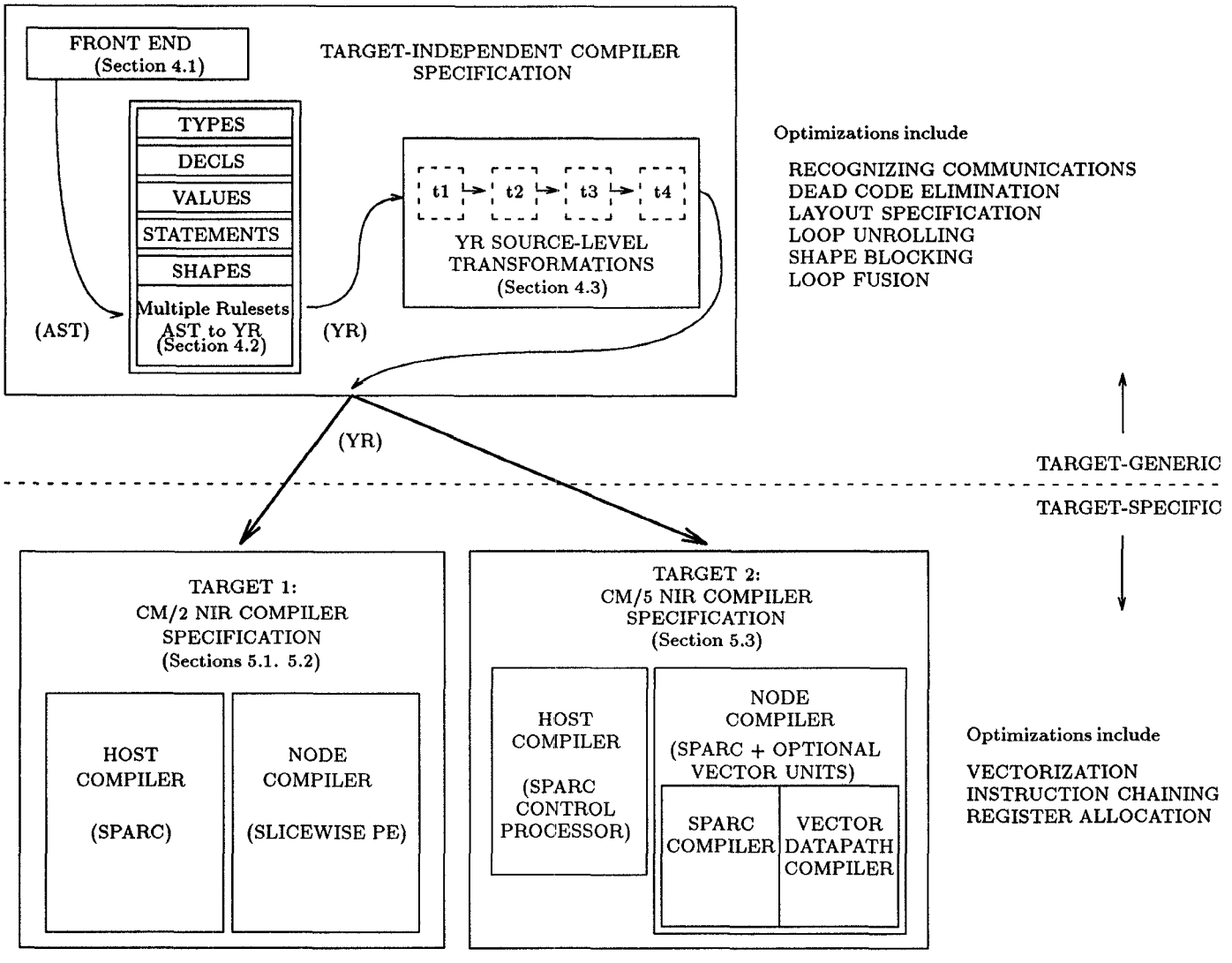

Figure 1: Fortran-90-Y specification structure

rial and parallel iteration to formulate an appropriate machine model.

These operators are collected in the central notation of the Fortran-90-Y compiler specification, *Yale Intermediate Representation* (YR). YR 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, the CM/5 SPARC node processor, and the CM/2 and CM/5 as a whole. The following sections describe YR and provide some examples of computations expressed in it. Then we describe the actual structure of the prototype compiler, and show how compilation to, transformation over, and code generation from YR take place.

The overall structure of the compiler, and YR's role in it, are shown in Figure 1.

3.1 YR Core Language

Core YR defines a series of semantic domains and sets of operators within each domain that model program actions. We will refer to these semantic domains as orthogonal *facets* of a semantic algebra defined by the YR operators, to avoid overloading the more usual interpretation of the term "domain" from the parallel processing literature. A partial listing of YR semantic facets and their operators are shown in Figure 2. Together, the core facets define a space of scalar program behaviors, and YR constructs are programs for the abstract machine characterized by these behaviors. Generating a compiler for a specific machine then reduces to the problem of compiling YR programs – constructs of this semantic algebra – to native code, by implementing the abstract machine for the intended target.

3.2 YR + Shapes

Using only the core YR 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 the addition of primitive shaping actions.

User code that iterates over data or time, as well as target hardware that 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 YR using *shapes*, a class of primitive semantic operators that model iteration. The basis for shapes in the current work is the basic Cartesian product space, 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 scalar YR, bridge actions are added to each of the other semantic facets. The additional operators are marked with asterisks in figure 2.

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 may be interpreted as a shape cross-product. Each value action has a declared or inferred shape and type, which can be checked statically at compile time.

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,S)`, which references array storage bound to identifier `i` through shape action `S`. Shape actions used as subscripts can further specialize the declared shape of an array variable, or default to the declared shape using the special shape action `all`. This allows us to construct array references that are implicitly parallel. Shapes can also be filled in with integer coordinate values using the `LOCALLY` value-constructor.

Imperative actions. Finally, we model serial and parallel iterations over spatial and temporal domains with the operators `DO(V,S,I)` and `DOALL(V,S,I)`. These operators carry out the action `I` at each point in the shape action `S` with either serial or concurrent semantics, borrowing the storage referenced by the value set `V` for maintaining the loop variables.

The body action `I` may itself be another `DO-` or `DOALL`-construct, providing a way to inductively define loops over Cartesian product spaces. We can then begin to define such standard transformations as loop interchange and loop fusion.

Moving data between like shapes 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, and the storage phase of the movement is executed under an optional mask.

TYPE FACET (T)	operator signature	interpretation * – Shape augmentation to core YR
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
dfield	`S*T->T`	* a field of elements of the given type
DECLARATION FACET (D)		
DECL	`id * T -> D`	simple declaration
DECLSET	`D list -> D`	multiple declarations
INITIALIZED	`id * T * V -> D`	declaration plus initial value
VALUE FACET (V)		
COMPUTE	`op*V*V -> V`	arithmetic computation
SVAR	`id -> V`	scalar variable
SCALAR	`T*s_rep -> V`	scalar constant
FUNCALL	`id*(T*V)list -> V`	function call
AVAR	`id*S -> V`	* array variable
LOCALLY	`S*int -> V`	* coord array constructor
IMPERATIVE FACET (I)		
PROGRAM	`I->I`	top-level program action
SEQUENTIALLY	`I list ->I`	sequential composition
CONCURRENTLY	`I list ->I`	concurrent composition
MOVE	`(V*V)list -> I`	data movement
MMOVE	`(V*(V*V))list -> I`	move 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	`D*I -> I`	execute in extended environment
SKIP	`I`	defines (SEQUENTIALLY nil)
DO	`V list*S*I -> I`	* execute over the given shape
DOALL	`V list*S*I -> I`	* ..in parallel
SHAPE FACET (S)		
point	`int -> S`	* single point
gen_interval	`S*S*int -> S`	* strided interval
interval	`S*S -> S`	* stride-1 interval
prod_dom	`S list -> S`	* shape cross-product
dyn	`V -> S`	* dynamically determined shape
all	`S`	* select all elements

Figure 2: Partial Listing of YR Domains and Operators

The Fortran 90 code mentioned earlier,

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

might be expressed in full YR as

```
WITH (DECLSET
  [DECL('k',dfield
   {shape=prod_dom
     [interval(point 1,point 128),
      interval(point 1,point 64)],
    element=integer_32}),

   DECL('l',dfield
   {shape=interval(point 1,point 64),
    element=integer_32})],
SEQUENTIALLY
[ ...
  MOVE [(SCALAR(integer_32,'6'),
              AVAR('l',all))],

  MOVE [(COMPUTE(bin Plus,
              [COMPUTE(bin Mul,
               [SCALAR(integer_32,'2'),
                AVAR('k',all)]),
               SCALAR(integer_32,'5')]),
              AVAR('k', all))],
... ]);
```

The program consists of a declaration section, wrapped around two data movements composed sequentially. The first assigns the scalar integer value 6 to all elements of array `l` in parallel; the second computes the function `2k+6` for all points in `k` simultaneously, and moves the result back into corresponding elements of `k`. Note that the `all` operator is used for the array variable subscript to specialize different shapes, with the meaning specified by context. `All` thus allows us to express elemental parallelism in data movement decoupled from the specific shape associated with the array variables.

4 Fortran-90-Y: Target-Independent Phase

The target-independent phases of the prototype compiler include modules for syntactic analysis, initial translation to YR and static semantic analysis, and YR source-to-source optimization. Each phase is specified as a set of Standard ML modules.

4.1 Syntactic Analysis

Syntactic analysis is performed by a front end system whose target is a generic AST format. The implementation of syntactic analysis by way of formal specification is well understood, and we elide further details here. The current front end in the Fortran-90-Y prototype, for example, is a custom subset lexer/parser in the form of a ruleset guiding the accumulation of lexical tokens into simple tree form, and the recognition and tagging of a number of multistatement constructs.

4.2 Static Semantic Analysis

Following syntactic analysis, the resultant AST is matched against a set of production rules that guide the construction of a valid YR program and perform static semantic checking. These rules are grouped according to the semantic facet within which their resultant program action is classified. The current prototype includes nine rule sets that govern the transformation of ASTs into YR program fragments according to their parent semantic facets: right and left-hand side expressions (valuations), types and shapes, simple and block-structured statements (imperatives), and variable and constant declarations.

The rule sets are mutually recursive, and call each other to fill in the missing parts of parameterized actions. For example, the `DO` operator for serial iteration is parameterized by a list of value-resources that may be borrowed to serve as loop variables, a shape over which the looping occurs, and a body instruction, possibly including the loop variables, to execute at each iteration. The production rule that creates a `DO`-statement is just one clause of the rule set `AST_to_Stmt`. One pseudocode form of the rule to produce a YR `DO`-statement from an AST encoding a simple lower-bound, upper-bound loop with unit stride is

```
AST_to_Stmt(env,
  [[do /loopvar/=/lb/,/ub/ /body/]]) =>

  let val bodyaction = AST_to_Stmt(body,env)
      and lvaraction = AST_to_LHS(env,loopvar)
      and lb_shape   = AST_to_Shape(env,lb)
      and ub_shape   = AST_to_Shape(env,ub)

  in DO([lvaraction],
        interval(lb_shape,ub_shape),
        bodyaction)
  end
```

If the user's program contains such a simple `DO`-loop, the AST fragment that encodes the loop will be matched against each rule in the ruleset `AST_to_Stmt` sequentially. It will trigger this particular clause, which will in turn extract the syntactic subtrees containing

the loop bounds (point shapes), the loop index variable, and the statement that forms the loop body. The rulesets `AST_to_[Stmt, LHS, and Shape]` are used recursively to construct their semantic values. These values are then combined under the DO-operator into a single entity, which is returned as the YR compilation of the original AST.

Similarly, the LHS-rule that translates unqualified identifiers to assignable variable actions is

```
AST_to_LHS(env,[[/vname/]]) =>

(case typeof(env,vname) of
     dfield{...} => AVAR(vname,all)
   | other       => SVAR vname)
```

Note that if the type associated with the variable name in the static environment **env** is an array type (**dfield**), the rule assumes that the whole-array reference is intended, and produces the correct YR encoding for an array variable used as a left-hand side.

Finally, the rule that handles simple assignment is just

```
AST_to_Stmt(env,[[/somelhs/=/somerhs/]]) =>

let val lhs' = AST_to_LHS(env,somelhs)
    and rhs' = AST_to_RHS(env,somerhs)
in
 if types_agree(env,lhs',rhs')
 andalso shapes_agree(env,lhs',rhs')

 then MOVE[(rhs',lhs')]
 else
   raise AST_to_Stmt_Error
    ("Mismatch in assign")
end
```

Here, the functions **types_agree** and **shapes_agree** look up or derive the type and shape of the supplied YR valuation actions and trivially check them for agreement before allowing the compilation to proceed. [2]

The final result from this phase should be a single imperative action that describes the entire user computation, and has been statically type- and shape-checked. No attempts at program optimization have been made. The YR action can then be passed through a series of optimizing transformations, or directly to a target-specific YR compiler for code generation.

4.3 YR Transformations

The current Fortran-90-Y optimization stage performs a chain of source-to-source transformations over YR code. This framework supports program transformation within each semantic facet, propagating the effects of transformations through the program by way of YR's bridging operators, where facets meet. The object is to produce programs in which computations over like shapes are blocked as much as possible, forming computation phases 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 such as exploiting communication patterns that are well-supported by the target.

Each is implemented as a single Standard ML functor, defining a transformation function whose type is `YR.statement -> YR.statement`. Each has a set of input assertions that must be satisfied, or the transformation fails, displaying a warning of assertion failure and passing the input program through unmodified. Like the static semantic phase, each transformation function is defined as a rule set partitioned along semantic facets. Generally, each transform is triggered by a particular context or YR construct, and so can be defined in just a few hundred lines of SML code.

The selection and ordering of these optimization modules are currently statically determined at the time the compiler is built to satisfy whatever input assertions the back end YR compilers may have, and to precondition code for good performance.

4.3.1 Constant propagation

The constant propagation module performs simple value tracking to simplify expressions in crucial contexts. This is primarily useful in statically resolving otherwise compile-time indeterminate subscript ranges, and thus can greatly extend the potential for later optimization. For example, if the user codes

```
B(2:np1) = A(2:np1) + B(1:n)
```

and variable **np1** is clearly only used to store the value **n+1** for some parameter **n**, then rewriting the array expressions to include that information will resolve a dynamic shape checking dilemma at compile time.

Simple value tracking is accomplished by passing over the input YR program with an environment that stores, for each variable, either a current constant valuation, an uninitialized marker, or an unknown status. This simple tracking does not cross the boundaries of basic blocks.

4.3.2 DOALL reduction

Unlike the Fortran 90 standard, but like most compilers that implement a subset of Fortran 90 features, the

[2] Nontrivial shapes, including arrays and intervals with runtime-determinate bounds, also exist in YR in the form of the **dyn** operator for dynamic shapes. Certain shape checking operations may therefore be postponed until runtime.

Fortran-90-Y prototype implements a FORALL statement, though the YR operator is named DOALL. Perhaps its most significant use is to allow pointwise computations in which the local array coordinates participate.

The subset of FORALL expressions that involve only local array elements and coordinates are detected and reduced by this transformation module to whole-array expressions that use the YR value-constructor LOCALLY. For example, the Fortran 90 statement

```
      forall (i=1:m, j=1:m)
    c   A(i,j) = i*B(i,j)+j
```

where A and B were defined over the common shape S equal to (1:m)*(1:m), would emerge from the static semantic phase as the YR fragment

```
DOALL([SVAR i, SVAR j], DOMAIN "S",
 MOVE

 [COMPUTE(bin Add,
   [COMPUTE(bin Mul,
    [AVAR("B",prod_dom
       [dyn(SVAR "i"),dyn(SVAR "j")]),
     SVAR "i"]),
    SVAR "j"]),

 AVAR("A",prod_dom
  [dyn(SVAR "i"),dyn(SVAR "j")])])
```

Although the individual shapes can be inferred, the exact subscript values remain unspecified, leaving complicated source and target expressions for the MOVE. The problematic subscript expressions each exactly match the list of variables that the DOALL borrows to store loop indices, triggering the DOALL reduction module. This transformation introduces parallel YR temporaries of the same shape and distribution as A and B using the YR LOCALLY operator, and reduces this YR fragment to the simpler pointwise computation

```
MOVE

[COMPUTE(bin Add,
  [COMPUTE(bin Mul,
    [AVAR("B",all),
     LOCALLY(S,dimension 1)]),
  LOCALLY(S,dimension 2)]),

 AVAR("A",all)]
```

4.3.3 Communication introduction

Expressions that involve mixed communication and computation must be broken into separate pieces executed serially, often with temporary storage to hold intermediate results. This module passes over the input YR program, using a rule set to detect such expressions and perform the necessary splitting transformations. For example, if the module sees YR code expressing

```
A = B + CSHIFT(B,DIM=2,SHIFT=2)
```

such as

```
MOVE

[COMPUTE(bin Add,
   [AVAR("B",all),
    PRIM("cshift",
      AVAR("B",all), ...)]),

  AVAR("A",all)]
```

it converts this to multiple moves with a local declaration

```
WITH(DECL("t0",dfield{shape=S,...}),
SEQUENTIALLY

[MOVE
 [PRIM("cshift",
      AVAR("B",all), ...),

    AVAR("t0",all)],

 MOVE
  [COMPUTE(bin Add,
     [AVAR("B",all),AVAR("t0",all)]),

    AVAR("A",all)]])
```

that computes the CSHIFT into locally scoped storage t0 of the appropriate shape before performing the addition and (presumably) releasing the local storage. Later transformation phases may choose to bring some outside computations within the scope of the generated temporary storage, increasing the length of the elemental code block at the expense of keeping some allocated heap space longer than necessary.

4.3.4 Parallel loop fusion

Finally, the loop fusion module passes over the YR program, looking for successive MOVE statements with purely elemental computation and like shape. Those it finds it groups together within a single MOVE set. In the CM/2 YR implementation, this results in these assignments being compiled into a single PEAC node routine, potentially resulting in increased vectorization. This is equivalent to fusing sequences of parallel loops, since in a synchronous context

```
SEQUENTIALLY[DOALL(V,S,I1),DOALL(V,S,I2)]
```

is exactly equivalent to

```
DOALL(V,S,SEQUENTIALLY[I1,I2])
```

5 Fortran-90-Y: Target-Specific Phase

5.1 CM2/YR Compiler

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

The source YR program will have been restructured by the optimization phase to consist of blocked computation and communication phases. The CM2/YR compiler passes over this series of blocks, cutting out and compiling a list of the elemental computation phases. Then it patches the remaining program skeleton with the appropriate YR calling code. Each extracted computation phase is passed to the CM/2 node YR compiler to be turned into a node procedure. The remainder is transformed into native frontend code by the CM/2 front end compiler.

5.2 CM2/Host and CM2/Node YR Compilers

The CM2/Host YR compiler translates the YR remainder progam into SPARC assembly code plus CM runtime system library calls. **DO**- and **MOVE**-constructs over serial shapes become explicit front-end iteration. References to front-end data, along with CM data used in a front-end context all become front-end code. Declarative YR constructs become memory allocations, with their home determined by usage. Certain primitive function calls that 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 node processors. The Node/YR compiler then passes over the chain of extracted loopnests, transforming each straightforwardly into a single PEAC node routine. An illustration of the code partitioning process can be seen in Figure 3.

The host and node compilers in the current CM/2 implementation represent the point where true specification ends. They compile YR fragments to native code directly, using hand-coded algorithms to achieve competitive performance. The formalism of YR is still

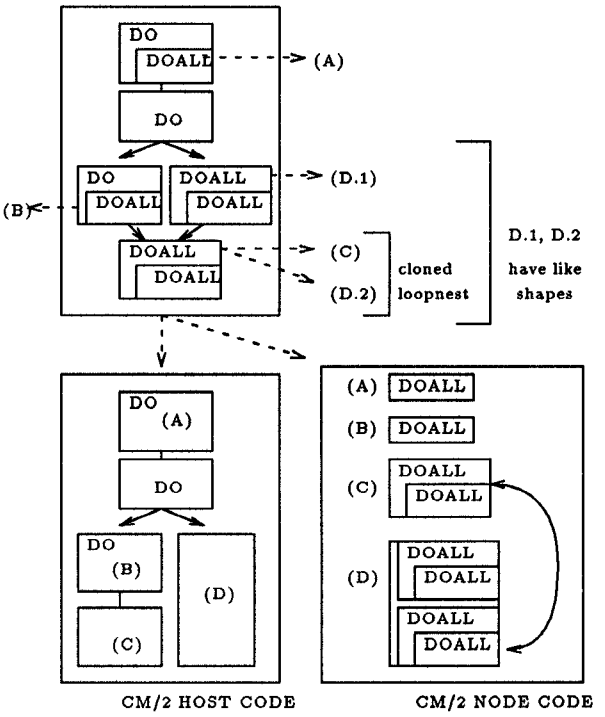

Figure 3: Decomposition to Host and Node Code

present, however, insofar as these implementations are organized internally at the top level as production rule sets. The CM2/Host compiler, for example, defines functions `compile_[i, v, d]action` that compile YR fragments from the imperative, valuation, and declarative semantic facets, respectively, to SPARC assembly code. The CM2/Node compiler performs a similar function, translating a narrow set of pointwise parallel loop constructs into optimized slicewise PE assembly code. Pushing the level of specification down into the FE and PE compilers is an ongoing process, guided by this facetwise organization.

Recall that earlier compiler stages defined the translation of AST fragments to YR fragments, and synthesized a single YR imperative program from them. The backend rulesets perform componentwise decomposition, rather than synthesis, of the constructed YR imperative, producing an equivalent program in the target language.

Having these separate compilers for the host and nodes allows effort to be expended intelligently to bring the prototype up to speed. In codes that contain a high degree of data parallelism, the node processor and runtime libraries' speeds are the limiting factor for performance. The SPARC front end can easily keep up, feeding PEAC code and data through the sequencer and calling the runtime system. As problem size increases, front end time comprises a negligible fraction of the overall execution profile. This allows SPARC

front-end performance questions to be postponed. The current front-end YR 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/2 node compiler is tuned for optimizing the virtual subgrid loop over the local data in each processor. Since the role of the processing elements is limited in this programming model to purely local, pointwise computations, the set of YR programs that the node compiler accepts can be restricted. Each computation burst is derived from computations over like shapes in the source program. The compiler therefore only needs to be able to 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. The node YR compiler emits an optimized PE assembly code routine for each subgrid loop passed to it by the CM/YR code partitioner.

After the host and node compilers have reduced their YR 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 CM5/YR Compiler

The initial CM/5 YR compiler retains the majority of its structure and, therefore, its specification from the CM/2 version. To use the CM/5 in SIMD mode to run codes written for the CM/2, the Fortran-90-Y prototype shares all code in common up to and including the CM/2 whole-machine YR compiler, which splits a single program into host and node code.

The CM/5 host compiler required less than 200 lines of additional code to handle both machines' front end requirements. A CM/5 node compiler that translates old node YR to regular C programs similarly allows CM/2 node code to compile transparently to the new machine. As a result, the total additional time required to construct this preliminary CM/5 prototype compiler was less than two weeks.

Compiling to a CM/5 augmented with optional vector hardware at each node will present more target levels due to the increased node complexity. In the new model a single YR 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 1. Most importantly, the new system will still be able to take advantage of the machine-independent block-lengthening YR transformations defined in the common compiler modules.

5.4 Other Computation Models

There are, in practice, no reasons 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 occasionally 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.

Likewise, a more flexible compiler will be able to take advantage of the MIMD features available in the CM/5 and other machines. Implementing new programming models only requires the specification of new host and node YR compilers, and adjustments to the top-level compiler to make use of them. The YR source transformation stage would 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 initial research into 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 CM/2 compiler began generating code for simple test programs by late October, and was compiling benchmark code by November. We gained access to a working CM/5 in February, and began compiling simple codes to it as well shortly thereafter.

Direction of Effort. By modeling the CM with a unified phase (CM/YR), we recognized that the machine taken as a whole is still a useful abstract target for compilation. But by dividing the labor of compilation between the 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 multiple target machines, and an interesting subset of the source language, in less than eight months.

Compiler Performance. The time and space requirements for building and using generated compilers have been traditional obstacles to their acceptance for production use. By constrast, on a moderately loaded Sun 4/390 with 32 MB of physical memory, the current prototype CM/2 compiler can be generated from specification source code in about ten minutes. During actual development, rebuilds generally take much less time – on the order of two or three minutes – due to specification modularity and Standard ML's separate compilation facilities. The resultant CM/2 compiler image is just over one megabyte in size; this compares favorably with other language tools available for the Connection Machine. Object files produced by the generated prototype system are, on average, close in size to those produced by CM Fortran. Compilation rates are generally within twenty percent of CMF as well.[3]

Object Code 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 2.77 gigaflops. The prototype Fortran-90-Y compiler, after the first eight months of development, produced code that attained a competitive sustained rate of 2.71 gigaflops. The initial CM/5 prototype, using untuned C program stubs for expressing node code, still attains roughly 85% of the gigaflop rate of CM Fortran on the CM/5.

7 Conclusions

In this paper we have presented a brief overview of an approach to parallelizing compiler design incorporating concepts from both formal specification and traditional high-performance computing. The original motivation of this work was to demonstrate that developing scalable, portable, competitive compilers from specification 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 on their way to being generally applicable to the real-world challenge of developing compilers for complex parallel targets at reasonable cost.

[3] Exclusive of linking time; as linking can comprise two-thirds or more of overall compile time, practical compilation rates are thus almost identical.

8 Acknowledgements

The authors thank Young-Il Choo for his advice regarding earlier versions of YR, 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.