Code Composition as an Implementation Language for Compilers

James M. Stichnoth and Thomas Gross
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Code composition is an effective technique for a compiler to implement complex high-level operations. The developer (i.e., the language designer or compiler writer) provides building blocks consisting of sequences of code written in, e.g., C, that are combined by a composition system to generate the code for such a high-level operation. The *composition system* can include optimizations not commonly found in compilers; e.g., it can specialize the code sequences based on loop nesting depth or procedure parameters. We describe a composition system, Catacomb, and illustrate its use for mapping array operations onto a parallel system.

1 Introduction

Application-specific programming languages capture properties of a particular problem domain. For many such problem domains, it is not necessary to design and implement a completely new language. Instead, a programming language like C or Fortran serves as the base and is augmented by operators or control constructs to capture specific information about a problem domain. For example, Fortran and C have been extended with array assignment statements and abstractions for computer vision [10, 28], or parallel looping or synchronization constructs have been added. These extensions can usually be expressed in the form of high-level operators. The benefits of such an extension are obvious: the extensions address the concerns of the problem domain, and the general-purpose language can be used for everything else.

There are two challenges in implementing such high-level operators. First, the implementation of such extensions in the compiler must be cheap, with respect to the time and effort that is required. Otherwise, the language may never be used due to the lack of a decent implementation. The second challenge is to devise a high-quality implementation. There are many dimensions of quality, but the two most critical are correctness and efficiency. While correctness is crucial for every language translator, one aspect of correctness that has sometimes been overlooked in the past is completeness; i.e., the translator must be able to deal with all possible legal inputs, and the implementor must cover and test all the boundary conditions. Efficiency is also important, since one of the reasons an application-specific language is used is that such a language can produce better-performing code. Unfortunately, these two demands (correctness and efficiency) tend to increase the cost of implementation, so there is increased interest in techniques to address these problems.

A straightforward implementation of such language extensions is to provide a subroutine for each of the new operators. Since these operators are at a high level and provide powerful operations, the implementation of such a library is far from simple. Furthermore, using a library either deprives the system of opportunities to optimize the code (if the library handles only the general case) or results in many variants (for different parameter values).

A composition system offers an attractive alternative. The developer of the high-level operations provides code sequences that are "stitched together" by a composition system. An analogy to conventional compilers may illustrate the concept. A compiler takes assembly-language or intermediate-code sequences, which have been determined as code for a statement or operator, and optimizes these low-level code sequences. The compiler discovers redundant operations and manages resources (e.g., registers) across boundaries. A composition system takes blocks of code in some suitable high-level general-purpose language and composes the code for the high-level operation by combining and optimizing the code sequences. Since the composition system sees a global view of the program, it can optimize these code sequences better than a conventional compiler.

In a conventional compiler, the compiler developer decides how the assembly language or intermediate code sequences are selected and optimized. However, to support a wide range of high-level operations in a composition system, the developer of the application-specific high-level operations must be able to express a wide variety of actions by the composition system. Thus the code composition must be programmable; i.e., there must be a programming language to control code composition. This programmability provided by the composition language is the key to the power of the composition system. In the remainder of the paper, we first provide a few examples of what we call "complex high-level operations" from different problem domains. Then we discuss code composition as a technique to address the two challenges mentioned above. Then we describe the Catacomb system that has been implemented and supports our claim of the practicality of the idea of code composition. There are other approaches to address the problem, and we summarize those after we present an evaluation of the Catacomb system for one class of high-level operations.

2 Examples of complex high-level operations

This work is motivated by the challenges faced in compiling "complex high-level operations." In this section, we briefly present three examples of complex high-level operations. They are described in more detail in Section 5.

2.1 Array assignment statement

The array assignment statement, which is a key component of High Performance Fortran (HPF) [10], effects a parallel transfer of a sequence of elements from a source array into a destination array. The canonical form of the array assignment statement is

$$A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B].$$

The *subscript triplet* notation $\ell\colon h\colon s$, also used in Fortran 90, describes a sequence of array indices starting with ℓ, with stride s, and having an upper bound h.

An important aspect of the array assignment statement is that the array elements are distributed across the processors of a multiprocessor system. This data distribution dramatically increases the complexity of an efficient algorithm to execute the statement.

Compiling the array assignment statement becomes much more difficult when we extend the canonical case to the general case. There are three extensions: multidimensional references (e.g, $A[1\colon m][1\colon n]$), multiple right-hand side terms (e.g., $B[\ell_B\colon h_B\colon s_B] + C[\ell_C\colon h_C\colon s_C]$), and a mix of subscript triplets and scalar indices (e.g., $A[x][\ell_A\colon h_A\colon s_A]$). While it is conceptually simple to lift the restrictions on the canonical example, it is much more difficult in practice to implement the general case within a compiler or runtime library framework. As such, most proposed implementations ignore the engineering issues of embedding general array assignment algorithms into a parallelizing compiler.

2.2 Data transfer in irregular applications

The array assignment statement is used to perform *regular* data transfer, in which the access pattern (as specified by the subscript triplet) and the data distribution both have regularity that can be exploited at compile time. When the data distribution becomes *irregular*, or the access pattern becomes irregular, the data transfer depends on values available only at run time. Irregular access patterns result from multiple levels of array indirection; e.g., $A[IA[\ell\colon h\colon s]]$. Irregular access patterns also result from explicit loops containing multiple levels of indirection.

Irregular array assignment statements and irregular parallel loops are examples of complex high-level operations. Engineering difficulties arise in a compiler when we add multiple levels of indirection, and when we distribute several dimensions of a multidimensional array.

2.3 Archimedes

The Quake project [4] at Carnegie Mellon focuses on predicting the ground motion during large earthquakes. At its heart is Archimedes [19], a system for compiling and executing unstructured finite element simulations on parallel computers. The compiler component of Archimedes, called Author, presents the programmer with a language nearly identical to C. The language is enhanced with additional aggregate data types for nodes, edges, and elements of a finite element mesh, as well as statements for iterating in parallel over the collection of such objects in the mesh (e.g., `FORNODE` and `FORELEM`). In addition, Author provides a crude mechanism that allows the programmer to extend the system with macro-like constructs that support type-checking of their arguments.

The statements that iterate in parallel over the aggregate data types are examples of complex high-level operations. In addition, there are high-level constructs that result in both regular and irregular communication, as described above.

3 Code composition

Traditionally, the compiler translates each input operation or statement into a small fixed sequence of statements at a lower level of abstraction (e.g., machine instructions or operations in an intermediate representation). For each input construct, there is typically a small sequence of instructions to perform the task at execution time. This compilation strategy is called *custom code generation*.

As a high-level language grows to be more complex, the complexity of individual operations increases as well, requiring more and more low-level operations to implement each complex input construct. At this point, the typical approach is to shift to the *runtime library routine* compilation strategy. In this strategy, the compiler translates each such high-level operation into a call to a runtime library routine. This approach tends to be much more manageable than generating custom code, because the code appears in a straightforward fashion in the library, rather than being buried in the compiler. However, it also tends to suffer in terms of runtime performance, because the runtime library routine does not have access to the specific parameters of the construct that are known at compile time, and cannot optimize accordingly.

The compilation strategies of custom code generation and runtime library routines trade off three important issues: efficiency, maintainability, and generality. Efficiency refers to the performance of the generated code at run time; i.e., being able to optimize the construct's runtime execution, based on all available compile-time information. Maintainability refers to how easy and straightforward it is to develop and maintain the algorithm, within the framework of the compilation system. Generality refers to whether the general case or merely a simplified canonical case is implemented.

Our solution is a technique called *high-level code composition*. Code composition is an approach related to custom code generation. The code sequences to be produced, called *code constructs*, appear external to the compiler. The instructions for piecing together the code sequences also appear externally, rather than being embedded in the compiler. These instructions are called *control constructs*. Code constructs and control constructs are bundled together into manageable-sized chunks called *code templates*, in the same way that the code in a typical program is a collection of manageable-sized functions. The code templates form a specialized language for directing the compilation process.

There is a *composition system* coupled with the compiler that uses these code templates to produce code. Figure 1 shows how the composition fits in with respect to the rest of the compiler. This structure is important for code reuse: the same composition system can be used in several different compilers, for several different problem domain. The composition system's function is to *execute* the code templates at compile time. Executing the code templates means following the instructions specified by the control constructs. Because the control constructs constitute a programming language, the composition system can be thought of as an interpreter of the control constructs.

The composition system is invoked by the compiler, which instructs the system to execute a template on a particular input. This input is a high-level programming language operation, such as an array assignment statement. The composition system then executes the template with full knowledge of all compile-time information (for the array assignment, information like number of array dimensions, dimension sizes, and distribution parameters). It uses this knowledge to produce the correct code for the input and to optimize the code.

The control constructs constitute a small language that the composition system interprets at compile time. As such, they must be designed as one would design a real programming language, containing variables, conditionals, procedure calls, and so forth. Furthermore, it is important to choose a syntax that is easily distinguishable from the syntax of the code constructs. There are several features that should be present in the control constructs: control procedures and procedure calls, control variables, control variable assignment, a control test, a control loop, and a concept called *variable renaming*. All of these features but the last are fairly self-explanatory. Variable renaming is a subtle point that is easy to overlook, but is important in practice. Often we need to compose the same template several times, but each composition needs a different set of variable names. For example, we might want to recursively compose a basic "loop" template several times to create a loop nest, but each loop induction variable name must be unique. To allow this, a variable renaming operator allows new variable names to be constructed during template execution, similar to Lisp's `gensym` function, only more controllable.

There is also the question of what, if any, relationship the control and code constructs should have toward each other. There are two styles in which the constructs can be interleaved: the *syntactic* style and the *lexical* style. With the syntactic style, code constructs and control constructs are required to fully nest within each other. With the lexical style, there is no such requirement.

Figures 2 and 3 demonstrate these two styles for writing templates, assuming C-like control constructs and Fortran-like code constructs. The purpose of the code in the figure is to produce an n-deep loop nest, where n is a parameter passed to the template. The template would be invoked through the control construct `call_template(loopnest, n)`. The lexical style uses a control loop to generate the `DO` statements and the matching `END DO` statements. The syntactic style has to use a recursive template to form a loop nest, calling itself recursively between the `DO` and the `END DO`.

It is preferable for the code and control constructs to interact through the syntactic style. The primary benefit of the syntactic style is readability of the template code.

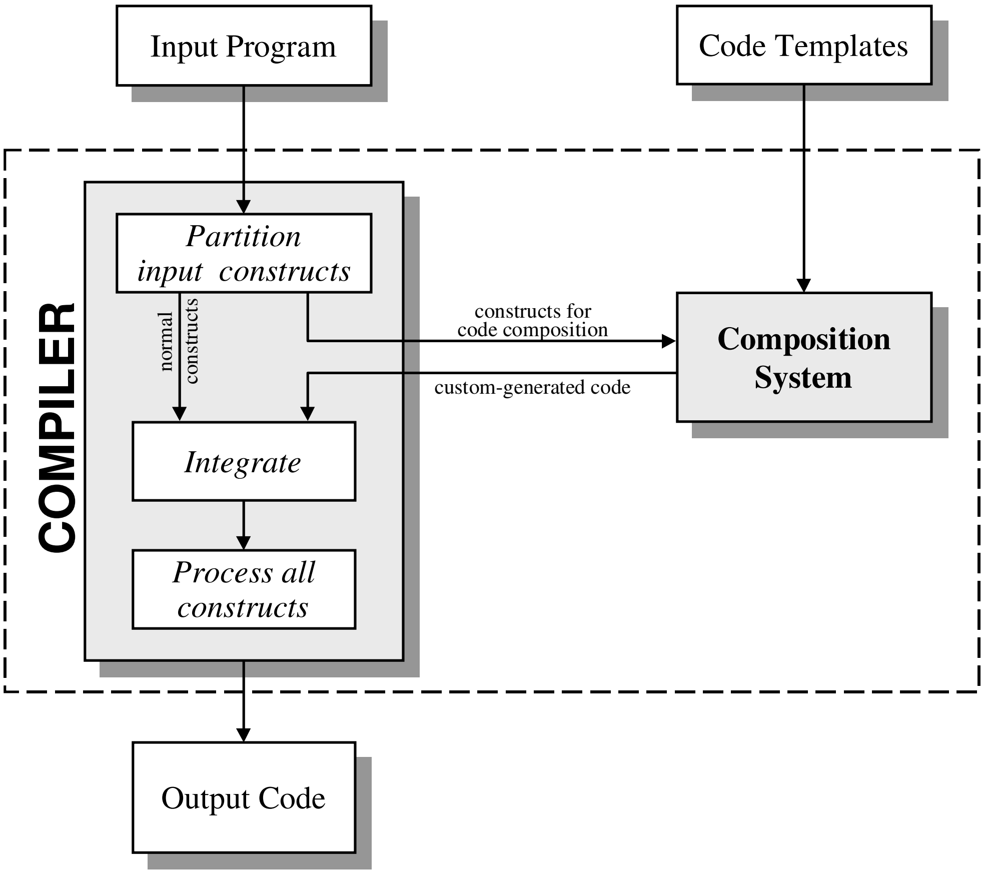

Figure 1: Integration of a composition system into a compiler.

```
TEMPLATE loopnest(depth)
{
  call_template(loopnest1, 0, depth);
}

TEMPLATE loopnest1(cur_depth, max_depth)
{
  if (cur_depth < max_depth) {
    DO I(cur_depth) = 1, 10
      call_template(loopnest1, cur_depth+1, max_depth);
    END DO
  } else {
    /* inner loop code goes here */
  }
}
```

Figure 2: A template for constructing a loop nest, using the *syntactic* style.

```
TEMPLATE loopnest(depth)
{
  for (count=0; count<depth; count++) {
    DO I(count) = 1, 10
  }
    /* inner loop code goes here */
  for (count=depth-1; count>=0; count--) {
    END DO
  }
}
```

Figure 3: A template for constructing a loop nest, using the *lexical* style.

It is easier to develop and maintain code written with this style, and it is also easier to automatically detect syntactic mistakes in the template code. Using the lexical style, it is much easier to make mistakes in matching up the syntactic constructs in the generated code (e.g., the `DO` and `END DO` in a Fortran loop). But with the syntactic style, syntax errors are obvious when the code or control constructs do not match up correctly, and can be easily detected when the composition system parses the templates.

It is important to stress that a composition language is *not* meant to be programmed by the end user. Rather, the composition system is a tool used by the *compiler writer* to facilitate the translation of complex high-level operations into lower-level code.

4 Catacomb

To illustrate the usefulness of a composition system, we have developed Catacomb, a composition system for generating C code. As such, its code constructs have the syntax and semantics of C constructs. The control constructs are C-like, but since they interleave with the code constructs using the syntactic style, the syntax is slightly different from C.

To illustrate the combination of code and control constructs, Figure 4 shows an annotated set of Catacomb templates. Its purpose is to construct a loop nest for setting the elements of a multidimensional array. Below the set of templates is a sample invocation and its corresponding result. The example consists of three templates:

- `loopnest`: The entry point. It takes three input arguments: the number of array dimensions (equivalent to the depth of the loop nest to be produced), the array whose elements are to be set, and the size of the array dimensions (for the sake of simplicity, all dimension sizes are assumed to be equal). The template verifies that `n`, the number of array dimensions, is a compile-time constant, and then calls the recursive `loopnest1` template.

- `loopnest1`: The recursive template. This template generates the outer loop, and then calls itself recursively to generate the rest of the loop nest. When it reaches the innermost loop, it generates the inner-loop assignment statement.

- `genlist`: Creation of the array subscript list. Because the number of array dimensions is an input parameter to the `loopnest` template, the subscript list for the inner-loop array reference has to be generated each time the template is called. The `genlist` template is responsible for building the subscript list.

This set of templates illustrates all of Catacomb's control constructs. Also in the figure is a sample invocation of the entry template, and the resulting C code. This is a complete working example of a Catacomb template and its output, with the caveat that the actual declaration of the input array is omitted; the array is assumed to be declared elsewhere.

The template header declares the template and its arguments, as well as template-local control variables, syntactically similar to a C function declaration. Control variables can be set using the `:=` control assignment operator. The `include` statement executes a control function call, passing a list of arguments to a template. By default, arguments are passed by value; the `var` keyword in the template argument declarations allows arguments to be passed by value-result, making it possible for a template to return results to the caller. The `cif` and `cwhile` are the control conditional and control loop, respectively; the condition must evaluate to a compile-time constant.

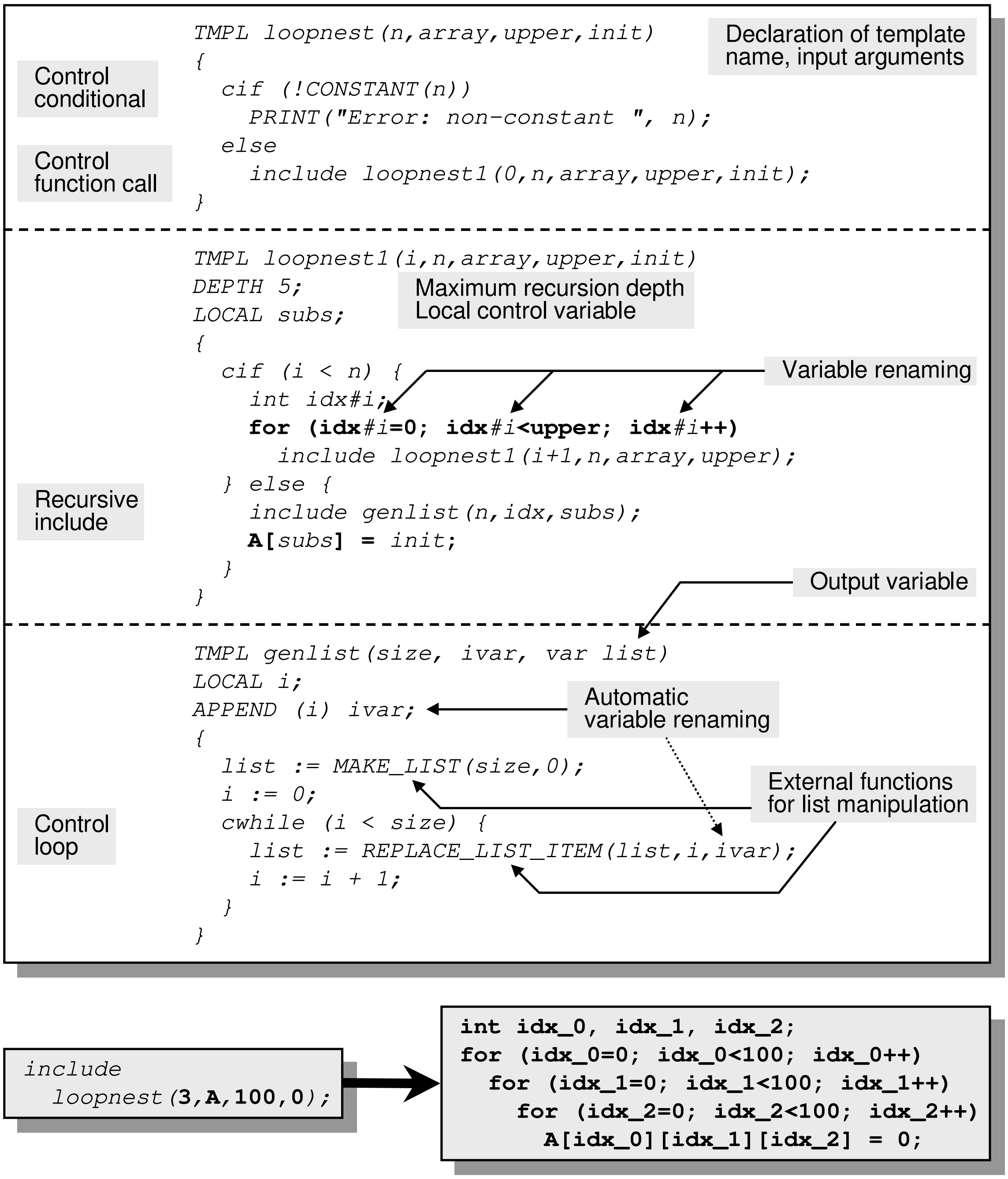

Figure 4: A set of Catacomb templates for constructing a loop nest that sets values in a multidimensional array. Also depicted are a sample invocation and the corresponding result.

Catacomb introduces the # operator for variable renaming. For example, `x#3` evaluates to the variable `x_3`, `y#4#5` evaluates to the variable `y_4_5`, and `foo#"bar"` evaluates to the variable `foobar`. (When the right operand is an integer constant, Catacomb also inserts an underscore character, "_", to help avoid variable name conflicts.) To aid in the conversion of library routines into templates, Catacomb provides the `APPEND` statement in the template header. With the statement `APPEND (i) x;` in the header, every occurrence of the variable `x` in the template body is automatically replaced with `x#i`.

In addition, Catacomb provides a number of control functions, called *external functions*, to perform operations not possible using the basic C operators on which the control constructs are based. For example, `CONSTANT` is used to test whether the input evaluates to a compile-time constant. There is a default set of external functions in Catacomb, and the set is easily extended (e.g., to add functions that query the distribution parameters of HPF arrays).

Catacomb implements several global optimizations [1, 8], as well as some nonstandard optimizations based on *bounds analysis*. Bounds analysis is based on the observation that sometimes, even though the compiler cannot determine a specific value for a variable or expression, it can determine that it must fall within a certain range of values. Catacomb uses bounds information to simplify expressions where possible. It uses copy propagation techniques to propagate the bounds across assignment statements. Catacomb also extracts bounds information from `if` conditions where possible. Because standard C optimizers do not implement bounds analysis, and some bounds information is available only within Catacomb (e.g., the number of processors in an array distribution is always positive), Catacomb needs to include these optimizations.

There is an additional issue related to the Catacomb implementation: the execution model of the interaction between control constructs and global optimizations. The most obvious and straightforward way to integrate them is to make them completely independent. This suggests a two-phase execution approach: in the first phase, Catacomb executes only the control constructs, leaving just the code constructs, and in the second phase, the resulting code constructs are passed off to the global optimizer, and then emitted. An attractive feature of the two-phase approach is the simplicity in both semantics and implementation.

Unfortunately, under this model, code composition decisions can only be made based on the values of control variables. For example, consider the following template code:

```
x=1; cif (x==1) { ...  }
```

Note that `x` is a code variable, not a control variable. Under the two-phase execution model, even though it is obvious that the value of `x` is always 1 at the `cif` site, the control constructs are executed before the global optimizations, and thus the `cif` has no way of knowing that the value of `x` is 1 at that point.

There are several alternative template programming styles, execution models, and semantics for allowing composition decisions to be made based on the values of code variables (these are discussed in greater detail elsewhere [20]). Most are insufficient and/or have unacceptably confusing semantics under different circumstances. The best alternative, although naturally the most difficult to implement, is a single-phase execution model, in which global optimizations are performed at the same time as the control execution. The single-phase model improves the efficiency of the generated code; the implementation details are beyond the scope of this paper.

5 Complex high-level operations

In this section, we consider two examples of complex high-level operations, and demonstrate why code composition is superior to the standard compilation techniques for these operations.

5.1 Array assignment statement

The array assignment statement is nontrivial to execute because of two properties: the elements of the arrays are distributed across different processors, and the sequence of elements indexing each array can be an arbitrary arithmetic sequence. Evidence of the complexity and importance of the array assignment statement can be found in the number of different algorithms that have been proposed for executing it efficiently [21, 11, 7, 13, 26, 14, 2, 25]. The compact syntax that hides the complexity of an efficient and complete implementation is what makes the array assignment statement particularly interesting.

The canonical form of the array assignment statement is

$$A[\ell_A : h_A : s_A] = B[\ell_B : h_B : s_B].$$

The statement is equivalent to the pair of sequential loops shown in Figure 5.

HPF arrays are allowed to have a *block-cyclic* distribution, in which fixed-size blocks of array elements are distributed to the processors in a round-robin fashion (see Figure 6). Two parameters characterize this distribution: the block size b and the number of processors P. The distribution defines an *ownership set* for each processor, which is the set of array elements mapped to

```
j = ℓ_B
DO i = ℓ_A, h_A, s_A
  T[i] = B[j]
  j = j + s_B
END DO
- - - - - - - - - - - -
DO i = ℓ_A, h_A, s_A
  A[i] = T[i]
END DO
```

Figure 5: Sample sequential code for the array assignment statement $A[\ell_A\!: h_A\!: s_A] = B[\ell_B\!: h_B\!: s_B]$.

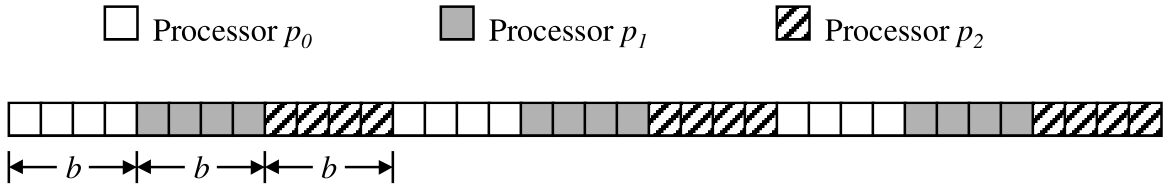

Figure 6: A block-cyclic data distribution in HPF, with three processors ($P = 3$) and block size $b = 4$.

the processor. Extremal cases of the block-cyclic distribution include the block distribution, in which a single, suitably large block is distributed onto each processor, and the cyclic distribution, in which the block size is 1. For the canonical array assignment statement, the challenge is to efficiently enumerate the *communication set*, the set of array elements to be transferred between a given pair of processors. The subscript triplets and the two processors' ownership sets determine the communication set. Implementations of the published algorithms for the array assignment require on the order of hundreds of lines of C code. For this reason, the array assignment is considered to be a complex high-level operation.

Compiling the array assignment statement becomes much more difficult when we extend the canonical case to the general case. There are three extensions: multiple right-hand side terms (e.g., $B[\ell_B\!: h_B\!: s_B]+C[\ell_C\!: h_C\!: s_C]$), multidimensional references (e.g, $A[1\!: m][1\!: n]$), and a mix of subscript triplets and scalar indices (e.g., $A[x][\ell_A\!: h_A\!: s_A]$). While it is conceptually simple to lift the restrictions on the canonical example, it is much more difficult in practice to implement the general case within a compiler framework.

Embedding an array assignment algorithm into a compiler severely degrades the *maintainability* of the algorithm. It becomes difficult to write the code within the compiler framework, and equally difficult to read and modify the code. For this reason, it is tempting to use the runtime library routine approach, writing the array assignment algorithm in a straightforward fashion in a runtime library.

Unfortunately, efficiency and generality suffer under the runtime library routine approach. Efficiency suffers because in general, it is no longer possible to compile all parameters that are known at compile time into the library. For the canonical array assignment statement, there are 14 parameters: 3 parameters for a subscript triplet, and 4 parameters for the distribution (block size, number of processors, array size, and processor number), all of which is multiplied by two because the statement operates on two arrays. Usually, most of these parameters are known at compile time, allowing more efficient code to be generated.

Generality suffers in a runtime library as we extend the canonical array assignment to the general case. This generality has three aspects: multidimensional arrays, multiple right-hand side terms, and scalar subscripts (as opposed to subscript triplets). Implementing the array assignment for multidimensional arrays requires iterating over a loop nest whose depth is proportional to the number of dimensions. There are two ways to implement such a loop nest in a runtime library. The first is to write a separate routine for each dimensionality, but this method imposes a limit on how many dimensions can be handled in an array, and it creates an exponential code explosion in the amount of code to write in the library. The other method is to write a function that recursively calls itself for each successive level of the loop nesting. However, this method lowers runtime efficiency by precluding function inlining and by imposing additional overhead in the inner loop.

Code composition is easily used to handle multidimensional arrays. Using a recursive template like that of Figure 4, the composition system generates a loop nest customized for the specific input array assignment statement. There are then no limits on the number of dimensions, nor is there extra inner-loop overhead. In addition, there are no exponential code explosion problems reducing maintainability.

A runtime library routine that handles multiple right-hand side terms is even more difficult to write than one for multidimensional arrays. Such a library must be able to handle an arbitrary set of arithmetic operators describing the right-hand side expression. Without runtime code generation techniques, the inner loop in which the computation is performed is bound to have a great deal of overhead. On the other hand, using code composition, the code templates create a customized inner loop, into which the specific operators are compiled. Once again, the increased generality does not cause any additional inner-loop overhead.

Although scalar subscripts generally do not cause efficiency problems in a runtime library, handling them generally decreases the maintainability of the runtime library. In fact, as each aspect of generality is added to a runtime library, there is a corresponding decrease in maintainability of the library. Code composition, on the other hand, allows the templates to directly manipulate subscript lists at compile time, providing generality with little impact on maintainability.

5.2 Data transfer in irregular applications

The array assignment statement is used to perform *regular* data transfer. The regularity of the transfer arises from the regularity of the subscript triplet and the block-cyclic distribution. Sparse and unstructured problems result in *irregular* data transfer, due to irregular data distributions and irregular access patterns (e.g., array references with multiple levels of indirection). In an irregular problem, the data transfer depends on values available only at run time; thus runtime analysis is required.

The key to the runtime analysis is the *inspector/executor* approach. This approach divides the runtime execution into two parts, the inspector phase and the executor phase. The inspector analyzes the global access pattern and calculates which array elements each processor needs to send, and where to send each element. The executor carries out the data transfer and the computation. The most effective means of executing irregular computations is through the use of the PARTI or CHAOS runtime libraries [24, 17, 18], developed by Saltz et al. These libraries contain sophisticated routines that help the programmer translate a sequential program into a parallel program that uses the inspector/executor approach.

More recently, several parallelizing compilers [6, 30] analyze the sequential loops in a program and automatically produce parallel loops with calls to the appropriate CHAOS routines. For many kinds of simple sequential loops, this translation is fairly straightforward. However, the translation becomes more complex as more levels of indirection in the array references are added.

For compiling irregular programs that use multiple levels of indirection in distributed arrays, Das, Saltz, and von Hanxleden use a technique called *slicing analysis* [9]. The idea behind slicing analysis is that for each loop containing a distributed array reference with multiple levels of indirection, that loop can be rewritten as several loops, each of which contains only a single level of indirection. The resulting program, containing only single levels of indirect array references, is then amenable to parallelizing techniques in existing compilers for irregular problems.

Because of the complex structure of an irregular parallel loop, the loop itself is not amenable to being implemented purely with a runtime library routine. Instead, a compiler is needed to translate the loop into a sequential loop, possibly with calls to a support library like CHAOS. The complexity of the code to be generated usually leads the compiler writer to sacrifice some amount of generality in the solution. For example, many compilers only allow a single level of indirection in array references, and existing compilers only allow a single dimension of an array to be distributed.

6 Evaluation

Along with our implementation of Catacomb, we developed templates to implement several solutions to the array assignment statement. These algorithms are known as the CMU algorithm [22], the OSU algorithm [11], and the LSU algorithm [26]. We divided our implementation into three components: preprocessing, architecture, and algorithm. The algorithm component determines the communication sets and packs/unpacks the communication buffers. The architecture component provides an interface to the architecture-specific communication features, such as the specific method for calling the send, receive, and synchronization primitives. The preprocessing component attempts to simplify the input array assignment statement into a form that is closer to the canonical array assignment statement. This division allows an arbitrary algorithm to be combined with an arbitrary architecture to form a complete implementation.

There are three issues that we can evaluate: efficiency, generality, and maintainability. Regarding efficiency, Catacomb's aggressive optimization framework results in code whose quality is close to that of hand-tuned code. Because this paper focuses more on the software engineering issues, we omit a discussion of the performance of the generated code; details are available elsewhere [20].

Regarding generality, the code templates make it simple and straightforward to support any regular array assignment statement, containing an arbitrary number of dimensions, and arbitrary number of right-hand side terms, and an arbitrary interleaving of scalar subscripts and subscript triplets. None of the implementations of the algorithms we studied handle more than the canonical case, yet the Catacomb template mechanism enables us to automatically extend the algorithm for the canonical case to the general case. In fact, this is the first implementation of the array assignment that allows an arbitrary algorithm to be coupled with an arbitrary architecture to form a complete implementation.

Evaluating maintainability is largely a subjective task. There is, however, an objective measurement that gives a rough idea of the maintainability of our template framework for the array assignment. This measurement consists of looking at the breakdown of the template code into control and code constructs, and comparing the amount of code constructs to the amount of control constructs. One could argue that as the amount of control constructs increases, the actual code being produced (i.e., the code constructs) becomes increasingly obscured within the control constructs, and the maintainability correspondingly decreases.

Figure 7 shows the breakdown of the CMU, OSU, and LSU template code, as well as the MPI communication architecture template code. This measurement was taken after removing comments and blank lines from the templates, and should only be considered as an approximation. We consider the number of lines of control constructs, code constructs, and external support libraries (i.e., code from the original implementation that did not need to be converted to template code). The breakdown shows that the control constructs are relatively evenly matched with the code constructs. In contrast, the original implementation of the CMU algorithm in the Fx parallelizing compiler [23] required roughly 15,000 lines of compiler code. Given that the CMU algorithm itself contains less than 1,000 lines of code, we can see that the vast majority of the 15,000 lines was dedicated to compile-time control. Because the Catacomb templates contain far less code devoted to compile-time control, the implementation is far more maintainable.

7 Related work

7.1 Templates and macro processing

Code composition includes control constructs that allow generalized computation at compile time. The concept of compile-time compilation has been around for some time. A widely-used example today is the C preprocessor. Its computational power is extremely limited, though; for example, looping is not possible, either directly or through recursion. Furthermore, its decoupling from the compiler prevents anything like the single-phase integrated execution model mentioned in this document at the end of Section 4. A consequence that many C programmers may be familiar with is the inability to perform preprocessor operations like `#if sizeof(int)==4`. PL/I [15] offers a more powerful preprocessor. However, it also is incapable of a single-phase execution mode, and neither it nor the C preprocessor is equipped to perform structural queries on general expressions, a feature critical to code composition.

The C++ template system provides a simple way to generate new functions and methods, tailored to a specific data type. Veldhuizen [27] has developed a mechanism called *expression templates*, which allows the template system to compose code in more complex ways, based on the structure of input expressions. For example, with the appropriate declaration of `x` and definition of `integrate`, the statement `double result = integrate(x/(1.0+x), 0.0, 10.0);` produces custom code at compile time to integrate the function $x/(1+x)$ over the domain $0 \leq x \leq 10$. While this is an interesting way to gain compile-time control over the structure of an expression, in practice the specifications end up being overly complex and unreadable.

There are other macro extensions to C (e.g., Safer_C [16] and Programmable Syntax Macros [29]) that offer many of the same benefits as Catacomb. These systems are generally not extensible like Catacomb, and do not offer an integrated single-phase execution model, thus precluding the use of global optimizations in the macro processing decisions.

Barrett et al. [5] use the concept of templates in a numerical computation context. Templates are designed and written in a high-level language to handle specific features of iterative solvers for linear systems (e.g., sparse or dense, convergence requirements, sequential or parallel, data layout). At compile time, the system automatically finds the right set of templates to match the needs of the user. This kind of system fits well within the code composition framework we describe.

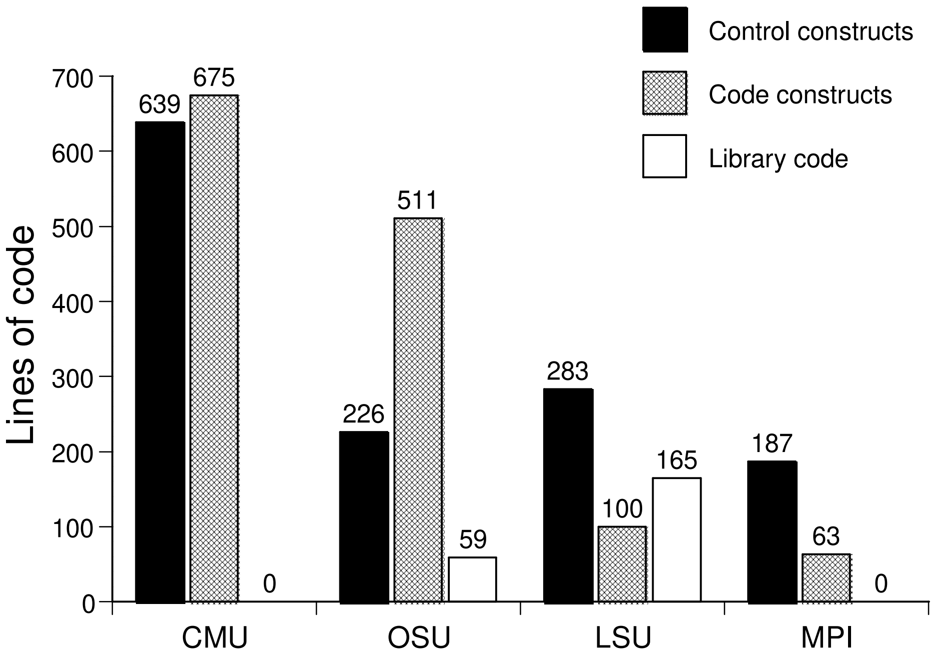

Figure 7: Comparison of the number of lines of control constructs and code constructs in the Catacomb templates for the array assignment, as well as lines of code in support libraries.

7.2 Partial evaluation

Like most optimizing compilation systems, Catacomb and the concept of code composition are related to the field of *partial evaluation* [12]. A partial evaluation system takes as input a program in a source language, and a set of known inputs to the program, and produces a *residual program* specialized for those particular inputs. The code templates are similar to a two-level version of an imperative input language. The control constructs, including the control variables, explicitly have a static binding time. The binding times of the code constructs are analyzed online using the global optimization framework, allowing some variables and statements to be classified as static, rather than the default of dynamic. Catacomb's technique of bounds analysis has some similarity to *bounded static variation*, in which the partial evaluator can restrict an otherwise-dynamic variable to a finite set of static values.

A notable difference between code composition and standard partial evaluation is the fact that control constructs follow a different flow of control from the code constructs. For example, a control assignment statement in the body of a loop is executed exactly once, regardless of the number of loop iterations at run time. This means that the straightforward translation of control constructs into their corresponding code constructs does *not* preserve the original semantics, in contrast to standard partial evaluation. Future work in this direction is to explore the issues of whether the Catacomb model (which is similar to the C preprocessor model) or the standard partial evaluation model presents the user with more "natural" semantics and ease of use, and whether there is in fact a realistic situation in which Catacomb's semantics are necessary.

7.3 Runtime code generation

Dynamic approaches attempt to improve the code at run time, and dynamic methods have lately received renewed attention (e.g., [3]). If the program notices that some parameter always has the same value, a runtime optimizer can customize the program by working with the known parameter values. Since these values are known, some tests may be resolved, or special instructions chosen, and such transformations have the potential to improve performance. However, dynamic methods too face a number of challenges: for one, the system must ensure that the overhead spent on detecting the occurrence of a common scenario is bounded and in relation to the expected benefits.

Runtime code generation should be used as an additional performance enhancement to code composition, rather than as a replacement. Optimizations should be performed in advance by the compiler whenever possible. In addition, using runtime code generation in place of code composition requires a full runtime library for the problem to be designed, which still trades off maintainability and robustness. (Efficiency is ignored, since it would be the responsibility of the runtime code generation system to provide runtime efficiency.)

8 Concluding remarks

We have identified a class of high-level languages formed by adding complex high-level operations to a base language. For these languages, traditional compilation techniques are inadequate. The traditional techniques, namely custom code generation and runtime library routines, fall short because custom code generation offers efficiency and robustness at the cost of maintainability, while runtime library routines offer maintainability at the cost of efficiency and robustness. We developed a new method called code composition and implemented a code composition system to demonstrate the practicality of this idea.

Code composition is under the control of the implementor of the complex operations, who uses a domain-specific language for directing the compilation process. Programmable code composition provides the union of the benefits of the traditional approaches: the composition system optimizes the code and thereby ensures efficiency, yet the composition templates are concise.

We designed and implemented Catacomb, a system for code composition, and explored several issues relating to the automatic optimization of the code it produces. We implemented several algorithms for the HPF array assignment statement in the context of Catacomb, and used the implementation to evaluate several aspects of efficiency, maintainability, and robustness. In our experience, use of a code composition system is a good way to control the translation of complex operations and provides for an elegant and effective approach to producing high quality code without undue implementation cost.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. Technical Report A-278-CRI, Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris, November 1995.

[3] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 1996. ACM.

[4] H. Bao, J. Bielak, O. Ghattas, D.R. O'Hallaron, L.F. Kallivokas, J.R. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Supercomputing '96*, Pittsburgh, Pennsylvania, November 1996.

[5] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, Pennsylvania, 1994.

[6] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[7] S. Chatterjee, J. Gilbert, F.J.E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[8] F. Chow. *A Portable Machine-Independent Global Optimizer – Design and Measurements*. PhD thesis, Stanford University, 1984.

[9] R. Das, J. Saltz, and R. von Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 152–168, Portland, Oregon, August 1993. Springer Verlag.

[10] High Performance Fortran Forum. High Performance Fortran language specification version 1.0, May 1993.

[11] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32(2):155–172, February 1996.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[13] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings*

of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 102–111, Santa Barbara, California, July 1995.

[14] S. Midkiff. Local iteration set computation for block-cyclic distributions. Technical Report RC-19910, IBM T.J. Watson Research Center, January 1995.

[15] S. Pollack and T. Sterling. *A Guide to PL/I and Structured Programming (Third edition)*. Holt, Rinehart, and Winston, New York, 1980.

[16] D.J. Salomon. Using partial evaluation in support of portability, reusability, and maintainability. In Tibor Gyimóthy, editor, *Proceeding of the Sixth International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 208–222, Linköping, Sweden, April 1996. Springer Verlag.

[17] J. Saltz, R. Ponnusamy, S.D. Sharma, B. Moon, Y.-S. Hwang, M. Uyasl, and R. Das. A manual for the CHAOS runtime library. Technical Report CS-TR-3437, Department of Computer Science, University of Maryland, March 1995.

[18] S.D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing '94*, pages 97–106, Washington, DC, November 1994.

[19] J.R. Shewchuk and O. Ghattas. A compiler for parallel finite element methods with domain-decomposed unstructured meshes. In David E. Keyes and Jinchao Xu, editors, *Proceedings of the Seventh International Conference on Domain Decomposition Methods in Scientific and Engineering Computing*, volume 180 of *Contemporary Mathematics*, pages 445–450. American Mathematical Society, October 1993.

[20] J. Stichnoth. *Generating Code for High-Level Operations through Code Composition*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1997.

[21] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[22] J.M. Stichnoth. Efficient compilation of array statements for private memory systems. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[23] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, California, May 1993.

[24] A. Sussman, G. Agrawal, and J. Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1, University of Maryland, December 1993.

[25] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 309–316, Knoxville, Tennessee, May 1994.

[26] A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data-parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing*, 38(2):188–203, November 1996.

[27] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[28] J. Webb. Steps toward architecture-independent image processing. *IEEE Computer Magazine*, 25(2):21–31, February 1992.

[29] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 156–165, Albuquerque, New Mexico, June 1993. ACM.

[30] H. Zima, P. Brezany, B. Chapman, P. Mehrota, and A. Schwald. Vienna Fortran – a language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.