Generating Code for High-Level Operations through Code Composition

James M. Stichnoth

August 1997

CMU-CS-97-165

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Thomas Gross, *chair*
David R. O'Hallaron
Jaspal Subhlok
Joel Saltz, University of Maryland

Effort sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Distribution Statement A. Approved for public release; distribution is unlimited.

Keywords: Compilers, code generation, parallelism, communication generation

Abstract

A traditional compiler translates each expression or statement in a high-level language into a sequence of lower-level target statements (e.g., operations in an intermediate representation, or machine instructions), in a manner fixed by the compiler writer. The output is then subject to further optimization. This compilation strategy is called *custom code generation*, as the compiler generates custom code for each input construct.

An alternative strategy is to generate a call to a runtime library for each high-level language construct. This approach is attractive if the source language contains complex, powerful constructs, like the distributed array assignment statement in High Performance Fortran (HPF). The decision between custom code generation and use of a runtime library involves tradeoffs between *efficiency* (performance of the generated code), *maintainability* (ease of developing and maintaining the algorithm), and *generality* (implementation of the general case, rather than merely a simplified canonical case).

I introduce a new compilation strategy, *high-level code composition*, which combines the advantages of custom code generation and runtime libraries. The compilation of each construct is controlled by *code templates*, which contain both target code to be generated and compile-time control instructions that specify how the templates are composed together. The templates are external to the compiler, making them easy to write and modify. The *composition system* executes the control code at compile time, automatically generating and optimizing the code to be executed at run time.

In this dissertation, I motivate and explore the language and implementation issues of code composition. I describe Catacomb, my implementation of a composition system, which integrates with a high-level compiler to generate custom C code. I describe the challenges in enabling Catacomb to automatically generate the best code without sacrificing a clean user model. In addition, I develop a framework for the HPF array assignment, allowing an arbitrary array assignment algorithm to be coupled with an arbitrary communication architecture to form a complete implementation. This framework leads to the first compilation system that can incorporate and compare several array assignment algorithms on several communication architectures, for the fully general array assignment statement.

Acknowledgements

It's been a long journey, and there are lots of people who made it possible, and sometimes even fun. At the top of the list is Thomas Gross, who always believed in me, supported me, and helped me to focus my ideas into a thesis and a dissertation. I am also grateful to the other members of my committee, Dave O'Hallaron, Joel Saltz, and Jaspal Subhlok. They each provided me with a different aspect of the guidance I needed to finish this work.

Mary Zosel and Mark Seager at Lawrence Livermore Labs helped guide me to the first glimmer of the ideas in this thesis, as I started developing what came to be the CMU algorithm during my first summer of grad school. When I returned to CMU, the Fx compiler was just getting underway. Thanks to a lot of hard work by Jaspal Subhlok, Dave O'Hallaron, Bwolen Yang, Susan Hinrichs, Tom Stricker, Peter Dinda, John Pieper, Peter Lieu, and undoubtedly several others whom I regret forgetting, Fx became a working research platform.

Jonathan Shewchuk was instrumental in helping me to form and focus my ideas through the years. He was always willing to listen to my latest ideas, and to be the voice of reason whenever I came close to making implementation decisions that I surely would have regretted later.

Joel Saltz helped me by giving feedback on my ideas in the context of irregular communication generation, and graciously hosted Jonathan Shewchuk and me for two days at Maryland as we met with his group to gain a deeper understanding of the CHAOS internals.

I thank Sid Chatterjee, J. Ramanujam, and P. Sadayappan for giving me the code for their implementations of the RIACS, LSU, and OSU array assignment algorithms. Also, many thanks to Lei Wang, who did an enormous amount of work to make the Supercomputing paper happen.

I also want to thank the iWarp/Nectar Seminar (or whatever name it goes by these days) for giving me and the other group members a forum for presenting and getting feedback on our research ideas. Particular thanks go to Peter Steenkiste and Allan Fisher, who always had very insightful questions and comments on my work.

The Zephyr community at CMU provided me with countless hours of instant help and advice on issues technical, theoretical, grammatical, practical, and impractical, and also provided a never-ending source of amusement and distraction. `zephyr++`

Finally, I thank Dafna Talmor, who was there for me through it all.

Contents

1 Introduction **1**

1.1 Compiling complex operations . . . 1
1.2 High-level code composition . . . 3
1.3 Organization . . . 4

2 The Array Assignment Statement **5**

2.1 Specification . . . 5
2.1.1 Canonical example . . . 5
2.1.2 General array assignment statements . . . 7
2.2 The CMU algorithm . . . 8
2.2.1 Communication set specification . . . 8
2.2.2 Set representation . . . 9
2.2.3 Regular set intersections . . . 10
2.2.4 Communication set . . . 11
2.2.5 Optimizations . . . 12
2.3 Other algorithms . . . 13
2.3.1 OSU algorithm . . . 13
2.3.2 RIACS algorithm . . . 14
2.3.3 Rice algorithm . . . 15
2.3.4 LSU algorithm . . . 15
2.3.5 IBM algorithm . . . 16
2.3.6 École des Mines algorithm . . . 17
2.3.7 Syracuse algorithm . . . 17
2.3.8 Previous approaches . . . 17
2.4 Discussion . . . 19

3 Code Composition **21**

3.1 Motivating code composition: compiling the array assignment . . . 22
3.1.1 Extending to the general case . . . 22
3.1.2 Custom code generation or runtime library routines? . . . 24
3.1.3 Architectural features . . . 29
3.1.4 Discussion . . . 30
3.2 Code composition . . . 30
3.2.1 Custom code generation . . . 31
3.2.2 Code templates . . . 32
3.2.3 Composition system . . . 32
3.3 Composition language issues . . . 33

3.3.1 Code constructs . . . 34
3.3.2 Control constructs . . . 34
3.3.3 Execution model . . . 36
3.3.4 Combining code and control constructs . . . 36
3.4 Summary . . . 38

4 Catacomb **39**
4.1 Terminology . . . 39
4.2 Language infrastructure: expressions . . . 40
4.3 Language design: code constructs . . . 41
4.4 Language design: control constructs . . . 41
4.4.1 Template header . . . 43
4.4.2 Control variables . . . 43
4.4.3 Control variable assignment . . . 43
4.4.4 Control function call . . . 44
4.4.5 Control conditional . . . 45
4.4.6 Control loop . . . 45
4.4.7 Variable renaming . . . 45
4.4.8 Declarations . . . 46
4.4.9 External functions . . . 46
4.5 Global optimization framework . . . 47
4.5.1 Standard global optimizations . . . 47
4.5.2 Bounds optimizations . . . 47
4.5.3 Bounds representations . . . 49
4.6 Interpretation of control constructs . . . 50
4.6.1 Two-phase execution model . . . 50
4.6.2 Problems with the two-phase execution model . . . 51
4.6.3 Lazy evaluation of external functions . . . 52
4.6.4 Inadequacy of lazy evaluation . . . 54
4.6.5 Single-phase execution model . . . 55
4.6.6 Implementing the single-phase approach using data flow techniques . . . 56
4.6.7 Implementing the single-phase approach: Catacomb's solution . . . 59
4.6.8 Checkpointing and backtracking . . . 62
4.7 Compiler interface . . . 63
4.7.1 Interface routines . . . 64
4.7.2 Language interface . . . 64
4.7.3 Extending the symbol data structure . . . 65
4.7.4 Adding external functions . . . 65
4.7.5 Adding new templates . . . 66
4.8 Summary . . . 66

5 Catacomb and the Array Assignment **67**
5.1 Structure . . . 67
5.1.1 Code template design . . . 68
5.1.2 Supporting task parallelism . . . 74
5.1.3 Catacomb extensions . . . 76
5.2 Evaluation: Efficiency . . . 78

5.2.1 Evaluation of the Syracuse approach . . . 78
5.2.2 Custom code versus library code . . . 83
5.2.3 Effects of the deposit model . . . 84
5.2.4 Effects of nonstandard global optimizations . . . 87
5.3 Generality and maintainability . . . 90
5.3.1 Generality . . . 90
5.3.2 Maintainability . . . 93
5.4 Summary . . . 95

6 Other Code Composition Domains **97**
6.1 Data transfer in irregular applications . . . 97
6.1.1 Background . . . 97
6.1.2 Applicability of code composition . . . 100
6.2 Archimedes . . . 101
6.3 Precompilation and query optimization in relational database systems . . . 103

7 Related Work **105**
7.1 Templates and macro processing . . . 105
7.2 Partial evaluation . . . 106
7.2.1 The relationship of code composition . . . 106
7.2.2 The relationship of Catacomb . . . 106
7.2.3 Differences in Catacomb and code composition . . . 107
7.3 Runtime code generation . . . 108

8 Concluding Remarks **109**
8.1 Contributions . . . 109
8.2 Criticisms of Catacomb . . . 110
8.3 Future directions . . . 111

Bibliography **113**

A External Functions in Catacomb **119**
A.1 Core set of external functions . . . 119
A.1.1 Numerical functions . . . 119
A.1.2 Structure queries . . . 119
A.1.3 Structure dissection . . . 120
A.1.4 Expression construction . . . 121
A.1.5 List manipulation . . . 121
A.1.6 Index permutations . . . 121
A.1.7 Miscellaneous . . . 122
A.2 External functions for Catacomb/Fx . . . 122
A.2.1 Distribution information . . . 122
A.2.2 Access to compiler flags . . . 123

Chapter 1

Introduction

1.1 Compiling complex operations

When compiling a high-level program, the traditional approach for a compiler is to translate each input statement into a small fixed sequence of lower-level statements (e.g., machine instructions). For each input statement or expression, there is typically a small sequence of instructions to perform the task at execution time. The compiler performs this translation on each input construct, yielding a large sequence of low-level operations. Before emitting the code, the compiler may perform optimizations on the code, analyzing the sequence of operations as a whole.

Typically, given an input construct, there are few possible variations on the code that can be produced. For example, consider a "plus" operation in the C programming language. When compiling to machine code, the resulting code for different "plus" operations varies only as a result of the operand types. If the operands have different types, then a type conversion or promotion may be necessary, and the specific registers and instruction to use may also depend on the type of the operands. As another example, for a procedure call, the number of values pushed onto the stack or loaded into argument registers depends on the number of arguments passed to the procedure. However, the operation sequence is typically straightforward, with few possible variations.

Each input construct is translated into lower-level operations on a case-by-case basis, and the compiler analyzes the specific details of each construct to generate a custom sequence of lower-level operations. For this reason, 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, and new challenges arise for compiling these operations. This dissertation specifically addresses the issues of compiling such complex high-level operations. As such, it is important to have a feel for the characteristics of this class of operations. The following are a few examples of problem domains containing complex high-level operations.

- **The array assignment statement.** High Performance Fortran (HPF) [24] uses the array assignment statement to perform a bulk transfer of array data, and to perform data parallel computation. 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 $\ell\colon h\colon s$ describes an arithmetic sequence of array indices beginning with ℓ, with stride s, and upper bound h. The arrays A and B are *distributed* across the processors, such that each processor is assigned a particular subset of the elements of the arrays. The set of array elements mapped to a processor is called the *ownership set*.

The key challenge of an array assignment algorithm is to efficiently enumerate the *communication set*: the set of array elements that need to be sent from a particular sending processor to a particular receiving processor. The solutions to this problem typically include solutions to linear Diophantine equations, solutions to small linear systems of equations, and loops nests ranging from one to three levels deep.

There are several ways to generalize the canonical case. First, the right-hand side can have multiple terms, such as $B[\ell_B\colon h_B\colon s_B] + C[\ell_C\colon h_C\colon s_C]$. Second, an array reference can be multidimensional, as in $A[1\colon m][1\colon n]$. Third, there can be a mix of subscript triplets and scalar indices, such as $A[x][\ell_A\colon h_A\colon s_A]$.

The array assignment is a high-level operation because one such operation has the capability of encoding a large amount of work at run time, with a compact syntax at compile time. The distributed array assignment is a complex operation, both because of the complexity of the code sequences required for efficient execution, and because of the many ways required to piece together the code sequences, depending on the specific parameters of the array assignment. Chapter 2 describes the complexity issues in more detail.

- **Data transfer in irregular applications.**

  The HPF array assignment statement is typically used for *regular* data transfer. Sparse and unstructured problems result in the need for *irregular* data transfer. This need arises from irregular data access patterns, as well as irregular distributions of array data and loop iterations.

  One way to specify irregular data transfer is through an array assignment statement, either a standard array assignment with an irregular data distribution or an array assignment involving indirection arrays, such as $A[IA[\ell_A\colon h_A\colon s_A]] = B[\ell_B\colon h_B\colon s_B]$. Both examples are as compact to express as the standard array assignment statement, but are more complex to execute at run time.

  The other way to specify irregular data transfer is through an explicitly parallel loop that includes irregularly distributed arrays and/or multiple levels of array indirection. In this case, the entire loop itself serves as a complex high-level operation. Section 6.1 describes the problem of irregular data transfer in more detail.

- **Archimedes.** Archimedes [53] is the language and compiler portion of the Quake project [7] at Carnegie Mellon, which is focused on predicting ground motion during earthquakes. Archimedes includes a domain-specific language and compiler for compiling and executing unstructured finite element simulations on parallel computers. The language provides aggregate data types for nodes, edges, and elements of a finite element mesh, and it provides high-level statements and operators to iterate in parallel over such an aggregate object. Because of the irregular distribution of the unstructured mesh's data structures, executing these high-level operations in parallel requires complex code. In addition, the irregular distributions result in complex communication requirements, similar to above. Archimedes is described in more detail in Section 6.2.

- **NESL.** NESL [10] is a nested data parallel language that operates on distributed segmented vectors. Each vector operation is considered a high-level operation because it operates on all elements of the vector at once. The implementation of NESL is based on VCODE (an intermediate language) and CVL (a library of vector operations), totaling tens of thousands of lines of code. The size of the implementation is indicative of the complexity of the high-level operations.

- **Query optimization.** An example outside the domain of parallel languages is *query optimization* in relational databases [37]. A query is a high-level operation because it is syntactically compact, yet operates on a large amount of data. Query optimization techniques are used to transform a query into an equivalent one whose execution is more efficient (e.g., a query that requires less intermediate work). Query optimization techniques are also used in the low-level implementation of the query, for example to reorganize the looping structure to minimize the number of disk accesses. The kind of flexibility inherent in query optimization makes queries another example of a complex high-level operation.

When the high-level operations become so complex, it is no longer feasible to embed the sequence of low-level operations in the compiler. Instead, 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 runtime library routine approach also suffers in terms of the *generality* of the implementation. For example, consider what happens as we extend the array assignment statement from the canonical case to the general case. As discussed above, there are three extensions to consider: multiple right-hand side terms, multidimensional array references, and a mix of subscript triplets and scalar indices. As each restriction is lifted, it becomes harder and harder for a runtime library to offer a correct and complete implementation of the array assignment. I explore these particular issues in detail in Chapter 3.

In compiling for such a domain, there are three issues to trade off: efficiency, maintainability, and generality. Efficiency refers to the performance of the code at run time, maintainability refers to how easy and straightforward it is to develop and maintain the algorithm, and generality refers to whether the general case or merely the canonical case is implemented. In a runtime library routine approach, it is hard to achieve the best possible efficiency, because the parameters known at compile time cannot be embedded into the library. The compiler or library writer can clone the library multiple times and custom-tailor each instance to a predicted set of compile-time parameters. However, doing so hinders maintainability, because of the resulting increase in the total amount of library code. Increasing generality in a runtime library typically comes at the cost of maintainability and/or efficiency; maintainability is lost when the routine is cloned to increase the number of cases it can handle, and efficiency is lost by structuring the library in a more abstract way that avoids the code explosion. A custom code generation approach offers efficiency by compiling all known values into the resulting code, and generality by being able to directly generate exactly the required code. However, maintainability suffers greatly when hundreds or thousands of lines of code to generate are buried in the compiler.

1.2 High-level code composition

To address these problems, I have developed a new approach to compiling complex high-level operations, called *high-level code composition*. The distinguishing features of code composition are *code templates* and a *composition system*. The code templates contain code to be generated, as well as compile-time control code that specifies how the templates fit together. They are written in a structured, straightforward form, not much differently than a corresponding runtime library routine. The composition system executes the control code at compile time, with full knowledge of all compile-time information, and generates code that can be inserted and further optimized.

Code composition is essentially a variation on the custom code generation approach, since a separate sequence of lower-level operations is generated for each input high-level construct. As such, it enables efficiency of the generated code, since all compile-time information is compiled into the resulting code. In addition, it enables generality in the problem being solved, because the compile-time control code allows the templates to be composed together in a way that addresses the specific structure of the input operation. Furthermore, code composition provides a maintainable framework in which to generate code: the structure of the algorithm being generated is preserved in the code templates, rather than being obscured within the compiler itself.

With this background, I can now state my thesis:

> When compiling complex high-level operations, the technique of *code composition* is a significant improvement over the traditional techniques (namely, custom code generation versus the use of runtime library routines), producing efficient, general code in a maintainable framework.

1.3 Organization

In this dissertation, I motivate, develop, and evaluate the concept of code composition. In addition, I describe and evaluate Catacomb, the system I created for code composition. I have explored the issues in depth in the context of the HPF array assignment statement, which I use as a running example throughout this dissertation.

I begin in Chapter 2 by describing the HPF array assignment statement. I present a diverse set of published methods, both old and new, for treating various aspects of the construct, and I describe the class of features in the array assignment statement that lead to problems for traditional compilation techniques.

In Chapter 3, I describe code composition in detail. Continuing with the array assignment theme, I demonstrate the problems faced by a compiler writer in implementing the array assignment statement. Then I develop the concept of code composition and the composition system, and I describe the issues therein, including the language issues related to the composition system.

I designed and implemented a composition system called Catacomb, which I describe in Chapter 4. After describing the feature set of Catacomb's control language, I describe Catacomb's global optimization framework, and I explore an important issue: how can Catacomb integrate the global optimizations with the compile-time control code to produce the best quality of code, without creating a semantics that is difficult for the template programmer to understand? I finish the chapter by describing the interface that makes it simple and straightforward to integrate Catacomb into a compiler.

To evaluate Catacomb, I used it to implement several array assignment algorithms, for execution on several different communication architectures. In Chapter 5, I describe the structure of this implementation, a framework that allows an arbitrary array assignment algorithm to be coupled with an arbitrary communication architecture. I evaluate the implementation, in terms of runtime efficiency, generality, and maintainability.

In Chapter 6, I describe several other domains for which code composition is applicable. In Chapter 7, I describe the relationship of code composition to other fields, particularly the fields of partial evaluation and runtime code generation.

Chapter 2

The Array Assignment Statement

A strong motivating example for the ideas in this dissertation is the array assignment statement. In brief, the array assignment statement, which is a key component of High Performance Fortran (HPF) [24], effects a parallel transfer of a sequence of elements from a source array into a destination array. The array assignment 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, many of which are surveyed in this chapter. The compact syntax that hides the complexity of an efficient and general implementation is what makes the array assignment statement particularly interesting.

The goal of this chapter is to present a detailed example of a domain in which traditional compiler techniques are inadequate. In Section 2.1, I describe the array assignment statement in greater detail, including the standard HPF block-cyclic data distribution. In Section 2.2, I describe my solution, referred to as the CMU algorithm, which I implemented in the Fx parallelizing compiler [61]. In Section 2.3, I compare and contrast other approaches to the array assignment statement problem. Finally, in Section 2.4, I summarize the features of the array assignment that make it relevant and important to my thesis.

2.1 Specification

The array assignment statement has a general form, as well as a simplified canonical form. Studies of the array assignment begin with the canonical form, which includes the important features, *subscript triplets* and *data distributions*.

2.1.1 Canonical example

The canonical form of the array assignment statement is

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

This statement represents a parallel assignment of elements of B into the corresponding locations of A. The *subscript triplet* notation $\ell\colon h\colon s$, also used in Fortran 90 [4], describes a sequence of array indices starting with ℓ, with stride s, and having an upper bound h. If s is negative, h is instead a lower bound. If the stride parameter is omitted, it defaults to 1.

The statement is equivalent to the pair of sequential loops shown in Figure 2.1. The first loop copies the array B into a temporary T, and the second loop copies from the temporary into the destination array A.

```
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 2.1: Sample sequential code for the array assignment statement $A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B]$.

This two-step procedure is necessary because the semantics of the array assignment statement dictate that the right-hand side array elements are first read, and then the left-hand side elements are written. If there are no dependences between the accessed elements of A and B, the temporary can be eliminated and only the first loop is necessary.

Executing the array assignment statement becomes considerably more complex when we distribute the arrays across several processors. Array distribution means partitioning the array elements into subsets, and assigning one subset to each processor of the target multiprocessor system. Distribution is necessary because the problems are usually too large to fit in the memory of one processor. If the arrays have "incompatible" distributions, then corresponding elements of A and B reside on different processors, requiring communication before the nonlocal elements can be stored.

The concept of array distribution is relevant even for shared-memory machines. Many such machines have a non-uniform memory access (NUMA) architecture, where some of the memory is "closer" to a particular processor and thus has a lower access time. On such a machine, arrays should be distributed across the "close" memories of the processors involved in the computation, and the work should be distributed to the processors in a way that minimizes accesses to the "far" memories. The same data distribution techniques apply to both shared-memory and distributed-memory architectures.

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 2.2). 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 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 a block distribution of an array with N elements, the block size is $\lceil N/P \rceil$.)

The definition of the ownership set leads to an important relation: the *ownership test*. Given an input array index and processor, the ownership test determines whether the array element is distributed to the processor. Let ℓ_p be the index of the first element of array A mapped to processor p (typically, $\ell_p = bp$, where b is the distribution block size, assuming that processors and array indices are numbered starting from 0). For a block-cyclic distribution, the ownership test on an array index i is defined as

$$Owns(p, i, A) = (i - \ell_p) \bmod bP < b.$$

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. Processor p sends array element $B[j]$ to processor q to be stored in $A[i]$ if the following conditions hold:

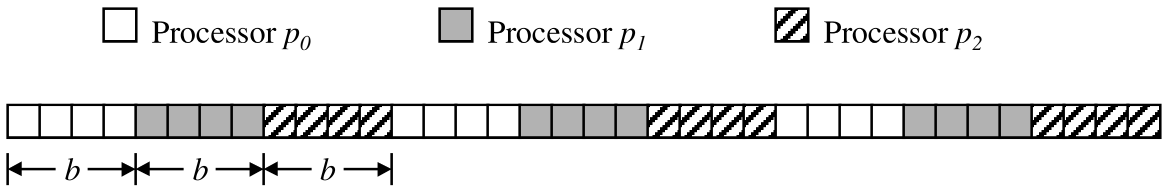


Figure 2.2: A block-cyclic data distribution in HPF, with three processors ($P = 3$) and block size $b = 4$.

1. j is in p's ownership set; i.e., $Owns(p, j, B)$ holds.
2. i is in q's ownership set; i.e., $Owns(q, i, A)$ holds.
3. i and j correspond to the same element in the respective subscript triplet sequences.

To verify the first two conditions, we use the ownership set formulas given above. The third condition is satisfied when $\ell_A \leq i \leq h_A$ and $\ell_B \leq j \leq h_B$, and there exists an integer n such that $n = (i - \ell_A)/s_A = (j - \ell_B)/s_B$.

2.1.2 General array assignment statements

Other examples of array assignment statements are:

1. $A[\ell\colon h\colon s] = k$
2. $A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B] + C[\ell_C\colon h_C\colon s_C]$
3. $A[\ell_{A1}\colon h_{A1}\colon s_{A1}][\ell_{A2}\colon h_{A2}\colon s_{A2}] = B[\ell_{B1}\colon h_{B1}\colon s_{B1}][\ell_{B2}\colon h_{B2}\colon s_{B2}]$

The first example, with no distributed arrays on the right-hand side, requires no communication (local computation only). However, each processor must determine and iterate over its *local computation set*. Array element $A[i]$ is part of processor p's local computation set if i is in p's ownership set and i is part of the subscript triplet sequence. Computing this set is similar to computing the communication set, although somewhat simpler.

The second example includes multiple terms on the right-hand side, as well as actual computation to be performed (the "plus" operation in this example). Figure 2.3 shows sequential code to execute this statement. The first two loops copy B and C into temporary arrays, both of which have the same size and distribution as A, and the final loop performs the computation and stores the results in A. Executing this statement in parallel reduces to executing the communication code for the first two loops, and executing the local computation code for the final loop. This code illustrates the commonly-used *owner computes* model, which specifies that the right-hand side computation for each iteration is to be performed on the processor where the result is stored.

The third example involves multidimensional arrays. Multidimensional arrays are handled by computing the individual sets for each dimension and taking their Cartesian product as the result. Taking a Cartesian product means producing a loop nest. Figure 2.4 shows simple sequential code to execute this two-dimensional array assignment statement.

In addition to distributions, HPF defines *alignments* of distributed arrays, and *templates* to which arrays can be aligned. Templates consume no resources at run time; they are merely a compile-time placeholder to which arrays can be aligned. When a template is distributed, the arrays aligned to the template are distributed as well. Arrays can be aligned to each other or to templates through a linear alignment function.

```
j = ℓ_B
DO i = ℓ_A, h_A, s_A
  T_B[i] = B[j]
  j = j + s_B
END DO
-------------------------
k = ℓ_C
DO i = ℓ_A, h_A, s_A
  T_C[i] = C[k]
  k = k + s_C
END DO
-------------------------
DO i = ℓ_A, h_A, s_A
  A[i] = T_B[i] + T_C[i]
END DO
```

Figure 2.3: Sample sequential code for the array assignment statement $A[\ell_A\!:h_A\!:s_A] = B[\ell_B\!:h_B\!:s_B] + C[\ell_C\!:h_C\!:s_C]$.

For example, if $C[i]$ is specified to be aligned to $D[ai + b]$, for integer constants a and b, then for each valid array index i, $C[i]$ is guaranteed to be mapped to the same processor as $D[ai + b]$. This alignment is commonly referred to as a *two-level mapping*, because array elements are mapped onto template elements and then templates are mapped onto processors. The two-level mapping adds complexity to the analysis because in general, such an array no longer has a true block-cyclic distribution. Several of the array assignment algorithms discussed later in this chapter deal also with the two-level mapping [14, 33, 68], but these extensions are beyond the scope of my thesis. Also beyond its scope are algorithms for determining the best alignments and distributions for the arrays in a program [15, 16, 73]; the discussion in this chapter assumes that distribution decisions have been made in advance, either automatically by the compiler or manually by the user.

2.2 The CMU algorithm

In previous work [54, 55] I derived an algorithm for determining and iterating across both communication sets and local computation sets for the array assignment statement in which the arrays have block-cyclic distributions. Following the nomenclature introduced by Wang, Stichnoth, and Chatterjee [71], I name this algorithm, as well as the algorithms described in Section 2.3, after their places of origin; hence the CMU algorithm here. In this section I give an overview of the CMU algorithm.

2.2.1 Communication set specification

Given the canonical array assignment statement

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

the goal is to determine the set of elements of B that processor p must send to processor q, to be stored in A. I define $Own(p, B)$ to be p's ownership set for array B; that is, the set of elements of B distributed onto p. Processor p can only send the elements of B that it owns and that are designated in the array assignment

```
x = ℓ_B1
DO i = ℓ_A1, h_A1, s_A1
   y = ℓ_B2
   DO j = ℓ_A2, h_A2, s_A2
      A[i][j] = B[x][y]
      y = y + s_B2
   END DO
   x = x + s_B1
END DO
```

Figure 2.4: Constructing a two-deep loop nest to execute the array assignment statement $A[\ell_{A1}\colon h_{A1}\colon s_{A1}][\ell_{A2}\colon h_{A2}\colon s_{A2}] = B[\ell_{B1}\colon h_{B1}\colon s_{B1}][\ell_{B2}\colon h_{B2}\colon s_{B2}]$.

statement; this is equal to the set

$$S_p = Own(p, B) \cap (\ell_B\colon h_B\colon s_B).$$

Similarly, processor q can only receive and store the elements of A that it owns and that are designated in the array assignment statement; this is equal to the set

$$S_q = Own(q, A) \cap (\ell_A\colon h_A\colon s_A).$$

For any integer i, if $\ell_A + is_A \in S_q$ and $\ell_B + is_B \in S_p$, then processor p sends $B[\ell_B + is_B]$ to q, which stores it in $A[\ell_A + is_A]$. To express this statement in terms of set intersections, I define a linear Map function, which associates $A[\ell_A + is_A]$ with $B[\ell_B + is_B] = B[Map(\ell_A + is_A)]$, on an index y as

$$Map(y) = \frac{y - \ell_A}{s_A} \cdot s_B + \ell_B.$$

The Map function is easily extended to a set or sequence in the obvious fashion. Hence for any integer i, if $Map(\ell_A + is_A) = \ell_B + is_B \in Map(S_q)$ and $\ell_B + is_B \in S_p$, then processor p sends $B[\ell_B + is_B]$ to q.

Using the Map function, the communication set is defined as

$$S_p \cap Map(S_q) = Own(p, B) \cap (\ell_B\colon h_B\colon s_B) \cap Map(Own(q, A) \cap (\ell_A\colon h_A\colon s_A)).$$

Because $Map(\ell_A\colon h_A\colon s_A) = (\ell_B\colon h_B\colon s_B)$, the $(\ell_B\colon h_B\colon s_B)$ term is unnecessary. Thus the communication set, expressed in terms of set intersections, is

$$Own(p, B) \cap Map(Own(q, A) \cap (\ell_A\colon h_A\colon s_A)).$$

2.2.2 Set representation

The CMU algorithm falls into the class of *set-theoretic* array assignment statement algorithms. It describes the communication and local computation sets as intersections and unions of regular sets. An advantage of using regular sets is that they are well suited to be efficiently enumerated by modern processors; a simple fixed-stride loop, like the loop implicit in a subscript triplet, suffices to enumerate a regular set. Another advantage is that regular sets are closed under intersection. Unfortunately, they are not closed under union.

The set denoted by a subscript triplet like $(\ell\colon h\colon s)$ is trivially a regular section. A block-cyclic ownership set, however, is not a regular section, although the set is compactly described as a union of regular sections.

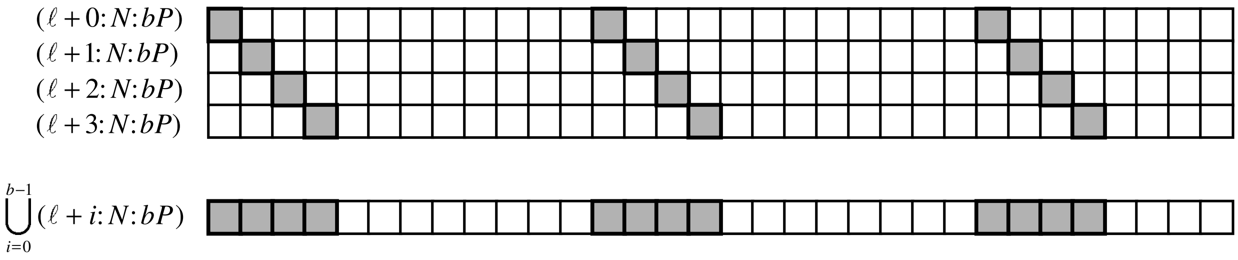


Figure 2.5: A block-cyclic set, represented as a union of regular sections.

Figure 2.5 illustrates how the CMU algorithm treats the block-cyclic set. Each regular set in the union shares the same stride and upper bound (keep in mind that in this regular set representation, the upper bound is not an exact upper bound), and each regular set is offset by 1 from its neighbors. The upper bound is equal to the highest-numbered index of the array, and the stride (which is the distance between the start of one block and the start of the next block) is equal to the block size b times the number of processors P over which the array is distributed. The formal definition of the block-cyclic set, then, is

$$\bigcup_{i=0}^{b-1}(\ell+i{:}\,N{:}\,bP),$$

where ℓ is the first element of the set and N is the highest-numbered index of the array.

An alternative representation of a block-cyclic set is as a union of contiguous blocks offset by bP, as illustrated in Figure 2.6. In this representation, the stride of each regular section is 1, but the lower and upper bounds have more complicated definitions. The formal definition of the block-cyclic set under this representation is

$$\bigcup_{i=0}^{\lceil N/bP\rceil}(\ell+ibP{:}\,\min(N,\ell+ibP+b-1){:}\,1),$$

where ℓ is the first element of the set and N is the highest-numbered index of the array. The OSU algorithm described in Section 2.3 uses both of these representations.

Using the first representation of a block-cyclic set yields the algorithm in Figure 2.7 for calculating the communication set. In the notation, ℓ_p is the index of the first element of B that p owns; ℓ_q is the index of the first element of A that q owns; h_p is the largest index of B; h_q is the largest index of A; s_p is the block size of B (i.e., b_p) times the number of processors over which B is distributed; and s_q is the block size of A (i.e., b_q) times the number of processors over which A is distributed.

2.2.3 Regular set intersections

The inner loop of the code in Figure 2.7 computes the intersection of three regular sets. Here I present an algorithm for computing a regular set intersection, optimized for computing the intersection of three or more regular sets at once.

My approach for finding the intersection of regular sections has three conceptual steps. The first step is to extend the lower bound and the upper bound of each sequence to $-\infty$ and $+\infty$, respectively, while remembering the original bounds. The second step is to find the intersection of these infinite sequences, which is either empty or another infinite sequence. The final step is to find the true lower and upper bounds of the intersection by using the remembered original bounds. This approach works well for finding the intersections of three or more regular sets, as it avoids the steps of converting intermediate intersections from finite sequences to infinite sequences, and vice versa.

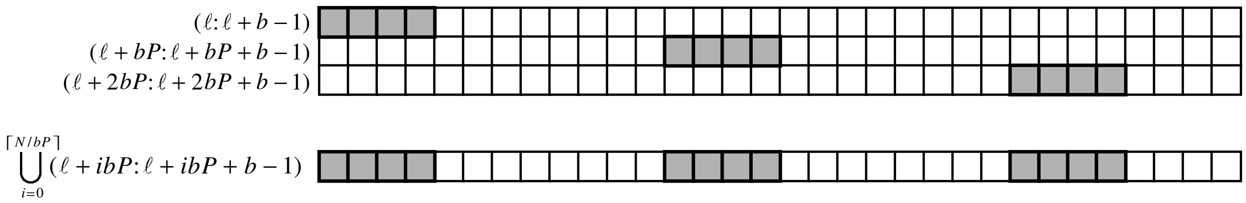


Figure 2.6: A block-cyclic set, represented as a different kind of union of regular sections.

I introduce a modified representation of a regular set, $(\ell : h : s : r)$, which represents a regular set with infinite bounds, as described above, where ℓ and h are the original remembered bounds. The s parameter serves the same purpose as before. The r parameter is a "representative" of the set, in the sense that any member of the set is equal to $r + ns$ for some integer n. For example, $(0 : 10 : 2 : 37)$ represents the same sequence as $(1 : 10 : 2)$, which is also the same as $(1 : 9 : 2)$. In this representation, the ℓ parameter need not be a representative of the set, in the same sense that the h parameter need not be a representative of the set. The only requirement is that for any integer n, $r + ns$ is a member of the original set if and only if $\ell \leq r + ns \leq h$ in the new representation.

The conversion from one representation to the other is simple. The set $(\ell : h : s)$ is the same as $(\ell : h : s : \ell)$, and the set $(\ell : h : s : r)$ is the same as $(\ell + ((r - \ell) \bmod s) : h : s)$. Note that in an implementation, $a \bmod b$ must return a non-negative value when b is positive; the `%` operator in the C programming language does not make this guarantee when a is negative.

Consider the intersection of the sets $(\ell_i : h_i : s_i : r_i)$ and $(\ell_j : h_j : s_j : r_j)$. A representative of the intersection is some integer r for which there exist integers m and n such that $r = r_i + ms_i = r_j + ns_j$. This equation tells us that $ms_i \equiv r_j - r_i \pmod{s_j}$.

Let x, y, and g be integers such that $xs_i + ys_j = g = \gcd(s_i, s_j)$. These integers are computed using Euclid's extended GCD algorithm [35, Volume 2]. A theorem of modular linear equations [19, Chapter 33] tells us that a solution exists if and only if $g | (r_j - r_i)$ [that is, g evenly divides $(r_j - r_i)$], and that one possible value for m is $m = (r_j - r_i)x/g$. (If g does not evenly divide $(r_j - r_i)$, then the intersection is empty.) By substituting, $r = r_i + ms_i = r_i + (r_j - r_i)xs_i/g$.

It is clear that the resulting stride of the intersection is $\mathrm{lcm}(s_i, s_j) = s_i s_j / g$. The new upper and lower bounds are the tighter of the two original bounds. Therefore

$$(\ell_i : h_i : s_i : r_i) \cap (\ell_j : h_j : s_j : r_j) = (\max(\ell_i, \ell_j) : \min(h_i, h_j) : s_i s_j / g : r_i + (r_j - r_i)xs_i/g).$$

One can see that for computing multiple intersections, this "relaxed" approach to computing the lower bound is preferable to computing the exact lower bound at each step. The relaxed approach only needs to compute the exact lower bound after all intersections have been computed in the four-parameter regular set representation.

2.2.4 Communication set

Through a tedious set of algebraic manipulations [55], I derived an algorithm for generating the communication set for the sending processor p. This algorithm generates the union of regular set intersections, including the application of the Map function, specified in Figure 2.7. Figure 2.8 gives the pseudo-code for this computation.

This algorithm ignores an important issue: the global-to-local array index mapping. Because each processor stores only a fraction of the array, it is necessary to compact the indices in each processor's

```
DO i = 0, b_q-1
  DO j = 0, b_p-1
    AddToSet((ℓ_p+j:h_p:s_p) ∩ Map((ℓ_q+i:h_q:s_q) ∩ (ℓ_A:h_A:s_A)))
  END DO
END DO
```

Figure 2.7: Pseudo-code for generating the communication set for an array assignment statement.

ownership set into a contiguous block, as illustrated in Figure 2.9. The local memory mapping function is defined in terms of the processor's distribution parameters ℓ, b, and s, as

$$LM(x) = \overbrace{\left\lfloor \frac{x-\ell}{s} \right\rfloor}^{block} \cdot b + \overbrace{((x-\ell) \bmod s)}^{offset}.$$

Applying further algebraic manipulations yields the communication set generation algorithm given in Figure 2.10.

The high-level communication model assumes the *message blocking* optimization, in which we send all data from p to q in a single aggregate message to minimize communication overhead. When q receives a message, it must execute a similar loop to store the received elements into the destination array A. This algorithm is similar to the message packing algorithm, except that the inverse of the Map function is applied to the global array indices, and the distribution parameters of the destination array A (rather than the source array B) are used in the LM function. Figure 2.11 shows the resulting algorithm for the receiver.

To complete the specification of the CMU algorithm, I show how to generate the computation set, which is somewhat simpler than the communication set. On processor q, the computation set for array $A[\ell_A\colon h_A\colon s_A]$ is $Own(q, A) \cap (\ell_A\colon h_A\colon s_A)$. Note that the Map function is not needed, but the LM function is needed to convert from the global to the local index space. Analysis similar to that of the communication set results in the algorithm shown in Figure 2.12.

2.2.5 Optimizations

The CMU algorithm works most efficiently with cyclic distributions. The cyclic distribution causes the outer loops to be small and the inner loop to be large, thus minimizing loop overhead. Unfortunately, the block distribution has the opposite effect: the outer loops are large and the inner loop is quite small, thus *maximizing* loop overhead.

Because the block distribution is so common, I had to develop separate communication and computation set algorithms specialized to the properties of the block distribution. The key property is that the ownership set for a block distribution can be treated as a single regular section, rather than a union of regular sections. This treatment eliminates one loop in the nest (two loops if both arrays have a block distribution), and lengthens the inner loop accordingly. Because the cyclic distribution can also be treated as a single regular set rather than a union, I derived specialized algorithms for cyclic distributions as well.

With the two new ways of representing ownership sets, there is now a total of three ways to represent the set. For the communication set algorithms, each array has one of three possible representations. With one left-hand side array and one right-hand side array, there are nine pack/send algorithms and nine receive/unpack algorithms to manage. For the computation set generation, there are three algorithms to manage. Refer to the original technical report [55] for the complete specifications of the additional algorithms.

$(x_1, y_1, g_1) = \mathrm{euclid}(s_A, s_q)$
$(x_2, y_2, g_2) = \mathrm{euclid}(s_B s_q / g_1, s_p)$
$s = s_B s_q s_p / g_1 g_2$
$h = \min(\ell_B + s_B \lfloor (\min(h_A, h_q) - \ell_A) / s_A \rfloor, h_p)$
$c = \max(\ell_B + s_B \lceil (\max(\ell_A, \ell_q) - \ell_A) / s_A \rceil, \ell_q)$
`DO j =` $\lceil (\ell_q - \ell_A)/g_1 \rceil, \lfloor (\ell_q - \ell_A + b_q - 1)/g_1 \rfloor$
  `DO i =` $\lceil (\ell_p - \ell_B - \mathtt{j} x_1 s_B)/g_2 \rceil, \lfloor (\ell_p - \ell_B - \mathtt{j} x_1 s_B + b_p - 1)/g_2 \rfloor$
    $r = \ell_B + \mathtt{j} x_1 s_B + \mathtt{i} x_2 s_B s_q / g_1$
    $\ell = c + ((r - c) \bmod s)$
    $\mathrm{Read}(B[\ell : h : s])$
  `END DO`
`END DO`

Figure 2.8: Communication set generation.

2.3 Other algorithms

In this section I present an overview of other array assignment statement algorithms. I continue to refer to each algorithm by the institution at which it was developed.

2.3.1 OSU algorithm

The OSU algorithm [27, 28, 29], developed by Gupta, Kaushik, Huang, and Sadayappan, is another set-theoretic algorithm, based on unions and intersections of regular sections. The principal difference between the OSU algorithm and the CMU algorithm is that the OSU algorithm is formulated to represent a block-cyclic set as shown in both Figures 2.5 and 2.6, whereas the CMU algorithm only uses the representation shown in Figure 2.5. This selection of representations allows the algorithm to choose a representation for each of the two arrays that leads to the most favorable looping structure; i.e., making the inner loops large and the outer loops small.

Block and cyclic sets are easier to treat than block-cyclic sets, so the OSU algorithm treats a block-cyclic distribution as a combination of a block distribution and a cyclic distribution. One way to treat it is as a block distribution over a set of virtual processors, where the virtual-to-real processor mapping is cyclic. This is called the *virtual-block* representation. The other way to treat it is as a cyclic distribution over a set of virtual processors, where the virtual-to-real mapping is block (the *virtual-cyclic* representation). The CMU algorithm is equivalent to the OSU algorithm where all array distributions are treated as virtual-cyclic.

The OSU algorithm allows both the left-hand side and the right-hand side array distributions to be treated independently as either virtual-block or virtual-cyclic, leading to four different formulations for the communication set generation. Each formulation results in a 3-deep loop nest that generates the same set, but in a different order, and with different looping overhead penalties.

Gupta et al. [28] also developed a simple runtime test to decide which of these four representations leads to the most favorable loop nest. However, this test does not consider the effect of the looping structure on the memory performance. While the virtual-block representation generates the array indices of the communication set in a monotonically increasing order, the virtual-cyclic representation generates indices in a non-monotonic order that makes many sweeps through the array. Unless the arrays are small, this access pattern results in poor cache performance.

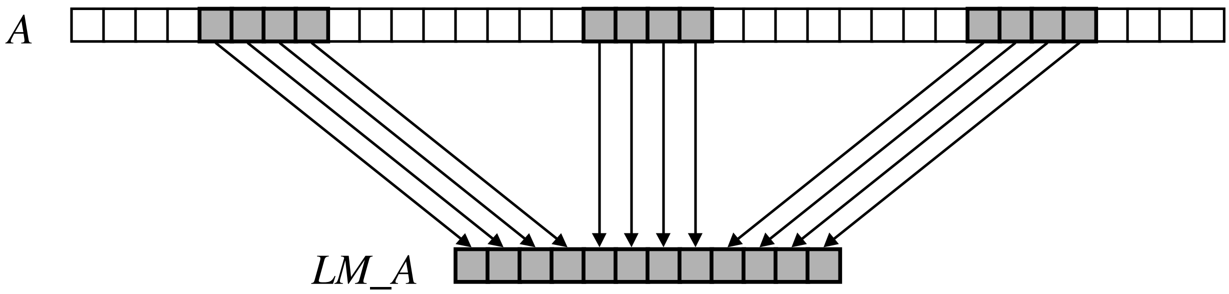


Figure 2.9: The local memory mapping function.

The OSU algorithm omits a formulation of the computation set algorithm. However, their analysis can be easily extended to handle computation set generation, including the runtime test to decide which of two representations leads to the more favorable loop nest.

2.3.2 RIACS algorithm

The RIACS algorithm [13, 14] is the first algorithm to be developed in the class of *table-driven* algorithms. The table-driven algorithms make use of the fact that the access pattern of the communication set repeats at some level. That is, if we look at the offsets between successive elements in the communication set, we find a pattern that repeats after a certain distance. The table-driven algorithms generate a table at run time that codifies these offsets.

The RIACS algorithm was originally formulated for just the computation set generation, so that the resulting loop makes a pass through the array, accessing only the elements assigned to the processor and that are specified in the array assignment statement. The algorithm was extended to enable it to generate the communication set by performing a brute-force calculation of the destination processor and location for each array index. This calculation involves integer division and remainder operations and is thus potentially expensive, unless, as often happens, the critical parameters (block size and number of processors) happen to be powers of two. In this case, cheap shift and bitwise operations can be used in place of divisions and remainders.

Rather than formulate a similar algorithm for the receiver, the RIACS algorithm specifies that the destination array indices are computed by the sender and included in the message. This decision increases the message sizes (typically by a factor of 2), but simplifies the decoding on the receiver.

Using the RIACS table to generate the communication set has some advantages over the CMU and OSU algorithms:

- The resulting loop is flat, in contrast to the three-deep loop nest structure of the CMU and OSU algorithms.
- It makes exactly one monotonic pass through the right-hand side array of the assignment statement. The CMU and OSU algorithms have to make at least one separate pass through the array for each destination processor (and they may make many more passes through the array when the virtual-cyclic representation is used).

On the other hand, there are some disadvantages:

- The RIACS algorithm generates a table of size proportional to the block size b and then sorts it, making the table generation complexity $O(b \log b)$. For the relatively common block distribution, the

```
(x_1, y_1, g_1) = euclid(s_A, s_q)
(x_2, y_2, g_2) = euclid(s_B s_q / g_1, s_p)
s' = s_B s_q s_p / g_1 g_2
s = s_B s_q b_p / g_1 g_2
h' = min(ℓ_B + s_B ⌊(min(h_A, h_q) − ℓ_A) / s_A⌋, h_p)
h = ⌊(h' − ℓ_p) / s_p⌋ b_p + min((h' − ℓ_p) mod s_p, b_p − 1)
c = max(ℓ_B + s_B ⌈(max(ℓ_A, ℓ_q) − ℓ_A) / s_A⌉, ℓ_q)
DO j = ⌈(ℓ_q − ℓ_A)/g_1⌉, ⌊(ℓ_q − ℓ_A + b_q − 1)/g_1⌋
   DO i = ⌈(ℓ_p − ℓ_B − j x_1 s_B)/g_2⌉, ⌊(ℓ_p − ℓ_B − j x_1 s_B + b_p − 1)/g_2⌋
      i' = g_2 i − ℓ_p + ℓ_B + j x_1 s_B
      r = ℓ_B + j x_1 s_B + i x_2 s_B s_q / g_1
      ℓ' = c + ((r − c) mod s')
      ℓ = (ℓ' − ℓ_p − i')/s_p + i'
      Read(B[ℓ:h:s])
   END DO
END DO
```

Figure 2.10: Communication set generation in terms of local memory indices.

table takes a considerable fraction of the overall execution time to build, and consumes approximately as much memory as the distributed array itself.

- The access stride of the inner loop is not fixed. Rather, each successive array reference is computed through a table lookup in the inner loop, thus hindering some potential compiler optimizations.
- No messages can be sent until all message buffers are packed and ready to go. This increases the amount of buffer space required (the CMU and OSU algorithms require only a single send buffer), and prevents overlap of communication and computation of the sets.

2.3.3 Rice algorithm

The Rice algorithm [34], also in the class of table-driven algorithms, is a modification of the RIACS algorithm. It improves the RIACS algorithm in two ways. First, it includes an algorithmic improvement in the complexity of the table generation phase. The complexity of the RIACS table generation phase is $O(b \log b)$, where b is the block size of the distribution; the Rice algorithm reduces the complexity to $O(b)$. Second, the Rice algorithm augments the table with destination processor and array index information, so that these values do not have to be computed by brute force in the inner loop. Like the RIACS algorithm, the table in the Rice algorithm becomes relatively large and expensive to generate as the block size becomes large.

2.3.4 LSU algorithm

The LSU algorithm [67, 68] is also a modification of the RIACS algorithm. Like the Rice algorithm, it reduces the table generation complexity from $O(b \log b)$ to $O(b)$. It does not, however, attempt to augment the table with destination address information like the Rice algorithm does. In practice, the LSU algorithm significantly outperforms both the RIACS and the Rice algorithms for generating the table [71]. Note that

$(x_1, y_1, g_1) = \mathrm{euclid}(s_A, s_q)$
$(x_2, y_2, g_2) = \mathrm{euclid}(s_B s_q / g_1, s_p)$
$s' = s_A s_q s_p / g_1 g_2$
$s = s_A b_q s_p / g_1 g_2$
$h' = \min(h_A, h_q, \ell_A + s_A \lfloor (h_p - \ell_B) / s_B \rfloor)$
$h = \lfloor (h' - \ell_q) / s_q \rfloor b_q + \min((h' - \ell_q) \bmod s_q, b_q - 1)$
$c = \max(\ell_A, \ell_q, \ell_A + s_A \lceil (\ell_p - \ell_B) / s_B \rceil)$
DO $\mathtt{j} = \lceil (\ell_q - \ell_A)/g_1 \rceil, \lfloor (\ell_q - \ell_A + b_q - 1)/g_1 \rfloor$
  $t = (\ell_A + \mathtt{j} x_1 s_A - \ell_q) \bmod s_q$
  DO $\mathtt{i} = \lceil (\ell_p - \ell_B - \mathtt{j} x_1 s_B)/g_2 \rceil, \lfloor (\ell_p - \ell_B - \mathtt{j} x_1 s_B + b_p - 1)/g_2 \rfloor$
    $r = \ell_A + \mathtt{j} x_1 s_A + \mathtt{i} x_2 s_A s_q / g_1$
    $\ell' = c + ((r - c) \bmod s')$
    $\ell = (\ell' - \ell_q - t) b_p / s_q + t$
    $\mathrm{Store}(A[\ell : h : s])$
  END DO
END DO

Figure 2.11: Communication set generation for the receiving processor, in terms of local memory indices.

the LSU algorithm generates the same table as the RIACS algorithm, so that using the table to iterate through the array is exactly the same; only the overhead of table generation changes. As such, like the other table generation algorithms, the LSU table is large and relatively expensive to generate in the presence of large block sizes.

2.3.5 IBM algorithm

Midkiff describes the IBM algorithm [40], one in a class of *linear algebraic* algorithms. It essentially uses symbolic Gaussian elimination at compile time to compute the intersection of a regular section (as specified by a subscript triplet) and a block-cyclic set. The method defines a system of linear Diophantine equations and finds a solution symbolically in a predetermined fashion. The resulting loop structure is similar to the virtual-block formulation of the OSU algorithm.

There are several weaknesses in this algorithm. First, like the RIACS algorithm, the IBM algorithm is formulated to generate only the local iteration set, and not the communication set. This means that inside the inner loop, it must compute the destination processor and local memory offset for each iteration. Second, the local iteration set algorithm is not formulated to incorporate the local index translation, so the translation must be done inside the inner loop. Third, because the algorithm is structured like the OSU virtual-block algorithm, its performance is expected to suffer in the presence of small block sizes (e.g., a cyclic distribution), because the outer loop is large and the inner loop is small.

The IBM algorithm does have some advantages, though. In comparison to the table-driven algorithms, the stride of the inner loop is fixed, rather than being the result of a table lookup. In addition, the algorithm is formulated to handle subscript functions that are more complex than those allowed in an array assignment statement. One such example is the coupled subscript like $A(i, i)$, which can arise as an explicit subscript in a parallel loop. Most of the array assignment techniques are not powerful enough to handle this kind of construct.

$$
\begin{array}{l}
(x_1, y_1, g_1) = \mathrm{euclid}(s_A, s_q) \\
s' = s_A s_q / g_1 \\
s = s_A b_q / g_1 \\
h' = \min(h_A, h_q) \\
h = \lfloor (h' - \ell_q) / s_q \rfloor b_q + \min((h' - \ell_q) \bmod s_q, b_q - 1) \\
c = \max(\ell_A, \ell_q) \\
\texttt{DO}\ \ \texttt{j} = \lceil (\ell_q - \ell_A)/g_1 \rceil,\ \lfloor (\ell_q - \ell_A + b_q - 1)/g_1 \rfloor \\
\quad j' = g_1 \texttt{j} + \ell_A - \ell_q \\
\quad r = \ell_A + \texttt{j} x_1 s_A \\
\quad \ell' = c + ((r - c) \bmod s') \\
\quad \ell = (\ell' - \ell_q - j') b_p / s_q + j' \\
\quad \texttt{DO}\ \ \texttt{i} = \ell,\ h,\ s \\
\qquad A[\texttt{i}] = T[\texttt{i}] \\
\quad \texttt{END DO} \\
\texttt{END DO}
\end{array}
$$

Figure 2.12: Computation set generation, in terms of local memory indices.

2.3.6 École des Mines algorithm

Another linear algebraic algorithm is the École des Mines algorithm [5]. Like the IBM algorithm, it uses matrix transformations to compute part of the access pattern at compile time. However, it uses a much larger set of transformations, allowing it to reduce the number of loops required, to improve the communication patterns, and to improve memory access patterns. Furthermore, it also addresses the issue of memory allocation and local memory mapping for two-level mappings.

2.3.7 Syracuse algorithm

The Syracuse algorithm, developed by Thakur, Choudhary, and Fox [66], is not a full array assignment statement algorithm. It only handles redistributions of whole arrays (rather than array sections) that are distributed over the same number of processors. The formulation does not fit into any of the above categories of algorithms. Instead, they present communication set generation as a brute-force computation, reminiscent of the *runtime resolution* approach (see below), for the general case, and they identify and present more efficient algorithms for several special cases (e.g., block-to-cyclic redistributions, and redistributions where the source block size is a multiple of the destination block size). Even though the Syracuse algorithm is not a fully general array assignment statement algorithm, I introduce it here because their high-level design forces them into some of the engineering compromises that my compilation approach and system (described in Chapters 3 and 4) are designed to avoid. In particular, they recognize the difficulty in producing a general system for redistributing d-dimensional arrays, and they address the problem with a simple solution that increases the amount of work at run time by a factor of d.

2.3.8 Previous approaches

Before High Performance Fortran brought the array assignment statement and the block-cyclic distribution together into a widely supported framework, there were several other systems that executed array assignment statements with distributed arrays. In this section I give an overview of some of this previous work.

Kali

The Kali language and implementation [36] was one of the first systems to address the issue of array assignment statements in the presence of block-cyclic distributions. Although Kali did not treat the array assignment statement directly, it included substantial compile-time analysis of FORALL loops with linear subscript functions. These loops have an equivalent access pattern to that of the array assignment statement.

The compile-time analysis for block and cyclic distributions has the same character as for that of the CMU, OSU, and Syracuse algorithms, involving the manipulation of regular sets. This compile-time analysis was not extended to the more general block-cyclic distribution. Instead, the block-cyclic distribution was handled using runtime analysis techniques.

ADAPTOR

The ADAPTOR system [12] is a freely available HPF compilation system. It translates data parallel Fortran programs enhanced with array assignment statements and parallel loops into SPMD message-passing Fortran code. One drawback is that it only supports block and cyclic distributions, and not the more general block-cyclic distribution.

ADAPTOR uses compile-time high-level code generation in conjunction with an extensive runtime library to translate the parallel program into SPMD code. As an example of the complexity of the runtime library, the communication generation routines handle up to 7-dimensional arrays, with separate code and separate optimizations for each dimensionality. As I discuss in Section 3.1.2, this decision adversely affects both the generality of the implementation and the potential for optimization based on compile-time knowledge.

Runtime resolution

The Cray MPP Fortran programming model [38, 42] supports array assignment statements with block-cyclic array distributions. However, their execution strategy for the array assignment is an extremely simplistic method known as *runtime resolution*, which does not scale up as the number of processors increases. To understand its problems, consider the communication set generation algorithm. The sending processor iterates across the elements in the subscript triplet. For each element, it computes several values:

- Whether the sending processor owns that element of the right-hand side array
- The sending processor's local index for the array element
- Which target processor receives the array element
- The local array index on the target processor

All of these decisions are made at run time, hence the name runtime resolution.

Each of these computations requires integer multiplies, divides, and remainder operations. Because of the high expected execution cost, MPP Fortran requires all distribution parameters (array size, distribution block size, and number of processors) to be powers of 2. This way, all addressing computations are expressed in terms of fast shift and bitwise operations.

Even with this restriction, runtime resolution still has a problem: it does not scale with the number of processors. As more processors are added to the array distribution (keeping the array size fixed), the amount of work per processor remains unchanged. With the other methods covered in this chapter, the work per processor is inversely proportional to the number of processors.

2.4 Discussion

There is a diversity of algorithms developed for executing the array assignment. Most of them are fairly dissimilar to each other, with the exception of obvious cases like the Rice and LSU algorithms, which are explicitly designed as improvements on the original RIACS algorithm. As different as the algorithms are, they share a few key features:

- **Complex code**. A single array assignment statement is syntactically simple and compact to specify, but requires a great deal of complex code to execute. Implementations of each of the algorithms reviewed in this chapter require, at a minimum, on the order of hundreds of lines of C code.

- **Widely varying code dependent on compile-time information**. Each algorithm reviewed in this chapter contains several branches, only one of which is executed at run time. In particular, the RIACS algorithm and its variants include a special case for which the communication pattern is determined to be a "shift" operations, and a general case. The OSU algorithm includes code for the virtual-block and virtual-cyclic representations; because the left-hand side array and the right-hand side array are treated independently, there are four possible cases. The CMU algorithm includes block-cyclic code as well as special-case code for both block and cyclic distributions, independently for each array, leading to a total of *nine* possible cases. The Syracuse algorithm includes four special-case algorithms in addition to the general case.

  In addition to this coarse level of special casing, the code within a single case can vary at a fine grain depending on the information available. For example, if certain parameters are known to evenly divide other key parameters, some boundary case computations can be eliminated. As another example, if a key parameter is known to be 1, then a gcd computation is trivial and can be eliminated.

- **Independent building blocks**. The array assignment statement algorithms are all specified for a simplified canonical case. As described in the next chapter, lifting the restrictions is conceptually simple, and involves decomposing the canonical case into a set of independent building blocks, and composing them together in new ways to solve the general case. However, attempting to enumerate all possible combinations of building blocks leads to an exponential code explosion.

As I discuss in the next chapter, these features motivate a new approach to compilation.

Chapter 3

Code Composition

There is a class of high-level programming language constructs for which traditional compilation techniques are inappropriate. These constructs share some or all of the features listed in Section 2.4:

- Executing them requires complex code, rather than just a few simple operations.
- There is a fairly wide variety of code sequences that can be executed at run time, dependent on the specific structure of the high-level construct (e.g., argument types, number of array dimensions, structure of an expression tree) and on other compile-time information (e.g., parameters known to be compile-time constants).
- They have an implementation structure based on defining a collection of independent building blocks for a simple case, and piecing together the building blocks to handle the general case, in a manner determined by its structure.

Traditional compilation techniques for this class of problem fall into two categories:

- *Custom code generation*: the compiler produces a separate code sequence for each instance of the construct, dependent on the information available at compile time.
- Static *runtime library routines*: there is a fixed set of routines in a runtime library, and the compiler "compiles" the construct into one or more calls to the appropriate library routines.

These techniques fall short due to a lack of at least one of three necessary properties:

- *efficiency*: being able to optimize the construct's runtime execution, based on all available compile-time information
- *generality*: being able to execute any instance of the construct, rather than just a special case
- *maintainability*: being able to write, maintain, debug, and tune the algorithm for executing the construct, all within the framework of the compilation system

The traditional compilation techniques fall short because custom code generation typically yields efficiency and generality at the cost of maintainability, while static runtime library routines offer maintainability at the cost of efficiency and generality.

In this chapter, I present a new approach for high-level code generation for this class of problem, called *high-level code composition*. Code composition is designed to provide all three of the above properties: efficiency, generality, and maintainability. Because it is a special formulation of the custom code generation approach, code composition offers efficiency and generality. And because it exposes the code generation framework to the user, rather than burying it inscrutably and inextricably within the compiler, it offers maintainability at the same time.

I begin in Section 3.1 by describing the implementational issues for the array assignment statement, and giving specific reasons for why the traditional compilation techniques for the array assignment statement lack one or more of these properties. In Section 3.2, I describe code composition in detail and how it can be used to compile the class of language constructs of interest. In Section 3.3, I discuss the language issues of a system for code composition.

3.1 Motivating code composition: compiling the array assignment

Chapter 2 describes at a high level a collection of algorithms for executing an array assignment. In this section, I discuss the lower-level implementation details needed for integrating the algorithms into a compiler. First, I describe how the canonical array assignment is conceptually extended to the general case. Then I describe the compiler issues for handling the general case. After that, I fill in some more details of the array assignment regarding the communication features of the target machine. Finally, I wrap up with a discussion of the compilation issues. The overall purpose is to demonstrate in detail why the traditional compilation techniques fall short for this class of problem.

3.1.1 Extending to the general case

In all of the communication/computation set algorithms, a simplified canonical array assignment statement is assumed; e.g. $A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B]$. Conceptually, the simplifying restrictions are easy to lift. The restrictions, and how to remove them, include:

- *Restriction 1: Each array is one-dimensional.* Multidimensional arrays are conceptually simple to handle, because each dimension can be treated orthogonally. For communication set generation, we construct a loop nest in which each level corresponds to one dimension of the array references. This loop nest produces the Cartesian product of the communication sets for the individual dimensions. Computation set generation is treated similarly.

  To illustrate with an example, Figure 3.1 shows how three copies of a loop for a one-dimensional array can be nested inside each other to form a loop nest for a three-dimensional array.

  There is one additional concern relating to multidimensional arrays. Consider a transpose operation. There is no way to express this operation using the array assignment syntax, yet the execution of a transpose is virtually identical to the execution of a multidimensional array assignment. The only change is to permute the order of the array subscripts in the inner loop array reference. For example, instead of $A[i][j] = B[i][j]$ in the inner loop, we could transpose the subscripts of B, yielding $A[i][j] = B[j][i]$. Thus a transpose, or any other *index permutation*, can be expressed as just a simple variation on the array assignment statement.

- *Restriction 2: There is only one array reference on the right-hand side of the assignment statement.* To support multiple right-hand side terms, we simply repeat the communication analysis independently for each term, moving the right-hand side term into a temporary array that has the same size, shape, and distribution as the right-hand side array. Then we merge the right-hand side computation into the inner loop of the computation set generation.

  As an example, consider the array assignment statement

  $$A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B] * C[\ell_C\colon h_C\colon s_C] + D[\ell_D\colon h_D\colon s_D].$$

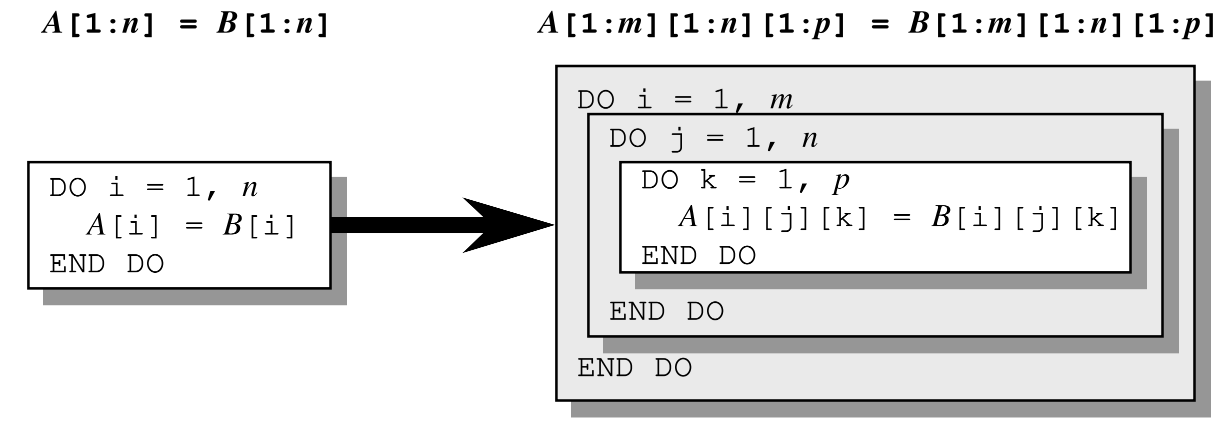


Figure 3.1: Forming the Cartesian product of three iteration sets by constructing a loop nest.

We can transform this statement into the sequence of statements shown in Figure 3.2. The first three statements are the simple canonical array assignment statements and can be processed with the standard methods. The final statement involves only local computation (no communication) because the sizes, distributions, and subscripts of the temporary arrays match those of A. Thus the inner loop of the computation set generation can perform the "$+$" and "$*$" computation.

Note that each temporary requires as much memory as the left-hand side array. To reduce the amount of temporary space needed, we can use standard compiler techniques [2] to reorder the computation to minimize temporaries. We can also use the right-hand side array A as a temporary, since it is overwritten by the computation.

- *Restriction 3: All array subscripts are subscript triplets.* If there are any scalar subscripts, two steps are required. First, we put the entire set generation code inside a conditional that tests whether the target processor owns the scalar subscript; i.e., whether the data distribution assigns that array index to the processor. Second, we apply the LM (local memory mapping) function to the scalar subscript when accessing it. To illustrate, Figure 3.3 shows an example of this modification to the communication set generation for the assignment statement $A[x][\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B][y]$, for a sending processor p and a receiving processor q.

There is another extension to the array assignment statement: the use of indirection arrays, as in

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

This extension completely changes the analysis of the access patterns. Instead of a *regular* linear sequence, the access pattern becomes an arbitrary *irregular* sequence dependent on the runtime contents of the indirection array IA. The analysis is further complicated by the fact that the indirection array is itself most likely distributed.

A similar situation holds when an array has an *irregular* distribution. With an irregular distribution, there is an auxiliary array at runtime whose contents specify the mapping of array elements to processors. Analysis of access patterns is similar to the analysis for indirection arrays.

Because of the enormous difference in complexity between executing regular and irregular array assignment statements, there is no simple or straightforward way to extend the canonical regular case to handle the irregular assignment. In Chapter 6.1, I give a more detailed analysis of the array assignment statement in the presence of indirection arrays and irregular distributions.

$$
\begin{array}{l}
T_B[\ell_A\!:\!h_A\!:\!s_A] = B[\ell_B\!:\!h_B\!:\!s_B] \\
T_C[\ell_A\!:\!h_A\!:\!s_A] = C[\ell_C\!:\!h_C\!:\!s_C] \\
T_D[\ell_A\!:\!h_A\!:\!s_A] = D[\ell_D\!:\!h_D\!:\!s_D] \\
A[\ell_A\!:\!h_A\!:\!s_A] = T_B[\ell_A\!:\!h_A\!:\!s_A] * T_C[\ell_A\!:\!h_A\!:\!s_A] + T_D[\ell_A\!:\!h_A\!:\!s_A]
\end{array}
$$

Figure 3.2: Translation of $A[\ell_A\!: h_A\!: s_A] = B[\ell_B\!: h_B\!: s_B] * C[\ell_C\!: h_C\!: s_C] + D[\ell_D\!: h_D\!: s_D]$ into simpler components.

3.1.2 Custom code generation or runtime library routines?

When algorithm designers present code to generate communication and computation sets, the code is meant to be used by a parallelizing compiler, not by an end user. To use the code, the compiler writer has two choices: custom code generation and runtime library routines. As stated at the beginning of this chapter, custom code generation means that the parallelizing compiler produces the code directly for each array assignment statement on a case-by-case basis. With the runtime library routine approach, the compiler writer writes a set of static support routines, and the compiler produces runtime calls to the appropriate routines for each array assignment statement. Figure 3.4 illustrates these two approaches. Each choice has its advantages and disadvantages, in terms of whether or not it includes three key properties: efficiency, maintainability, and generality. I discuss these properties and their relation to the code generation approaches below.

Efficiency

To achieve the best runtime performance, custom code generation is more desirable. The code generated by the compiler contains all the constants and other information available at compile time, and thus has more optimization potential. A runtime library routine has to be designed to handle the most general input possible, and so the potential for optimization is severely limited. It is certainly possible to include specially optimized libraries for common cases, but in practice, it is unlikely that all optimizable cases can be anticipated.

All is not lost with runtime libraries, though. In general, using procedure inlining in conjunction with runtime libraries can improve runtime performance. In particular, for the array assignment statement, inlining can help quite a bit, because of the number of input parameters. Consider a standalone function that executes the array assignment. For each of the two arrays, there are three input parameters specifying the subscript triplet and three parameters specifying the array distribution (block size, number of processors, and array size). In addition, the function takes the two arrays as input, and it may also require the sending and/or receiving processor IDs, for a total of 14 to 16 input arguments. Usually, most of these parameters are known at compile time (e.g., access strides, array sizes, and distribution block sizes), so inlining is likely to yield significant benefits.

Maintainability

Runtime performance is not the only issue, though. Maintainability and ease of development of the code being generated are important as well. For supporting this goal, a runtime library is far superior to custom code generation. A runtime library routine is easy to develop, test, and maintain, and is an ideal way to formulate and develop a complex algorithm like communication set generation. By contrast, custom code generation usually requires modifying the compiler itself to make any changes. This modification tends to be difficult because the structure of the algorithm becomes obscured or lost inside the code generation framework.

```
IF (Owns(p,B,y) && Owns(q,A,x))
  DO i = ...
    ... = B[i][LM(y)]
  END DO
END IF
```

Figure 3.3: Modifications to the communication set generation to handle scalar subscripts in the array assignment statement $A[x][\ell_A\!: h_A\!: s_A] = B[\ell_B\!: h_B\!: s_B][y]$.

An additional practical concern leads a compiler writer to favor the runtime library routine approach for the array assignment statement problem. The compiler writer may wish to incorporate and explore the performance of several different array assignment statement algorithms. This approach is reasonable even for a production HPF-like compiler, because in practice, no single algorithm outperforms all other algorithms under all circumstances [71]. The simplest way to incorporate a new algorithm into the compiler is to directly use the author's own implementation, and the most easily exchangeable form of an implementation is a runtime library routine.

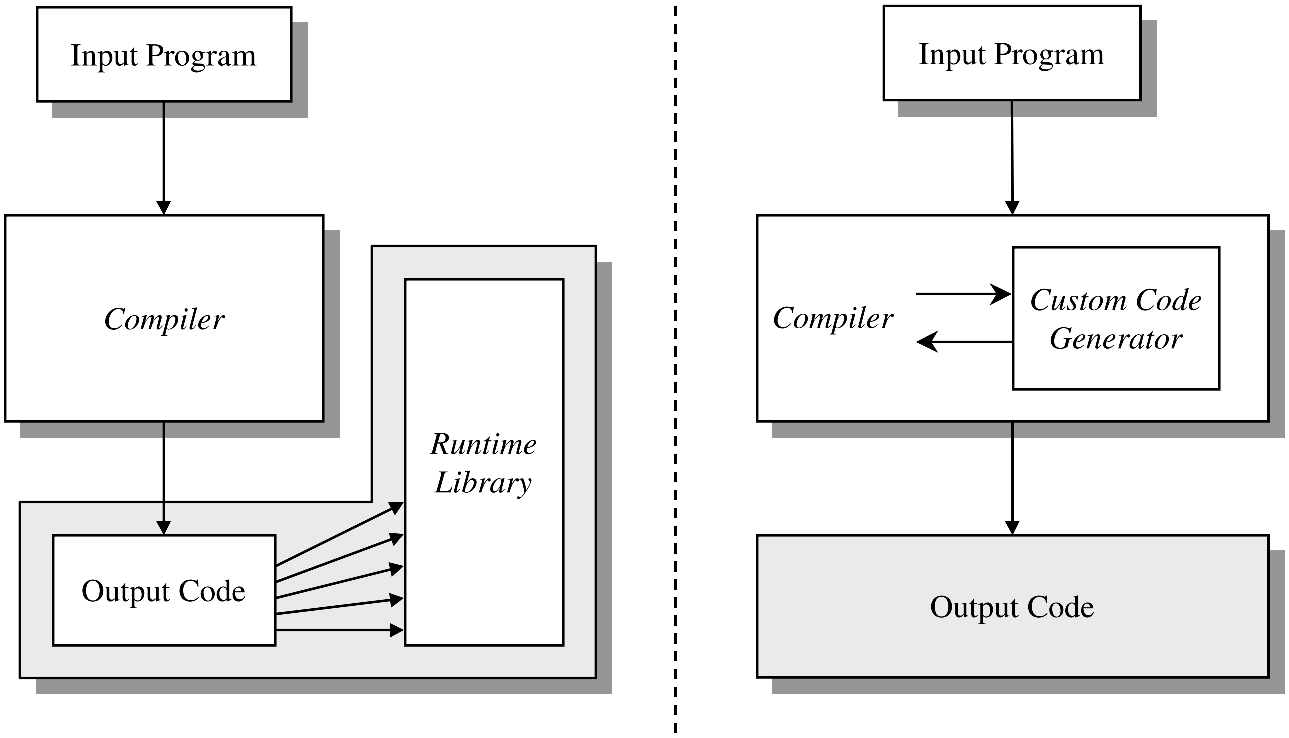
Generality

At this point, the most desirable choice for compiling the array assignment statement appears to be to produce calls to a runtime library routine, perhaps making use of procedure inlining for more efficient runtime execution. The huge increase in maintainability surely makes up for the small loss of runtime performance. However, there remains the issue of *generality* of the implementation; i.e., whether the implementation can handle all the variants on the array assignment statement, and not just the canonical case.

Here I show that because of the generality concern, a runtime library routine is unsuitable for implementing the array assignment statement in its full generality. Consider what happens as we try to lift the restrictions on the canonical array assignment statement and develop a general set of runtime library routines for the fully general array assignment statement.

- **Multidimensional arrays.** This restriction is the hardest to lift, in terms of both maintainability and runtime performance. First, let us assume that we want to write a single routine that handles arrays of arbitrary dimensionality. This routine needs at least 12 input parameters per dimension, as discussed above, as well as other parameters such as the input arrays and the number of dimensions. The routine must execute a loop nest whose depth is proportional to the number of dimensions. This kind of a loop nest, where the depth is unknown at compile time, is notoriously difficult to write and to execute efficiently in a procedural language like C (which is the most common language for implementing runtime libraries to support compiler-generated code). The most straightforward way to write such a routine is to use recursion: write the routine for the canonical one-dimensional case, and the body of the loop either performs the assignment statement or recursively invokes the routine for the next dimension. While the resulting routine is straightforward, the recursion prevents optimizers from instantiating compile-time constants through function inlining.

  Figure 3.5 demonstrates this kind of recursive loop nest. The parameters `depth` and `n` indicate the current and maximum level of recursion (i.e., which dimension of the arrays `A` and `B` is being processed). The vector `v` is an array containing all the parameters of the loop nest. In this example, there are 6 parameters per array dimension: $\ell_A, h_A, s_A, \ell_B, h_B, s_B$. For the distributed array assignment statement, there are about 8 additional parameters per array dimension, representing the array distri-



(a) Runtime library routine approach

(b) Custom code generation approach

Figure 3.4: Two approaches to compilation, the runtime library routine approach and the custom code generation approach.

```
void arrayassn(A, B, depth, n, v[], subs[][2])
{   /* all variables are integers */
  if (depth < n) {
    ℓ_A = v[0]; h_A = v[1]; s_A = v[2];
    ℓ_B = v[3]; h_B = v[4]; s_B = v[5];
    j = ℓ_B;
    for (i=ℓ_A; i<=h_A; i+=s_A) {
      subs[depth][0] = i;
      subs[depth][1] = j;
      arrayassn(A,B,depth+1,n,&v[6],subs);
      j += s_B;
    }
  } else {
    i0 = subs[0][0]; j0 = subs[0][1];
    i1 = subs[1][0]; j1 = subs[1][1];
    ...
    A[i0][i1][...] = B[j0][j1][...];
  }
}
```

Extract parameters (ℓ_A:h_A:s_A) and (ℓ_B:h_B:s_B)

Execute next loop in nest

Recursive call

Extract array subscripts

Inner loop assignment

Figure 3.5: A recursive function that simulates a loop nest.

butions. Similarly, `subs` contains the vector of loop indices for the current iteration. Other tricks are necessary for the inner loop assignment to deal with the fact that the number of dimensions of `A` and `B`, as well as their dimension sizes, are not compiled into the library. There are obviously ways to improve the performance of this example at the source level. For example, instead of copying a single element at the bottom level of the recursion, copy an entire array section at the bottom level (in effect, manually inlining the final recursive call). Nonetheless, the recursion still prevents the compiler from inlining any of the other recursive calls.

An alternative approach allows optimization through function inlining, but at the cost of maintainability. This approach is to write a separate routine for each dimensionality, thus imposing an upper bound on how many dimensions an array assignment statement can have. However, when writing one of these routines, there is the problem of exponential blowup in the amount of code to write, for the following reason.

A d-dimensional array assignment requires a loop nest whose depth is proportional to d. The table-driven algorithms use a loop nest of depth exactly d; the linear algebraic algorithms use a loop nest of depth $2d$; the OSU algorithm's depth is $3d$; and the CMU algorithm's depth ranges from d to $3d$, depending on the distributions. For each dimension, there is an independent choice of loop nests to use. The table-driven algorithms have at least two choices of looping structure: one for "shift" communication and one for default communication. The OSU algorithm has four choices, depending on whether each distribution is viewed as virtual-block or virtual-cyclic. The implementation of the CMU algorithm actually has *nine* choices, depending on the distributions. For any particular algorithm, if there are d dimensions and k choices for each dimension, then k^d different loop nests must be written. This task is tedious even for small values of d and k, and severely hampers the maintainability of the routines.

Because of the complexity of dealing with multidimensional arrays, the Syracuse algorithm [66] actually advocates a separate communication phase for each dimension of the array. For example, to change an array distribution from (block,block) to (cyclic,cyclic), the algorithm first attacks one dimension by changing to, e.g., (block,cyclic), and then going from there to (cyclic,cyclic). For a d-dimensional array, this approach multiplies the total work (both communication and computation) by a factor of d. Although they claim an overall benefit in performance, their code appears to be almost an order of magnitude slower than the code produced by my system (see Chapter 5).

- **Multiple right-hand side terms.** This restriction can be lifted, for the most part, with a small amount of help from the compiler and few if any changes to the runtime library routines. Consider once again the array assignment statement

$$A[\ell_A\colon h_A\colon s_A] = B[\ell_B\colon h_B\colon s_B] * C[\ell_C\colon h_C\colon s_C] + D[\ell_D\colon h_D\colon s_D].$$

In a true runtime library implementation, the library has to be prepared to deal with an arbitrary number of right-hand side terms. However, a small amount of compiler support significantly reduces the complexity of the library. The compiler can first produce code that allocates the temporaries and calls the communication routines for each temporary. The communication in this example then reduces to executing the following three array assignment statements, each of which has a single right-hand side term:

$$\begin{aligned} T_B[\ell_A\colon h_A\colon s_A] &= B[\ell_B\colon h_B\colon s_B] \\ T_C[\ell_A\colon h_A\colon s_A] &= C[\ell_C\colon h_C\colon s_C] \\ T_D[\ell_A\colon h_A\colon s_A] &= D[\ell_D\colon h_D\colon s_D] \end{aligned}$$

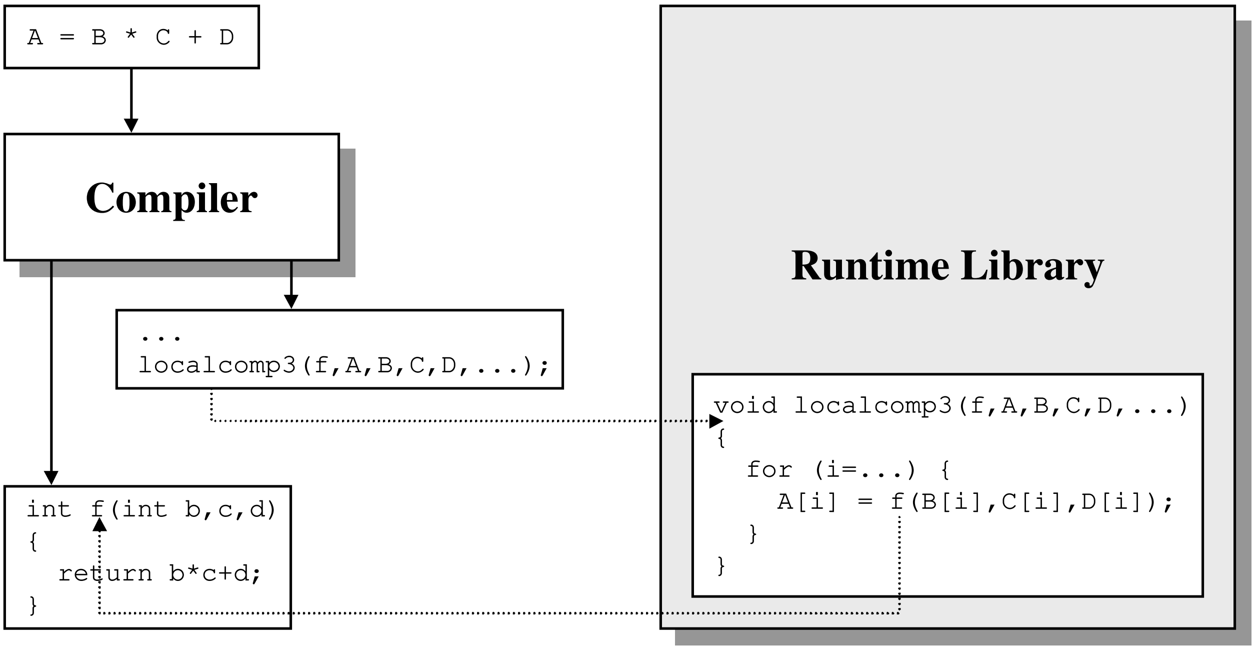


Figure 3.6: Using a runtime library routine to execute the local computation portion of an array assignment statement. In this example, the compiler produces a call to the runtime library routine `localcomp3`, and it generates a custom function `f`, which the runtime library routine calls, for executing the body of the inner loop.

The tricky part is in dealing with the local computation loop at the end; i.e., in this example,

$$A[\ell_A\!:h_A\!:s_A] = T_B[\ell_A\!:h_A\!:s_A] * T_C[\ell_A\!:h_A\!:s_A] + T_D[\ell_A\!:h_A\!:s_A].$$

The local iteration set computation (and thus the corresponding loop structure) is identical to that of the canonical array assignment statement. The divergence from the canonical case happens because, instead of a simple copy, both the "$+$" and the "$*$" operations must execute inside the inner loop. A fully general runtime library must be able to support an arbitrary number of such operators. A number of schemes to support this kind of library are imaginable. For example, the compiler can custom-generate the body of the inner loop, and produce code that passes a function pointer to the runtime library routine, as illustrated in Figure 3.6. Or the compiler can produce code that passes an expression tree that encodes the operators and the operands of the right-hand side expression, requiring the library routine to traverse the tree in the inner loop. However, this is a clear case in which efficiency and generality are mutually exclusive.

- **Scalar subscripts.** This restriction is the simplest to lift, although it also requires a modicum of compiler support. For dealing with scalar subscripts, there are two requirements beyond the standard requirements for the canonical case. First, the execution must be guarded by an ownership test of the scalar subscripts. Second, for the inner-loop array element accesses, the local memory mapping function must be applied to the scalar subscripts.

  Suppose that the array reference has s scalar subscripts and t subscript triplets (for a total of $s + t$ dimensions). We would like to pass this array to a library routine and have it treated as a t-dimensional array. The parallelizing compiler can easily produce the s ownership tests and compute the s local

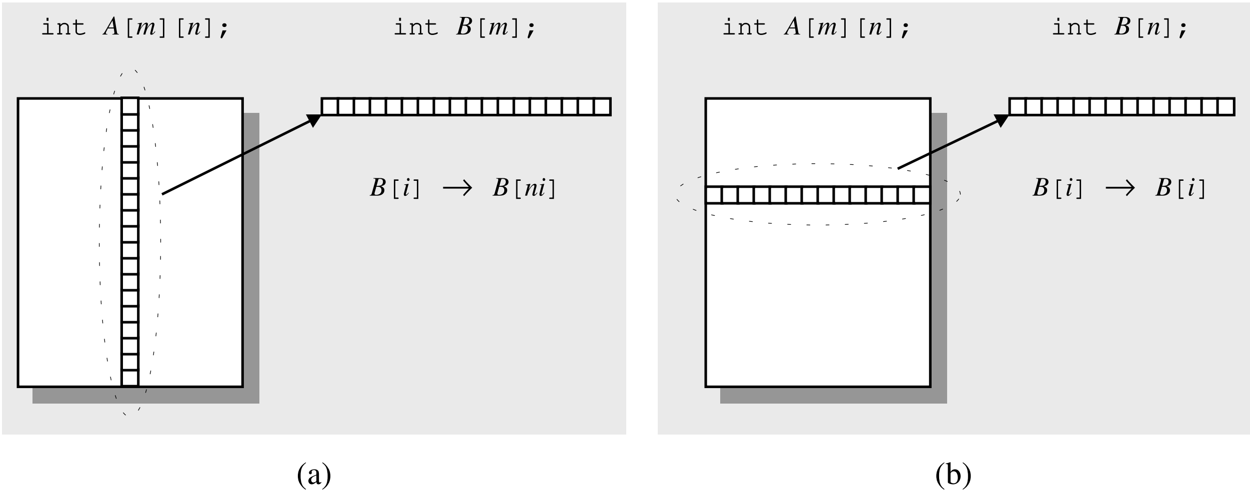


Figure 3.7: Treating a slice of a multidimensional array as a one-dimensional array by applying a multiplier to the subscript.

memory mappings before calling the runtime library routine, without a runtime performance penalty. The tricky part is to modify the library routine, which expects to be processing a t-dimensional array, to add the s scalar subscripts at arbitrary points within the t-dimensional array.

One way to accomplish this is for the compiler to compute a "multiplier" to be applied to each subscript in the library routine. To illustrate, consider Figure 3.7, which shows a one-dimensional slice of the two-dimensional array A passed to a function, where it is to be treated as a one-dimensional array B. In example (a), the slice is $A[0\colon m-1][j]$. Assuming a row-major ordering of array elements, each element of the slice is actually separated by n elements of the original array A. Thus the library routine must use n as its multiplier, changing every $B[k]$ reference to $B[n \times k]$. In example (b), the slice is $A[i][0\colon n-1]$, where adjacent elements of the slice are also adjacent in the original array A. Thus the multiplier passed to the library routine is simply 1.

For array references with more than one subscript triplet, the compiler computes a separate multiplier for each subscript triplet.

This discussion illustrates the difficulties in implementing a set of general runtime library routines for the array assignment statement (that is, libraries that handle the full array assignment in all its generality). As each restriction of the canonical array assignment is lifted, more and more complexity is added to the library implementation, until it is rendered unmaintainable. At the same time, runtime efficiency of the library degrades. Thus achieving generality has cost the library its maintainability and efficiency.

To summarize, the custom code generation approach offers runtime efficiency and generality at the cost of maintainability, whereas the runtime library routine approach offers maintainability at the cost of generality and possibly runtime efficiency. Thus the traditional compiler techniques are unsuitable for this kind of problem domain.

3.1.3 Architectural features

To complete the array assignment statement implementation, there are a few more requirements in addition to generating the communication sets and computation sets. These requirements are dependent on the architectural features of the target parallel machine.

- The processors must do any synchronization necessary to ensure that messages from different communication phases are not confused.
- Each processor must allocate all necessary communication buffers at the beginning and deallocate them at the end.
- The processors must interleave the sends and receives as necessary to prevent communication deadlock.
- On distributed memory machines with shared memory support, such as the Cray T3D [1, 9] and T3E [49], the algorithm might need to be modified so that the communication set loop writes directly to remote memory rather than going through communication buffers [56].

The specifics of these implementational details vary depending on the communication architecture model used (e.g., PVM [25, 63], MPI [39], or Intel NX [20]). Thus to be portable across architectures, the compiler-generated code (or the runtime library routines) must be easily modifiable at a high level to enable plugging in code for new architectural features.

3.1.4 Discussion

Most compiler systems for executing the array assignment statement are built on top of the runtime library routine model, due primarily to the ease in development, testing, and experimentation. To address runtime performance issues, the systems include special-case code for the more common occurrences. To address the generality issue, the systems also include extra code for, e.g., different numbers of dimensions, or else they simply ignore all but the canonical case. Only one system, the Fx compiler [61], uses a full custom code generation approach. The improved efficiency and generality come at a cost: in the original version of Fx, the custom code generation component consisted of over 15,000 lines of C code in the compiler, and implemented only the CMU algorithm. Generating code for a different algorithm would require an additional 10,000 to 15,000 lines of C code for each algorithm added to the Fx compiler. Interspersed with the algorithm component is the architecture component, also making it difficult to retarget the code to a different communication architecture.

Faced with these two choices (15,000 lines of C code per algorithm to generate efficient custom runtime code versus the far more manageable but less runtime-efficient and less general library routine approach), it is not surprising that developers choose the runtime library routine approach. The extra effort in writing additional optimized libraries for the common cases seems a small price to pay for not having to put customized code generation into a compiler.

In the next section, I present a customized code generation approach that provides maintainability in addition to runtime performance and generality.

3.2 Code composition

Several domains in addition to the array assignment statement have high-level constructs with the properties described at the beginning of this chapter, as I discuss in Chapter 6. The compiler writer in such a domain is faced with the problem of how to produce general and efficient code within a maintainable framework. A solution to this problem is obviously one that preserves the advantages of the two traditional approaches while eliminating the disadvantages. This means being able to write the algorithm code in a compact, straightforward way, but in a way that leads to good runtime performance, and in a way that easily generalizes to handle both simple canonical instances of the problem domain and more general but less common instances.

While a runtime library routine approach easily provides maintainability, it is in general hard for it to provide good efficiency as well. It is possible to write optimized versions for common cases, but it is difficult to predict in advance all the optimizable common cases likely to occur; furthermore, as the number of specially optimized cases increases, maintainability correspondingly decreases. Harder still is to make the runtime library general while holding onto maintainability and efficiency, as illustrated in Section 3.1.2 for the array assignment statement. For this reason, my solution is based on a custom code generation approach. This approach automatically provides efficiency and generality; what remains is to develop a method on top of custom code generation that provides maintainability as well.

The solution I propose is a technique called *high-level code composition*. In this customized code generation approach, the high-level code to generate, as well as instructions for generating it, appears in a straightforward fashion external to the compiler. There is a *composition system* loosely coupled with the compiler that uses these compile-time instructions and runtime code sequences to produce code. The key features of code composition are:

- It is a custom code generation approach, rather than a runtime library routine approach.
- The code sequences to be produced appear external to the compiler. These code sequences are called *code constructs*.
- 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.

In the remainder of this section, I discuss these points in greater detail.

3.2.1 Custom code generation

Code composition is classified as a custom code generation approach, rather than as a runtime library routine approach. I chose the term "composition" for two reasons. One definition of the word composition is "A putting together of parts or elements to form a whole." [41] This definition reflects the process of building up a customized sequence of code at compile time, to be executed at run time. There is also a mathematical sense of the word, as in "function composition." Function composition is a method of taking two or more functions and putting them together in a way that results in a more complex and more interesting function. This sense of the word reflects the idea of fitting together basic building blocks in a way that matches the structure of the particular construct being compiled.

As a custom code generation approach, code composition retains the advantage of producing more efficient code than the runtime library approach, with compile-time constants compiled into the runtime code. In addition, it makes it straightforward to produce code for general instances of the problem domain as well as the canonical case.

Any custom code generation approach has these two properties. To combine these benefits with the maintainability benefits of the runtime library approach, special care must be taken. The composition system must be designed so that it is easy to write, read, and modify the code that is to be produced. This is where the initial version of the Fx compiler went wrong. The expansion of a few hundred lines of algorithm code into 15,000 lines of compiler code rendered the CMU algorithm virtually unmaintainable within the Fx framework.

3.2.2 Code templates

To support maintainability, code composition requires that all the code sequences to generate, as well as instructions for composing the code sequences, appear external to the compiler, rather than being embedded in the compiler. These constructs are divided into two groups: *code constructs* and *control constructs*.

- **Code constructs** are the sequences of real code that the compiler produces, to be executed at run time. In a system for code composition, the code constructs are written almost exactly as they would appear in a runtime library. Entire loops, conditionals, and other constructs appear intact and are simple to read and write. Because the code constructs are not obscured in the internals of the compiler, they are easy to maintain.

- **Control constructs** are the compile-time instructions specifying how to put together the code constructs to form code that executes an entire high-level operation at run time. The control constructs actually form a language that is executed (or interpreted) at compile time. There are two types of control needed at compile time: control needed for selecting optimized special-case code versus the general case, and control needed for generalizing from a canonical case. For special-case code selection, the control constructs needs to select one of several possible code blocks to produce. For generalizing, the control constructs needs to piece together several blocks of code to form the whole.

In the same way that a program is divided into functions, the control constructs (as well as the code constructs that they enclose) are broken up into *code templates*. A code template can be thought of as a function in the control language. It serves two purposes: it is an abstraction for breaking up the code into manageable chunks, and it is a unit that can be called as a subroutine at compile time.

Because the control and code constructs appear external to the compiler, they are not inextricably locked into the compiler itself. This decoupling provides a higher degree of maintainability: development, maintenance, and bug fixing of the algorithm do not require rebuilding the compiler for each change. Note that the runtime library routine approach has a similar property, in that changes to the runtime library routines usually only require rebuilding the libraries and not the whole compiler.

3.2.3 Composition system

Code composition requires a *composition system* to generate code at compile time from the code templates. The composition system can be tightly coupled with the compiler (i.e., directly built into the compiler), or it can be loosely coupled (i.e., not built in, but invoked as a separate tool or library). A loosely coupled composition system can be more easily coupled with a new compiler, but it requires a careful design of the protocol for communicating with the compiler. Figure 3.8 shows the design of a compiler loosely coupled to a composition system.

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.

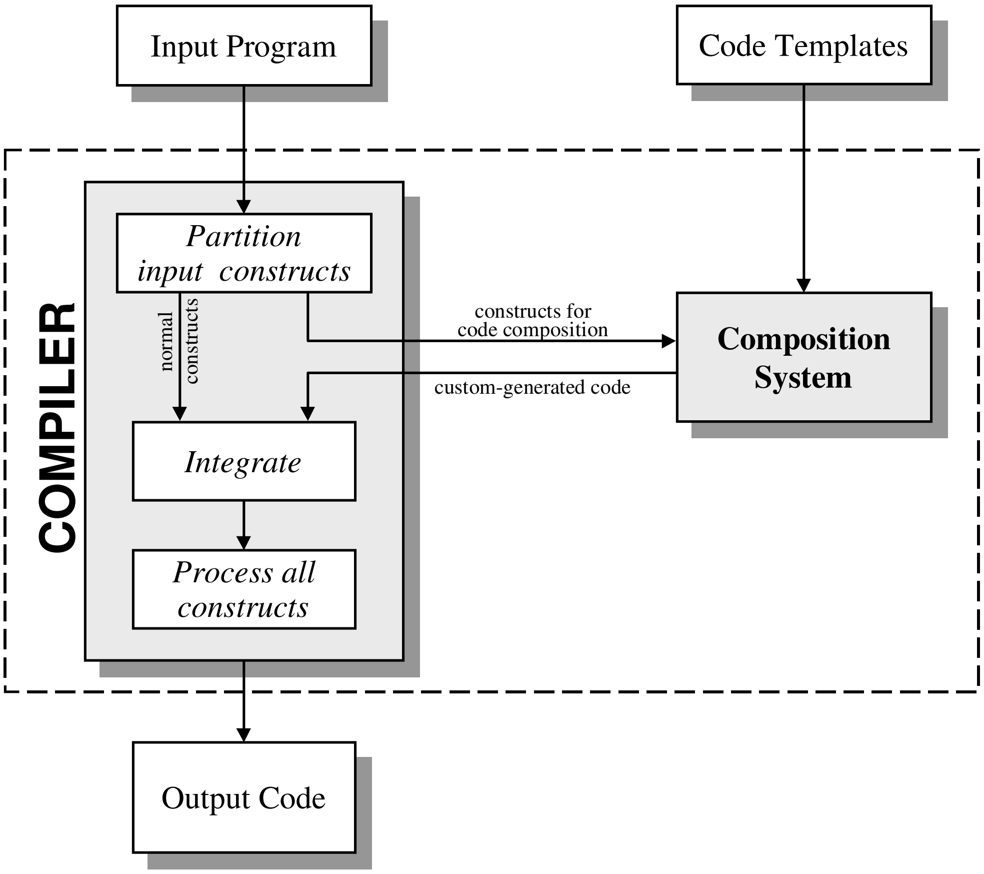


Figure 3.8: A loosely coupled composition system. A tightly coupled composition system is integrated directly into the compiler, simplifying the protocol for data exchange.

The code templates are a combination of control and code constructs; after the control execution completes, only code constructs remain. These code constructs are returned to the compiler, with the intent that they be integrated into the runtime executable that the compiler produces. Before returning the code constructs to the compiler, the composition system has the option of performing a set of standard compiler optimizations. Even though the compiler is likely to be doing optimizations as well, there are reasons, as discussed in Section 4.6, for the composition system to perform some of these optimizations as well.

3.3 Composition language issues

A language for composing code at compile time is a meta-language consisting of two levels. The language of the control constructs forms the higher level, and the language of the code constructs forms the lower level. Together, they form the composition language in which the code templates are written. The language at each level is itself a separate programming language, and as such, there are practical concerns that drive the design of each language. In this section, I discuss these practical concerns, in terms of the features needed in the code constructs and those needed in the control constructs. In addition, I discuss the syntactic and semantic relationship between the control and code constructs.

3.3.1 Code constructs

Code constructs are the pieces of actual code that the composition system composes to form the resulting function. There are no strict requirements on the form of the code constructs. In principle, they can be arbitrary pieces of text; the target language is arbitrary. Feasible choices include assembly code, a high-level language like C, or an intermediate representation language.

In practice, two concerns drive the choice of language for the code constructs. The first concern is motivated by the need to quickly and correctly convert runtime library code into a template framework. As an example, consider the array assignment statement. Because the communication set generation code is so complex, the easiest way to test the algorithm is to develop a standalone runtime library routine for just the canonical case. After the algorithm is tested and debugged, it can be converted into template code to be used to handle the general array assignment statement. To make the conversion to templates as straightforward and error-free as possible, it is wise to choose the language in which the original library was implemented as the language of the code constructs.

The second concern relates to optimizations. As I describe in Section 4.5, to achieve the best efficiency in the generated code, it may be necessary for the composition system to include a global optimization framework. Some language constructs, such as the `goto` statement, add considerable complexity to the analysis required for the optimization framework, and might be eliminated from the code constructs for the sake of simplicity, especially for such rarely used constructs. In addition, to support common domain-specific operations, we may choose to add some relevant operators to the language to increase optimization potential. For example, if `floor()` and `ceil()` are common operations, they could be added to the language as first-class unary operators, so that optimizations like constant folding can be done simply.

3.3.2 Control constructs

Control constructs provide a description of how to compose the templates. They constitute a small language that the composition system interprets at compile time. As such, the control constructs must be designed as one would design a real programming language, containing variables, conditionals, procedure calls, and so forth. It is important to choose a syntax that is easily distinguishable from the syntax of the code constructs. Beyond that, there are a few basic programming language constructs that should be a part of the control constructs.

- **Control procedures and procedure calls.** As a full-featured programming language, the control constructs need the abstraction of a procedure. A single control procedure is called a *code template*. When the template is invoked, the control constructs in the template are executed and the resulting code constructs are emitted (or perhaps globally optimized before being emitted).

  The template needs to be able to take parameters as input and to return results to the calling template. There are several reasonable ways to pass parameters (e.g., call by value, call by reference, call by value-result, or a combination of these), and several reasonable ways to return results (e.g., using output parameters or having an explicit return value). If the template explicitly returns a result, and templates are allowed to have side effects, then the evaluation order of components of a control expression needs to be well defined. For example, if templates `t1` and `t2` both have side effects, and the control expression `call(t1)-call(t2)` is evaluated, then the resulting control expression can depend on depend on which of `call(t1)` and `call(t2)` is evaluated first. Note that there is no such ambiguity if templates return results only through output parameters, as in `call(t1,result1); call(t2,result2); result1-result2`.

- **Control variables.** The control constructs need a variable space that is separate from that of the code constructs. The control constructs can modify the control variables, and whenever a control variable appears in a code construct, its value is substituted. Because of this substitution, a control variable never appears in the final generated code.

The input parameters to a template constitute a set of control variables. In addition, the template writer might need to declare a set of scratch variables to perform compile-time calculations; these scratch variables are also control variables. In a functional control language, scratch variables are declared and set using the `let` construct; in an imperative/declarative language, scratch variables are explicitly declared, and set using a control assignment statement (see below).

It may also be useful to have a set of global control variables. Without proper attention to the language design, though, this can lead to a potential problem: how to distinguish one class of variable from another. Altogether, there are three classes of variables in the templates: local control variables, global control variables, and code variables. (A code variable is a variable referenced in the program that the composition system generates.) The class of a variable usually cannot be determined purely from context, since many syntactic constructs allow variables of any class to appear therein.

Without global control variables, distinguishing local control variables from code variables is fairly simple: local control variables are declared within the template. With global control variables, though, distinguishing them from code variables may be difficult. Code variables are similar to global control variables in that their life extends beyond the execution of the template, and their declarations may appear in a completely different template.

One way to solve this problem is to require *all* variables, rather than just local control variables, to be declared in each template in which they are used. Another possibility is to use a different syntax for global control variables. This change in syntax should be reasonably non-intrusive, since the use of global control variables in a well-structured set of templates should be fairly rare.

- **Control variable assignment.** Once control variables are declared, there must be a mechanism or mechanisms to set their values. During a control function call, there is implicit control variable assignment: the template parameters are instantiated to the passed-in values. Other more explicit assignment constructs are possible; e.g., a `let` construct for a functional composition language, or an explicit control assignment statement for an imperative/procedural language. Naturally, the control assignment syntax must be different from the assignment syntax in the code constructs.

- **Control test.** One of the control constructs must be a control conditional statement, which tests the value of an expression and, depending on the result, executes one of the branches. The typical use is to test whether certain information is known at compile time, composing a specially optimized piece of code if so, and employing the default general case otherwise. Another common use is to provide a guard for a recursive control function call.

  When executing the control test, the conditional expression must evaluate to a constant so that the composition system can determine which branch to take during control execution. If the expression contains a reference to a code variable whose value is indeterminate at compile time, the system will be unable to determine which branch of the control conditional to compose. In this situation, it must signal a compile-time error.

- **Variable renaming.** This 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 do this, we need a control construct that produces a new variable name. A reasonable way to do this is, e.g., to provide a control function or operator that takes a string and an integer constant as input, and produces a new symbol whose name is the concatenation of the two inputs. A more cumbersome approach is to use a function like Lisp's `gensym`, which gives less control over the resulting symbol name, making the template code somewhat harder to manage and the generated code less readable.

- **Control loop.** Using the above set of basic control constructs, the way to write a loop is to use a recursive control procedure call guarded by a control conditional statement. This kind of programming style is clumsy to write in, as every loop has to be isolated into its own recursive template. For this reason, it is useful, although not strictly necessary, to add an explicit loop to the set of control constructs. Executing a control loop causes potentially several copies the same code to be generated repeatedly until the control loop terminates. As in the control conditional, the termination condition of the control loop must always evaluate to a constant.

3.3.3 Execution model

Most programmers are familiar with the concept of code that is executed at compile time. For example, the C preprocessor fits into this category. It has compile-time directives, such as `#define`, `#include`, and `#if`, that can be used to perform an extremely limited amount of computation. Under the standard execution model, the compile-time code is first executed, leaving only runtime code. To make a composition system intuitive for the developer to use, the execution semantics should follow this model.

On the surface, this requirement seems unnecessary. After all, what other choices are there? In Section 4.6.5, I describe an execution model that essentially makes a single pass through the templates, executing the control constructs and performing global optimizations on the resulting runtime code at the same time. Because of the difficulty of this kind of implementation, it is tempting to add complexity to the execution semantics so that the implementation can be simplified. However, for the composition system to be usable, it is wise to make the execution semantics as straightforward as possible, and put some extra work into the implementation to support the simpler semantics.

3.3.4 Combining code and control constructs

One of the properties of code composition is that both code and control constructs appear together in the code templates. There is then the question of what, if any, relationship the 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. We can think of the C language as an extremely crude composition system, in which the C preprocessor commands are control constructs and the rest are code constructs. Under this model, the constructs are interleaved using the lexical style, because the preprocessor commands are allowed to surround any lexical piece of the C code, and not limited to surrounding entire statements.

Figures 3.9 and 3.10 demonstrate these two styles for writing templates, assuming C-like control constructs and Fortran-like code constructs. The purpose of the code in the figures 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`.

There is an additional style, called the *emit* style, that is an instance of the lexical style. Figure 3.11 demonstrates the emit style for the template of Figure 3.10. Syntactically, the emit style has no code constructs. Instead, it uses a special control function called EMIT to generate each piece of the resulting code.

```
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 3.9: 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 3.10: A template for constructing a loop nest, using the *lexical* style.

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

Figure 3.11: A template for constructing a loop nest, using the *emit* style, which is an instance of the lexical style.

Its advantage over the strict lexical style is that it is easier to construct a grammar for the composition language; because there are no syntactic code constructs, there is no need to disambiguate the syntax of the control constructs from that of the code constructs. In addition, if the target language changes, there is no need to modify the grammar of the composition language; only the templates need to be changed. The disadvantage of the emit style is that it is much more tedious to convert an existing runtime library routine into a template. Instead of simply copying the relevant statements from the library into template code, the statements have to be inserted into EMIT statements.

Although not a strict requirement of a composition system, it is preferable that the code and control constructs interact through the syntactic style. The primary benefit of the syntactic style is readability of the template code. It is easier to develop and maintain code written with this style, and it is also easier to 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.

3.4 Summary

Complex high-level operations like the array assignment statement present problems for the traditional compilation approaches (i.e., custom code generation and the use of runtime library routines). These approaches trade off at least one of three important properties: efficiency, generality, and maintainability. My solution, a technique called code composition, provides all three properties. Code composition consists of code templates that contain control and code constructs, and a composition system that composes the templates at compile time to create a function that is executed at run time.

The control constructs form a separate programming language that is executed at compile time, and should be designed with a set of features described in this chapter. The code constructs should have syntax and semantics as close as possible to the language in which runtime libraries are written (usually the C language). Within the code templates, the control and code constructs should be combined using the syntactic style, rather than the lexical style.

I have outlined a minimal framework for designing the control constructs, the code constructs, and the composition system. In the next chapter, I describe Catacomb, an implementation of a composition system that represents one point in the design space.

Chapter 4

Catacomb

To illustrate the usefulness of a composition system, I developed Catacomb, a composition system for generating C code. Catacomb is not a standalone compiler; rather, it is designed to operate as a library that is linked into a compiler, generating code for a particular class of high-level language constructs. From the input source program, the compiler separates out these high-level constructs, passing each one in turn to Catacomb. When invoked, Catacomb generates code for that construct, and this code is combined with the compiler's output to form the runtime executable. To make Catacomb as inter-operable as possible with a variety of compilers and problem domains, I provided the following features:

- To make the conversion from runtime libraries to Catacomb templates as easy as possible, the composition language is based on C, which is the most common language choice for writing such libraries.
- Communication of data structures between Catacomb and the compiler is handled by a small, well-defined set of interface routines, to localize the work required to integrate Catacomb into a new compiler.
- The control constructs can be easily extended to meet the needs of a new compiler or problem domain, without any modification to Catacomb itself. In the same way, new domain-specific information can be attached to the control variables in the composition language without modifying Catacomb.

In this chapter, I describe the control and code constructs of Catacomb's composition language, and I describe implementation and optimization issues and challenges. In addition, I describe in detail how Catacomb interfaces to a compiler, and how it can be extended for new problem domains.

4.1 Terminology

In this chapter, I use a number of terms and phrases whose precise meanings are important to understand. Some are first used in this chapter; others were introduced in the previous chapter.

- **Compile-time** and **runtime execution**: The purpose of a composition system is to generate code by processing code templates. I refer to the act of processing the code templates as *executing templates*, or *control execution*. More precisely, executing code templates means executing or *interpreting* the control constructs in the code templates, leaving behind a sequence of code constructs. One template can *invoke* another template (or the compiler can invoke a template), causing the template to be executed. The composition system executes the templates at compile time, and the sequence of code constructs produced is executed at run time.

- **Control variables** and **code variables**: As discussed in Section 3.3, a control variable is a variable that is used only at compile time during control execution. The control variables are not referenced in code that is generated, and thus they require no storage in memory at run time. The value of a control variable is always known during control execution.

  A *code variable* is a variable used by the program that the composition system generates, for which explicit storage is allocated at run time. Unlike a control variable, the value of a code variable at a particular point, called its *runtime value*, cannot in general be determined by the compiler.

- **Strict** and **lazy evaluation**: During control execution, control expressions are formed and have to be evaluated. Under a *lazy evaluation* execution model, evaluation of these control expressions is delayed until the result is actually needed. Under a *strict* execution model, the control expressions are fully evaluated immediately after they are formed.

- **Propagation value**: One of the tasks of a global optimizer is to perform constant propagation. For each reference of a code variable, if the optimizer can prove that its runtime variable is always a particular constant, then the compiler assigns that constant to the variable as its *propagation value*. (The optimizer can also set the propagation value to a variable or expression, rather than a constant, to support copy propagation.) If a variable's runtime value is indeterminate (e.g., it could have two possible values, depending on which branch of an `if` statement is taken at run time), it has a *null* propagation value.

- **Setting/clearing a propagation value**: The optimizer analyzes the program and *sets* propagation values when it can prove that a code variable has a particular runtime value. When a runtime value becomes indeterminate, the optimizer *clears* the variable's propagation value; i.e., it assigns the null propagation value.

4.2 Language infrastructure: expressions

Both the code and control constructs use *expressions* as a key component of the implementation. Because Catacomb is designed to be C-like, the expressions are also designed to be close to those of C.

Values and expressions in Catacomb consist of the following:

- **Constants**. This includes integer, floating point, and string constants. Integer constants are represented with the `int` type in C, floating point constants with `double`, and string constants with `char*`.

- **Symbols** (i.e., simple variable references).

- **Expressions**. This includes unary, binary, and ternary expressions. It also includes type casts and the `sizeof` operator. The `sizeof` operator has to be treated specially because its argument can be either an expression or a type.

- **Array references and function calls**. Array references are optionally tagged with an index permutation (see Section 3.1.1).

- **Subscript triplets**. This data type is needed for handling array references in High Performance Fortran. The inclusion of subscript triplets in Catacomb was the result of an early design decision when targeting HPF. All three components of the subscript triplet are optional; when absent, default values are used instead. The default for the lower bound is 0; the default for the upper bound is one less than the corresponding array dimension size; and the default for the stride is 1. Note that even though the subscript triplet is represented internally, there is no explicit syntax in Catacomb to create or manipulate a subscript triplet.

In addition, I added several useful binary operations as first-class binary operators: `min`, `max`, `pmod` ("positive modulo"), `fldiv` ("floor-div"), and `ceildiv` ("ceiling-div"). `min` and `max` are the obvious operators. `pmod` is similar to the C modulus operator, except that the property $0 \leq \texttt{pmod}(a, b) < b$ is guaranteed when $b > 0$. (In C, the sign of the result is implementation-dependent; some mathematical formulas require the positive result.) For integers a and b, $\texttt{fldiv}(a, b)$ and $\texttt{ceildiv}(a, b)$ are defined as the floor and ceiling, respectively, of a/b. Including these as first-class operators simplifies optimizations like expression simplification and constant folding. These additional operators are also represented only internally, having no explicit syntax in the code or control constructs.

4.3 Language design: code constructs

The code constructs in Catacomb consist of constructs from the C programming language. To simplify the analysis in the optimization phase (see Section 4.5 and Section 4.6), I omitted the `break`, `continue`, `switch`, and `goto` statements, and the `enum` declaration. In addition, C function definitions are not included in the code constructs, since Catacomb is designed to produce a single function. Apart from these modifications, and a small change in variable declarations (see Section 4.4.8), the remainder of C is intact in the code constructs.

Why choose C as the language of the code constructs? The primary reason is to satisfy one of the goals of code composition: to make it simple to convert runtime library routines into a code composition framework. These runtime libraries are most commonly written in C, making C the obvious choice for minimizing the amount of work required for the conversion. A side effect of this choice is that Catacomb produces C code as output, because of the obvious straightforward translation from the input code constructs.

4.4 Language design: control constructs

Because Catacomb's code constructs have the syntax of C, its control constructs are designed to have the feel of syntactic extensions to C. As such, the control constructs have C-like syntax, and they interleave with the code constructs using the *syntactic style* (as opposed to the *lexical style*; see Section 3.3.4). This decision makes it easy to extend a C parser to handle the control constructs, and it makes programming of the control and code constructs straightforward. In this section I describe in detail Catacomb's control constructs.

To aid in the presentation of the control constructs, I use a running example in this section. Figure 4.1 shows an annotated set of Catacomb templates whose 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.

Control conditional

Control function call

Declaration of template name, input arguments

```
TMPL loopnest(n,array,upper)
{
  cif (!CONSTANT(n))
    PRINT("Error: non-constant ", n);
  else
    include loopnest1(0,n,array,upper);
}
```

Maximum recursion depth
Local control variable

Variable renaming

Recursive include

```
TMPL loopnest1(i,n,array,upper)
DEPTH 5;
LOCAL subs;
{
  cif (i < n) {
    int idx#i;
    for (idx#i=0; idx#i<upper; idx#i++)
      include loopnest1(i+1,n,array,upper);
  } else {
    include genlist(n,idx,subs);
    A[subs] = 0;
  }
}
```

Output variable

Automatic variable renaming

External functions for list manipulation

Control loop

```
TMPL genlist(size, ivar, var list)
LOCAL i;
APPEND (i) ivar;
{
  list := MAKE_LIST(size,0);
  i := 0;
  cwhile (i < size) {
    list := REPLACE_LIST_ITEM(list,i,ivar);
    i := i + 1;
  }
}
```

```
include
  loopnest(3,A,100);
```

```
int idx_0, idx_1, idx_2;
for (idx_0=0; idx_0<100; idx_0++)
  for (idx_1=0; idx_1<100; idx_1++)
    for (idx_2=0; idx_2<100; idx_2++)
      A[idx_0][idx_1][idx_2] = 0;
```

Figure 4.1: 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.

- `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.

4.4.1 Template header

A control function in Catacomb is called a template. Each template contains a mixture of control and code constructs. The template definition itself is syntactically similar to a C function declaration. The template header contains the name of the template and a list of arguments passed to the template. After the template header come three optional declarations:

- The maximum recursion depth (see Section 4.4.4)
- A list of template-local control variables (see Section 4.4.2)
- A set of "automatic variable renaming" declarations (see Section 4.4.7)

Following the header comes the actual template code, enclosed in a pair of C-style curly braces.

4.4.2 Control variables

For compile-time computations, Catacomb provides control variables. The syntax of a control variable is the same as that of a code variable—a standard symbol. Catacomb internally maintains separate symbol tables for control and code variables. When a variable is referenced, and the context allows for either a control variable or a code variable, it is treated as a control variable if it appears in the control symbol table, and as a code variable otherwise. Thus control variables override code variables of the same name.

Control variables are further partitioned into local and global variables. Local control variables are the ones declared in the current template; these consist of the template arguments and the set of control variables declared to be local in the template header. All other control variable references are global. Catacomb uses static scoping to resolve control variable references.

Control variables are untyped, and can hold values of any type (as listed in Section 4.3) supported in Catacomb. In addition to these, control variables can contain *lists*. Catacomb templates use a list data structure to construct function argument lists and array subscript lists, when the length of the list is not fixed, but rather is dependent on some other compile-time parameter. Catacomb provides a mechanism for constructing lists and for querying and changing list elements; the `genlist` template in Figure 4.1 uses two *external functions* (see Section 4.4.9) to create a list and set its values.

To complement these data types, Catacomb provides control constructs to query the type of an expression (see Section 4.4.9). It is crucial to the concept of code composition for Catacomb to be able to ascertain an input expression's type, using the information to produce the right code sequence for the context.

4.4.3 Control variable assignment

Catacomb adopts the `:=` operator for control variable assignment. The control assignment statement takes the form "`LHS := RHS;`". The left-hand side of the assignment statement must be a control variable (or must be a "variable-renaming" expression that evaluates to a control variable; see Section 4.4.7). When executing a control assignment statement, Catacomb evaluates the right-hand side expression and sets the left-hand side variable's value to the result. Subsequently, every time the left-hand side control variable appears in an expression (regardless of whether the expression is used in a control or code context), the variable's value is substituted. Because of this strict evaluation scheme, a control variable's value can never contain another control variable; only constants, code variables, and expressions are possible.

This design decision led to an unfortunate side effect. In a control assignment statement, if the left-hand side symbol is not a local control variable, it is treated as a global control variable. If the variable was not previously in the global control symbol table, it is added. Afterwards, it is no longer possible to directly reference a code variable of the same name. For example, if `x` is not declared as a local control variable, the code

```
x:=3; x=y; print(x);
```

evaluates to the (illegal) code

```
3=y; print(3);.
```

After the control assignment statement executes, `x` subsequently can never be accessed as a code variable. This is a problem because a poor choice of a global control variable name (or a simple typo) can shadow a code variable's name, long before the code variable's use and in a template far away, with no chance of removing the shadowing. Note that a local control variable can also shadow a code variable's name, but the effect is limited to the template in which the local control variable is declared, hence short-lived and easy to track down.

A more appealing way to reference global control variables, and thus solve this problem, is to introduce explicit syntax for global control variable references. For example, the syntax `G(x)` could be used to instruct Catacomb to treat `x` as a global control variable reference. Although this kind of syntax is a bit cumbersome to use, a well-structured program should minimize its use of global variables, so its impact should be minimal. This simple change will appear in a future version of Catacomb.

4.4.4 Control function call

Catacomb uses the `include` statement to execute a control function call. This statement contains a template name and a list of arguments to pass to the template. An example of the statement (as in Figure 4.1) is

```
include loopnest(3,A,100);
```

In Catacomb, the template conceptually has type "void" and does not return a value to the caller.

By default, Catacomb uses *call-by-value* to pass arguments to the template. It also allows values to be passed to the caller through the use of *call-by-value-result*, which differs from call-by-value in that the final value of the argument is passed back to the caller. In the template header, the template writer flags these output parameters with the `var` keyword, similar to the syntax of passing arguments by reference in Pascal. The `genlist` template in Figure 4.1 declares `list` as an output parameter. In the `include` statement, the corresponding argument must be a control variable (obviously, only control variables, and not constants or expressions, can be passed by reference). When the template execution completes, the final value of the argument is copied back into the control variable that was passed in.

A template can recursively include itself. If the recursive call is not properly guarded by a conditional, it can lead to an infinite include loop. To help prevent this infinite recursion, Catacomb imposes an upper limit of 20 on the number of concurrent includes of each template. Associated with each template is a counter, initialized to 0, that is incremented each time the template is included, and decremented at the end of the template's execution. If the counter exceeds the limit, Catacomb assumes infinite recursion and aborts all execution. If needed, the maximum recursion depth can be set to any value (or removed altogether) by using the `DEPTH` statement in the template header.

The `loopnest1` template in Figure 4.1 uses the `DEPTH` statement to specify a maximum include depth of 5. Once called by the `loopnest` template, it can include itself up to four more times, allowing a loop nest to be generated with maximum depth four. It is important to realize that this `DEPTH` limitation is provided as a programming safeguard, not because of any implementational difficulties, and is easily overridden by the template programmer if necessary.

4.4.5 Control conditional

Catacomb's control conditional statement is modeled after the `if` statement in C, and is thus called `cif` (i.e., "control-if"). Like the `if` statement, the `cif` statement includes a THEN branch and an ELSE branch. When executing a template, Catacomb first evaluates the conditional expression. If the expression evaluates to a nonzero constant, the THEN branch is executed; if the expression evaluates to zero, the ELSE branch (if any) is executed. The expression *must* evaluate to a constant; if not, Catacomb signals a compile-time error and aborts.

4.4.6 Control loop

Catacomb provides a control looping mechanism, modeled after the `while` statement in C, and thus called `cwhile` (i.e., "control-while"). Like the `cif` statement, the conditional expression must evaluate to a constant. Unlike the `include` statement, there is no facility to detect or prevent potentially infinite control loops.

The `genlist` template in Figure 4.1 uses a `cwhile` to iterate across the array subscript list, setting each value in turn. In this example, the control loop contains no code constructs; if it did, one instance of the code construct would be generated for every iteration of the loop.

4.4.7 Variable renaming

To create new control and data variables, Catacomb provides the left-associative binary # operator, called the concatenation operator. The left operand, which must evaluate to a symbol, is the base variable name, and the right operand is an expression that must evaluate to an integer or string constant. The result is a new variable whose name is the concatenation of the two parts. (When the right operand is an integer constant, Catacomb also inserts an underscore character, "_", between the two components of the final variable name, to help avoid variable name conflicts.) 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`. If the left operand does not evaluate to a variable, or the right operand does not evaluate to an integer or string constant, Catacomb signals an error.

When converting a library routine into a Catacomb template, the intent often is to compose that block of code several times, with each occurrence using a unique set of variables. To make the variables unique in each instance of the template, the template programmer uses the concatenation operator to append the instance number to each variable occurrence, and provides the instance number in a control variable. Figure 4.1 demonstrates this concept for creating a set of unique loop induction variables. In the `loopnest` template, `idx` is the base variable name, and `i` is the control variable containing the instance number.

The problem with this approach is that, when converting a library routine, every such variable reference in the template needs to be changed to a concatenation expression. This source code transformation is tedious and error-prone for the template programmer. To address this problem, Catacomb provides a mechanism for automatic variable renaming within a template (illustrated in the `genlist` template of Figure 4.1). In the template header, there can be a number of lines of the form

```
APPEND (expr) x,y,z;.
```

In this example, throughout the template, each occurrence of variable `x` is changed to `x#expr`, and similar for variables `y` and `z`. The end result is that the template programmer need only identify the variables that need to be unique, and list them in the template header, and Catacomb automatically performs the concatenation operations. This allows templates to look virtually identical to the original runtime libraries on which they are based, and yet be automatically composed multiple times without name conflicts.

4.4.8 Declarations

Catacomb treats declarations of code variables as a control construct, because creating declarations for code variables sometimes requires more functionality than provided by standard C declarations. In particular, a variable may need to be created whose type is dependent on the type of another variable, or whose array dimension sizes are dependent on those of another array.

Declarations of code variables are allowed anywhere in the template. A code variable's declaration may even follow the variable's first use, as long as the declaration is encountered somewhere during control execution. Catacomb allows all of the basic C types in declarations. In addition, it provides the `typeof()` operator, whose result can be used to declare a variable's type. It takes a variable as input and returns the type of the variable. For example, if `x` is a float, then the declaration

```
typeof(x) *p;
```

makes `p` a pointer to a float.

For array declarations, if a dimension size argument evaluates to the special "list" data type in Catacomb, then that dimension size declaration is expanded into a sequence of dimension size declarations. Without this feature, it would not be possible to create new arrays (e.g., temporaries) without knowing the number of dimensions at template writing time.

If a declaration for a code variable is omitted, Catacomb assumes it to be an integer and declares it accordingly. (This default typing decision was a poor design choice; all variables should have explicit type declarations, if only to help discover typing mistakes in the templates.) However, for all function call references, the function must be explicitly declared. The principal reason for this is that beginning template writers have a tendency to occasionally forget to use the `include` keyword in a control function call, making it look like a normal function call, and leading to linking errors.

Catacomb includes two additional keywords for type declarations: `nodecl` and `macro`. The `nodecl` keyword indicates that the variable or function is already declared in a system include file, and that Catacomb should not emit a declaration for it. Common examples are

```
nodecl malloc();      nodecl printf();      nodecl NULL;
```

The `macro` keyword is similar to `nodecl`, but in addition, it tells Catacomb that the function call is actually an invocation of a C macro, and thus any symbol arguments passed in are subject to modification. This information is needed by Catacomb's optimization framework to prevent too-aggressive optimization.

4.4.9 External functions

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. In general, these control operations are used to test certain high-level properties of their inputs. For example, there are functions to test whether the input evaluates to a constant, or a variable, or a binary expression, or an array reference. Other functions return the number of dimensions of an array, or the nth subscript of an array reference. There are external functions to manipulate lists, and others to provide access to the additional first-class binary operators (described in Section 4.3) that Catacomb adds. An external function is essentially an escape hatch that allows the template programmer to execute an arbitrary C function at compile time. The templates in Figure 4.1 use several external functions: `CONSTANT` (tests whether its input evaluates to a compile-time

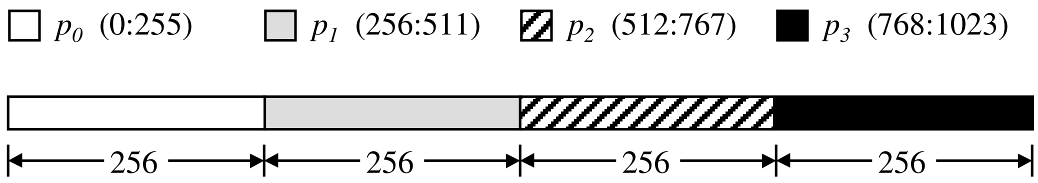


Figure 4.2: Bounds information available to Catacomb. The figure represents a 1024-element array with a block distribution over 4 processors (i.e., the block size is 256).

constant), `PRINT` (displays a message at compile time), `MAKE_LIST` (creates a list of a given size, with all elements initialized to a particular value), and `REPLACE_LIST_ITEM` (sets a particular element of a list).

Catacomb provides a default set of general-purpose external functions. It is easy to extend the set of external functions (see Section 4.7). For example, the array assignment statement module extends the set to include functions that query the block size, number of processors, and other parameters of an array distribution. I provide a complete list with descriptions in Appendix A.

4.5 Global optimization framework

4.5.1 Standard global optimizations

Catacomb implements several standard global optimizations [2, 17]:

- **Constant folding** and other expression simplifications. Every time a new expression is created, Catacomb tries to fold constant expressions and exploit algebraic identities to simplify the resulting expressions.
- **Constant/copy propagation**.
- **Dead assignment elimination**.
- **Unreachable code elimination**. This involves eliminating loops and branches of `if` statements when the condition is a compile-time constant.
- A limited form of **loop peeling**. If Catacomb determines that a loop body will be executed for at most one iteration, it transforms the loop into either straight-line code (exactly one iteration), dead code (exactly zero iterations), or an `if` statement (either zero or one iteration).
- **Code hoisting**. When possible, assignments are hoisted out of loops whose bodies are executed at least once.

These are optimizations that should be found in any optimizing C compiler that compiles Catacomb's output. Regardless, Catacomb needs these optimizations to support *bounds optimizations* (described below) and to support the model of control execution that I describe in Section 4.6.5.

4.5.2 Bounds optimizations

In addition to these standard global optimizations, Catacomb implements several nonstandard global optimizations. All of these optimizations are based on *bounds analysis*. Bounds analysis, which is similar to the technique of *generalized constant propagation* [70], is based on the following observation. Sometimes, even though the compiler cannot determine a specific value for a variable or expression, it can determine that it must belong to a range of values. Consider an example from the domain of the distributed array assignment statement. Figure 4.2 depicts a 1024-element array block distributed across 4 processors. With just this information, we can determine the following:

- A given processor ID p falls within the range $[0{:}\,3]$.
- The first index in any processor's ownership set falls within the range $[0{:}\,768]$.
- The last index in any processor's ownership set falls within the range $[255{:}\,1023]$.

Catacomb associates with each variable or expression a lower and upper bound. If no information is available about a lower or upper bound, it is assigned $-\infty$ or $+\infty$, respectively. Note that it is possible for one bound to be known but not the other. For example, without knowing the upper bound, we still know that a lower bound on a distribution block size is 1.

Using bounds information, Catacomb implements a set of nonstandard global optimizations:

- **Bounds folding and expression simplification**. When new expressions are created, Catacomb uses a simple bounds calculus to assign bounds to the new expression. For example, if $a \le x \le b$ and $c \le y \le d$, then properties like $a + c \le x + y \le b + d$ and $a - d \le x - y \le b - c$ and $\min(a, c) \le \min(x, y) \le \min(b, d)$ hold. There are more interesting cases as well. For example, if $0 < x$, and k is a positive constant, then $0 \le (x \bmod k) \le k - 1$. In addition, if $a \le x \le b$ and $0 < c \le y \le d$, then $\lfloor a/d \rfloor \le \lfloor x/y \rfloor \le \lfloor b/c \rfloor$.

  In some cases, expressions can be simplified purely by using properties of their bounds. To illustrate, if $a \le x \le b$ and $c \le y \le d$ and $b \le c$, then $\min(x, y) = x$ and $\max(x, y) = y$. Paraphrasing, if the two subexpressions have non-overlapping bounds, then the min or max expression simplifies to one of the subexpressions. Furthermore, if the lower and upper bounds turn out to be identical, then the entire expression can be replaced with the (constant) bound. As an example, taken from Figure 4.2, where $0 \le p \le 3$, the expression $\lfloor (1023 - p)/4 \rfloor$ simplifies to 255.

- **Bounds propagation**. In conjunction with constant and copy propagation, Catacomb propagates the bounds of the right-hand side expression in an assignment statement.

- **Propagation within a scoping level**. When the condition of an `if` statement has the form `(var == c)`, and `c` is a constant, then the value of `c` can be propagated for `var` within the scope of the appropriate branch of the `if` statement. Similarly, when the condition has the form `(var <= c)`, then `c` can be propagated as the upper bound of `var` within the branch. When the condition is a conjunction like `(e1&&e2&&e3)`, then each conjunct can be individually propagated within the branch. Finally, the negation of the condition can be propagated through the `else` branch of the statement.

  Catacomb implements this optimization for the `if` statement, and it propagates similarly within the bodies of `for` and `while` loops.

- **Detection of monotonic loop induction variables**. When a variable is modified within a loop body, it is generally the case that no value can be propagated into or out of the loop for that variable. However, it is sometimes possible to allow bounds information to propagate into or out of the loop.

  Within a loop body, Catacomb tries to detect variables that either monotonically increase or monotonically decrease. For monotonically increasing variables, it is safe to propagate the lower bound; similarly, for monotonically decreasing variables, it is safe to propagate the upper bound.

4.5.3 Bounds representations

Catacomb uses two values for its bounds representation: a lower bound and an upper bound. This two-value representation is surprisingly effective in providing optimization potential, in the sense that my studies of Catacomb-generated code for the array assignment statement, using the block and cyclic data distributions, show that no additional bounds information could further optimize the code. However, when considering the more general block-cyclic distribution, this two-value approach fails to capture all of the bounds information needed to generate the best code. As an example that arises in practice, consider the expression $\min((1023-2p)\%8,1)$, where p has bounds $[0\colon 3]$. With a little thought, one can realize that this expression simplifies to 1. However, the two-value bounds representation can only produce the bounds $[0\colon 1]$ on the expression.

Extensions to this two-value bounds representation are possible. For example, we could add a positive *stride* parameter to the representation. A stride parameter would allow us to specify that the expression modulo the stride is fixed. To be useful, though, we need to know the precise value of the expression modulo the stride. This information could come from the lower and/or upper bound, or we could explicitly add a fourth parameter giving its value. This fourth parameter would serve the same role as the *representative* of a regular set described in Section 2.2.3. (Recall that a representative r of a subscript triplet sequence $\ell\colon h\colon s$ is a value for which any member of the sequence is equal to $r + ns$ for some integer n.) As an example, if an integer expression has lower bound 1, upper bound 100, stride 8, and representative 6, then we know that the expression is in the sequence $(6, 14, 22, \ldots, 94)$, and that the expression modulo 2 is 0. As another example, in Figure 4.2, the first index in any processor's ownership set falls within the range $[0\colon 768\colon 256]$, and the last index falls within the range $[255\colon 1023\colon 256]$.

This enhancement to the two-value bounds representation, along with the associated bounds calculus, improves the quality of the generated code for the array assignment statement with block-cyclic distributions. Consider the above expression, $\min((1023 - 2p)\%8, 1)$, where p has bounds $[0\colon 3]$. For simplicity, let us use a three-value strided bounds representation, giving p the bounds $[0\colon 3\colon 1]$. The expression $2p$ has bounds $[0\colon 6\colon 2]$, giving the expression $1023-2p$ the bounds $[1017\colon 1023\colon 2]$. Because 8 (the divisor) is a multiple of 2 (the stride), the expression $(1023 - 2p)\%8$ has bounds $[1\colon 7\colon 2]$. Finally, using this resulting lower bound of 1, we have proven that $\min((1023 - 2p)\%8, 1) = 1$.

There are a few reasons for Catacomb to provide these optimizations. First, the optimizations are important for the performance of the resulting code that executes on the target machine, and are motivated by real examples. The problem is that the bounds optimizations I describe above are not provided by most optimizing C compilers, and thus Catacomb needs to provide them. Providing them requires implementing a full global optimization framework that includes the other kinds of standard propagation and expression simplifications described above.

Why do standard optimizing compilers not provide bounds optimizations? The primary reason is that human programmers rarely write code that is amenable to bounds optimizations. The code tends to be studied and simplified algebraically before it even reaches the compiler. Only machine-generated code depends so heavily on this kind of optimization framework to improve its performance; such code is, however, relatively rare. But suppose an optimizing compiler were to include bounds optimizations. It would still achieve only limited success, because some bounds information is available only within the context of Catacomb itself. For example, an array distribution's block size and number of processors are known only within Catacomb. (On the other hand, consider the fact that Catacomb provides external functions that allow the template programmer to annotate lower and upper bounds on variables. Although it seems unlikely, perhaps the target optimizing compiler would have similar annotations, allowing Catacomb to pass its bounds information along.)

The second reason for Catacomb to include bounds and other optimizations is that an optimization framework is important for giving the template programmer (and the composition system designer) feed-

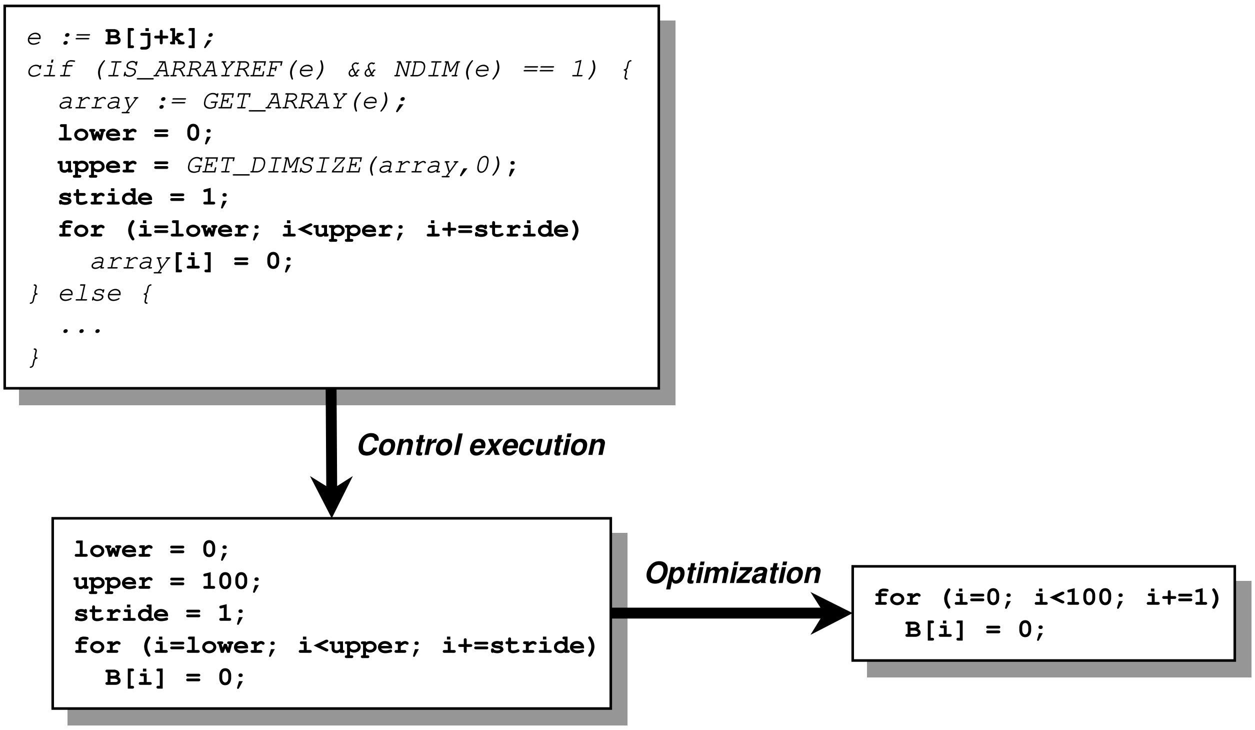


Figure 4.3: A two-phase execution model. In the first phase, control constructs are executed, leaving only code constructs. In the second phase, global optimizations are performed on the remaining code constructs.

back on the quality of the machine-generated code. The template programmer typically wants to examine the resulting code to see whether special-case templates are necessary and feasible to write. Without an optimization framework, it can be hard to judge the quality of the generated code; the alternative is to study the assembly code produced by the target optimizing compiler.

There is a further reason for Catacomb to provide a global optimization framework: the need to provide the control constructs with access to propagation information. I discuss this problem next.

4.6 Interpretation of control constructs

So far, I have discussed Catacomb's control constructs and how they are executed. In addition, I have discussed Catacomb's global optimization framework, including its standard and nonstandard set of global optimizations. However, I have not discussed how (or whether) the control constructs and the global optimization framework should fit together.

4.6.1 Two-phase execution model

The most obvious and straightforward way to integrate the execution of the control constructs and the global optimizations in a composition system is to make them completely independent. This suggests a two-phase execution approach, as illustrated in Figure 4.3. In the first phase, the composition system executes only the control constructs, leaving just the code constructs. No optimizations are performed in this phase, except for simple operations like constant folding and expression simplification required to make composition

```
x = ... ;
cif (x == 0) {
  /* special case */
} else {
  /* general case */
}
```

(a)

```
x = ... ;
cif (CONSTANT(x) && x == 0) {
  /* special case */
} else {
  /* general case */
}
```

(b)

```
x = ... ;
if (x == 0) {
  /* special case */
} else {
  /* general case */
}
```

(c)

```
x = ... ;
if (CONSTANT(x) && x == 0) {
  /* special case */
} else {
  /* general case */
}
```

(d)

Figure 4.4: Code composition decisions based on a code variable, in a two-phase execution model. Example (a) results in a compile-time error because `x`'s value is unknown during control execution. Example (b) *always* results in the general case code sequence. Example (c) may cause both code sequences to be generated, depending on the expression assigned to `x`. Example (d) always results in the general case code sequence, unless lazy evaluation of external functions is used.

decisions. For example, in Catacomb, we need to fold constants and simplify expressions occurring in the conditional of a `cif` test or a `cwhile` loop to present the template programmer with the expected semantics of the statements.

After all of the control constructs have been processed, the resulting code constructs are passed off to the global optimizer, and then emitted. Note that after the control constructs are executed, the code could be emitted, without going through the global optimization phase. In a system like Catacomb, however, this would mean losing the advantage of the framework for nonstandard global optimizations.

An attractive feature of the two-phase approach is the simplicity in both semantics and implementation. The semantics are easy for the template programmer to understand, provided that control variables and other control constructs are easily distinguishable from the code constructs. The implementation, divided into two independent steps, is much simpler than the alternatives described below.

4.6.2 Problems with the two-phase execution model

There is a key disadvantage to the straightforward two-phase approach to code composition: code composition decisions can only be made based on the values of control variables. As an example, consider the code in Figure 4.4a. The goal is to be able to compose exactly one of the two branches based on compile-time knowledge of `x`. Note that `x` is a code variable, not a control variable (the assignment to `x` uses a normal assignment statement). This particular example actually causes a compile-time error, because the value of `x` is unknown during control execution, and thus the comparison to 0 is indeterminate. Even with the error-free code in Figure 4.4b, the `else` branch is always taken, for the same reason.

With certain changes in template coding style and execution semantics, it is possible to come closer to the goal. For example, we could change the `cif` statement into an `if` statement, as depicted in Figure 4.4c.

If a constant is assigned to `x` in the first line, the combination of constant propagation and unreachable code elimination leaves only the code in the one of the branches of the `if` statement (depending on whether the constant is 0). However, in general, if the optimizer is unable to determine that `x` is a constant, then both branches of code are produced. But this defeats the purpose of the original `cif` coding style, which is to compose a specially optimized piece of code if the value of `x` is known at compile time, and a default piece otherwise; *exactly* one branch is meant to be generated. The style in Figure 4.4d guarantees that the `if` condition evaluates to a constant and thus exactly one branch of the statement is produced; unfortunately, as in Figure 4.4b, the general case is *always* produced, because the external function `CONSTANT` is evaluated during the control phase instead of during the optimization phase.

To summarize the problem so far:

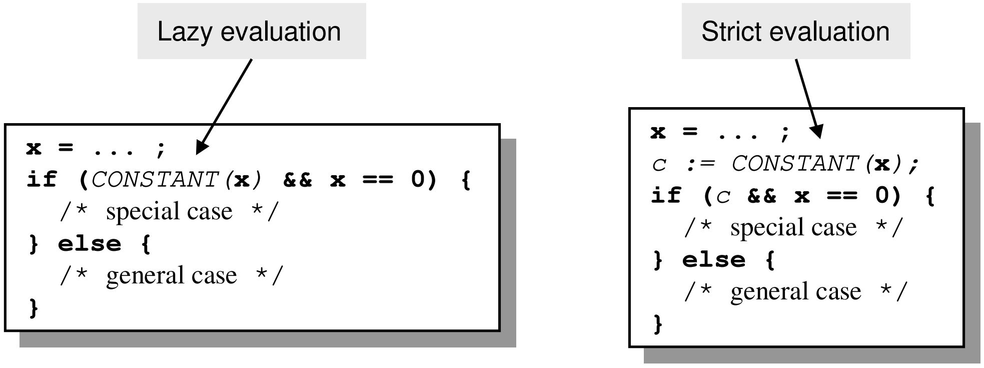
- We want to generate either the general case code or the special case code, but not both, based on whether we know the runtime value of a code variable at compile time.
- Control constructs like `cif` statements and external functions cannot be used, because they force a decision to be made before the optimizer has a chance to discover runtime values of code variables.
- Code constructs like `if` statements also cannot be used reliably, because the optimizer may not be able to prove one of the branches unreachable, and thus leave both the general case code and the special case code.

4.6.3 Lazy evaluation of external functions

One possible solution to this problem is to make the external functions *lazy* wherever possible. This means that by default, they are evaluated during the later optimization phase. In certain contexts, though, the external functions *must* be evaluated during the control execution phase. For example, the conditions of `cif` and `cwhile` statements must be strictly evaluated during control execution, as must the right-hand portion of the variable-renaming `#` operator. Using an `if` construct in the code in Figure 4.4d, in conjunction with lazy evaluation of the `CONSTANT` external function, leads to the desired effect of producing exactly one of the branches of code, without the surrounding `if` statement.

Unfortunately, lazy evaluation of external functions leads to compile-time performance problems (i.e., Catacomb takes an unacceptably long time to execute). The primary reason for this is that part of the process of handling the array assignment statement in a fully general fashion is to use control constructs to break down the input expressions into their constituent pieces, analyze each piece, and then build a new expression out of those pieces. At this point, each array reference turns into a large expression involving external functions that decompose and rebuild the original array reference. Expressions that extract parameters from the new array reference (e.g., dimension sizes or distribution block sizes) become even larger. Expressions comprised of these large expressions become huge. Frequent copying and evaluation of such huge expressions causes the compile-time performance problems. This execution strategy (breaking down expressions, analyzing their components, and rebuilding them) is not limited to the array assignment statement; any problem domain that operates on such expressions will also use this strategy.

The explosion in expression sizes results from the use of lazy evaluation in *control assignment statements*. The obvious question, then, is whether strict evaluation of external functions could be applied to the right-hand side expressions of control assignment statements (in addition to `cif` and `cwhile` statements and right-hand portions of the concatenation operator, where strict evaluation is already required). The answer, unfortunately, is no. Consider the code in Figure 4.5. If external functions in control assignment statements are strictly evaluated, but external functions in `if` conditionals are lazily evaluated, then the two code fragments have different semantics. In code fragment (a), the evaluation of the `CONSTANT`



(a) (b)

Figure 4.5: Unexpected semantics result from strictly evaluating external functions in control assignment statements. The evaluation of `CONSTANT` in code fragment (a) is delayed as expected, allowing the special-case code to be generated if `x` is proven to be a constant. The control assignment statement in fragment (b) forces strict evaluation at that point, precluding any chance of generating the special-case code.

external function is delayed, allowing the special-case code to be generated if the optimizer can prove that `x`'s runtime value is a constant. But in fragment (b), the external function is strictly evaluated during control execution, forcing the general-case code always to be generated. A template programmer should emphatically *not* be required to understand this subtle difference between two programs that intuitively appear to have identical semantics.

Clearly, then, strict evaluation of external functions should not be applied to control assignment statements if lazy evaluation is used with `if` statements and other code constructs. However, there are other possible modifications to the lazy evaluation approach, each of which improves the compile-time performance and has "less questionable" semantics. Define a *strict evaluation site* to be a point in the template where strict evaluation of external functions during control execution is required; strict evaluation sites include the condition if a `cif` or `cwhile` statement, as well as the right-hand portion of a variable-renaming expression. The modifications to the evaluation scheme are the following:

1. Lazily evaluate the external functions on the right-hand side of array assignment statements, as usual. However, any time a control variable appears anywhere in a strict evaluation site, do the following. Evaluate the variable as usual, including the required strict evaluation of all resulting external functions. But after this evaluation, assign the result to the control variable, so that it no longer contains any external functions. (Normally, the control variable would be left unchanged.) Put another way, we *collapse* the value of a control variable any time the control variable appears in a strict evaluation site.

   The semantics of this approach are only slightly different from the fully lazy (but inefficient) approach. Figure 4.6 demonstrates a difference. In the fully lazy approach, the control variable `c` evaluates to `CONSTANT(x)` everywhere it is used. Because the evaluation of `CONSTANT` in the `if` statement is delayed until the optimization phase, the value of `x` that is tested is known to be 0 (i.e., a constant) at that point, and thus the condition evaluates to 1. In the modified approach, the `cif` statement locks `c` to the value 0, and thus it is still 0 at the `if` statement. It is unclear which behavior is preferable.

2. Alternatively, do the same as above, but instead of collapsing the value of each control variable at a strict evaluation site, collapse them when they are passed as arguments to control procedures.

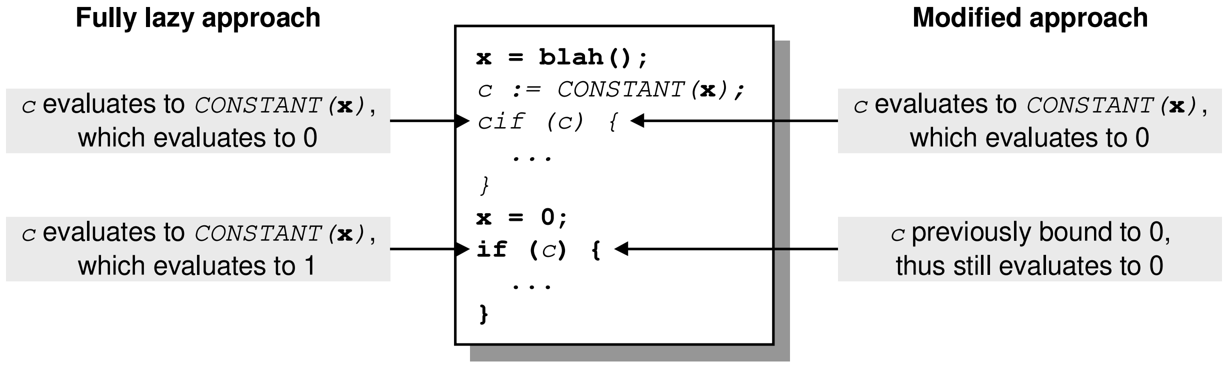


Figure 4.6: A semantic difference between the fully lazy approach for external function evaluation and the modified approach in which external functions in control variables are collapsed at strict evaluation sites. In the fully lazy approach, the value of control variable `c` changes without an explicit control assignment, but in the modified approach, its value stays fixed after the first strict evaluation site.

Unfortunately, the semantics of this approach suffer from the same problems as the approach of strictly evaluating the right-hand sides of control assignment statements. To illustrate, Figure 4.7 gives two code fragments that seem like they should behave the same. However, fragment (a) delays evaluation of `CONSTANT` in the `if` statement, whereas in fragment (b), control variable `c` has been collapsed as a result of the `include` statement before it reaches the `if` statement (thus the evaluation of `CONSTANT` is effectively strict).

3. Yet another possible approach is to put the strict-versus-lazy evaluation rules explicitly in the hands of the template programmer. In this scheme, all external functions are by default strictly evaluated, and there is a special annotation to force a particular instance of an external function to be evaluated lazily (or conversely, lazy evaluation could be the default). However, the correct choice of whether or not to use the annotation requires the template programmer to have sophisticated knowledge of the implementation.

To summarize, the simple two-phase execution model causes a problem because it does not allow the template programmer to make code composition decisions based on propagation values of code variables. By introducing lazy evaluation semantics for the external functions, and encouraging a style of template programming that favors the use of `if` rather than `cif`, we can alleviate the problem to a certain degree. Unfortunately, the cost is a huge drop in compile-time efficiency. The compile-time efficiency can be brought back to acceptable levels by creating a set of rules that specify additional points at which external functions must be strictly evaluated. The cost of this change, though, is a new semantics that essentially require the template programmer to understand the low-level implementational details of the composition system.

4.6.4 Inadequacy of lazy evaluation

Unfortunately, the two-phase execution model in conjunction with lazy evaluation of external functions still prevents other kinds of optimizations. As an example, consider the template shown in Figure 4.8. This template is designed to compute the greatest common divisor (gcd) of two positive integers, using Euclid's gcd algorithm [35, Volume 2]. The basic idea of the template is to test whether the two inputs are both compile-time constants, and if they are, compute their gcd at compile time, otherwise produce code that

Lazy evaluation

```
x = ... ;
c := CONSTANT(x);
if (c && x == 0) {
   /* special case */
} else {
   /* general case */
}
```

(a)

Strict evaluation

```
x = ... ;
c := CONSTANT(x);
include do_nothing(c);
if (c && x == 0) {
   /* special case */
} else {
   /* general case */
}
```

(b)

Figure 4.7: Unexpected semantics result from collapsing the external functions in control variables passed as arguments to control functions. Under this model, the condition of the `if` statement in code fragment (a) evaluates to 1, whereas in code fragment (b), the condition evaluates to 0. This model is problematic because most programmers would expect the two code fragments to produce identical results.

calculates their gcd at run time. Note that this simplified version is incomplete in two ways. First, a more complete version would also directly compute the gcd if either of the inputs is known to be 1 at compile time. Second, in practice we usually need to implement Euclid's extended gcd algorithm to compute three values: given inputs a and b, compute x, y, and g such that $ax + by = g = \gcd(a, b)$. Computing the additional parameters x and y requires only minimal changes to the basic gcd algorithm.

The idea behind this gcd template is that if invoked with the code

`include gcd(4,6,g);`,

the resulting code `g=2;` is produced. But what happens when invoked with

`a=4; b=6; include gcd(a,b,g);`?

Because both input parameters to the template are symbols rather than constants, the `cif` statement causes the runtime loop code to be produced. Ideally, though, we would like to have some way of having the simple code `g=2;` produced. The trick of changing the `cif` to an `if` statement and using lazily-evaluated external functions does not work here, because during control execution, the `cwhile` statement does not have access to the results of the constant/copy propagation phase. The external function `CONSTANT` only has access to the propagation values because the lazy evaluation delays its execution; the `cwhile` condition is a strict evaluation site, and thus the evaluation of the external function cannot be delayed.

It is important to note that even though the complex code generated is equivalent to the simple `g=2;` code, there is little chance that any optimizing compiler will attempt to unroll the `while` loop and discover that simple code. Thus it is important for the composition system to be able to produce the simple code sequence directly.

4.6.5 Single-phase execution model

To address this problem, as well as the various semantic problems of the variations on the two-phase approach described above, I developed a single-phase execution model and implemented it in Catacomb. The basic idea behind the single-phase approach is to perform global optimizations (in particular, constant, copy, and bounds propagation, as well as unreachable code elimination) at the same time as control construct execution, so that the control constructs can use propagation information to make decisions based on the

```
TMPL gcd(a,b,result)
LOCAL u, v, q, t;
{
  cif (CONSTANT(a) && CONSTANT(b)) {      Compute gcd at compile time if
    u := a;  v := b;                      input parameters are constants
    cwhile (v != 0) {
      q := u/v; t := u-v*q; u := v; v := t;
    }
    result = u;
  }
  else {
    tmp_u = a; tmp_v = b;                 Otherwise, produce code that
    while (tmp_v != 0) {                  computes gcd at run time
      tmp_q = tmp_u / tmp_v;
      tmp_t = tmp_u - tmp_v*tmp_q;
      tmp_u = tmp_v; tmp_v = tmp_t;
    }
    result = tmp_u;
  }
}
```

Figure 4.8: A simplified Catacomb template for computing the gcd of two values.

values of code variables. In the gcd template example, under the single-phase execution model, the `cif` and `cwhile` constructs have access to the fact (proven by the optimizer) that the input parameters are 4 and 6, respectively (rather than just code variables `a` and `b`), and thus Catacomb produces the simpler code sequence.

While the single-phase approach is attractive in theory, it is hard to implement. The key problematic issue is that in the template code, the control constructs and the code constructs follow different threads of control. For example, consider a `for` loop that contains a control assignment statement in the loop body. Regardless of how many times the loop body executes at run time, the semantics dictate that the control assignment statement is executed exactly once at compile time. The same holds for any control construct that appears within a loop body. Similarly, control constructs are executed in both branches of an `if` statement, regardless of which branch is taken at run time, or whether the code constructs in one branch are eliminated at compile time.

4.6.6 Implementing the single-phase approach using data flow techniques

The usual method of performing propagation in an optimizer is through the data flow method. However, for performing propagation in conjunction with control execution, standard data flow techniques are inappropriate. The main problem is that the structure and contents of the control flow graph itself are dependent on the control construct execution. Thus, not only do propagation values fluctuate in the control flow graph until convergence, but the control flow graph itself fluctuates until convergence.

It is possible, however, to use data flow techniques for propagation by first making a set of transformations on the input templates. The idea is to transform the combination of code and control constructs into purely code constructs, while preserving the original execution semantics of the control constructs. The basic transformation rules are the following:

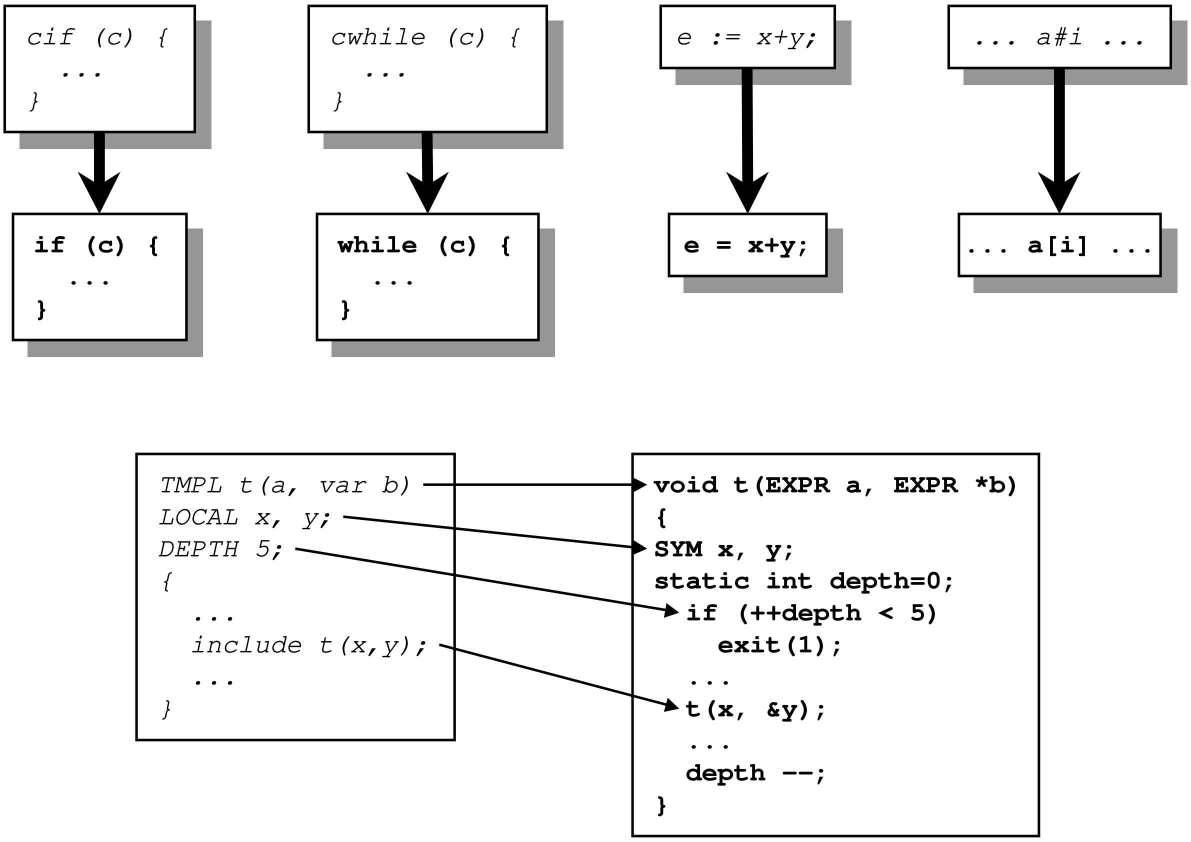


Figure 4.9: Transforming Catacomb control constructs into a form amenable to data flow analysis.

- **`cif`**, **`cwhile`**: change to standard `if` and `while`. See Figure 4.9.
- **Templates**: change to C functions. See Figure 4.9.
- **`include`**: change to a function call. See Figure 4.9.
- **Control assignment**: change to a standard assignment statement. See Figure 4.9.
- **Concatenation operator**: change to an array reference. See Figure 4.9.
- **`if`**: Conditionals are a bit trickier, since we have to ensure that control constructs inside the branches of the `if` statement are executed regardless of the condition. We start by evaluating the condition. Then we execute both branches in sequence, wrapping the condition (or its negation) around each contiguous block of code constructs, and applying this `if` rule recursively. Control constructs are left unchanged. See Figure 4.10.
- **Loops**: A loop remains a loop, but the termination condition is modified if necessary to ensure that the body is executed at least once to catch any control constructs. Then, the control constructs are masked so that they are only executed during the first iteration, and the code constructs are similarly masked so that they do *not* execute for a 0-iteration loop. See Figure 4.10.

The transformations of statements inside loops and `if` statements are similar to the technique of *if-conversion*, originally proposed by Allen, Kennedy, Porterfield, and Warren [3]. If there is a sequence

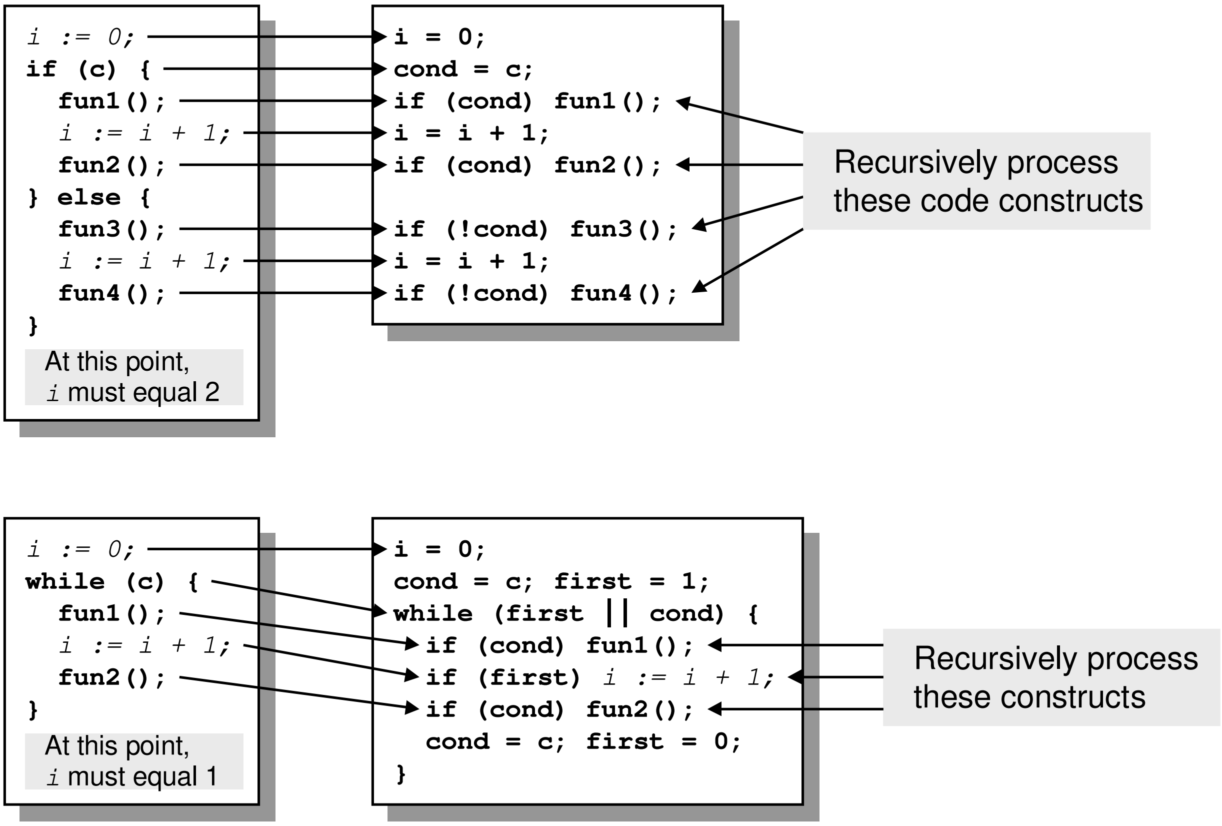


Figure 4.10: Transforming Catacomb code constructs that enclose control constructs into a form amenable to data flow analysis.

of statements in a branch of an `if` statement, if-conversion transforms the construct into a sequence of statements, each of which is guarded by a separate `if` statement. Applying the `if` condition separately to each statement makes the code more amenable to vectorization, using, e.g., the `WHERE` construct of Fortran 90.

After these transformations are complete, the resulting code can be analyzed with standard data flow techniques, with the following caveat. For every control construct in the original set of code templates, its corresponding code construct in the resulting transformed flow graph must be "removed", in the same sense that control constructs are removed after control execution. This means that function calls to templates must be fully inlined; loops that result from `cwhile` statements must be completely unrolled; and assignments that result from control assignment statements must be fully propagated (meaning that anywhere such a variable appears in the flow graph, a precise value can be assigned). To allow this fully aggressive propagation, we also have to enforce the requirement that the condition of a `cif` or `cwhile` statement is a compile-time constant.

After applying these transformations, problems still remain. Optimizations that appear straightforward in the original template code become considerably less apparent in the transformed code. For example, consider the code in Figure 4.11. Before the transformation, the `z=2` assignment is obviously unreachable and can be eliminated. However, after the transformation, sophisticated analysis techniques are required to discover that the corresponding assignment is also unreachable in the transformed code. Essentially, any control construct has the effect of separating the prior code from the following code, making it difficult to propagate values from one section to the other.

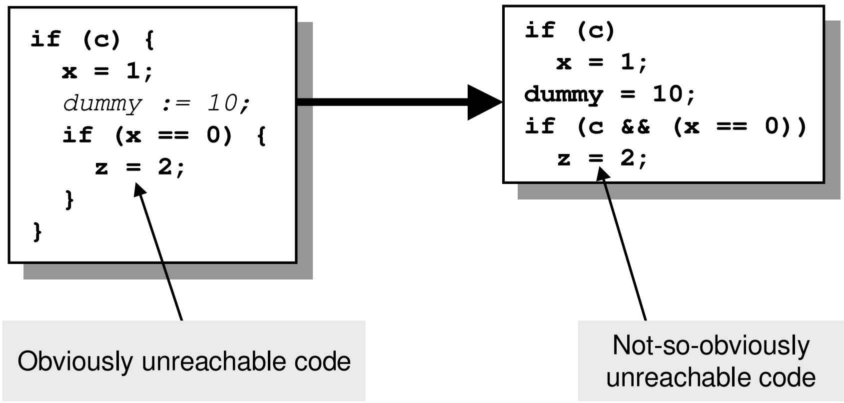


Figure 4.11: Some transformations make it much more difficult to realize the full benefits of copy propagation and unreachable code elimination.

4.6.7 Implementing the single-phase approach: Catacomb's solution

In my implementation of Catacomb, I take a different approach. In this approach, Catacomb begins at the entry point of the template and executes one statement at a time, in the forward direction. If the statement is a control construct, it is executed in the obvious fashion (exactly as it would be executed under the simple two-phase model). This includes external functions; because the execution model has only one phase, there is no need for lazy evaluation.

At the same time as Catacomb executes the control constructs, it performs all of its global optimizations (except for dead assignment elimination, which requires traversing the flow graph in the reverse direction). To deal with the chicken-and-egg problem between the control and code constructs, I use a *backtracking* mechanism, which I describe at the end of this section. This backtracking facility allows Catacomb to make a *checkpoint* at any time during its execution, and, later in its execution, to backtrack to the checkpoint if necessary.

Catacomb's code constructs can be divided into three categories: loops, conditionals, and expressions (including assignments and function calls). In the remainder of this section, I describe how the code constructs are processed under the single-phase execution model.

Assignments

For assignments, Catacomb performs constant/copy propagation as well as bounds propagation. Copy propagation is limited to expressions that Catacomb deems "simple enough"; an expression is defined to be simple enough if the depth of the expression tree is at most 1, and it contains no function calls or array references. For example, 1 and x and $x + y$ are simple expressions, but $x + y + z$ is not. (Clearly, this definition subsumes constant propagation, since constants fit the definition of "simple.")

To process an assignment statement, Catacomb first rebuilds the right-hand side expression by substituting the propagation values for variables in the expression. (No substitution is performed when the address of the variable is taken.) The right-hand side of the assignment statement is replaced with this rebuilt expression. If the following conditions hold:

- the new right-hand side expression is deemed "simple enough"
- the left-hand side is a simple variable (as opposed to an array or structure reference)
- the left-hand side variable does not appear as a term in the new right-hand side expression

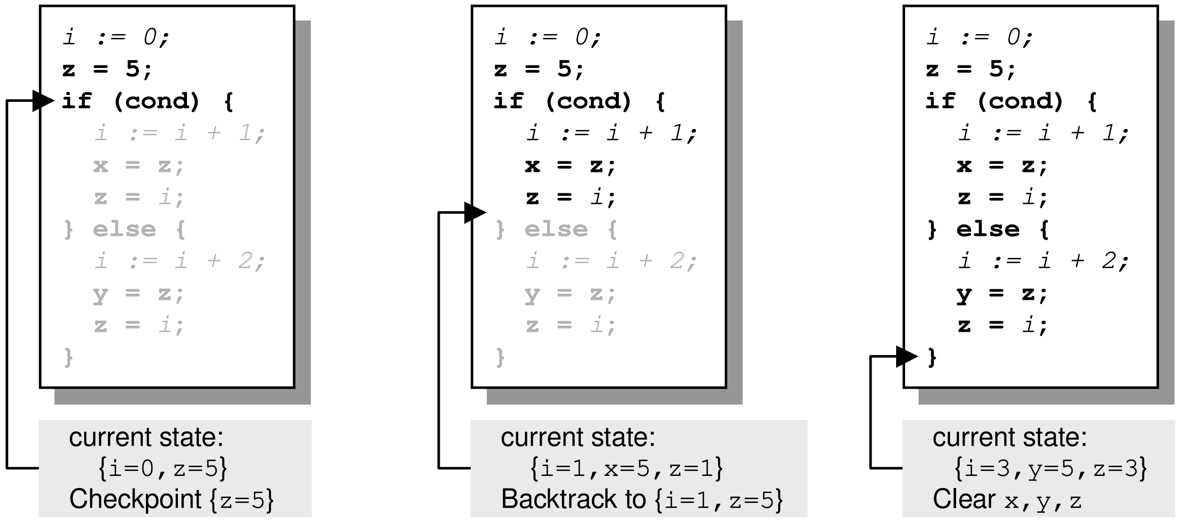


Figure 4.12: Executing the single-phase approach on an `if` statement.

then Catacomb sets the left-hand side variable's propagation value to the new expression. If the expression is not simple enough, then the left-hand side variable's propagation value is cleared and the expression is not propagated. Regardless of whether the right-hand side expression is propagated, its bounds are.

Finally, if the left-hand side of the assignment statement is a simple variable (as opposed to an array or structure reference), then any variable that contains this left-hand side variable as part of its propagation value must have its propagation value cleared.

Conditionals

Conditionals (`if` statements) are slightly more complicated. Catacomb first evaluates the conditional expression. If the condition evaluates to a constant, Catacomb eliminates the conditional test and the dead branch, and executes the live branch. If the condition does not evaluate to a constant, Catacomb has to keep and execute both branches. It executes the first branch of the statement, keeping track of which code variables are modified in the branch. At the end of the branch, Catacomb backtracks to the original propagation values and executes the other branch similarly. Note that it does *not* perform any backtracking for the control variables. After processing both branches, Catacomb clears the propagation values of any code variables that were modified in either branch. Also note that even when one branch of the `if` statement is known to be dead, Catacomb must still process the dead branch as though it were live, so that the control constructs can be executed within the branch.

Figure 4.12 demonstrates the single-phase processing on an `if` statement. Upon reaching the start of the `if` statement, the value of control variable `i` is 0, and the propagation value for `z` is 5. Catacomb checkpoints the propagation values of the code variables, but it does not checkpoint the control variables. After processing the first branch of the `if` statement, `i` is 1, and the propagation values for `x` and `z` are 5 and 1, respectively. Next, Catacomb backtracks to the checkpoint, restoring `z`'s propagation value to 5, and clearing `x`'s propagation value (since `x` had no propagation value coming into the `if` statement). Notice that the value of the control variable `i` remains 1. Finally, after processing the other branch of the `if` statement, `i` is 3, and the propagation values of `y` and `z` are 5 and 3, respectively. At this point, before proceeding, Catacomb clears the propagation values of all code variables set in the statement (namely, `x`, `y`, and `z`). Note that `i`'s value is 3, exactly as it would be at this point under the simple two-phase execution model.

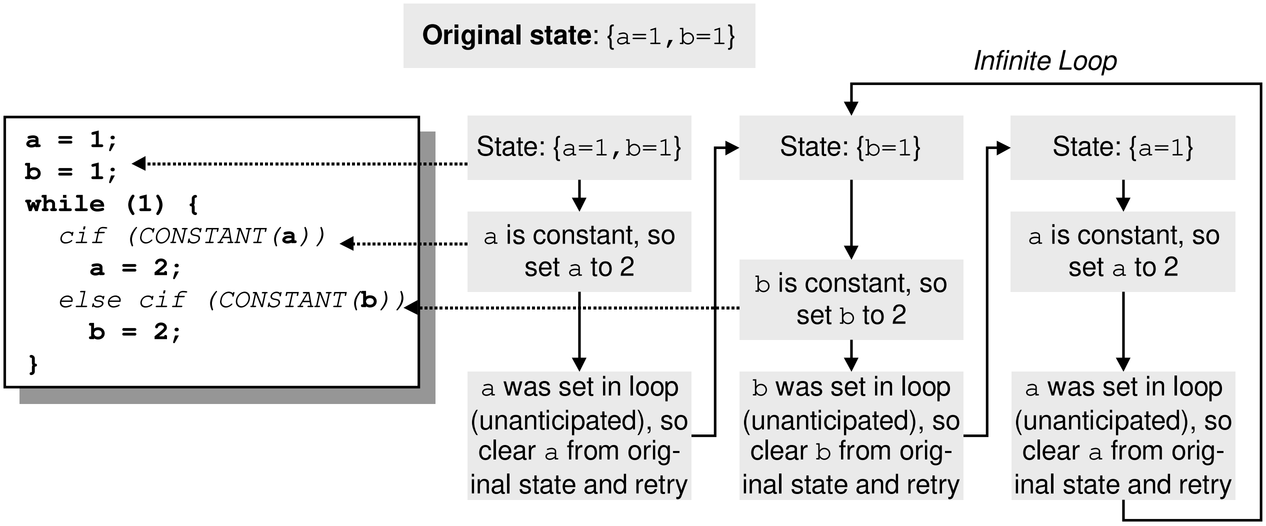


Figure 4.13: Incorrect processing of a loop under the single-phase execution model. If propagation values are not cleared monotonically, as shown here, the processing may fail to terminate.

Loops

The hardest constructs to process are the loops. The idea behind processing the loops is that we first scan the loop body to determine which variables are modified in the body. Then we clear those variables' propagation values and process the body. Finally, we once again clear the propagation values for the variables modified in the loop. The reason for clearing the propagation values is that because we cannot determine the number of loop iterations, variables set in the loop cannot have values propagated into or out of the loop.

The problem here is that we cannot scan the loop body for modified variables without first executing the control constructs to determine what code is actually in the loop body. But because the control construct execution depends on the results of the optimizer, we cannot safely execute the control constructs until the propagation values of the modified variables have been cleared.

To solve this problem, Catacomb begins by assuming that no variables are modified within the loop body, and executing the loop body optimistically without first clearing any propagation values. At the end of the loop body, it checks whether this optimistic assumption was correct (i.e., conservative enough). If not, it backtracks both the control variables and the propagation values to their state at the beginning of the loop, clears the propagation values of all modified code variables, and repeats. This execution repeats until the set of variables modified in the loop is a subset of the variables whose propagation values were cleared going into the loop.

Note that to ensure convergence, propagation values must be cleared for all variables modified in any previous execution of the loop body, not just the most recent execution. Consider the example in Figure 4.13. Coming into the `while` loop, `a` and `b` both have propagation values of 1. In Catacomb's first execution of the loop body, `a` is set to 2 and `b` is left unchanged, meaning that to ensure correctness, `a`'s propagation value should have been cleared coming into the loop. Catacomb backtracks to the beginning of the loop, where `a` and `b` both had propagation values of 1, clears `a`'s propagation value, and tries executing the loop again. This time, `b` is set to 2 and `a` is left unchanged, meaning that `b`'s propagation value should have been cleared, so Catacomb backtracks again and restores both values to 1. Now, if Catacomb chooses only to clear `b`'s propagation value without also clearing `a`'s value, it will end up in an infinite loop, oscillating between clearing only `a` and clearing only `b`. To guarantee convergence, both `a` and `b` must be cleared at the

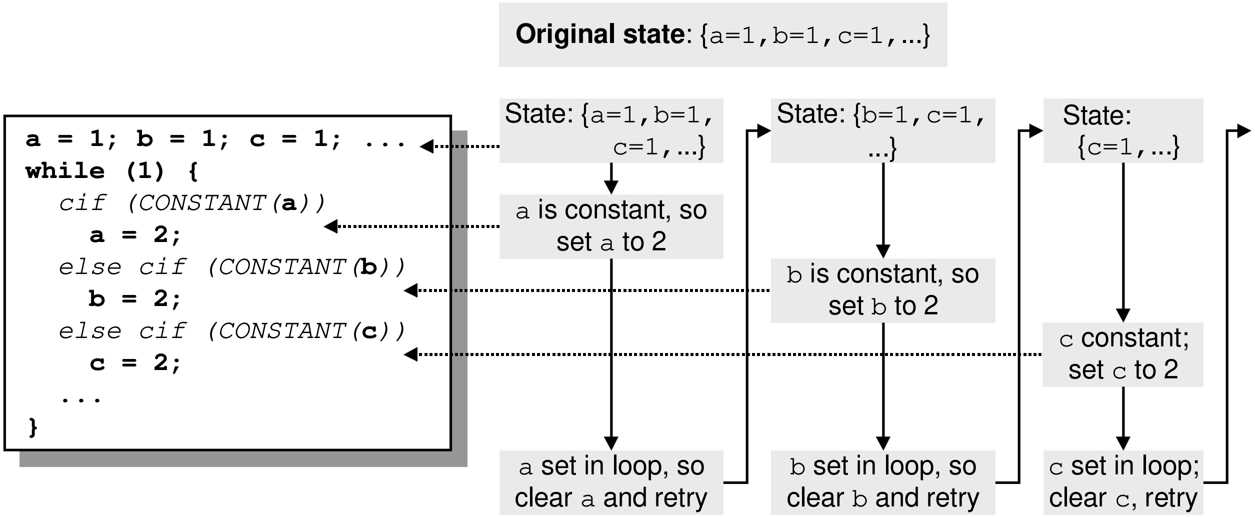


Figure 4.14: Correct processing of a loop under the single-phase execution model. Propagation values are cleared monotonically, guaranteeing convergence.

start of the loop. When this is done, both of the `cif` conditions evaluate to false, and neither assignment in the loop is executed. Thus the propagation values of all variables set in the loop body (none in this instance) are cleared before processing the loop, and the correctness requirement is met.

For compile-time efficiency, the number of passes required over the loop body is an issue. Without using nested loops, it is possible to construct a loop body that requires a number of passes proportional to the size of the loop body. Consider the example in Figure 4.14. Every time Catacomb processes this loop body, it discovers a new variable whose propagation value needs to be cleared, and so the number of passes required is equal to the number of variables set in the loop body. When multiple such loops are nested, the number of passes required can become exponential. In practice, though, the number of passes required is small, usually only 2 or 3.

4.6.8 Checkpointing and backtracking

The key to the backtracking infrastructure in Catacomb is a checkpointing facility. At any point during execution, Catacomb can create a new *checkpoint*. When this happens, a snapshot is taken of the propagation values and/or the control variables, so that at any point in the future, Catacomb can backtrack to that state. When creating the checkpoint, one of the options is to specify whether both control variables and propagation values should be checkpointed, or only propagation values. This option is needed because when processing a dead branch of an `if` statement (or other unreachable code), it is necessary at the end to backtrack the propagation values but not the control variables, whereas when processing a loop, both propagation values and control variables must be backtracked. Catacomb allows multiple checkpoints to be active at once, to support processing of nested loops, nested conditionals, and combinations of these. When the checkpoint is no longer needed (e.g., when processing of the loop is completed), it must be explicitly deallocated.

The backtracking mechanism provides two additional features that help in the optimizations. First, it keeps track of the set of code variables that were modified since the checkpoint was taken. This makes it easy to find the set of variables modified in a loop body or in a branch of an `if` statement so that their propagation values can be cleared where appropriate. The loop-processing code also uses this to test whether

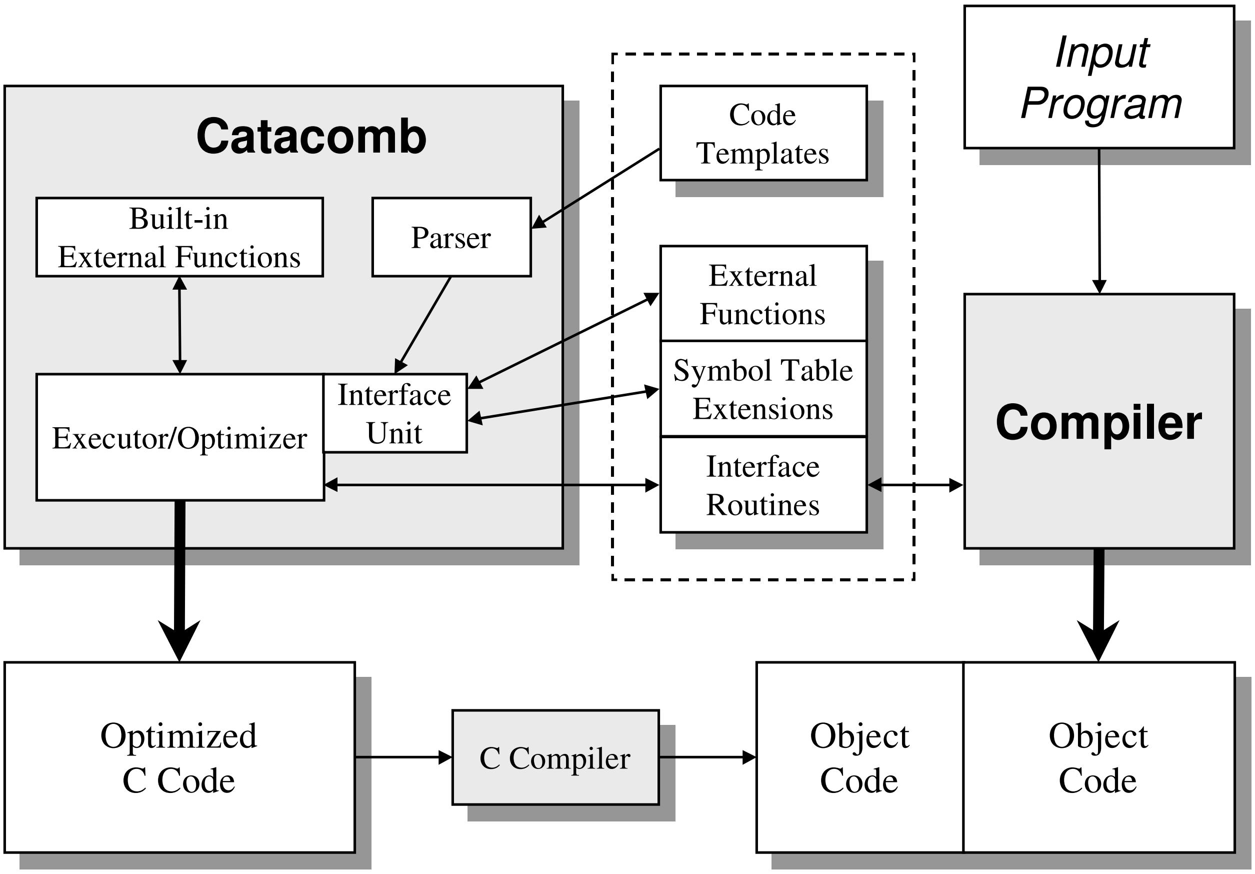


Figure 4.15: The high-level structure of Catacomb. The portions inside the dashed box (code templates, external functions, symbol table extensions, and interface routines) are implemented by the compiler writer when integrating Catacomb into a compiler.

the set of variables modified in the loop is a subset of the variables whose propagation values were cleared going into the loop, to determine whether to backtrack to the start of the loop and try again. Second, it can return the original bounds of all variables modified since the checkpoint was taken. For the monotonic induction variable detection optimization, these initial bounds are compared to the final bounds to determine which variables' upper or lower bounds can be preserved.

4.7 Compiler interface

Catacomb is designed to be plugged into a new or existing compiler with only minor modifications required. It consists of a core set of routines that are linked into the compiler (as a shared library where possible). To invoke Catacomb, the compiler invokes execution of a template, providing Catacomb with the template name and a list of template arguments, similar to the `include` construct in Catacomb. Catacomb executes the template with the given arguments, producing a C function that is written to an output file. The C function is given a unique name, and Catacomb returns the calling sequence for the function to the compiler. An alternate invocation returns the intermediate representation to the compiler for direct manipulation or inclusion into the compiler's output. Figure 4.15 shows the high-level structure of Catacomb and how it interfaces with a compiler.

4.7.1 Interface routines

The crucial part of the interface between the compiler and Catacomb is the translation of data structures. As there is little chance that the two systems use the same data structures to represent expressions and expression components, *interface routines* are required to translate the data structures. Interface routines are responsible for translating the following structures to and from Catacomb:

- **Constants**. Catacomb directly supports `int`, `double`, and `char*` constants.
- **Variables**. A symbol consists of the name, the type, the number of array dimensions (0 if it is a scalar), and the list of dimension sizes. Note that, unlike C, the array dimensionality and dimension sizes are not integrated into the type of the variable. This reflects an early design decision to make it easier to treat arrays as first-class objects in Catacomb.

  In addition, the symbol contains a *final dimension size* field, which is a virtual function that returns the final declared size of a particular array dimension. This function is provided particularly for distributed arrays, since only a portion of each such array should be declared locally on each processor. The default value of this field is the identity function.
- **Types**. Catacomb supports all the standard C types, except for function prototypes.
- **Array dimension sizes**. These are allowed to be arbitrary expressions, unlike in C, where dimension sizes are required to be compile-time constants. Where necessary, Catacomb produces the extra code necessary to convert declarations for and accesses to such arrays into valid C code.
- **Expressions**. Catacomb supports all of the unary, binary, and ternary expressions in C. In addition, to facilitate expression simplification optimizations, it internally supports five additional binary operators: `min`, `max`, `pmod`, `fldiv`, and `ceildiv`. These operators are described in Section 4.3.
- **Array references and function calls**. Catacomb represents these as an expression and a list. The expression represents the actual array or function, and the list consists of the array subscripts or function arguments.

4.7.2 Language interface

Catacomb produces C code, and the compiler that invokes Catacomb at compile time is expected to produce code that calls the Catacomb-produced functions at run time. For example, the Fx compiler produces Fortran-77 code that contains calls to the Catacomb-generated C functions. Catacomb tries to be flexible in the way it produces functions so that they can be easily invoked by a variety of compilers on a variety of systems. These features include the following.

- Catacomb allows the compiler to specify a prefix and suffix for the functions produced in the output C file. For example, most Fortran compilers append an underscore character to the names of external symbols, both for symbols that the compiler provides and for symbols that the compiler accesses. In this case, the Fortran compiler would generate a call to `foo`, but Catacomb would need to produce the function `foo_`.
- Catacomb produces function headers in both ANSI C style and the older K&R style, to support older C compilers.
- The Fortran compiler on the Cray T3D [42] produces external symbols and external symbol references in all uppercase. Catacomb provides an option to support this.

4.7.3 Extending the symbol data structure

Catacomb is expected to be used in conjunction with domain-specific compilers. In such a domain, variables often have extra attributes associated with them. For example, in Fx, each array has an extensive amount of distribution information associated with it. When the interface routines create the Catacomb symbols corresponding to these distributed arrays, they also need to create the associated distribution information within Catacomb, and link the distribution information to the Catacomb symbol.

Adding this kind of domain-specific information to Catacomb symbols requires modifying the symbol data structure of Catacomb itself. To make this task as painless as possible, Catacomb provides a mechanism for easily and dynamically extending the symbol data structure with additional fields. This extension takes the form of a "module" interface, with three operations:

- **Register the module with Catacomb**. At the beginning of the compiler's execution, the compiler needs to notify Catacomb of the existence of any such modules that it provides. Catacomb returns a module identifier, which is required for subsequent module operations. Catacomb allows an unlimited number of modules to be added to the symbol data structure, each module independently adding data to the symbols.

- **Associate module data with a symbol**. When the compiler wants to add module information to a Catacomb symbol, it passes to Catacomb the module identifier, the symbol, and a single pointer to data to associate with the symbol. The data should be contiguous in memory, as Catacomb treats it as a `void*`. Any previous module data associated with the symbol is discarded.

- **Access a symbol's module data**. Given the module identifier and the symbol, Catacomb returns the module data previously associated with the symbol. If no data was previously associated, Catacomb returns NULL.

Using this kind of module structure to extend the symbol data structure means that the data structure, as compiled into Catacomb, need not actually be changed. In fact, Catacomb does not even need to be recompiled to add additional modules; the new code can instead be bundled with the interface routines and linked along with Catacomb into the compiler executable.

4.7.4 Adding external functions

After extending the symbol data structure with domain-specific information, the compiler needs to be able to make this information accessible to the templates. To fill this need, Catacomb provides a mechanism for adding new external functions. Like the symbol extension modules, external functions can be added without modifying or recompiling Catacomb. There is a Catacomb function that allows new external functions to be registered; this registration associates a C function with the name of the external function, causing the C function to be called whenever the external function is invoked. Like the symbol extensions, the external function registration should be called at the beginning of compiler execution, before any templates have been executed. This mechanism also allows the definitions of existing external functions to be changed, in case the compiler needs nonstandard behavior from one of the core external functions.

The external function writer should expect that occasionally the template writer will make an error in a call to an external function. Examples of errors include the wrong number of arguments, wrong argument type, or argument out of range. Except in special cases, the external function should not print an error message, and should not call `exit()`. Instead, it should return an "error node," which is a special type of expression that causes an error to be printed only when the code is actually produced in its final form. The reason is that the value returned by the external function might never actually be used; the expression can become dead through, e.g., an unreachable `if` branch, or the short-circuiting semantics of the `||` and `&&` C operators.

Except in special cases, the external function should not produce any side effects. The reason for this is that Catacomb's single-phase approach to control execution and optimization (see Section 4.6.7) means that certain sections of template code are repeatedly executed and backtracked until convergence is reached, and thus one instance of an external function can actually be executed several times by Catacomb. As an example, Catacomb provides a `PRINT` routine to give feedback at compile time. This external function contains the side effect of printing out a value. When Catacomb executes a template containing a `PRINT` statement, it is not uncommon to see it executed more than once.

4.7.5 Adding new templates

Catacomb provides a simple mechanism for adding new templates to execute. There is a function to load all template files from a particular directory. A Catacomb template file is defined to be a file whose name ends in the ".ccom" extension. Syntax errors in a template result in that particular template being discarded. If two templates from two different directories have the same name, the template from the earlier-read directory is discarded. (If the two templates with the same name are in the same directory, one or the other is discarded, in a system-dependent fashion.) This feature is useful for experimenting with new algorithms or installing quick bug fixes: the user can set up a template directory, which is processed last, containing the updated overriding templates.

4.8 Summary

In this chapter, I have described the design and implementation of Catacomb, a composition system for producing C code. Although the design is fairly simple, there are many subtle implementational issues for producing the highest quality code while preserving an intuitive semantics for the template programmer. The end result is a more complex implementation structure, but one with a powerful optimizer and clean semantics. In the next chapter, I evaluate Catacomb, including the optimization framework, in the context of the array assignment statement.

Chapter 5

Catacomb and the Array Assignment

In Chapter 3, I discussed in detail the concept of code composition, and in Chapter 4, I described Catacomb, a specific implementation of a composition system. To study the effectiveness of Catacomb and code composition in practice, I investigated the use of Catacomb for the array assignment statement.

The "core" implementation of Catacomb consists of routines that load templates, execute the templates on an input, and optimize the resulting code. An infrastructure for the array assignment requires three extensions to the core Catacomb implementation: interface routines to allow Catacomb to pass data to and from an existing parallelizing compiler, external functions to provide the code templates with access to array distribution information, and a set of code templates for the array assignment. The code templates require a fair amount of design and planning to organize them in such a way that an arbitrary communication set algorithm can be combined with an arbitrary communication architecture to yield a complete implementation.

The result of this framework is that Catacomb is the first system able to incorporate and compare several array assignment algorithms targeted to several communication architectures in a single compiler, for the fully general array assignment statement.

This chapter describes the design, development, and performance of the Catacomb implementation of the array assignment. This framework was implemented in the context of Fx parallelizing Fortran compiler [61]. To distinguish the implementation of the array assignment from other possible uses of Catacomb, I refer to it in this chapter as Catacomb/Fx. In Section 5.1, I describe in detail the structure of the Catacomb/Fx implementation of the array assignment, focusing on the modular design of the templates that allows combining arbitrary algorithms and architectures. In Sections 5.2 and 5.3, I evaluate the performance of Catacomb/Fx, in terms of the three key components of code generation: efficiency, generality, and maintainability.

5.1 Structure

As discussed in Section 4.7, an implementation of the array assignment in Catacomb/Fx has three requirements:

1. Interface routines must be written so that Catacomb can be coupled with a parallelizing compiler. For this evaluation, I integrated Catacomb with the Fx parallelizing Fortran compiler at Carnegie Mellon. Before the integration, Fx compiled array assignment statements using the CMU algorithm, for a single communication architecture model. The details of the integration appear below in Section 5.1.3.

2. Catacomb must be extended, as described in Section 4.7, to provide the code templates with access to array distribution information. This includes extensions to the symbol data structure in Catacomb, so

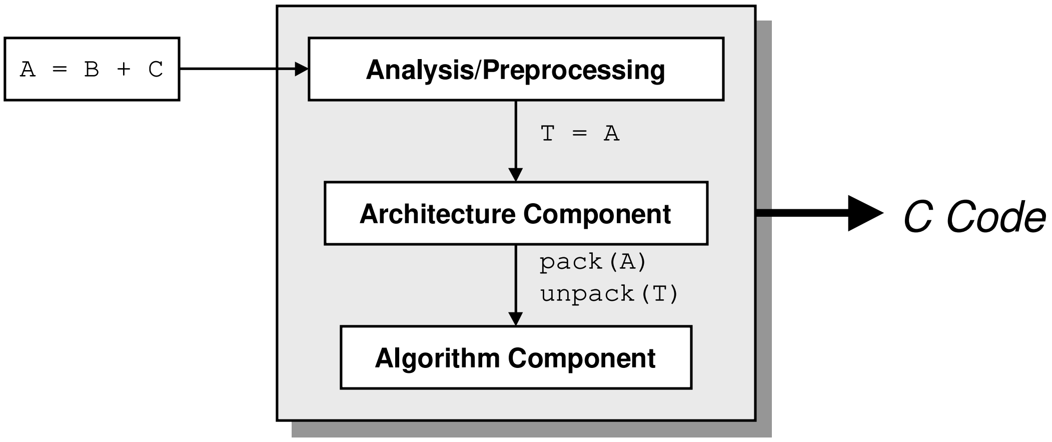


Figure 5.1: The high-level structure of the Catacomb/Fx templates for the array assignment.

that variables can hold compile-time distribution descriptors, and new external functions, for use by the code templates. These extensions are also detailed in Section 5.1.3.

3. The most obvious requirement is that code templates must be written to handle the array assignment statement. It may seem that, given a runtime library routine for the array assignment, it is simply a mechanical exercise to translate it to template code. However, as I describe in this section, it is important to design *frameworks* for the algorithm and the architecture portions of the array assignment. With well-specified frameworks in place, it becomes possible to combine any array assignment algorithm with any communication architecture in a modular fashion, to form a complete implementation.

In this section, I begin by giving a high-level description of the code templates for the array assignment statement. Then I describe the extensions for supporting *task parallelism*. Finally, I describe the interface to Fx, and the Catacomb extensions needed to support array distribution information in Catacomb/Fx.

5.1.1 Code template design

The code templates for the array assignment are divided into three levels. At the highest level is the *analysis and preprocessing component*. The next level consists of the *architecture component*, and at the lowest level is the *algorithm component*. Figure 5.1 illustrates this hierarchy. There is a separate architecture component written for each target communication architecture, and there is a separate algorithm component written for each communication set algorithm.

Analysis and preprocessing component

Compilation of the array assignment statement begins with the analysis and preprocessing component. In this phase, the Catacomb/Fx templates analyze the specific array assignment statement, breaking it down into simpler constructs where necessary. Figure 5.2 depicts the operations performed in this component, which include:

- **Broadcast single distributed elements**. If an array subscript or a component of a subscript triplet consists of a single element of a distributed array, then that distributed array element is broadcast. For example, if the left-hand side array reference is $A[1\colon n][X[i]]$, then the preprocessing phase creates and compiles the statement $t = X[i]$ and replaces the array reference with $A[1\colon n][t]$. Similarly, if the

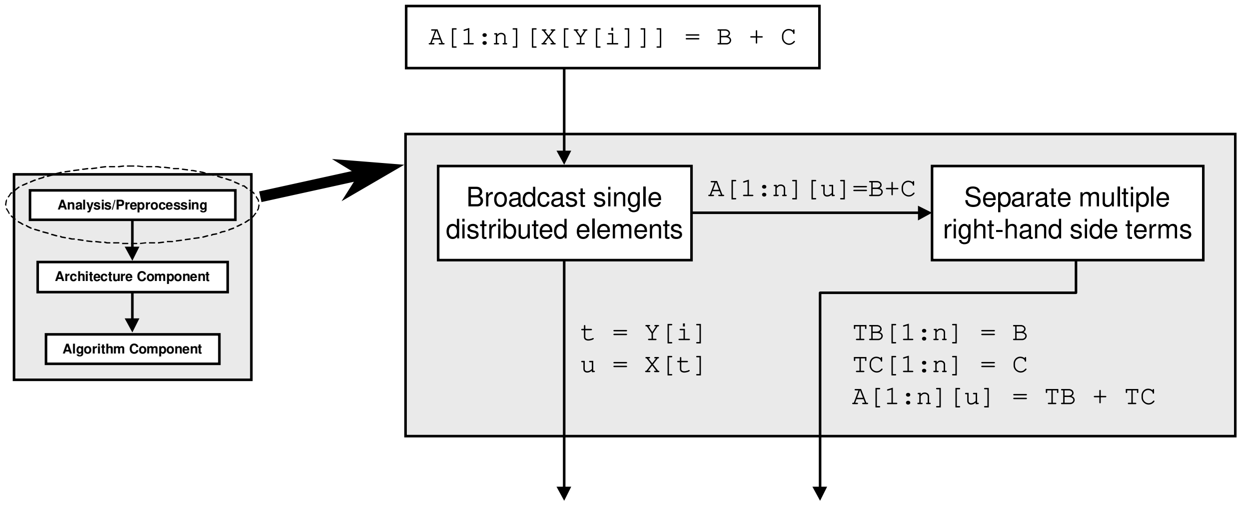


Figure 5.2: The structure of the analysis and preprocessing component of the Catacomb/Fx templates for array assignment.

input array reference is $A[1\!:\!n][X[Y[i]]]$, as in Figure 5.2, the two statements $t = Y[i]; u = X[t]$ are compiled and the input array reference is replaced with $A[1\!:\!n][u]$. For the intermediate statements, the processor that owns the distributed array element broadcasts its value.

- **Separate multiple right-hand side terms**. If the right-hand side of the array assignment statement contains multiple terms, then the preprocessing component breaks the statement down into several assignment statements, as illustrated in Figure 5.2, where only the last statement (i.e., $A[1\!:\!n][u] = TB + TC$) contains multiple right-hand side terms (but no communication). This last statement is given to the local computation templates for compilation, and the other statements are given to the communication templates to be compiled.

These steps are always performed the same way, independent of the particular communication set or computation set algorithm used or the specifics of the communication architecture. After the steps are performed, the only array assignment statements remaining either have a single right-hand side term or involve only local computation. Regardless, no terms contain distributed array references like $A[i]$ that need to be broadcast. The preprocessing component then passes the statements requiring communication to the architecture component, which generates the necessary communication. Finally, if it is necessary to include local computation, the relevant statement is passed to the algorithm component.

Architecture component

The architecture component is a set of code templates written for the specific details of the target communication architecture. It takes as input an array assignment statement with a single term on the right-hand side. The key duty of the architecture component is to produce the *communication schedule*. The communication schedule is responsible for interleaving all of the operations in the appropriate order, including the architecture-specific operations like sends, receives, and synchronization, as well invoking the algorithm-specific details like packing and unpacking the communication buffers.

Before going into the details of the operation of the architecture component, I discuss two capabilities that are desirable for the system to provide. The first capability is the *deposit model* or the *direct deposit*

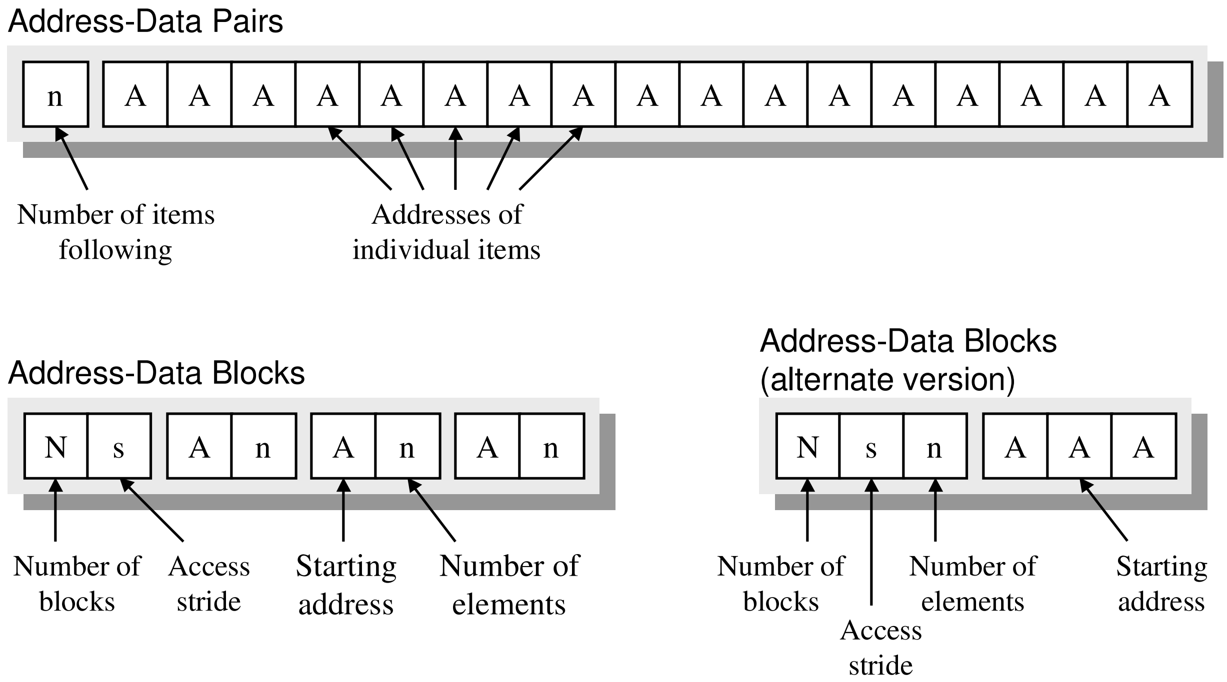


Figure 5.3: Three examples of address relations. The most basic form, *address-data pairs*, includes no compression; each address is meant to be paired with a single data item. *Address-data blocks* provide some compression when the sequence of addresses consists of long sub-sequences with a fixed stride. A modified form of address-data blocks can be used when the length of each fixed-stride block is the same.

model [56, 59] of communication. In the standard model of communication, the sending processor executes a communication set algorithm to pack elements from the source array into a communication buffer, and then sends the buffer to the receiving processor. The receiver executes an analogous communication set algorithm to unpack the buffer and store the elements into the destination array. Under the deposit model, the sender also executes the communication set algorithm on behalf of the receiver, thereby computing each element's destination address (or offset into the destination array) on the receiver. This addressing information is included in the communication buffer, so that the unpacking algorithm is a simple, flat loop. The motivation is that the packing and unpacking algorithms have some common computations that can be shared by merging the algorithms; on a machine with fast communication hardware, the decrease in unpacking time may be larger than the corresponding increase in communication time.

The direct deposit model takes this idea one step further. On machines with fine-grained remote store capabilities, like the Cray T3D [1, 9] and T3E [49], as well as on true shared-memory machines, the direct deposit model allows the sender to compute the destination addresses and perform remote stores for each element, rather than using communication buffers and explicit communication operations. The extra benefit here is the decrease in buffering and memory bandwidth due to the direct deposit model not using explicit communication buffers [57, 58].

The second desirable capability is to support the saving and reuse of communication patterns. For some array assignments in conjunction with some array assignment algorithms, the loop overhead in the communication set generation is large in comparison to the actual data accesses. This happens particularly with the harder-to-analyze block-cyclic data distribution (as opposed to the simpler block and cyclic distributions). If, during the course of the program, the same array assignment executes many times, it may be advantageous to encode the communication pattern as a compact *address relation* [23]. This address relation is a

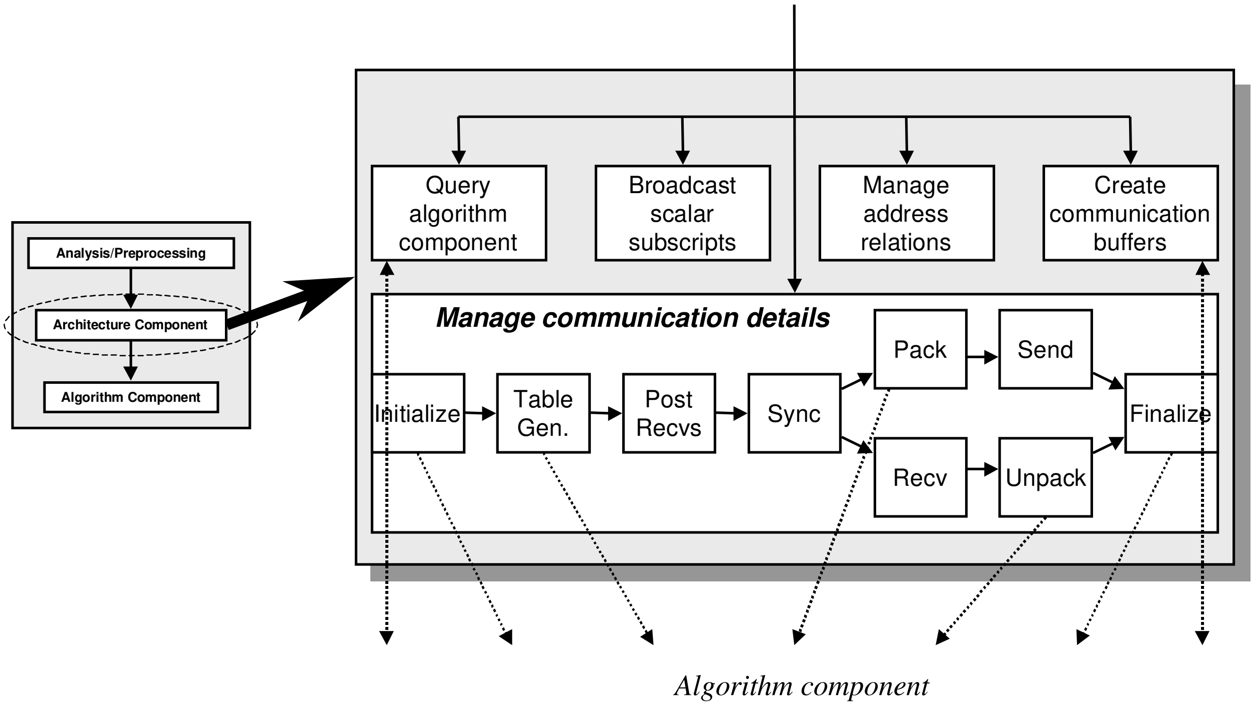


Figure 5.4: The structure of the architecture component of the Catacomb/Fx templates for array assignment.

representation, possibly compressed, of the access pattern of the communication set generation on the sender and/or receiver. Figure 5.3 shows three examples of address relations. If the address relation can be decoded faster than the communication set algorithm can execute, and the memory requirements of the address relation are modest, then it may be desirable to store and reuse the address relation for frequently-executed array assignment statements.

The architecture component has several duties, as illustrated in Figure 5.4:

- **Determine the characteristics and capabilities of the algorithm component**. Some of the communication set algorithms have characteristics that the architecture component needs to know about. The specific structure of the code produced depends on the algorithm's treatment of the following features.

  Pack-and-send vs. bulk pack: Some of the algorithms, like the table-driven algorithms, are designed to pack all communication buffers at once. Others, like the CMU and OSU algorithms, are designed to pack a single communication buffer, send it off, and repeat for each destination processor. The architecture component needs to query the algorithm component on this issue, both to determine how many communication buffers to allocate and to determine the interleaving of packing and sending. In general, the single-buffer approach is preferred when possible, both because it reduces buffer memory requirements and because it can reduce congestion in the communication network (compared to sending all messages at once). However, there are some architectures for which a carefully optimized communication schedule can allow an exchange of all buffers at once without congestion [30]; the architecture component can target such a system if desired.

  Deposit: To use the deposit model or the direct deposit model, the communication set algorithm must be able to compute destination addresses at the same time that it computes source addresses. Most of the communication set algorithms, with the notable exception of the OSU algorithm, support this model. (The authors' implementation of the OSU algorithm [32], which I used as a basis for the OSU code templates, gives no easy way to modify it to add this support.) If the algorithm does not support the deposit model, the architecture component must use the standard model instead.

Empty messages: When executing an array assignment, one possible communication optimization is to suppress the sending and receiving of empty messages. It is easy to suppress the send of an empty message, but it is hard to suppress the receive of an empty message, because the receiving processor does not know that no data was sent until it has executed its buffer unpacking algorithm. The CMU algorithm includes a `should_receive` function that determines whether a non-empty message is expected from a particular sender, but the other algorithms do not have this functionality. Thus only the CMU algorithm is amenable to the empty message suppression optimization.

- **Manage communication details**. Different communication architectures have different semantics for achieving the best communication performance. For MPI [39] and NX [20], the most efficient communication ordering is:

  1. Post all receives in advance, using `MPI_Irecv` or `irecv`.
  2. Synchronize all processors, using `MPI_Barrier` or `gsync`.
  3. Execute all packs and sends, using `MPI_Send` or `csend` for sending.
  4. Wait for messages to arrive and unpack them, using `MPI_Wait` or `msgwait` to wait on a message.

  By posting receives in advance, incoming messages can be stored directly in the posted user buffer, rather than having to go into a system buffer and be copied to the user buffer later. The barrier synchronization ensures that all receives are posted before any sends can start, and it provides a fence between messages from two array assignments. The disadvantage of this approach is that a receive buffer must be allocated for every potential sender, thus increasing the memory requirements.

  Unfortunately, PVM [25, 63] does not allow receive buffers to be posted in advance. Furthermore, the implementation, rather than the PVM specification, determines how much buffer space is available to temporarily store messages that have already arrived but for which no receive has been issued. Thus the architecture component needs to produce code that interleaves sends and receives in an order that guarantees freedom from deadlock. One possible ordering, for execution on processor p, is:

  1. Synchronize all processors, using `pvm_barrier`.
  2. Pack and send to all processors q for which $q < p$, using `pvm_send`.
  3. Receive and unpack from all processors, using `pvm_recv`.
  4. Pack and send to all processors q for which $q > p$.

  This approach requires only a single receive buffer to be allocated. As above, the barrier synchronization is necessary to prevent intermixing of messages from different array assignments.

  For the direct deposit communication architecture, no explicit sends or receives are required. All that is needed is to invoke the communication set generation for all destination processors. Once again, though, a barrier synchronization is needed before beginning the direct deposits, to ensure that live data on the receiver is not overwritten.

- **Handle scalar subscripts**. Section 3.1.1 describes the modifications needed to handle scalar subscripts in the array assignment statement. It is the responsibility of the architecture component to apply the ownership test and the local memory mapping function, as depicted in Figure 3.3.

- **Create communication buffers**. The architecture component must allocate and deallocate the communication buffers. It first invokes a template from the algorithm component to find the required length of each buffer, as well as the type of the buffer element (see page 74). Then it allocates as many send and receive buffers as necessary to meet the requirements of the algorithm and the communication architecture.

- **Invoke templates of the algorithm component**. Where necessary, the architecture component invokes the templates from the algorithm component. Because the templates of the architecture component invoke those of the algorithm component, I classify the algorithm component as being at a lower level in the overall structure than the architecture component. The specifics of the algorithm component templates are described below.

- **Manage address relations**. Before packing or unpacking communication buffers, the array assignment code needs to query the address relation library to find out whether it should reuse an address relation for packing or unpacking. The architecture component is responsible for producing code that decides at runtime whether to use the standard packing and unpacking algorithms or the address relation library to pack and unpack the communication buffers.

Algorithm component

The following are the operations that fall into the domain of the algorithm component, to be invoked by the architecture component.

- **Table generation**. Some of the algorithms require the generation of auxiliary data to be used in the set generation loops. For example, the table-driven algorithms (RIACS, Rice, and LSU) build up tables that describe the access patterns over the arrays. Although not classified as a table-driven approach, the OSU algorithm also performs computation outside of the set generation loops, the results of which are used inside the loops. The CMU algorithm does not require any such computation, so its code template for table generation is empty.

  The architecture component invokes the table generation template before any buffer packing or unpacking. The table generation template uses a `cwhile` loop to repeat the table generation step for each dimension of the array assignment. In this way, the table generation step for the canonical array assignment statement is automatically composed to handle multidimensional arrays.

- **Buffer packing**. The communication set generation algorithm includes a loop or a loop nest that iterates over the elements of the source array, packing the required elements into a communication buffer(s) to be sent to the destination processor(s). For a multidimensional array, as described in Section 3.1.1, the buffer packing template recursively includes itself once for every array dimension, creating a loop nest that packs the Cartesian product of the per-dimension communication sets. The recursive include is the way in which the buffer packing step is automatically composed for a multidimensional instance of the array assignment.

  If the deposit model or the direct deposit model is enabled, and the algorithm component supports computation of remote address information, then the buffer packing template must compute the destination address and either pack the address into the communication buffer (under the deposit model) or store the element directly to the destination processor's memory (under the direct deposit model). The actual mechanism of direct deposit is clearly an architectural feature, which means that the buffer packing template invokes a template in the architecture component to effect the actual fine-grained transfer.

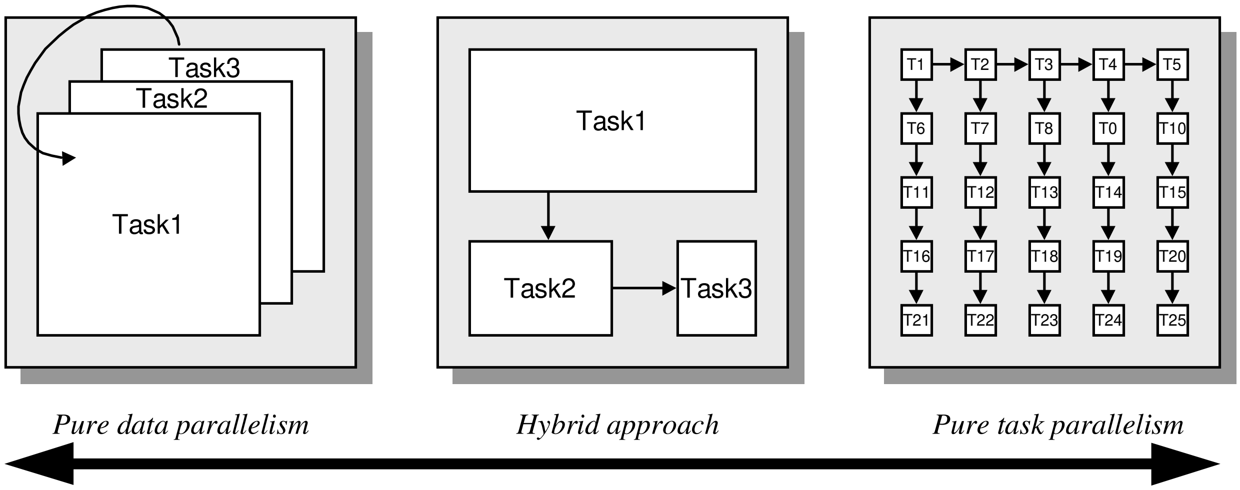


Figure 5.5: A spectrum of data and task parallelism.

- **Buffer unpacking**. The buffer unpacking template is analogous to the buffer packing template. It creates a loop or loop nest that iterates over the communication buffer after a message is received and stores the elements into the destination array. Like the buffer packing template, the buffer unpacking template uses a recursive include approach to compose the one-dimensional buffer unpacking loop to create a Cartesian product loop for a multidimensional instance of the array assignment.

  When the deposit style of message passing is used, the communication buffer is a flat array of data and addressing information. Unpacking and storing the data requires only a simple loop that works for array assignments of any dimensionality.

- **Determination of communication buffer type and size**. This template requires the algorithm component to compute the size of a communication buffer and the type of each buffer element. Distribution information, as well as the values of the subscript triplets, can be used to determine the maximum number of array elements that can be transferred from a source processor to a destination processor.

  The type of a communication buffer element is usually the same as the type of the right-hand side array. This kind of declaration is made possible by using Catacomb's special `typeof()` operator. However, when using the deposit model of communication, destination address information has to be packed into the buffer along with the data. In this case, the type of a buffer element might need to be different. For example, it could be declared as a struct containing a data element and an address.

- **Initialization, finalization**. There is an initialization template and a finalization template allowed for whatever purposes deemed necessary by the algorithm component. A particular use of the initialization template is for the algorithm and architecture components to exchange data (as described above), such as whether the algorithm needs to pack all buffers at once, whether the algorithm supports the deposit/direct deposit model, and whether the algorithm can predict empty messages in advance. A particular use of the finalization template is to produce code to deallocate memory allocated during the table generation phase.

5.1.2 Supporting task parallelism

There exist a variety of ways to decompose parallelism into data parallel methods and task parallel methods [61], as shown in Figure 5.5. At one end of the spectrum, data parallel programs are typically expressed

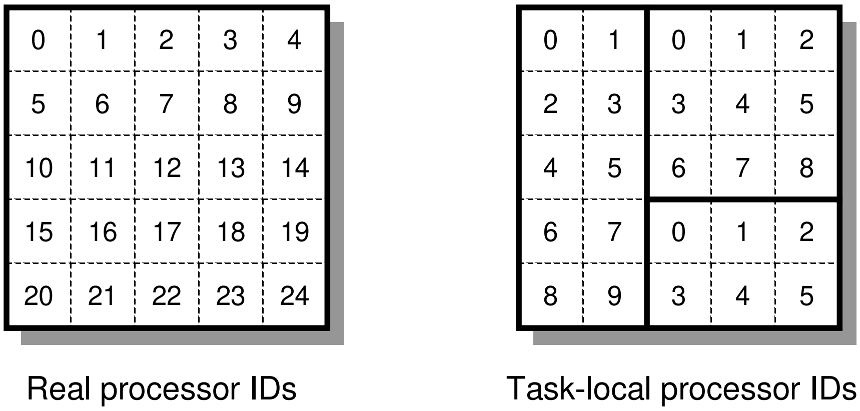


Figure 5.6: Three tasks, in which the task-local processor ID is different from the real processor ID.

as a single thread of control operating on data sets that are distributed across all processors. At the other end of the spectrum, task parallelism can be specified as a collection of sequential processes with explicit or implicit communication between them. A hybrid model allows each task to be mapped onto a subset of processors, as a data parallel task. Under this model, communication between tasks becomes difficult, because it has the same character as the standard data-parallel array assignment statement. For each communication step, a section of a distributed array needs to be transferred from the source task to the destination task, and stored into another distributed array on the destination task.

The Fx compiler supports a model of data parallel tasks. The programmer specifies the granularity of the tasks and the data dependences between the tasks, and Fx automatically determines the mapping of tasks to processors [26, 62]. Fx also takes into consideration memory, bandwidth, and latency requirements in determining the task mapping [60].

Data transfer in such a task parallel program can be divided into three categories: intratask data transfer, split/join data transfer, and intertask data transfer. Split/join data transfer happens when a task splits itself into several smaller subtasks, or when the subtasks complete and rejoin the parent task. The split and join operations require arrays to be redistributed between the two tasks; note that the subtask's processors are a subset of the parent task's processors. Intratask data transfer occurs as the result of normal data parallel computation (e.g., array assignment statements) within one task; the data transfer is isolated to the processors of the task. Intertask data transfer is required when one separate task needs to send data to another task, and the tasks share no processors.

The data transfer requirements for all of these categories can be specified in terms of array assignment statements, with some caveats. The standard array assignment algorithms can be used to generate communication sets for the data transfer. Thus the algorithm component need not be modified. However, some additional support is needed in the architecture component.

There is a modification necessary for all three types of task communication, relating to processor numbering. The algorithms for data parallel communication assume that processors are numbered sequentially, starting from 0. This numbering is important not only for matching the assumptions of the architecture-specific communication routines, but also for the calculation of ownership sets. However, when a task does not consist of all the processors, the processors in that task are likely not to be numbered starting from 0, and they need not even be numbered sequentially. Therefore, the processors within a task need to have a virtual numbering scheme that starts at 0 and is sequential, with the real processor ID used only for the low-level communication routines. Figure 5.6 illustrates this task-local processor numbering scheme.

Intratask communication

Communication here can be treated identically to the standard array assignment communication, except that any barrier should synchronize only the processors of the task, and not the entire set of processors.

Split/join communication

Like intratask communication, a barrier should synchronize only the processors of the larger task, of which the smaller task's processors are a subset. In implementing this subset barrier, one must keep in mind that several subtasks will be trying to synchronize with the parent task at the same time, so the subset barriers should have unique tags.

A more subtle concern is the following. A processor in the main task has one virtual ID, but its virtual ID in the subtask may be different. This means that as a sender, it must adopt one virtual ID, but as a receiver, it must adopt another.

Intertask communication

The communication here has the same concerns as the split/join communication: subset barrier synchronization and virtual processor numbering issues. Note that since the processor sets for the two tasks are unique, one task performs all sends and the other task performs all receives. For this reason, there is no need for the architecture component to interleave sends and receives as described on page 72.

5.1.3 Catacomb extensions

The first step in integrating Catacomb and Fx is to write the interface routines to translate data structures between the two. The only complications, albeit simple ones, arise from the fact that the Fx data structures represent Fortran constructs, whereas the Catacomb data structures represent C constructs. When translating array references and array declarations, the ordering of the dimensions must be reversed. Also, when translating array references, the indices must be translated between C's 0-based indexing and Fortran's 1-based indexing. Another complication relates to type translations. On most systems, a Fortran `INTEGER` is the same as a C `int`, a Fortran `REAL` is the same as a C `float`, and a Fortran `DOUBLE PRECISION` is the same as a C `double`. However, on the Cray systems, `INTEGER` and `long` are equivalent,[1] and `REAL` and `double` are equivalent (i.e., Cray Fortran uses only 64-bit data). The rules for type translation are simple to implement in the interface routines, but can lead to hard-to-find errors when a system does not follow the usual conventions.

The second step in the integration was to design the extensions to Catacomb, in terms of the extension to the symbol data structure (as described in Section 4.7.3) and the design of the external functions (as described in Section 4.7.4).

To understand all of the details, it is necessary to understand the Fx model of alignment and distribution, which is based on the HPF model. As outlined in Section 2.1.2, arrays are aligned to *templates*, and the templates are distributed across the processors in a block-cyclic fashion. The Fx alignment and distribution model is essentially the same as HPF, with a few exceptions. First, Fx only supports alignment functions of the form $i + b$ (where b is known as the *alignment offset*), rather than the more general linear alignment function $ai + b$. Second, Fx allows an *overlapped* alignment, where a range of array elements can be aligned

[1] At the time of this writing, an `int` was 32 bits wide and a `long` was 64 bits wide in the Cray T3D C compiler. At some later point, `int` was changed to be 64 bits wide. Trying to keep abreast of such a moving target makes an even stronger argument for putting the type translation routines in one central location.

to each template element (e.g., aligning $A[i-1\!:\!i+1]$ to $T[i]$). Thus the alignment function also includes the *left offset* and the *right offset*.

The data added to the Catacomb symbol for an array (whether distributed or not) is a pointer to a struct containing the following fields:

- A flag telling whether the array itself is distributed. If the array is not distributed, the rest of the fields in the struct are set to default values.
- The name of the *runtime distribution descriptor*. At run time, distribution information is encoded in a descriptor array. This descriptor is needed because of function calls. Normally the compiler knows the distribution parameters of the arrays when compiling a function. However, the programmer might want to write a function that can work on an input array of any distribution. In this case, it has to query the runtime distribution descriptor to determine the distribution parameters.
- A descriptor of the template to which the array is aligned. The template descriptor contains the number of template dimensions and the size of each dimension.
- A descriptor of the "distribute" statement used to distribute the template. This descriptor contains, for each template dimension, the distribution block size and the number of processors over which the template is distributed.
- A descriptor of the "align" statement that aligns the array to the template. This descriptor primarily describes the *alignment permutation*. For example, if the align statement specifies that $A[i][j]$ is aligned with $T[j][i]$, the alignment permutation allows it to be distinguished from $A[i][j]$ being aligned with $T[i][j]$. In addition, if $B[i]$ is aligned with $T[i][10]$, then the descriptor also specifies the constant 10 in the second dimension of template T.
- For each array dimension, a description of its distribution. The per-dimension description includes a "distributed" flag (whether that particular dimension is distributed), the distribution block size, and the number of processors over which it is distributed. It also indicates whether the distribution type is known to be block, cyclic, or only the more general block-cyclic. It is easy to know that a distribution is cyclic, because that is equivalent to its block size being 1. However, we can know that a distribution is block without knowing its exact block size, because the template can be explicitly distributed as block even without knowing the exact size of the template dimension or the number of processors over which it is distributed.

To access these fields within code templates, I added the following external functions in Catacomb/Fx:

- Querying compiler flags. These flags include: the total number of processors that Fx is compiling for; whether the deposit or direct deposit model of message passing is desired; and whether to suppress all actual communication calls, for use in measuring only the computation costs.
- Determining whether an array is distributed. There is a function to test the array as a whole, and one to test whether a particular array dimension is distributed.
- Querying the per-dimension distribution parameters, including block size, number of processors, and the distribution type (i.e., block, cyclic, or block-cyclic).
- Querying specific parameters from the template, align, or distribute statements.

5.2 Evaluation: Efficiency

In this section, I evaluate some of the aspects of the efficiency of the Catacomb-generated code for the array assignment statement. To perform the experiments, I used Catacomb/Fx to generate C code for a variety of array assignment statements. I evaluated the performance of three array assignment algorithms, the CMU, OSU, and RIACS algorithms, each of which I developed templates for, based on their original implementations.

I must stress that it is *not* the purpose of these experiments to compare the performance of the array assignment algorithms head to head. Wang, Stichnoth, and Chatterjee compared these algorithms elsewhere [71], using Catacomb/Fx to generate the code, and found that for each algorithm, there is a class of array assignment statements for which that algorithm outperforms the others. By presenting the performance of all three algorithms together, I show that the performance improvements offered by Catacomb are useful beyond just a single algorithm.

I begin in Section 5.2.1 by evaluating some of the claims made in the Syracuse approach to redistributing multidimensional arrays. In Section 5.2.2, I consider the performance benefits of knowing parameters of the array assignment at compile time and compiling them into a custom function, versus using a general runtime library routine. In Section 5.2.3, I examine the effect of the deposit model and the direct deposit model of communication on the overall performance of the array assignment. Finally, in Section 5.2.4, I consider the effects of the optimizations I discussed previously in Sections 4.5 and 4.6.5.

5.2.1 Evaluation of the Syracuse approach

Section 2.3.7 describes the Syracuse algorithm [66] for redistributing arrays. The Syracuse algorithm is not a general array assignment algorithm, as it only allows redistributions of entire arrays, rather than arbitrary array sections (i.e., the subscript triplets have fixed values). The algorithm is specified for the general block-cyclic case, as well as for some special cases (e.g., block-to-cyclic redistributions and vice versa, and block-cyclic redistributions in which one block size is a multiple of the other). It is designed for use in the runtime support library of an HPF compiler.

The notable feature of the Syracuse approach is not the actual algorithm for determining the communication sets, but rather the treatment proposed for multidimensional arrays. The authors consider a *direct* and an *indirect* approach, as illustrated in Figure 5.7. In the indirect approach, only one dimension is redistributed at a time. For example, in redistributing a two-dimensional array from a (block,block) distribution to (cyclic,cyclic), the indirect approach first redistributes to (block,cyclic), and continues to (cyclic,cyclic) from there. Alternatively, it may use the (cyclic,block) distribution as the intermediate step. Regardless of which intermediate distribution is used, the indirect approach increases the total data movement by a factor of d, when d corresponding dimensions of the two arrays have different distributions. The direct approach, on the other hand, copies the data exactly once, redistributing all dimensions at the same time.

What advantage could the indirect approach's increased costs of communication and data movement provide over the direct approach? The authors correctly observe that extending the special cases of the canonical algorithm to a multidimensional algorithm causes an exponential blowup in the amount of code to write:

> This method requires different algorithms for different numbers of dimensions and different types of redistributions and these algorithms cannot be optimized much. [66]

To avoid the exponential blowup in the direct approach, optimized special cases can be applied to at most one array dimension; the rest of the dimensions must be handled using the potentially inefficient general case. The indirect approach, though, allows the special-case optimizations to be applied during each step of

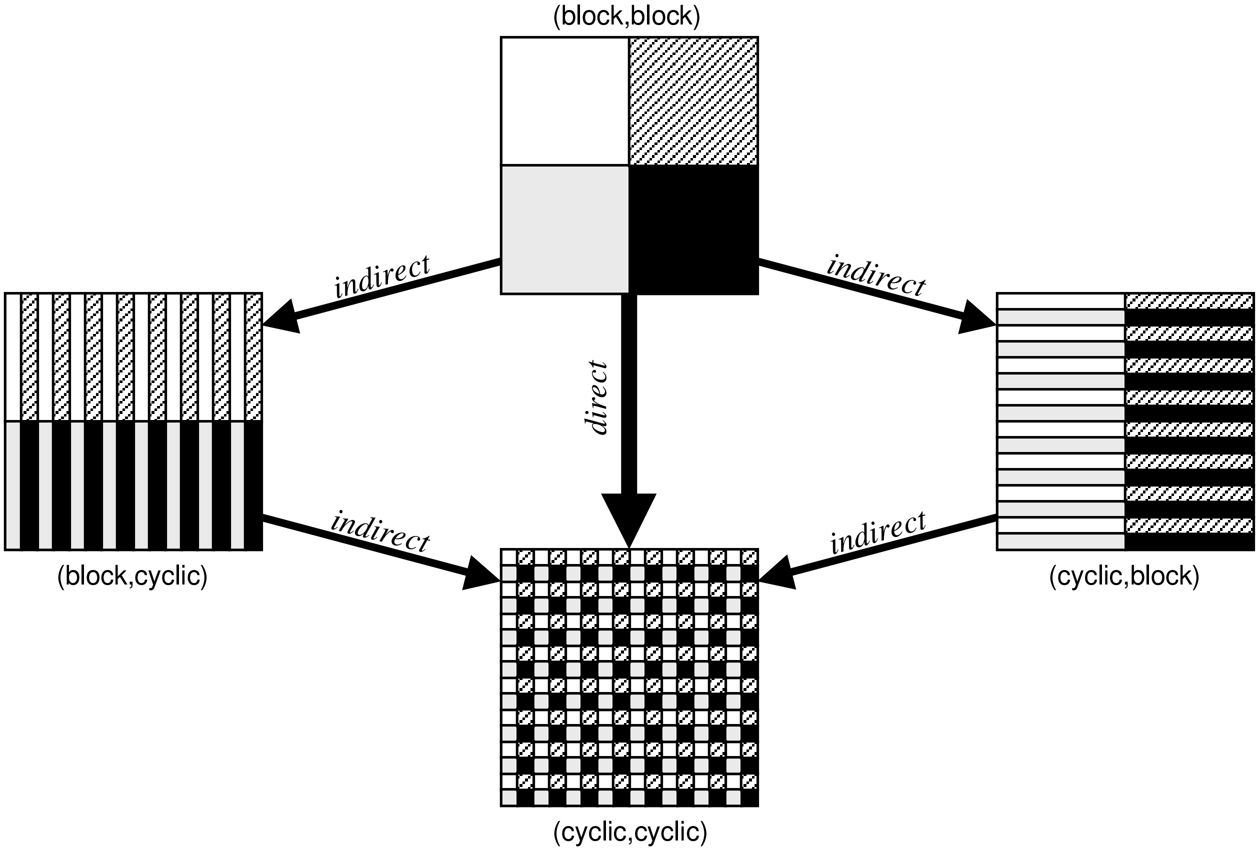


Figure 5.7: Illustration of the direct and indirect approaches to redistributing a two-dimensional array from (block,block) to (cyclic,cyclic). Each array dimension is distributed over two processors. The direct approach redistributes both dimensions in a single step. The indirect approach takes two steps, passing through either the (block,cyclic) or the (cyclic,block) intermediate distribution.

the redistribution. If the special-case optimizations yield enough of an improvement over the general case, then the indirect approach may outperform this unoptimized or partially-optimized implementation of the direct approach.

The key observation here is that in seeking to add generality to the algorithm (i.e., extending it to handle the general multidimensional case), the authors are forced to sacrifice either efficiency or maintainability. Acquiring efficiency means losing maintainability as they write an exponential number of special cases for multidimensional arrays, and acquiring maintainability means losing efficiency by copying the arrays several times. However, the authors argue that there is no such tradeoff: they present performance data indicating that the indirect approach outperforms the direct approach. I present data below indicating that the indirect approach *does* inflict the expected performance penalty, and that the technique of code composition achieves generality without sacrificing efficiency or maintainability.

The original Syracuse paper [66] presents performance data for redistributing a 1024×1024 element array from a (block,block) distribution to a (cyclic,cyclic) distribution. Their experiment was performed on an Intel Paragon, with the number of processors varying from 2 to 32. The data indicates that the indirect approach results in better overall performance, despite the doubling in overall data movement. Figure 5.8 shows their results, as well as some performance results for Catacomb/Fx-generated code. For my evaluation, I considered the performance of three array assignment algorithms: CMU, OSU, and RIACS.

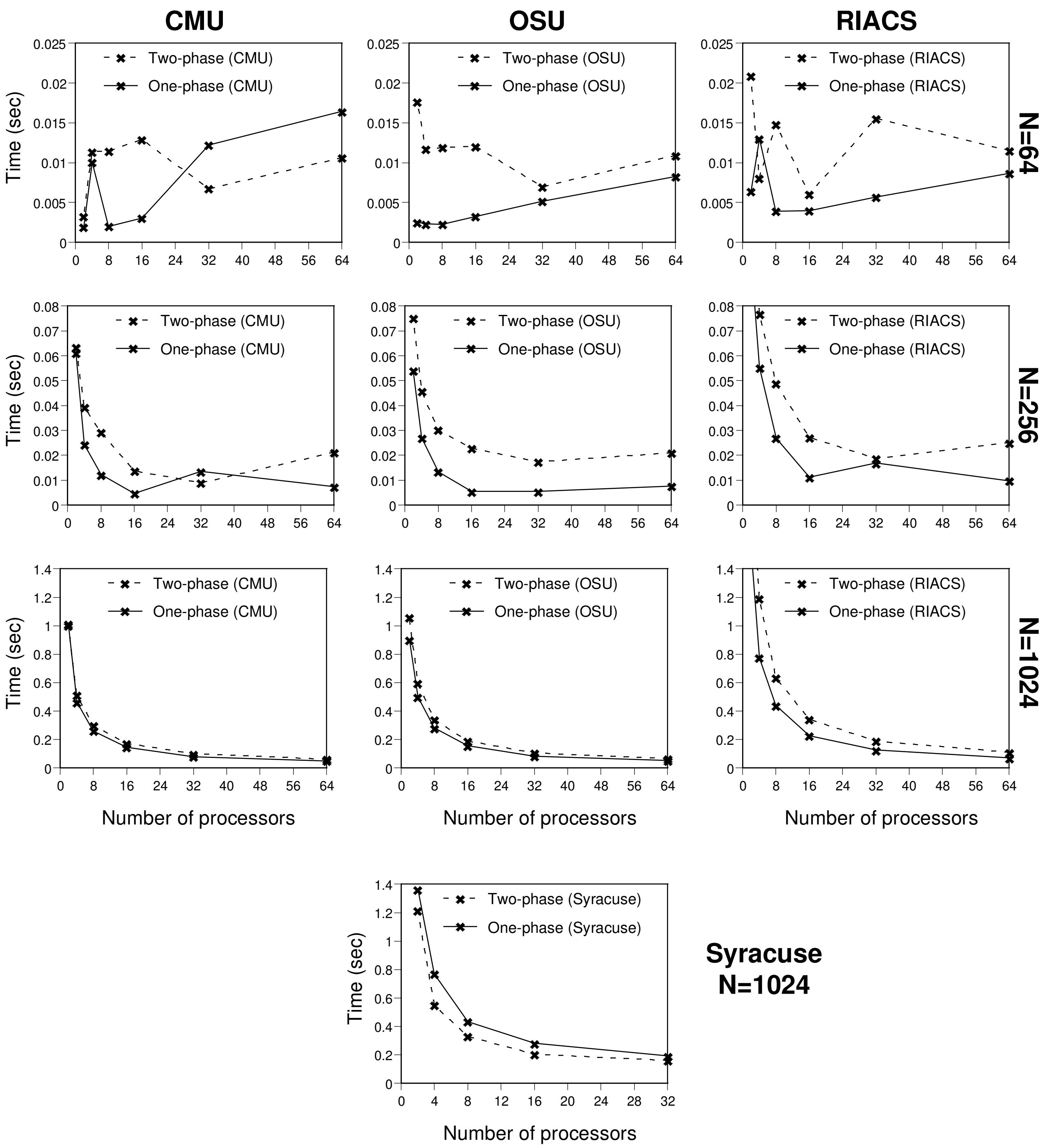


Figure 5.8: Performance comparison of the one-phase (direct) approach and the two-phase (indirect) approach for redistributing a two-dimensional $N \times N$ array on an Intel Paragon, from a (block,block) distribution to a (cyclic,cyclic) distribution. Note that the number of processors in the published Syracuse data ranges from 2 to 32, whereas the Catacomb/Fx experiments use 2 to 64 processors.

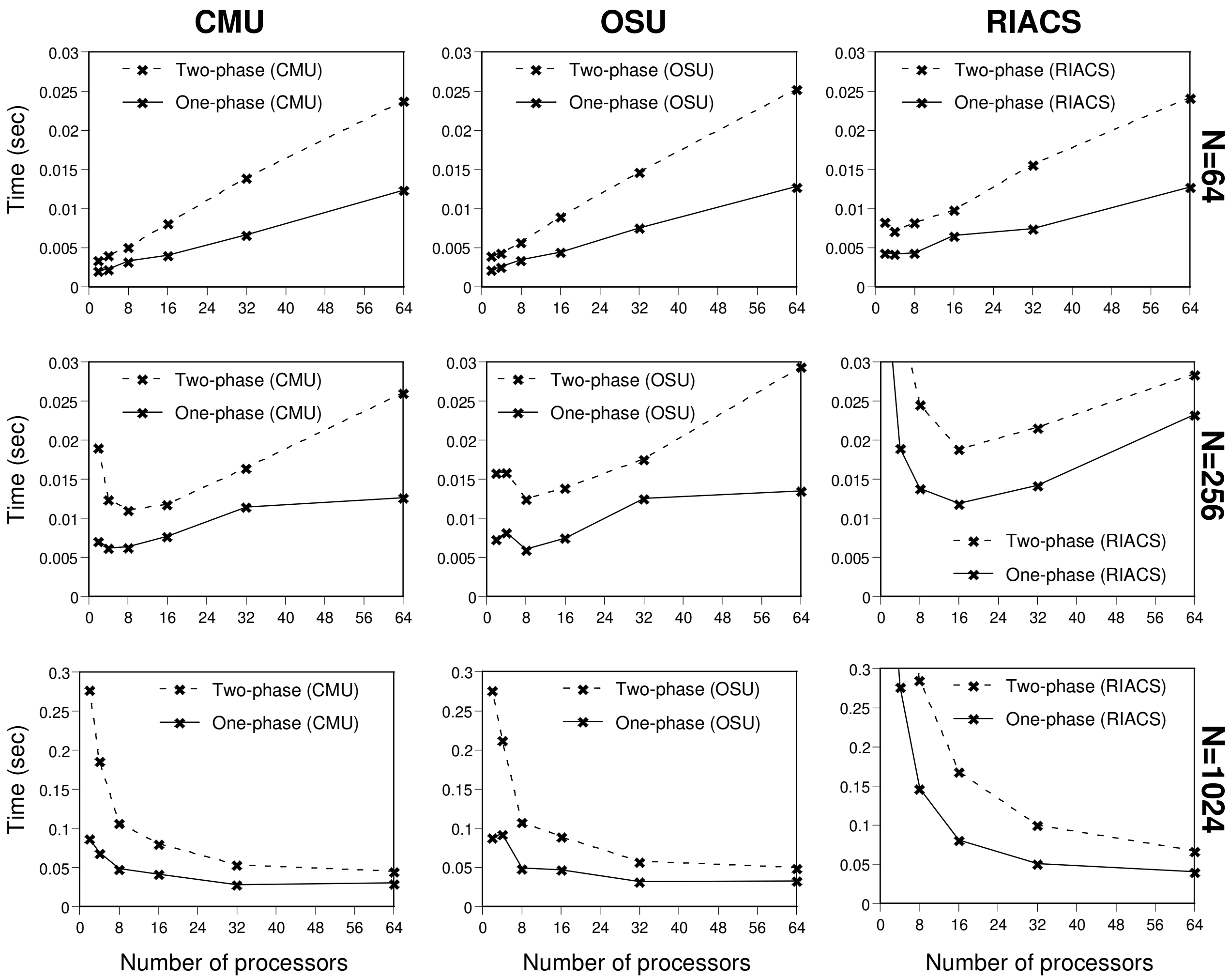


Figure 5.9: Performance comparison of the one-phase (direct) approach and the two-phase (indirect) approach for redistributing a two-dimensional $N \times N$ array on a Cray T3D, from a (block,block) distribution to a (cyclic,cyclic) distribution. Note that the Catacomb/Fx-generated code (i.e., the CMU, OSU, and RIACS algorithms) exhibits the expected performance: the two-phase approach is roughly twice the cost of the one-phase approach.

I compared the direct and indirect approaches on an Intel Paragon, varying the number of processors from 2 to 64, and testing array sizes ranging from 64×64 to 1024×1024. Each data point represents the performance of Catacomb/Fx-generated code in which all parameters are known at compile time. In other words, the array size and the number of processors are compiled into the executable.

The results from the Catacomb/Fx-generated code are overall the intuitive results. The data for the small values of N appears to be somewhat noisy, but in the larger problem sizes, the direct approach clearly outperforms the indirect approach. To validate the performance of the code generated by Catacomb/Fx, note that when comparing to the published Syracuse data, the CMU, OSU, and RIACS algorithms generally outperform the Syracuse algorithm.

I repeated this experiment on a Cray T3D, using the same set of parameters as in the experiment on the Paragon. The results are shown in Figure 5.9. The key observation here is that *all* of the Catacomb/Fx-generated examples exhibit the intuitive behavior: the indirect approach is roughly twice as expensive as

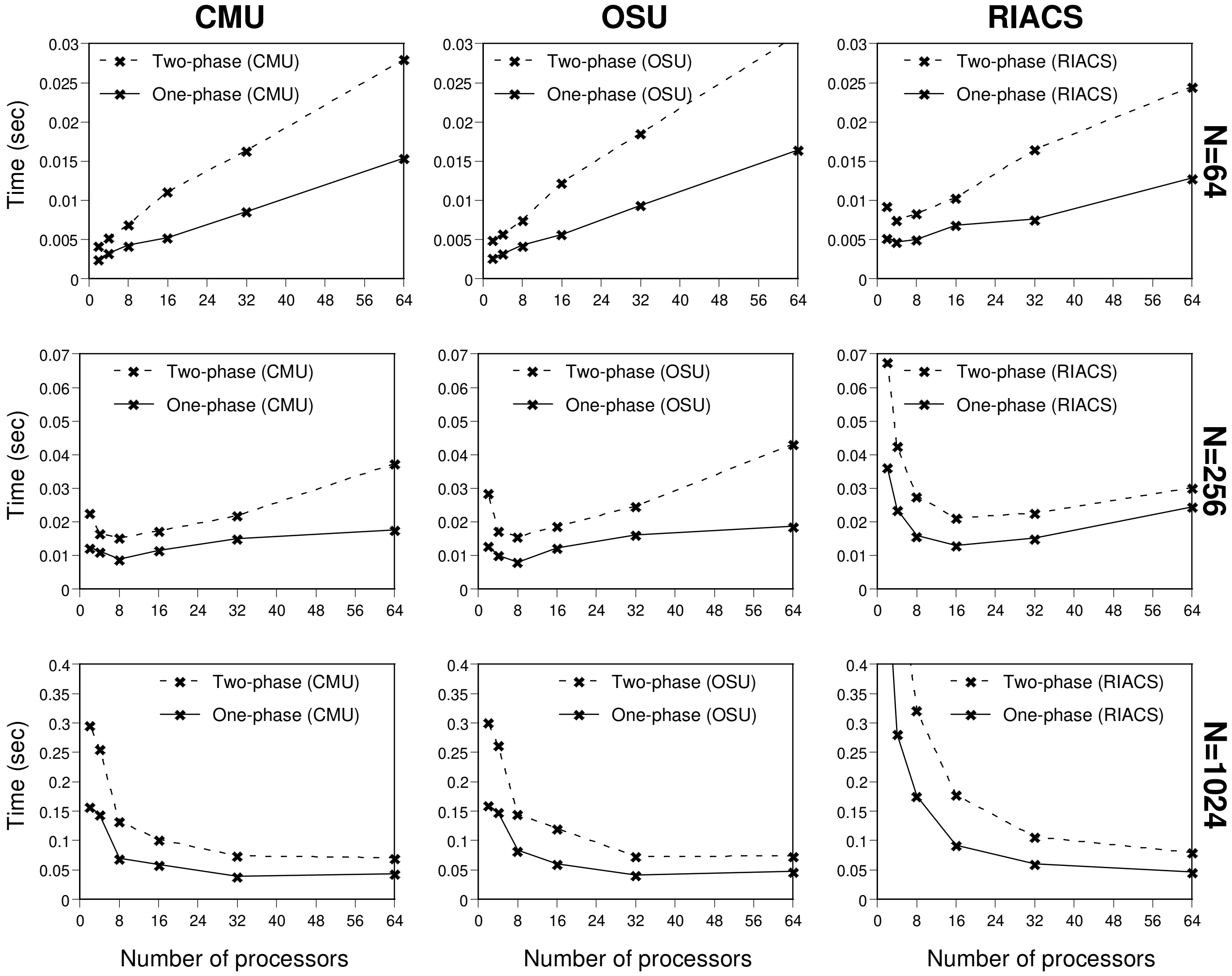


Figure 5.10: Performance comparison of the direct and indirect approaches for redistributing multidimensional arrays on a Cray T3D. The experimental parameters are identical to those of Figure 5.9, except that the array size and the number of processors are *not* compiled into the executable.

the direct approach, for redistributing two dimensions of an array. Over a fairly wide range of problem and machine sizes, the two approaches never come close in performance. The same behavior is observed even when the array size and the number of processors are *not* compiled into the executable, as shown in Figure 5.10. This experiment models the performance of a general library routine for redistributing two-dimensional arrays. Although the performance is a small factor worse than the corresponding graphs of Figure 5.9, the overall factor of two performance difference between the direct and indirect approaches remains evident.

To summarize, the developers of the Syracuse approach have noticed the engineering difficulties that arise when extending an algorithm for the canonical array assignment to multiple dimensions. In a runtime library implementation, the exponential explosion in the number of special cases precludes maintainability, leaving two choices: use the direct redistribution approach with no optimized special cases, or use the multi-step indirect approach. The indirect approach allows the one-dimensional optimized special cases to be used, at the cost of increasing the overall amount of data movement. My experiments show that code composition allows the canonical array statement algorithm to be automatically extended to and specialized for multiple dimensions, without a runtime performance penalty.

5.2.2 Custom code versus library code

One of the advantages of code composition is the ability to instantiate known compile-time parameters into the resulting code, without having to rely on the target system's compiler to inline library routines. In this experiment, I look at a few examples of array assignment statements, and I show the effect of various amounts of compile-time information on the runtime performance. All experiments were performed on a Cray T3D system with 150 MHz processors, and the experiments examine the performance of the CMU, OSU, and RIACS algorithms.

For the first test, I considered a cyclic-to-block redistribution of a one-dimensional array of 64-bit values. This redistribution can be expressed as the array assignment statement $A[0\colon N-1\colon 1] = B[0\colon N-1\colon 1]$, where A has a block distribution and B has a cyclic distribution. I consider the effect of knowing the specific parameters of the subscript triplets (i.e., $0\colon N-1\colon 1$ versus the more general $\ell\colon h\colon s$), as well as the effect of knowing the distribution parameters. Knowing both sets of parameters represents the full custom code generation case, while knowing none of these parameters represents a general one-dimensional array assignment statement library routine. Knowing the subscript triplet parameters at compile time but not the distribution parameters represents a library for redistributing one-dimensional arrays. On the other hand, knowing the exact distribution parameters but not the subscript triplet parameters is a less likely candidate for a runtime library routine, and thus is not measured here.

Figure 5.11 shows the effect of compile-time knowledge on the performance of a one-dimensional cyclic-to-block redistribution, for the CMU, OSU, and RIACS algorithms. The data labeled "All" represents the case in which all parameters of the array assignment statement are compiled into the code, and the data labeled "None" represents the case in which none of the parameters is compiled in (i.e., a fully general library routine for the one-dimensional array assignment statement). The "Redist" data represents an intermediate point in which the subscript triplet parameters are known at compile time but the array distributions are not; this is equivalent to a runtime library for redistributing a one-dimensional array. The array size varies from 1K (2^{10}) elements to 1024K (2^{20}) elements, and the number of processors varies from 2 to 64.

The data shows that for all three algorithms, a small but noticeable performance improvement results from knowing the parameters of the array assignment statement at compile time. In addition, relatively little performance is gained by using a redistribution library routine instead of a library routine for the general array assignment. All three methods become closer and closer together as the problem size increases. This is because the presence or absence of compile-time information ultimately does not affect the overall structure of the inner buffer packing/unpacking loop, where the bulk of the work is done.

Figure 5.12 illustrates the difference in the quality of the generated code when the parameters are known at compile time. The large block of code pictured is the buffer-packing portion of the CMU algorithm, when all parameters are unknown at compile time. The smaller block of code results from instantiating the parameters to values that make it a cyclic-to-block redistribution of a 1024-element array distributed over 8 processors. The code blocks fill a communication buffer with values to be sent from processor `rcellid` to `proc`. These blocks of code were used in the measurements of Figure 5.11. Both blocks of code were automatically generated and optimized (where possible) by Catacomb/Fx. One can see that a large amount of overhead is automatically eliminated when the parameters are known at compile time.

For the second test, I considered the redistribution of a two-dimensional array from (cyclic,cyclic) to (block,block). Once again, I examined the effect of knowing the subscript triplet parameters and the distribution parameters at compile time versus a fully general two-dimensional array assignment library routine. In addition, I considered an intermediate step, in which the subscript triplet parameters are known at compile time but the distribution parameters are not. Essentially, this is the same experiment as above, except that it

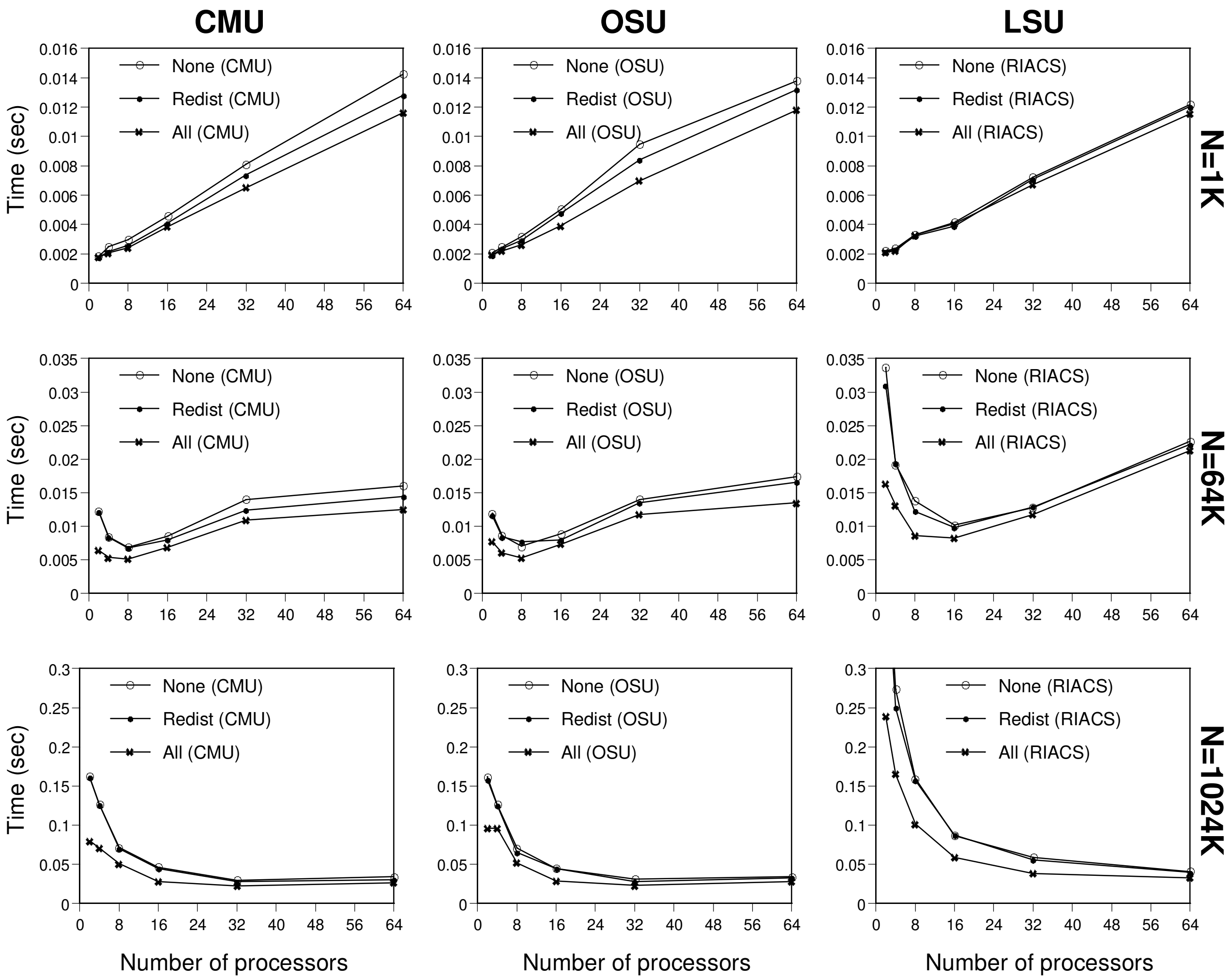


Figure 5.11: The effect of compile-time knowledge on the performance of a one-dimensional redistribution.

extends the one-dimensional test to two dimensions.

Figure 5.13 shows the performance results of this experiment. As before, "All" represents the case where all parameters are known at compile time, "None" represents the case where no parameters are known at compile time, and "Redist" represents the case where only the distributions are unknown at compile time. The array size is $N \times N$ for N ranging from 64 to 1024, and the number of processors varies from 2 to 64. The CMU, OSU, and RIACS algorithms are tested.

The results are nearly identical to those of the one-dimensional redistribution. There is a noticeable performance improvement between knowing all parameters at compile time and knowing none, and the performance difference between knowing none of the parameters and knowing only the subscript triplet parameters is in general fairly small.

5.2.3 Effects of the deposit model

The structure of the Catacomb/Fx templates for the array assignment allows for the use of the deposit model and the direct deposit model of communication. On an architecture that supports direct deposit, any algorithm that uses the deposit model can automatically use the direct deposit model as well.

```
euc_u1 = 1;
euc_u3 = s1;
euc_v1 = 0;
euc_v3 = bbs * Pb;
while (euc_v3 != 0) {
  euc_t1 = euc_u1 - (euc_v1 * (euc_u3 / euc_v3));
  euc_t3 = euc_u3 - (euc_v3 * (euc_u3 / euc_v3));
  euc_u1 = euc_v1;
  euc_u3 = euc_v3;
  euc_v1 = euc_t1;
  euc_v3 = euc_t3;
}
x1_0 = euc_u1;
g1_0 = euc_u3;
euc_u1 = 1;
euc_u3 = (s2 * (bbs * Pb)) / euc_u3;
euc_v1 = 0;
euc_v3 = cbs * Pc;
while (euc_v3 != 0) {
  euc_t1 = euc_u1 - (euc_v1 * (euc_u3 / euc_v3));
  euc_t3 = euc_u3 - (euc_v3 * (euc_u3 / euc_v3));
  euc_u1 = euc_v1;
  euc_u3 = euc_v3;
  euc_v1 = euc_t1;
  euc_v3 = euc_t3;
}
sstride_0 = ((s2 * (bbs * Pb)) * (cbs * Pc)) / (g1_0 * euc_u3);
lmsstride_0 = ((s2 * (bbs * Pb)) * cbs) / (g1_0 * euc_u3);
slast_0 = MIN(Nc + (-1), l2 + (s2 * FLDIV(MIN(h1, Nb + (-1)) - l1, s1)));
lmslast_0 = (FLDIV(slast_0 - (rcellid * cbs), cbs * Pc) * cbs) +
            MIN(PMOD(slast_0 - (rcellid * cbs), cbs * Pc), cbs + (-1));
scc_0 = MAX(l2 + (s2 * CEILDIV(MAX(l1, proc * bbs) - l1, s1)),
            rcellid * cbs);
pl_0 = CEILDIV((proc * bbs) - l1, g1_0);
pu_0 = FLDIV((((proc * bbs) - l1) + bbs) + (-1), g1_0);
for (outer_0 = pl_0; outer_0 <= pu_0; outer_0++) {
  t_0 = ((rcellid * cbs) - l2) - ((outer_0 * x1_0) * s2);
  pui_0 = FLDIV((t_0 + cbs) + (-1), euc_u3);
  for (inner_0 = CEILDIV(t_0, euc_u3); inner_0 <= pui_0; inner_0++) {
    r_0 = ((rcellid * cbs) - t_0) +
          ((((inner_0 * euc_u1) * s2) * (bbs * Pb)) / g1_0);
    sfirst_0 = scc_0 + PMOD(r_0 - scc_0, sstride_0);
    iprime_0 = (euc_u3 * inner_0) - t_0;
    lmsfirst_0 = ((((sfirst_0 - (rcellid * cbs)) - iprime_0) * cbs) / (cbs * Pc))
                 + iprime_0;
    for (pack_0 = lmsfirst_0; pack_0 <= lmslast_0; pack_0 += lmsstride_0)
      sbuf_0[sbufptr++] = c[pack_0];
  }
}
```

```
slast_0 = (proc * 128) + 127;
lmslast_0 = (slast_0 - rcellid) / 8;
scc_0 = MAX(proc * 128, rcellid);
sfirst_0 = scc_0 + PMOD(rcellid - scc_0, 8);
lmsfirst_0 = (sfirst_0 - rcellid) / 8;
for (pack_0 = lmsfirst_0;
     pack_0 <= lmslast_0;
     pack_0 += 1)
  sbuf_0[sbufptr++] = c[pack_0];
```

Figure 5.12: Generated code in the presence and absence of compile-time information. Pictured is the buffer-packing loop of the CMU algorithm for a general one-dimensional array assignment statement. The smaller code block highlighted in gray is the code generated and optimized by Catacomb/Fx for the cyclic-to-block redistribution of a 1024-element array distributed over 8 processors.

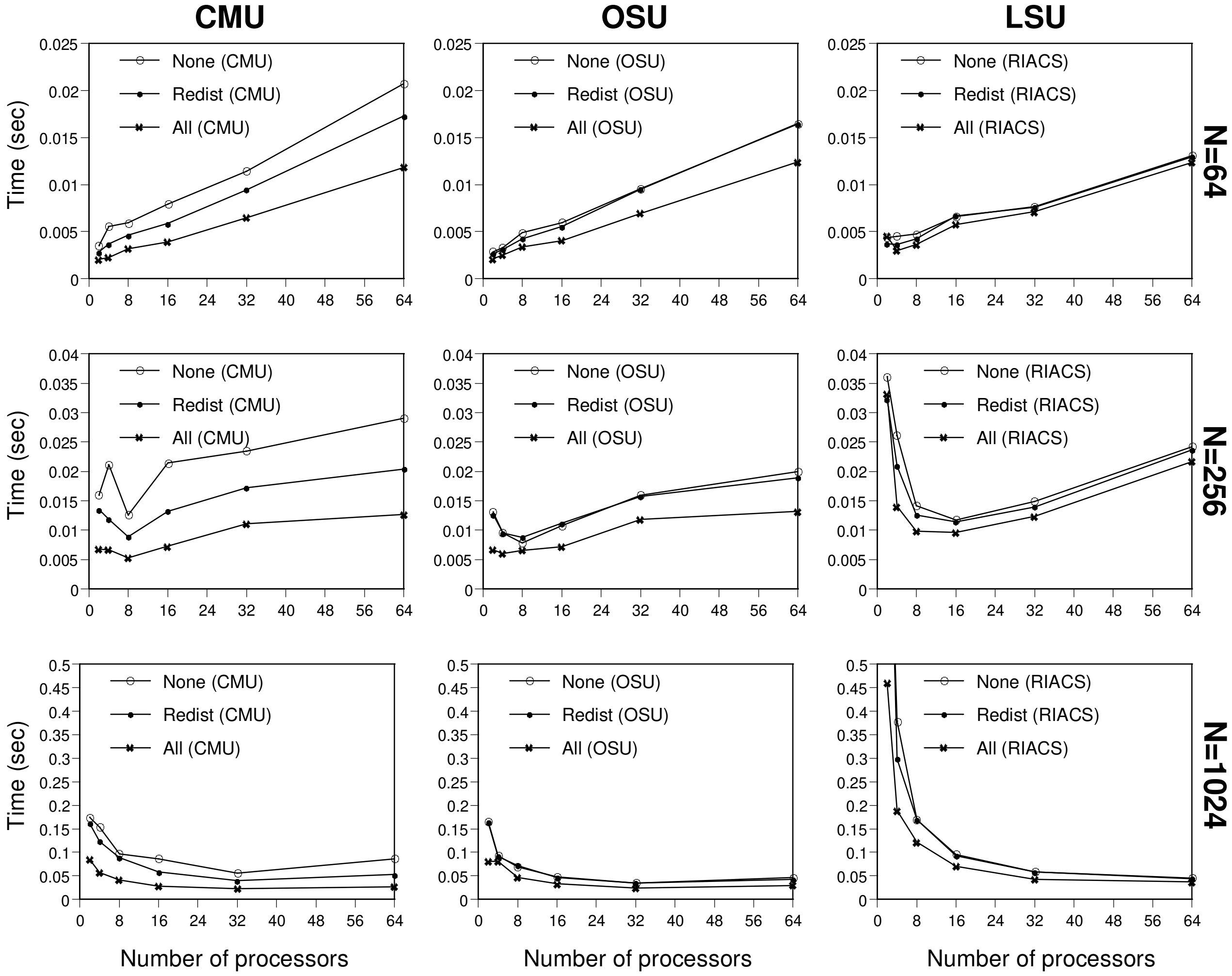


Figure 5.13: The effect of compile-time knowledge on the performance of a two-dimensional redistribution of an $N \times N$ array.

In this experiment, I show that the direct deposit model of communication is highly beneficial to the overall performance. I measured the performance of the CMU algorithm using the standard model, the deposit model, and the direct deposit model of communication. In addition, I measured the performance of the RIACS algorithm using the deposit and direct deposit models (the RIACS algorithm is formulated to use the deposit model, so no comparison to the standard model is possible). Unfortunately, there is no specification of the OSU algorithm using the deposit model, so neither the deposit model nor the direct deposit model is possible to compare.

I measured the performance of two array assignment statements for this experiment. The first array assignment statement is a redistribution of a one-dimensional cyclic array to a block distribution. The second array assignment statement is similar: the redistribution of a two-dimensional $N \times N$ (cyclic,cyclic) array to a (block,block) distribution. I varied the number of processors from 2 to 64 in powers of 2, and I examined a variety of array sizes. For the one-dimensional redistribution, I varied the array size from 1K (2^{10}) elements to 1024K (2^{20}) elements, and for the $N \times N$ two-dimensional redistribution, I varied N from 64 to 1024. In both cases, the arrays consist of 64-bit integer values.

For both array assignment statements, neither the problem size nor the number of processors is known at compile time. The array distributions are known at compile time (for the block distributions, the actual block size is unknown because the array size and number of processors are unknown, but the Catacomb/Fx templates still know that the distribution is guaranteed to be block). Regarding the subscript triplet values, I ran two versions of each experiment, one where the values are known and one where they are unknown at compile time.

For the experiment, I used a Cray T3D system, with the computation nodes operating at 150 MHz. The deposit model and the standard model both use MPI routines for the communication, while the direct deposit model uses the T3D's shared memory features to effect the transfer of data. Unfortunately, a limitation of the T3D prevents the full implementation of direct deposit for the RIACS algorithm (as well as the other related table-generation algorithms). The *DTB Annex* [1] is a 32-entry table that allows a T3D node to send data to up to 32 nodes at once. However, in the context of direct deposit, the RIACS algorithm expects to communicate with all other nodes in a fine-grained interleaved fashion, which is not possible when the array is distributed over more than 32 processors. Thus under the direct deposit model, the RIACS algorithm packs communication buffers as usual, but instead of using MPI routines for the communication followed by an unpacking loop, it loops through each communication buffer in turn, using the direct deposit method to remotely store the individual elements. On the other hand, the CMU algorithm is explicitly formulated to focus on one destination processor at a time, so the direct deposit model can be used without any intermediate communication buffers.

Figure 5.14 shows the performance of the one-dimensional redistribution for three different problem sizes: 1K, 64K, and 1024K element arrays. Figure 5.15 shows the performance of the two-dimensional redistribution for 64×64, 256×256, and 1024×1024 element arrays. "Standard" refers to the standard communication model, "deposit" refers to the deposit model in conjunction with MPI primitives, and "direct" refers to the direct deposit model. In all of these graphs, the parameters of the subscript triplets are known at compile time. The most striking observation is the degree to which the direct deposit model improves the performance in all cases. For small problem sizes, the overhead in sending many small messages via MPI dominates the execution time, whereas the fine-grained remote stores allow the problems to scale reasonably well as the number of processors increases. For larger problem sizes, where per-message overheads are amortized over large messages, direct deposit still improves performance by a factor of around 2, due to the decrease in the number of times the data is copied.

There are two other points to mention with regard to this experiment. First, even when the subscript triplet parameters are not known at compile time, the results have the same character. Although the absolute times are around 10–15% higher, the direct deposit model shows the same level of improvement over the standard message passing models. (Because the results are so similar, I omit presenting the graphs here.) Second, it is interesting to note that the CMU algorithm shows little difference between the standard and deposit models, yet without the algorithm for the deposit model, direct deposit would not be possible.

To summarize, Catacomb/Fx's modular template structure allows any array assignment algorithm that supports the deposit model to be matched up with any architecture that supports shared memory style writes to form a direct deposit algorithm. As seen on the T3D, direct deposit can lead to a substantial performance improvement.

5.2.4 Effects of nonstandard global optimizations

Sections 4.5 and 4.6.5 describe several nonstandard global optimizations that Catacomb implements. This includes the optimizations based on bound analysis, as well as the single-phase execution model of the control constructs and the global optimizations. In this section, I discuss the effectiveness of these optimizations.

In general, for the array assignment statement, these extra optimizations provide little additional perfor-

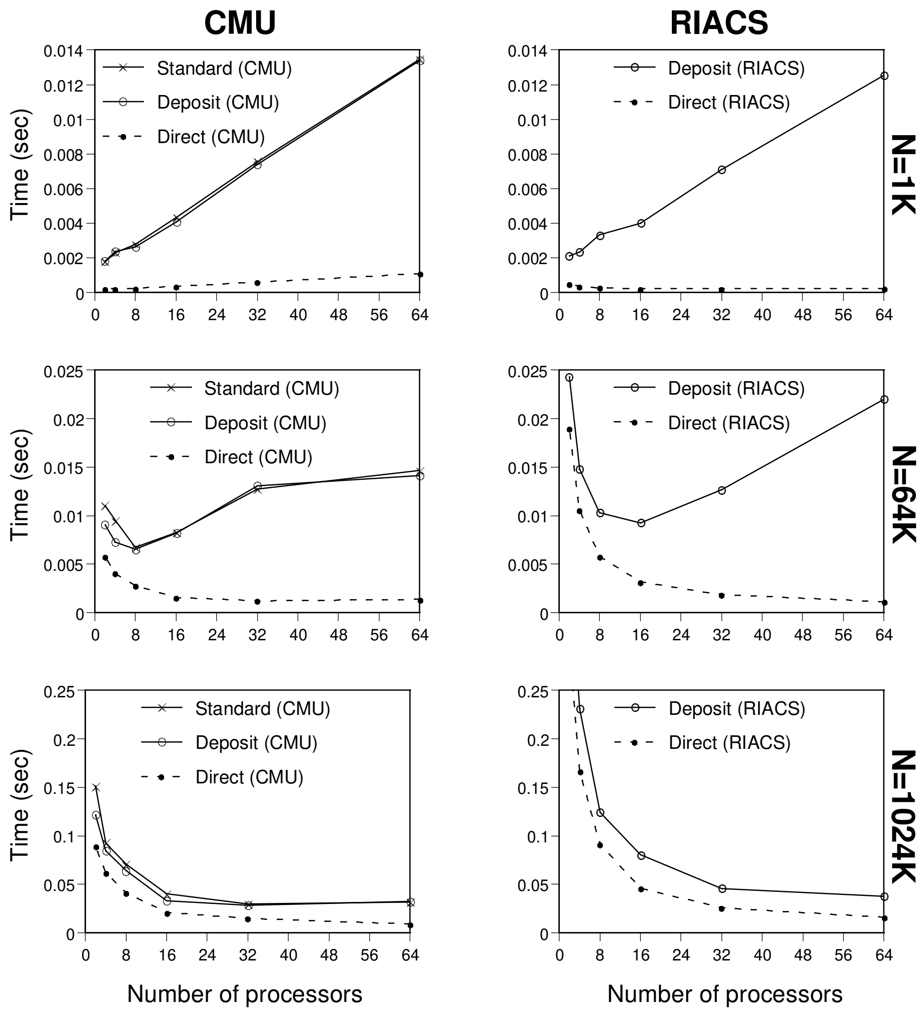


Figure 5.14: Performance of the direct deposit model of communication for the redistribution of a one-dimensional array of size N from cyclic to block.

mance. The reason is that the optimizations tend to eliminate fixed overhead, rather than per-element costs. Because most of the time is spent in the inner loop of the generated code, as the problem size increases, the performance benefit of the extra optimizations decreases proportionally.

Figure 5.16 shows the effect of the optimizations on the performance of the CMU algorithm for redistributing an N-element one-dimensional array from block to cyclic. For small problems, there is a large improvement, but as the problem size increases, the improvement becomes negligible. Figure 5.17 shows similar behavior for a two-dimensional redistribution of an $N \times N$ element array. In all of the tests, all parameters of the array assignment statement are compiled into the executable. Furthermore, the performance results were measured using the direct deposit communication method (as discussed in Section 5.2.3) on the Cray T3D, meaning that message passing overhead has already been eliminated. Thus, when using a more standard message passing model (e.g., MPI), the performance improvements of the nonstandard global optimizations are lost in the noise.

These optimizations are still useful for a number of reasons. First, it may not be possible in practice to scale up the size of the array assignment statement to amortize constant overhead, depending on the specific problem being solved. Second, the optimizations may be more effective in a different problem domain.

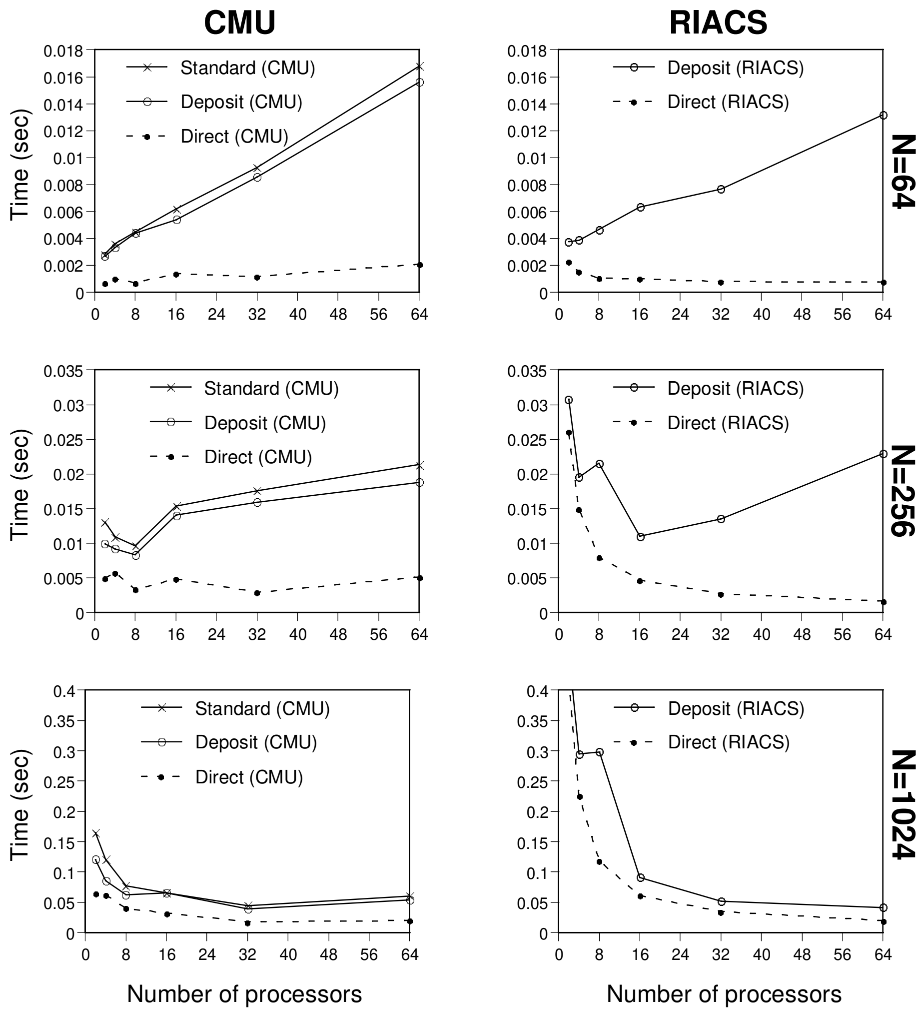


Figure 5.15: Performance of the direct deposit model of communication for the redistribution of a two-dimensional $N \times N$ array from (cyclic,cyclic) to (block,block).

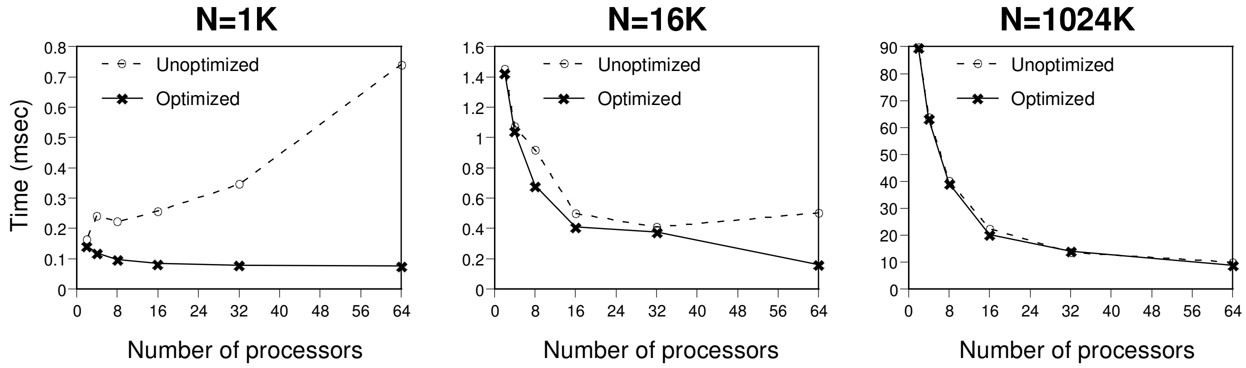


Figure 5.16: The effect of Catacomb's nonstandard global optimizations on the performance of a one-dimensional redistribution of an N-element array. The performance is measured using the CMU algorithm to redistribute the array from block to cyclic. The test uses the direct deposit communication style on the Cray T3D.

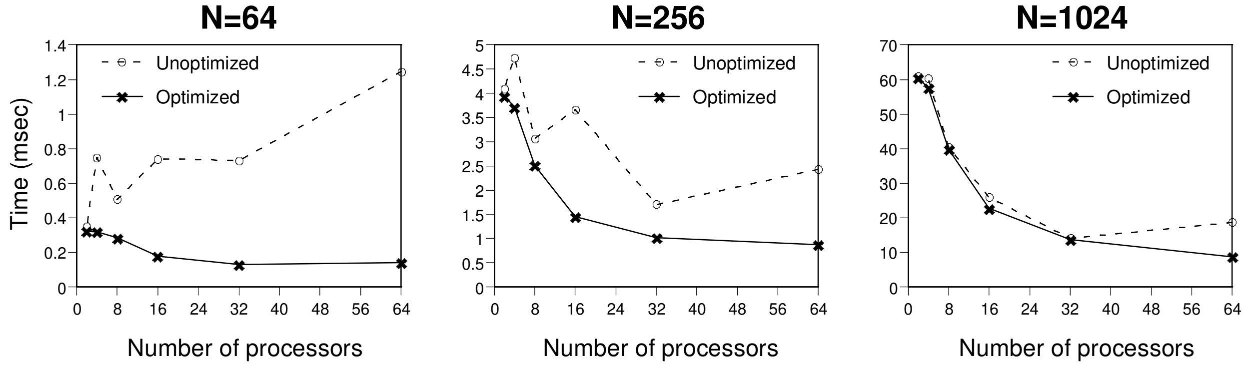


Figure 5.17: The effect of Catacomb's nonstandard global optimizations on the performance of a two-dimensional redistribution of an $N \times N$ array. The performance is measured using the CMU algorithm to redistribute the array from (block,block) to (cyclic,cyclic). The test uses the direct deposit communication style on the Cray T3D.

Third, the optimizations improve the quality of the generated C code, which can be a great help to the template writer. To illustrate, Figures 5.18 and 5.19 show actual Catacomb-generated code fragments for the two-dimensional redistribution, whose performance is shown in Figure 5.17. The purpose of each code fragment is to iterate over the elements of the right-hand side array. Figure 5.18 contains the code produced without the additional optimizations, and Figure 5.19 contains the code with all optimizations.

The code in Figure 5.19 is still not optimal. For example, it is possible to prove that `lmsfirst_1` can be simplified to `(proc%8)*16`, leaving the assignments to `scc_1` and `sfirst_1` dead. Bounds analysis is insufficient to prove this, so Catacomb is incapable of automatically producing the simpler code. In such a case, if the better code is required, the template writer is responsible for writing the special-case templates that detect and optimize this case. Clearly, it is easier to discover this optimization potential by studying the code of Figure 5.19 than by studying the code of Figure 5.18.

5.3 Generality and maintainability

Beyond runtime efficiency, two additional goals of a code composition system are to provide generality in the input it can handle, and maintainability in the specification of the code to generate. In this section, I discuss these issues as they relate to the Catacomb/Fx implementation of the array assignment. Because of the difficulty in quantitatively evaluating generality and maintainability in Catacomb/Fx, I report here the qualitative aspects.

5.3.1 Generality

Catacomb/Fx correctly handles all instances of the array assignment statement, provided that there are no vector-valued subscripts (e.g., $A[IA[1\!:n]]$). The three conceptual problems in extending the canonical example to the general case are multidimensional arrays (including index permutations), multiple right-hand side terms, and scalar subscripts mixed in with the subscript triplets. These problems and their solutions are discussed in Section 3.1.1. The Catacomb/Fx templates correctly apply the necessary transformations to produce correct code.

Figure 5.20 summarizes the generality-related features of the array assignment statement that the Catacomb framework, as well as a number of other array assignment implementations, supports. The "branching

```
tmppid_0_0 = (proc % 64) / 8;
tmppid_2_0 = (rcellid % 64) / 8;
if ((proc < 64) && (rcellid < 64)) {
  sbufptr = 0;
  euc_u1 = 1;
  euc_u3 = 1;
  euc_v1 = 0;
  euc_v3 = 8;
  while (euc_v3 != 0) {
    euc_t1 = euc_u1 - (euc_v1 * (euc_u3 / euc_v3));
    euc_t3 = euc_u3 - (euc_v3 * (euc_u3 / euc_v3));
    euc_u1 = euc_v1;
    euc_u3 = euc_v3;
    euc_v1 = euc_t1;
    euc_v3 = euc_t3;
  }
  lmsstride_0 = 1 / euc_u3;
  slast_0 = MIN((tmppid_0_0 * 128) + 127, 1023);
  lmslast_0 = FLDIV(slast_0 - tmppid_2_0, 8);
  scc_0 = MAX(MAX(tmppid_0_0 * 128, 0), tmppid_2_0);
  if (((-tmppid_2_0) % euc_u3) == 0) {
    r_0 = (tmppid_2_0 * euc_u1) / euc_u3;
    sfirst_0 = scc_0 + PMOD(r_0 - scc_0, 8 / euc_u3);
    lmsfirst_0 = (sfirst_0 - tmppid_2_0) / 8;
    for (pack_0 = lmsfirst_0; pack_0 <= lmslast_0; pack_0 += lmsstride_0) {
      euc_u1 = 1;
      euc_u3 = 1;
      euc_v1 = 0;
      euc_v3 = 8;
      while (euc_v3 != 0) {
        euc_t1 = euc_u1 - (euc_v1 * (euc_u3 / euc_v3));
        euc_t3 = euc_u3 - (euc_v3 * (euc_u3 / euc_v3));
        euc_u1 = euc_v1;
        euc_u3 = euc_v3;
        euc_v1 = euc_t1;
        euc_v3 = euc_t3;
      }
      slast_1 = MIN(((proc % 8) * 128) + 127, 1023);
      lmslast_1 = FLDIV(slast_1 - (rcellid % 8), 8);
      scc_1 = MAX(MAX((proc % 8) * 128, 0), rcellid % 8);
      if (((-(rcellid % 8)) % euc_u3) == 0) {
        r_1 = ((rcellid % 8) * euc_u1) / euc_u3;
        sfirst_1 = scc_1 + PMOD(r_1 - scc_1, 8 / euc_u3);
        lmsfirst_1 = (sfirst_1 - (rcellid % 8)) / 8;
        for (pack_1 = lmsfirst_1; pack_1 <= lmslast_1; pack_1 += 1 / euc_u3)
          sbuf_0[sbufptr++] = c[pack_0][pack_1];
      }
    }
  }
}
```

Figure 5.18: C code generated by Catacomb for iterating over the right-hand side elements of a two-dimensional array, using only standard global optimizations.

```
sbufptr = 0;
slast_0 = ((proc / 8) * 128) + 127;
lmslast_0 = (slast_0 - (rcellid / 8)) / 8;
scc_0 = MAX((proc / 8) * 128, rcellid / 8);
sfirst_0 = scc_0 + PMOD((rcellid / 8) - scc_0, 8);
lmsfirst_0 = (sfirst_0 - (rcellid / 8)) / 8;
for (pack_0 = lmsfirst_0; pack_0 <= lmslast_0; pack_0 += 1)
  {
    slast_1 = ((proc % 8) * 128) + 127;
    lmslast_1 = (slast_1 - (rcellid % 8)) / 8;
    scc_1 = MAX((proc % 8) * 128, rcellid % 8);
    sfirst_1 = scc_1 + PMOD((rcellid % 8) - scc_1, 8);
    lmsfirst_1 = (sfirst_1 - (rcellid % 8)) / 8;
    for (pack_1 = lmsfirst_1; pack_1 <= lmslast_1; pack_1 += 1)
      sbuf_0[sbufptr++] = c[pack_0][pack_1];
  }
```

Figure 5.19: C code generated by Catacomb for iterating over the right-hand side elements of a two-dimensional array, using all of Catacomb's global optimizations.

factor" entry refers to the number of cases the implementation provides for a one-dimensional array assignment. If the one-dimensional array assignment has b branches, a d-dimensional assignment requires b^d cases. Note that the Catacomb framework and the original Fx compiler are the only implementations that provide all of the features listed, but the original Fx compiler was much more limited and much larger than the Catacomb framework.

The most challenging aspect related to generality was to design the algorithm and architecture frameworks, which allow an arbitrary array assignment algorithm to be coupled with an arbitrary communication architecture to form a complete array assignment implementation. I discussed the design of these frameworks in Section 5.1.1. As an example of the complexity of interaction between the two frameworks, consider the issue of communication buffer allocation. The architecture component allocates the send and receive buffers, but the algorithm component determines the length of the buffer and the type of each buffer element. The number of buffers to allocate depends in part on the algorithm component and in part on the architecture component. Some algorithms need only one send buffer, and others require multiple send buffers that are packed simultaneously, in an interleaved fashion. The same holds for receive buffers, although the architecture component may choose to allocate many receive buffers to optimize communication performance, regardless of whether the algorithm component needs only one receive buffer. Another issue relates to the allocation of multiple buffers. Some algorithms prefer to treat multiple buffers as a one-dimensional array of buffers; others prefer a multidimensional array of buffers, where each dimension corresponds to a distributed dimension of the source array in the assignment statement. For example, if one dimension of the source array is distributed over 3 processors and another dimension is distributed over 4 processors, then the algorithm component may wish to have a linear array of 12 buffers, or it may prefer a 3×4 array of buffers instead. The architecture component needs to be prepared to allocate the buffers and treat them in the appropriate manner. On the other hand, when using the direct deposit model of communication, it may be that no buffers need to be allocated at all.

After carefully studying the requirements of the various array assignment algorithms, I was able to develop the frameworks of the algorithm and architecture components, and provide generality between the two.

	Catacomb	CMU (Fx)	OSU	RIACS	LSU	Syracuse	ADAPTOR	IBM
Handles communication	✓	✓	✓	✓	✓	✓	✓	
Handles local computation	✓	✓		✓	✓		✓	✓
Arbitrary subscript triplets	✓	✓	✓	✓	✓	✓	✓	✓
Arbitrary block-cyclic distributions	✓	✓	✓	✓	✓			✓
Arbitrary number of dimensions	✓	✓				✓	✓	
Branching factor per dimension	varies[(a)]	1,4,9[(b)]	4[(c)]	2[(d)]	2[(e)]	5[(f)]	1[(g)]	1
Handles multiple RHS terms	✓	✓					✓	
Handles scalar subscripts	✓	✓					✓	
Supports deposit model	✓	✓		✓	✓			

(a): determined by which algorithm is used.
(b): an implementation parameter determines whether 1, 2, or 3 branches per array are generated.
(c): each array can independently be treated as virtual-block or virtual-cyclic.
(d): a runtime test selects either the standard method or the “shift” optimization.
(e): the implementation strip-mines the loop, resulting in two loops per dimension.
(f): one general case plus four special cases.
(g): generates one branch for each dimension, but the last dimension may have special cases.

Figure 5.20: Summary of the generality-related features of Catacomb, as well as other existing implementations of the array assignment statement.

5.3.2 Maintainability

Compared to the original Fx implementation of the CMU algorithm, Catacomb/Fx offers a huge improvement in maintainability. Catacomb offers two key improvements. First, it uses a simple and straightforward syntax in the template files for generating code, and for testing and computing values at compile time, rather than constructing expressions through primitives like `mk_const()` and `mk_expr()`. Second, the use of code templates provides a much more structured approach to writing code than embedding the code generation in the compiler.

As a more concrete measure of maintainability, I considered the size and makeup of the template files. The idea is to compare the amount of code constructs to the amount of control constructs in the templates. 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.

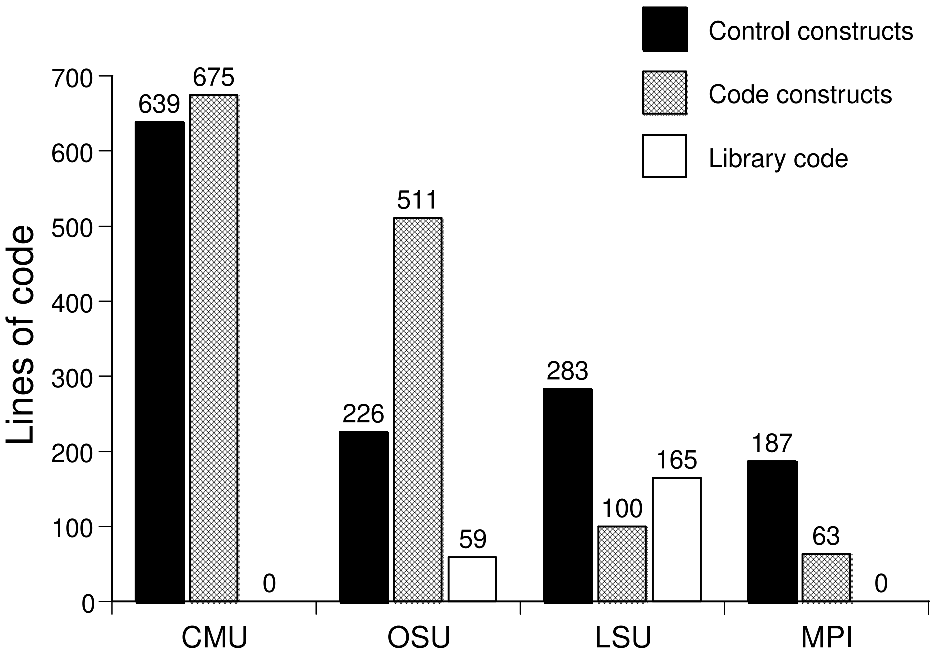


Figure 5.21: Comparison of the number of lines of control constructs and code constructs in the Catacomb/Fx templates, as well as lines of code in support libraries.

Figure 5.21 shows the breakdown of the CMU, OSU, and RIACS template code, as well as the MPI architecture template code. This measurement was taken after removing comments and blank lines from the templates, and should only be considered as an approximation. I 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 Fx implementation of the CMU algorithm 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/Fx templates contain far less code devoted to compile-time control, the implementation is far more maintainable.

Another important feature that enhances maintainability is the fact that the code templates are read and interpreted when Catacomb executes, rather than being compiled into the system. In Fx, making a change to the code that was generated required recompiling the file in question and relinking the system before the change could be tested. In Catacomb, changing a code template does not require any changes to Catacomb itself; thus the latency of making a change is reduced by one or two orders of magnitude. In addition, because Catacomb is built as a shared, dynamically linked library where possible, changes to the Catacomb infrastructure itself generally do not require rebuilding the compiler, such as Fx, to which Catacomb is linked.

On the other hand, in its current state, Catacomb has a serious deficiency: the lack of good debugging support. Given that code templates in Catacomb form a programming language, the template programmer is bound to make mistakes and want to debug his templates. Currently, the only debugging facility available is the `PRINT()` external function. There is no mechanism for stepping through the template program, setting breakpoints, and displaying values of control variables, propagation values, and bounds values. As a debugging aid, for every statement Catacomb generates, it also includes a comment that describes which line of which template was responsible for generating that line of C code. However, Catacomb's optimizations like copy propagation, dead code elimination, and code hoisting tend to make this information less useful than one might expect. Furthermore, this information does little to help debug control constructs, since no control constructs remain in the final code produced by Catacomb.

It would not be difficult to add support for a Catacomb debugger. The debugger would be integrated into Catacomb itself, so that each time Catacomb executes a control or code construct, it prompts the user for a debugging command if a breakpoint was set for that statement. However, Catacomb's single-phase execution model (see Section 4.6.7) presents an interesting problem. When processing a loop, Catacomb makes repeated passes over the loop body, until it reaches convergence. (It backtracks to the beginning of the loop and repeats execution if, at the end of the loop body, it discovers that more code variables were modified in the loop body than it anticipated.) If a breakpoint is set inside such a loop, the template programmer may become confused when Catacomb stops repeatedly at the same point. One possible solution is for Catacomb to disable all breakpoints when processing a loop, until it reaches convergence. After convergence, Catacomb can backtrack once more and execute the loop with breakpoints enabled, knowing that there will be no further need to backtrack to the beginning of this particular loop. While this method of execution presents a cleaner model to the template programmer, it brings up yet another problem. The programmer may be trying to debug an error in control execution; e.g., an infinite control loop, or a fatal error in which the condition of a `cif` or `cwhile` statement does not evaluate to a compile-time constant. If such an error occurs while Catacomb has temporarily disabled breakpoints, then the programmer will be unable to use the debugger to track down such a bug.

5.4 Summary

By carefully studying the details of the various array assignment algorithms, as well as the interaction between the algorithms and the various possible communication architectures, I developed a general framework for integrating an arbitrary array assignment algorithm with an arbitrary communication architecture. This overall framework has three components. The analysis and preprocessing component breaks the specific array assignment statement into simpler statements that are closer in form to the canonical example. The two main duties are to broadcast single distributed array elements and to separate multiple right-hand side terms. The architecture component is responsible for operations like allocating communication buffers and managing the specific communication details, such as scheduling the individual sends and receives. The algorithm component is responsible for the algorithm-specific details of communication set generation such as table generation, buffer packing, and buffer unpacking.

By combining this framework for the array assignment with a code composition system, Catacomb/Fx becomes the first system capable of compiling the general array assignment statement for several array assignment algorithms.

I implemented code templates for the CMU, OSU, and RIACS algorithms, and used them to evaluate some aspects of the runtime performance of Catacomb-generated code. I showed that the Syracuse approach to redistributing multidimensional arrays is flawed. The Syracuse approach advocates breaking a d-dimensional array redistribution into a sequence of one-dimensional redistributions, increasing the total work by a factor of d. This approach improves maintainability of a runtime library, but my experiments show that the direct approach to redistributing multidimensional arrays is more efficient. I also showed the performance benefits of using custom-generated code over library code. In addition, I showed that the use of the direct deposit model of communication leads to a significant performance improvement.

Finally, I discussed the generality and maintainability aspects of the Catacomb code templates for the array assignment. The most significant issue here is that the current implementation of Catacomb lacks a good infrastructure for debugging, and that Catacomb's single-phase execution model presents some difficulties in the design of the user interface for such a debugger.

Chapter 6

Other Code Composition Domains

In the previous chapters, I have used the HPF array assignment statement as a running example of a domain in which code composition benefits compilation. The purpose of this chapter is to show examples of other domains that can also benefit from code composition.

Recall from Chapter 3 the features of complex high-level operations that make them suitable for code composition:

- Executing them requires complex code, rather than just a few simple operations.
- There is a fairly wide variety of code sequences that can be executed at run time, dependent on the specific structure of the high-level construct and on other compile-time information.
- They have an implementation structure based on defining a collection of independent building blocks for a simple case, and piecing together the building blocks to handle the general case, in a manner determined by its structure.

I describe in this chapter several examples of compilers with constructs that match these criteria. Most are from the domain of parallel languages, whose implementations often have to operate on large aggregate objects, or move data from one processor to another. One example, however, comes from the rather unrelated field of relational database operations.

6.1 Data transfer in irregular applications

6.1.1 Background

Chapter 2 describes how a compiler can partially analyze the communication pattern of an array assignment statement, producing code that quickly enumerates the communication set at run time. The key issue that makes it possible for the compiler to perform this analysis is the amount of information available at compile time. First, the compiler knows that the reference pattern is a regular section, as specified by the subscript triplet. Second, the compiler knows that the ownership set of each array is a block-cyclic set (or, better yet, a simpler block or cyclic set, which can be described as a regular section). The subscript triplet forms a *regular* access pattern, and the block-cyclic set forms a *regular* data distribution, resulting in a *regular* communication pattern. Armed with this knowledge at compile time, the compiler is able to produce fast, straightforward code, even if it does not know the specific values of the parameters.

All this changes, though, when the communication pattern becomes *irregular*. This happens in one of two ways: either the access pattern is irregular, or the data distribution is irregular. The simplest way to express an irregular access pattern is through the use of an indirection array. For example, in the array assignment statement

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

the sequence of left-hand side indices of A is dependent on the runtime values of the indirection array IA. As such, it is no longer amenable to the same kind of compile-time analysis as the regular array assignment. Instead, runtime analysis techniques are 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. If the computation (e.g., the array assignment statement) is repeated several times, and the contents of the indirection array do not change, the results of the inspector phase can be saved and reused.

While it is clear that the executor phase involves a global communication pattern, the inspector phase generally involves a global communication pattern as well. This is because, due to its large size, the information that describes the data transfer is itself distributed across the processors, while the information is needed globally. By contrast, in a regular data transfer, the pattern is described by a small number of parameters, and thus the complete information can be replicated on all processors. Because the inspector phase includes a global communication pattern, considerable effort can be saved by reusing the inspector phase whenever possible.

Irregular computation is not limited to the above array assignment statement example. Other classes are the following:

- Indirection arrays can appear on both the left-hand side and the right-hand side, as in the example

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

  In addition, the indirect array references can be nested multiple levels, as in the example

  $$A[B[C[\ell_A\colon h_A\colon s_A]]] = D[\ell_D\colon h_D\colon s_D].$$

- The distributions of the arrays can be irregular, in which an auxiliary array defines the mapping of array elements to processors. (This auxiliary array can be treated similar to an indirection array in the analysis.) Note that irregular distributions are orthogonal to the use of indirection arrays in the array assignment statement.

- Although it is unclear whether such an extension is useful in practice, irregular array references may be multidimensional, as in

  $$A[B[\ell_B\colon h_B\colon s_B]][C[\ell_C\colon h_C\colon s_C]] = D[E[\ell_E\colon h_E\colon s_E]][F[\ell_F\colon h_F\colon s_F]].$$

- Instead of an array assignment, irregular computation can be expressed in terms of a more general loop construct, where each iteration is proven (or asserted) to be independent. In such a loop, the loop bounds can contain distributed array references, there can be nested loops, the array references can contain multiple levels of indirection, and array distributions can be irregular.

  Consider the code in Figure 6.1, which computes the sparse matrix-vector product $y = Ax$, using a compressed sparse row representation. Here we have a 2-deep loop nest, where the bounds of the inner loop depend on a distributed array. The loops are not perfectly nested, due to the initialization of $y(i)$. Also note that the reference $y(col(j))$ effectively has two levels of indirection, not one, because j depends on the value of another array.

```
DO i = 1, n
  y(i) = 0.0
  DO j = idx(i)+1, idx(i+1)
    y(i) = y(i) + A(j) * x(col(j))
  END DO
END DO
```

Figure 6.1: Code for computing a sparse matrix-vector product, using a compressed sparse row representation. The code contains a nested loop and multiple levels of array indirection.

To date, the most effective means of executing irregular computations is through the use of the PARTI or CHAOS runtime libraries [64, 48, 50], developed by Saltz et al. These libraries contain sophisticated routines that help the programmer or compiler translate a sequential program into a parallel program that uses the inspector/executor approach. In addition to inspector and executor routines, it contains routines that help the programmer or compiler to repartition the data arrays and the loop iterations in a way that improves load balance and minimizes the amount of communication required. CHAOS supports the resulting irregular distributions by creating and managing *translation tables*, which describe irregular distributions in terms of the processor and local memory mapping of each array element. Because the translation table is large (approximately the same size as the corresponding distributed array), the table itself needs to be distributed. CHAOS supports a *paged* translation table [22], meaning that the translation table itself has a block-cyclic distribution. CHAOS also allows partial replication of the translation table, similar to Fx's data distribution with overlapped alignment described in Section 5.1.3.

Although PARTI/CHAOS was designed to be used as the target of a parallelizing compiler [47], it took some time for compiler support to catch up with the library development. As such, the application programmer had to use the routines manually to transform a sequential program into a parallel version. Under this model, the programmer decides which loops should be parallelized and which arrays should be distributed, as well as what kinds of distributions should be used for the array elements and the loop iterations. After making these decisions, it is up to the programmer to convert each loop or loop nest into the inspector and the executor, making use of the routines in the CHAOS library.

More recently, several parallelizing compilers [74, 11, 45, 75] 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* [21]. 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. As each level of indirection is peeled away, the technique builds up a *slice graph*. The slice graph is used to identify redundant preprocessing steps, and a topological sort of the slice graph yields an ordering in which to compute the slices at run time. The resulting program, containing only single levels of indirect array references, is then amenable to parallelizing techniques in existing compilers for irregular problems.

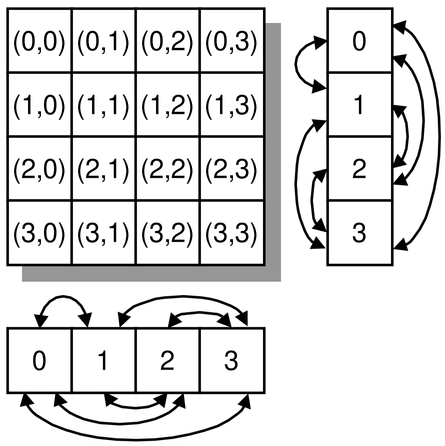


Figure 6.2: Communication for a two-dimensional irregular array assignment is accomplished by analyzing each dimension separately and composing the results.

6.1.2 Applicability of code composition

Generality in the irregular array assignment

Recall that the three principal benefits of code composition are efficiency, maintainability, and generality. Unfortunately, code composition is unlikely to provide a noticeable increase in efficiency over the traditional approaches, because most of the time is spent in the communication routines, and little can be done at compile time to speed up this communication. However, for the irregular array assignment statement, code composition offers the same *generality* benefits as the regular array assignment, and more. In addition to multidimensional arrays, multiple right-hand side terms, and scalar subscripts, code composition adds the capability of handling irregular data distributions and multiple levels of array indirection, without the maintainability problems of exponential code explosion inherent in a runtime library implementation.

The issue of multidimensional irregular communication deserves special attention here, since the issues are not discussed in the literature. An example of this kind of array assignment is

$$A[B[\ell_B\colon h_B\colon s_B]][C[\ell_C\colon h_C\colon s_C]] = D[E[\ell_E\colon h_E\colon s_E]][F[\ell_F\colon h_F\colon s_F]],$$

where all arrays are distributed (either regularly or irregularly). Another example is the canonical two-dimensional array assignment

$$A[\ell_1\colon h_1\colon s_1][\ell_2\colon h_2\colon s_2] = B[\ell_3\colon h_3\colon s_3][\ell_4\colon h_4\colon s_4],$$

where each array dimension of A and B independently has an irregular distribution.

As in the regular array assignment, each dimension is analyzed orthogonally, and the resulting communication pattern is formed by composing the individual communication patterns for each dimension independently. Consider the communication for a two-dimensional array assignment, as depicted in Figure 6.2. The boxes represent individual processors. If each dimension is distributed over 4 processors, we consider the processor numbering scheme to be a pair of values between (0,0) and (3,3), depending on the processor's position in the two-dimensional space. To determine the communication between processor (a, b) and processor (c, d), we compose the communication between a and c in the first dimension with the communication between b and d in the second dimension.

Notice that in a naive approach, more work may be performed in the inspector phase than is necessary. For example, in Figure 6.2, the communication pattern between processors (1,2) and (1,3) is identical to that of (2,2) and (2,3) in the second dimension. In the naive approach, this analysis would be duplicated by both sets of processors, which is a problem due to the high cost of the inspector phase. A better approach is to divide this work among all the processors that are performing the same analysis, using simple multicasts to deliver the results to the other processors. Determining the best methods for dividing the work and combining the results is a subject of future research.

Analysis of explicit loops

A composition system can be used as a simple compiler for an irregular parallel loop nest, converting a loop nest into a pair of inspector/executor loops containing the appropriate calls to the CHAOS routines. For the simple irregular loops, the translation is straightforward, and such a system is unlikely to gain additional efficiency, maintainability, or generality benefits over the traditional compiler approaches.

However, code composition is well-suited to the problem of analyzing loops with multiple levels of indirection. The technique of slicing analysis bears some similarity to the method described in Section 3.1.1 for compiling array assignment statements with multiple right-hand side terms. For such an array assignment, each right-hand side term is "peeled away" as a separate array assignment with a single right-hand side term, allowing all communication to be expressed in terms of the simpler statements. In addition, when we add multidimensional distributed arrays to the parallel loops, code composition provides the benefits of generality.

The compiler interface to a composition system is not as simple as for the array assignment statement, though. The array assignment statement is syntactically simple and limited in its expressibility, resulting in a well-defined set of operations for the compiler to perform, as well as a small amount of information to be passed to the composition system.

On the other hand, for a parallel loop, the analysis must be applied to an entire loop nest, which means that the compiler must pass a representation of the entire loop nest to the composition system. The composition system operates on the input loop nest and returns a new loop nest to the compiler. Passing a loop nest is clearly much more complex than passing a simple array assignment, because different kinds of loops have different semantics (e.g., C has three separate looping constructs). To make such a composition system portable to different compilers for different input languages, one needs to carefully design a "canonical loop" representation to describe a variety of looping constructs. This representation should contain the loop bounds, as well as a sequence of statements. A statement can be another (nested) loop, or any other kind of flat statement (e.g., a simple assignment statement) or hierarchical statement (e.g., a conditional), provided that all distributed array references are accessible to the composition system. Once the compiler interface is designed, the composition system can effectively perform the analysis, creating a new set of inspector/executor loop nests that it returns to the compiler.

6.2 Archimedes

The Quake project [7] at Carnegie Mellon focuses on predicting the ground motion during large earthquakes. At its heart is Archimedes [53], a system for executing unstructured finite element simulations on parallel computers. Figure 6.3 shows the structure of Archimedes.

An Archimedes problem consists of two inputs. One input is a geometric description of the problem domain, in terms of a 2- or 3-dimensional set of points, segments, and faces, describing boundary and interior features. The other input is a program written by an engineer in a high-level language that describes the numerical solution to a partial differential equation over the input problem domain.

The input problem domain is first given to Triangle [51] (a two-dimensional Delaunay triangulation mesh generator) or Pyramid [52] (a three-dimensional Delaunay-based mesh generator) to create a triangular

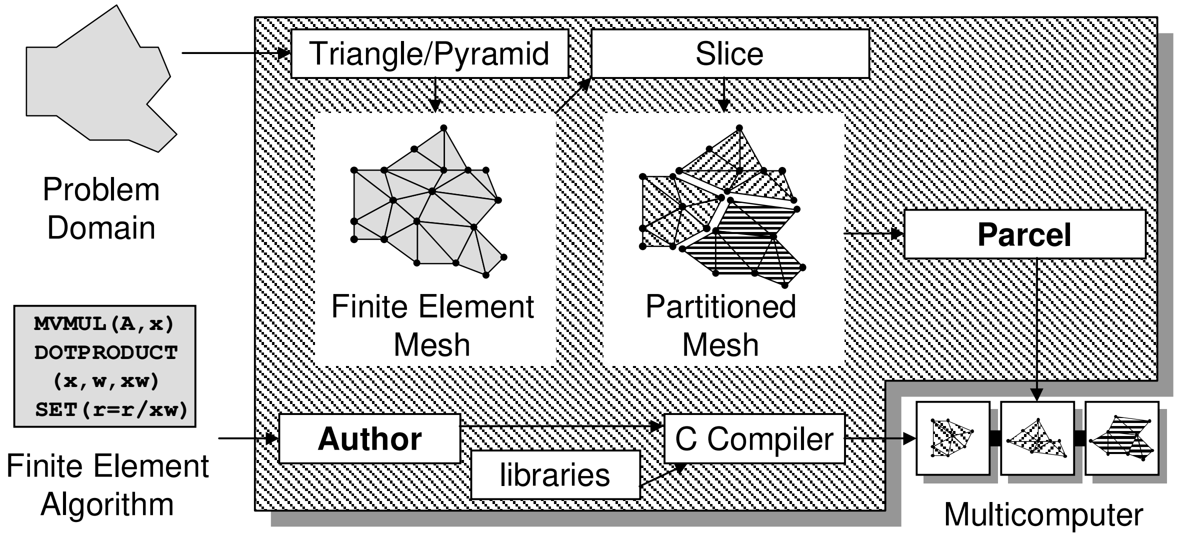


Figure 6.3: The Archimedes toolchain.

or tetrahedral finite element mesh. Next, a component called Slice partitions the mesh into one sub-mesh per processor, using a geometric partitioning algorithm [65]. Slice also tries to optimize placement and routing of the sub-meshes onto the processors. Finally, a tool called Parcel compiles the irregular communication patterns into the resulting executable.

In addition, there is a compiler component called Author, which presents the engineer/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, somewhat like code templates, that allows the programmer to extend the system with macro-like constructs that support type-checking of their arguments.

Code composition can provide several benefits in a system like Archimedes. First, it can provide a more general communication framework. Archimedes is currently limited to non-adaptive finite element problems, in which the finite element mesh remains the same throughout the computation. As such, the Parcel component compiles the communication patterns directly into the executable. This decision prevents the parallel solution of adaptive problems, because the data is fixed in its original distribution. Another consequence is that the arrays cannot be redistributed into other distributions (e.g., regular distributions) that may be needed by other algorithms and components of the solution (e.g., the I/O subsystem). To add this flexibility, we can use code composition to add a general-purpose irregular communication framework to Archimedes, as described in Section 6.1.

Another benefit that code composition can provide is in the compiler portion. The developers and users of Author have found it useful to have extensibility in the system in a macro-like fashion that bears similarity to code templates. For example, the programmer might write a `MVMUL` routine to compute a matrix-vector product.

The current template mechanism in Author requires a separate template to be written for every combination of input types. For example, different templates would need to be written for sparse and dense matrices. The problem is that there is a code explosion due to the number of independent axes in the problem: single versus double precision, symmetric versus asymmetric matrices, sparse versus dense matrices, the specific sparsity representation (e.g., compressed sparse row), and number of "degrees of freedom" in the problem. Trying to provide full generality thus results in a degradation in maintainability, since most of the axes are largely independent. Using a more general code template system with powerful compile-time control over the code generation, as described in this dissertation, allows a single, reasonably compact, abstract high-level template to be written for all matrix-vector multiplies, and to have it specialized at compile time to produce the correct version to be executed at run time.

6.3 Precompilation and query optimization in relational database systems

An example outside the domain of parallel languages in which code composition is beneficial is *precompilation* and *query optimization* in relational databases [37]. Database application programs are written in a *host language*, with embedded commands to the database manager written in a *data manipulation language* or *query language*. Clearly, the host language syntax is in general quite different from the query language syntax. Therefore, either the host language must be extended with additional query syntax, or the query syntax may appear in comments in the host language program. For example, the host language could be C, and the query language could be SQL, with SQL commands appearing as specially-annotated C comments.

A compiler for such a system has two tasks. First, like a standard compiler, it must translate the host language program into object code (or some other program at a lower level of abstraction). Second, it must compile the queries into calls to the *database manager*, handling the details of converting the input and output data between the representations used by the host language and the database manager.

To deal with the second task, the compilation system includes a precompilation phase. In this phase, a preprocessor converts the embedded data manipulation commands into the appropriate procedure calls in the host language. To perform this conversion, the precompiler must parse the embedded queries, generate the necessary procedure calls in the host language to execute the query at run time, and deal with the conversion of data structures between the host language and the database manager.

By itself, precompilation does not derive much benefit from the basic ideas of code composition. However, when precompilation and query optimization (see below) are combined, code composition becomes practical and attractive for this domain.

For nontrivial queries, there are generally a number of different ways in which the query can be processed. For example, algebraic manipulation of a query may lead to an equivalent query that requires less intermediate data (similar to the classic dynamic programming problem of optimal parenthesization of matrix multiplication [19, Chapter 16]). An important part of any database system is the query optimizer, which performs several functions to try to minimize the amount of work required to process the query.

There are many ways in which a query optimizer can reduce the amount of work. For example, it can recognize common subexpressions in a query, computing that part of the result only once and reusing it. In addition, it can use heuristics like "perform selection and projection operations as early as possible, favoring selection" to reduce the size of intermediate relations.

Other lower-level query optimizations are possible. For example, the obvious way to implement Cartesian product and join operations is through a loop nest of depth two, accessing the corresponding tuples of the relations in the inner loop. For relations that are too large to fit in main memory, this simple loop nest may result in more disk accesses than necessary, due to the fact that an entire disk block can be read and stored in memory at virtually the same cost as a single element. An optimized loop nest uses strip mining and loop tiling techniques to result in a deeper loop nest, but one that requires fewer overall disk accesses.

When we combine the concepts of precompilation and query optimization in a relational database system, we find that the techniques of code composition, as well as the software engineering issues of a composition system, are relevant. First, note that a query qualifies as a complex high-level operation: a query is syntactically simple and compact, yet there may be many possible ways to execute a given query, each of which results in a significant amount of code. Second, a composition system that is loosely coupled with a host language compiler can be reused in other host language compilers.

Let us consider how we might use a system like Catacomb to help with precompilation and query optimization in such a relational database system. To start with, consider the structure of the host language compiler. Under the standard approach (i.e., without the use of a composition system), the precompiler must extract and parse the queries from comments or directives in the input program. With the code composition approach, the compiler still needs to perform this task. However, instead of directly generating code to interface to the database manager, the compiler passes the query to the composition system.

The code templates are divided into two components. The first component deals with the issues of converting the input and output data between host language and database manager representations, as discussed above. This is a simple and straightforward task, and not representative of the power of code composition. This power is illustrated by the second component, which deals with the issues of query optimization and the low-level implementation of the query. The query optimization component applies the rules and heuristics to restructure the query into one that can be executed most efficiently, and it generates the low-level loops and other code to execute the query at run time.[1]

To conclude the discussion, I demonstrate a similarity between code composition in this domain and code composition for the array assignment statement. Consider the join or Cartesian product of 3 or more relations; e.g., $r \bowtie s \bowtie t$. One way to simplify this query is to use the associativity of the join operator to parenthesize the query as either $(r \bowtie s) \bowtie t$ or $r \bowtie (s \bowtie t)$. This query can now be processed as two binary joins. However, for the low-level looping structure, it is better to treat the query as a single operation, in terms of the loop tiling concerns. This means generating a 6-deep loop nest, or in general, a $2n$-deep loop nest for a join of n relations. Thus we have the problem of generating a loop nest whose depth is determined by the structure of the input, in much the same way that the depth of the loop nest for the array assignment statement is dependent on the dimensionality of the input array assignment. The same code generation techniques apply in both cases.

[1] Note that this simple treatment ignores the handling of issues like security and schema evolution.

Chapter 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 described in Section 4.6.5. A consequence that many C programmers may be familiar with is the inability to perform preprocessor operations like `#if sizeof(int)==4`. Because the preprocessor executes before the compiler, the preprocessor has no way of evaluating `sizeof` expressions.

PL/I [44] offers a more powerful preprocessor. However, it also is incapable of a single-phase execution model, and neither it nor the C preprocessor is equipped to perform structural queries on general expressions, a feature critical to code composition. The reason is that these macro processors use *lexical* macros, which process streams of tokens, rather than *syntactic* macros, which operate on complete syntax trees like expressions or statements.

The C++ template system provides a simple way to generate new functions and methods, tailored to a specific data type. Veldhuizen [69] 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 [46] and Programmable Syntax Macros [72]) that offer many of the same benefits as Catacomb. Safer_C allows the programmer to annotate code execution at one of 5 times: compile time, link time, load time, frame activation time, and (of course) run time. Programmable Syntax Macros are proposed as a replacement for the standard C preprocessor, using powerful syntax macros rather than the simple lexical macros. (However, the unusual syntax for specifying these macros makes it rather unlikely for this particular system to catch on.) 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.

At a higher level, Barrett et al. [8] use the concept of templates in a numerical computation context. In their system, templates are designed and written in a high-level language to handle specific features of iterative solvers for linear systems. For example, the programmer could write different libraries to solve sparse or dense systems, or to use different algorithms depending on convergence requirements, or to write sequential or parallel versions, or to handle different data layouts. At compile time, the template system automatically finds the right set of templates to match the needs of the user. The code is composed at the algorithm or procedure level, which is at a much higher level than the statement or expression level advocated in this dissertation. However, even at such a high level, this kind of specialized system fits well within the code composition framework I describe.

7.2 Partial evaluation

7.2.1 The relationship of code composition

Like most optimizing compilation systems, Catacomb and the concept of code composition are related to the field of *partial evaluation* [31, 18]. The goal of a system for partial evaluation is to attempt to perform as much of the computation as possible in advance (e.g., at compile time), yielding a more efficient program. In general, partial evaluation is performed when one or more of the input parameters is known in advance. However, partial evaluation techniques are also applicable for simplifying programs through, e.g., constant propagation and expression simplification, even when no input parameters are known. The result of partial evaluation is a *residual program* or *specialized program*.

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. An interesting extension of partial evaluation is the following. Suppose we have a partial evaluator, an interpreter for some language, and a program in that language. If we apply the partial evaluator to the interpreter and the program, the result can be considered a compiled version of the program. If we apply the partial evaluator to just the interpreter, the result is a compiler for the language. Furthermore, applying the partial evaluator to itself yields a compiler generator.

In a composition system, the code templates form a program in some input language. The input language consists of the combination of the control constructs and the code constructs. The composition system itself can be thought of as a partial evaluator applied to an interpreter for the control constructs (and the code constructs as well, if the composition system is equipped to analyze and perform optimizations on the code constructs). The composition system is thus a compiler (which is itself a partial evaluator) that takes as input a program (i.e., the code templates) and a known input (i.e., the input high-level construct for which code is to be generated), and produces a specialized program.

The first step in partial evaluation is to form a *division* of variables into *static* and *dynamic* variables. Static variables are ones whose values are known statically during the partial evaluation, and the values of dynamic variables are not known until execution time. Forming this division is called *binding time analysis*. Binding time analysis can also be applied to executable constructs like conditionals and assignments. In code composition, binding time is made explicit: all control constructs are static, and must be evaluated by the composition system. Binding times of code variables are dynamic by default, but through propagation and expression simplification, the composition system may be able to treat some code variables as static in certain regions of the program.

7.2.2 The relationship of Catacomb

Catacomb is an example of a system for partial evaluation of an imperative language. For the most part (see Section 7.2.3 below for an exception), the code templates form a two-level version of C, where the control constructs (e.g., control assignment, `cif`, and `cwhile`) are versions of their corresponding C constructs annotated for static binding time (i.e., compile-time execution). The template itself is a function annotated to be unfolded (i.e., inlined) statically.

The binding times of the code constructs are not necessarily dynamic. Catacomb's global optimization framework provides an *online* mechanism for making the binding times of some code variables static. The optimizations also allow conditionals and some loops to be marked as *eliminable*, when the conditions are static; e.g., if the condition of an `if` statement evaluates to a compile-time constant, the `if` statement can be evaluated, eliminating one of the branches. By contrast, binding times of control variables are trivially determined *offline*—all control variables are static. Because of this, Catacomb can be considered to perform *mixline partial evaluation*, because the binding time analysis is a mixture of offline and online analysis.

In an annotated two-level program, there is the issue of whether the annotations must be *consistent*; i.e., whether static functions are required to take static arguments. Catacomb does not require consistent annotations. For example, the `include` statement allows arbitrary expressions, not just static expressions, to be passed to templates. When the expression is dynamic, the expression is simply substituted wherever the corresponding argument appears in the template. In addition, there are external functions (e.g., `CONSTANT`) that can take dynamic expressions as input, even though all external functions are by definition static. In one sense, such external functions also serve to test whether the binding of an expression is static.

While performing partial evaluation, a variable may be found to have *bounded static variation*. This means that although the value of the variable is not statically known, it is restricted to a finite set of values. For example, after the statement

```
if (c) x=10; else x=20;
```

the value of `x` must be either 10 or 20. The partial evaluator may choose to specialize for some or all feasible values. Catacomb's bounds information (described in Section 4.5.2) provides a similar kind of information to the optimizer.

7.2.3 Differences in Catacomb and code composition

Many of the features in code composition and Catacomb are shared by other systems for partial evaluation, as described above. There is one key difference in the design of code composition, though. By design, the code templates do not actually form a program in the target language. The difference is that the control constructs follow a different flow of control from the code constructs. For example, in Catacomb, a control assignment statement that appears in the body of a loop is executed exactly once, regardless of the number of loop iterations at run time, and a control assignment that appears in a branch of an `if` statement is executed exactly once, regardless of whether that branch is taken at run time. Section 4.6.6 describes techniques for converting code templates into a form that is purely C, and thus amenable to optimization through data flow analysis, but the result is much less straightforward to analyze than the original templates.

In comparison to the field of partial evaluation, the mixture of control and code constructs in the code templates is similar to the concept of annotating the static portions of the input program. In contrast, though, in a standard partial evaluator, if these annotations are removed, the resulting program is semantically equivalent to the original program. As I discussed above, in code composition, a straightforward conversion of control constructs to code constructs, without the additional transformations described in Section 4.6.6, does *not* preserve the original semantics. 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

Recently, dynamic approaches to code generation have received renewed interest and attention. The static approach to code generation involves generating all possible code sequences at compile time, and selecting among them at run time. Dynamic approaches, on the other hand, augment the static approach by allowing code sequences to be generated on the fly at run time, and then executed.

A certain form of runtime code generation has been around since the days of Lisp, in the form of the *eval* construct. The eval construct allows the programmer to build a syntax tree (or a string representation of a syntax tree) dynamically at run time, and then to execute it. Eval-like constructs also exist in other languages, such as perl, tcl, and some implementations of ML. The eval construct is useful when, e.g., the types of arguments are dynamic and unknown at compile time. In this case, the correct code can be dynamically produced at run time when all information is available.

Eval is understandably absent from most compiler-based languages (e.g., C), due both to the problem of implementing an interpreter or compiler in the runtime system, and to the typically large performance difference between compiled and interpreted code. The tcc compiler for the ‘C language [43] takes on this challenge by augmenting the C language with the backquote and eval constructs, similar to Lisp. It deals with the performance problem by adding dynamic compilation to the runtime system, including such optimizations as partial evaluation and register allocation. Note that all code fragments that are dynamically compiled are explicitly annotated by the programmer.

A different approach is taken at the University of Washington [6]. Using a combination of automatic methods and programmer annotations, their system for runtime code generation dynamically creates and optimizes code based on runtime constants. The programmer annotates which regions should be dynamically compiled and which variables in the region are runtime constants, as well as certain optimizations (e.g., loop unrolling) that should be performed dynamically. The static compiler produces machine-code templates, with holes to be filled in at run time with runtime constants. For both this approach and the tcc compiler, it is critical to minimize the cost of the dynamic overhead so that a reasonable performance improvement can be obtained.

The principal similarity between the dynamic methods and code composition is that both approaches begin with an abstract specification of the problem to be solved, and then construct a lower-level solution based on the specific details of the input. For code composition, the abstract specification is the set of code templates, which are instantiated with a specific input to create a specialized static code sequence. For the dynamic methods, the annotated program itself is the abstract specification, and the dynamically-created code is the lower-level solution.

To contrast the two approaches, code composition is used at a higher level than the dynamic methods, in that it is used earlier in the compilation sequence. Whereas runtime code generation may require different modifications for each target architecture, code composition requires only minimal support from a compiler, and no additional runtime support.

Chapter 8

Concluding Remarks

8.1 Contributions

In this dissertation, I identified a class of complex, high-level constructs in programming languages, which cause problems for traditional compilation strategies. These constructs share some or all of the following features:

- Executing them requires complex code, rather than just a few simple operations.
- There is a fairly wide variety of code sequences that can be executed at run time, dependent on the specific structure of the high-level construct and on other compile-time information.
- There is a small number of "building block" code sequences that can be put together in an exponential number of combinations, depending on the specific structure of the construct, and yet the building blocks are largely independent of each other.

The traditional compilation strategies consist of custom code generation, in which the compiler produces a separate code sequence for each input construct, and the use of runtime library routines, in which each input construct is "compiled" into one or more calls to a fixed set of runtime library routines. These strategies are problematic for complex high-level operations because they trade off three key properties: efficiency, maintainability, and generality.

Efficiency refers to the performance of the generated code. Custom code generation offers the best efficiency because all compile-time information is compiled into the resulting code, whereas a fixed set of runtime library routines can only hope to optimize some of the input constructs.

Maintainability refers to the ease of writing, maintaining, and debugging the algorithm for executing the construct, all within the framework of the compilation system. Maintainability of a runtime library is usually straightforward, whereas it is easy to lose the structure of the algorithm when it is embedded in the compiler.

Generality refers to whether the implementation solves the fully general case, or just a simplified canonical case. For most complex high-level constructs, it is relatively easy for custom code generation to produce fully general code, whereas it is much more difficult to make a runtime library abstract enough to be fully general.

I used the High Performance Fortran array assignment statement to demonstrate in detail the tradeoffs between efficiency, maintainability, and generality when using the traditional compilation strategies. I then developed the concept of code composition, using code templates and a composition system, as a way to achieve efficiency, maintainability, and generality, without imposing tradeoffs. The code templates form a two-level programming language, with control constructs that are executed at compile time and code constructs that are executed at run time.

I developed a composition system called Catacomb for composing C code, in which the code constructs are straightforward C constructs and the control constructs resemble syntactic extensions to the C language. A global optimization framework enabled experimentation with nonstandard global optimizations that are based on bounds analysis. I also showed that the simplistic two-phase model of execution, in which the control constructs are executed separately from the global optimizations, misses certain kinds of "obvious" optimizations, and I showed that a single-phase execution model, while somewhat more difficult to implement, enables these optimizations without sacrificing clean semantics.

I evaluated Catacomb on the array assignment statement, in terms of the desirable properties of efficiency, maintainability, and generality. For generality, I developed a framework for the array assignment that allows an arbitrary algorithm to be coupled with an arbitrary communication architecture to form a complete solution; this is the only system currently available that allows this interchanging. Starting with array assignment code for a simple one-dimensional canonical example, this framework allows Catacomb to automatically generalize the code to the fully general case. For efficiency, I measured the performance of the array assignment under various degrees of compile-time knowledge, showing that even though most of the optimizations decrease overhead rather than per-element costs, the performance benefits are still significant. For maintainability, I showed that the amount of control constructs, which might be construed as "overhead" by some definition, is on the same order as the amount of code constructs. The resulting maintainability is quite good, especially compared to that of the 15,000 lines of code in the original Fx compiler.

Finally, I described how code composition can be used in domains other than the array assignment; in particular, irregular communication generation, the Archimedes system for the parallel solution of finite element problems, and precompilation and query optimization in relational database systems.

8.2 Criticisms of Catacomb

I did not develop the ideas of code composition in isolation. At the same time, I implemented the Catacomb composition system, and I used it extensively for the implementation and evaluation of the array assignment statement. The implementation of the array assignment is over 6,000 lines of template code, and it includes template code for the CMU, OSU, LSU, and RIACS algorithms, as well as the MPI, PVM, Intel NX, and Cray T3D direct deposit communication architectures.

I have found the implementation of Catacomb to be quite usable and robust, and I have the same confidence in it as any standard compiler that I use on a daily basis. However, in using Catacomb, I have learned a few lessons about language design and engineering:

- Default typing of code variables is a bad idea. In Catacomb, if a scalar code variable does not have an explicit type declaration, it defaults to an integer. This design decision results in subtle and hard-to-find bugs when the template programmer commits an oversight. While this decision is conceptually simple to fix, implementing the fix would then require sifting through several thousand lines of existing template code.

- The use of undeclared global control variables is also a bad idea. In Catacomb, global control variables are never explicitly declared. Instead, any symbol that appears as the left-hand side of a control assignment statement, and that is not already declared as a local control variable, automatically becomes a global control variable for the remainder of the Catacomb execution. If a code variable shares that name, then after the control assignment executes, that code variable will no longer be accessible (except through certain "tricks" that require knowledge of the underlying implementation).

To fix this problem, a global control variable should be explicitly declared within the template in which it is used, similar to the local control variables. Another possibility is to require a special syntax for global control variable references. For example, one could use `G(foo)` to refer to global variable `foo`, where `G()` is reserved syntax in Catacomb. As an added benefit, the clumsiness of such syntax may encourage the template programmer to minimize the use of global control variables.

- I made a poor decision when designing the framework for the array assignment statement. The analysis and preprocessing component (as described in Section 5.1.1) invokes the templates of the architecture component through a fixed set of template names. Similarly, the architecture component invokes the templates of the algorithm component through another fixed set of template names. When writing a new architecture component or algorithm component, the template writer must provide templates with these specific names, and must call templates in other components using these specific names.

  The result of this design is that, since all Catacomb templates are loaded at the beginning of program execution, it is impossible to use more than one architecture component and one algorithm component at a time. This is a problem because for an extremely high-quality implementation of the array assignment statement, it is necessary to be able to select between several algorithms, either at compile time or run time, depending on the specific parameters of the array assignment. Unfortunately, the current framework only allows a single algorithm component to be used.

8.3 Future directions

There are several directions one could take to extend the work of this dissertation.

- **Resolving the differences between code composition and partial evaluation.** Code composition assumes a two-level language model, in which the control constructs are executed at compile time, and the code constructs are emitted as code to be executed at run time. Thus if a control construct appears inside a loop that is a code construct, the control construct is executed exactly once at compile time, regardless of the number of loop iterations executed at run time. (Users of the C preprocessor and other macro processing systems are already aware of this kind of execution model.) This model is different from the standard model in the partial evaluation literature, which assumes a uniform set of program semantics.

  It is an open question whether the code composition model deserves special attention in the context of partial evaluation. It may be that any template that makes use of the code composition semantics can be easily transformed into an equivalent template that follows the partial evaluation model. Or there may be a convincing example of a template that cannot be cleanly transformed in this way.

- **Exploring the use of code composition for runtime code generation.** While code composition was designed to be used for compile-time code generation, the template mechanism, consisting of control constructs and code constructs, could be used to orchestrate the process of runtime code generation. The challenges would be to demonstrate a convincing need for such a system, in terms of efficiency, maintainability, and generality, and to design a lightweight composition system that executes at run time with minimal cost.

- **Implementing irregular communication generation.** I have demonstrated code composition to be extremely effective for implementing regular communication resulting from the array assignment statement. While code composition is also expected to be effective for the irregular array assignment, it remains to be seen how simple or difficult it is to design a framework for the treatment of general parallel loops with irregular communication requirements.

In addition, the issues of irregular communication for arrays distributed in multiple dimensions should be studied. For example, even though code composition can provide this generality, it is unclear what kind of applications would benefit from this kind of communication. For such communication, it is also important to explore the issues of further distributing the inspector phase over the set of processors.

Bibliography

[1] D. Adams. CRAY T3D system architecture overview. Technical report, Cray Research Inc., September 1993. Available as `http://www.cray.com/PUBLIC/product-info/mpp/T3D_Architecture_Over/T3D.overview.html`.

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

[3] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages*, pages 177–189. ACM, January 1983.

[4] American National Standards Institute. *Fortran 90*, May 1991. X3J3 internal document S8.118.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] R. Barriuso and Knies A. SHMEM user's guide for C. Technical report, Cray Research Inc., June 20 1994. Revision 2.1.

[10] G.E. Blelloch, S. Chatterjee, J.C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[11] 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.

[12] T. Brandes. ADAPTOR programmer's guide version 4.0. Available as `ftp://ftp.gmd.de/GMD/adaptor/docs/pguide.ps` via `http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`, April 1996.

[13] S. Chatterjee, J. Gilbert, F.J.E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 149–158, San Diego, California, May 1993.

[14] 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.

[15] S. Chatterjee, J. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28, Charleston, South Carolina, January 1993. ACM.

[16] S. Chatterjee, J.R. Gilbert, L. Oliker, R. Schreiber, and T. Sheffler. Algorithms for automatic alignment of arrays. *Journal of Parallel and Distributed Computing*, 38(2):145–157, November 1996.

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

[18] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM.

[19] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[20] Intel Corporation. *Paragon User's Guide*. Intel Corporation, Portland, Oregon, June 1994.

[21] 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.

[22] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, September 1994.

[23] P. Dinda and D. O'Hallaron. Fast message assembly using compact address relations. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS'96)*, pages 47–56, Philadelphia, Pennsylvania, May 1996.

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

[25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994. Available as `http://www.netlib.org/pvm3/book/pvm-book.html`.

[26] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[27] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report OSE-CISRC-4/94-TR19, The Ohio State University, April 1994.

[28] 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.

[29] S.K.S. Gupta, S.D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *Proceedings of the International Conference on Parallel Processing 1993*, pages 301–305, St. Charles, Illinois, August 1993.

[30] S. Hinrichs, C. Kosak, D. O'Hallaron, T. Stricker, and R. Take. An architecture for optimal all-to-all personalized communication. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 310–319, Cape May, New Jersey, June 1994.

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

[32] S. Kaushik. Implementation of the OSU algorithm, 1995. Code available from `ftp://ftp.cis.ohio-state.edu/pub/hpce/compiler/Source/arraysect/t3d/`.

[33] S.D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling array statements for efficient execution on distributed-memory machines: Two-level mappings. In *Proceeding of the Eighth Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 209–223, Columbus, Ohio, August 1995. Springer Verlag.

[34] 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.

[35] D.E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1981.

[36] C. Koelbel. *Compiling Programs for Nonshared Memory Machines*. PhD thesis, Purdue University, November 1990.

[37] H. Korth and A. Silberschatz. *Database System Concepts*. Advanced Computer Science Series. McGraw-Hill, 1986.

[38] T. MacDonald, D. Pase, and A. Meltzer. Addressing in Cray Research's MPP Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 161–172, Austrian Center for Parallel Computation, July 1992.

[39] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995. Available as `http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html`.

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

[41] William Morris, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, Massachusetts, 1982.

[42] D.M. Pase, T. MacDonald, and A. Meltzer. *MPP Fortran Programming Model*. Cray Research, Inc., Eagan, Minnesota, 1993.

[43] M. Poletto, D. Engler, and M.F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, page to appear, Las Vegas, New Mexico, June 1997. ACM.

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

[45] R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions on FORTRAN 90D/HPF compilers. Technical Report CS-TR-3268, Department of Computer Science, University of Maryland, May 1994.

[46] 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.

[47] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[48] 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.

[49] S.L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Cambridge, Massachusetts, October 1996.

[50] 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.

[51] J.R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In *First Workshop on Applied Computational Geometry*, pages 124–133, Philadelphia, Pennsylvania, May 1996. ACM.

[52] J.R. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.

[53] 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.

[54] 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.

[55] 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.

[56] T. Stricker. *Direct Deposit: A Communication Architecture for Parallel and Distributed Programs*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. To appear.

[57] T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Portofino, Italy, June 1995.

[58] T. Stricker and T. Gross. Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, San Antonio, Texas, February 1997.

[59] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proceedings of the Ninth International Conference on Supercomputing*, pages 1–10, Barcelona, Spain, July 1995. ACM.

[60] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda, and J. Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proceedings of Supercomputing '94*, pages 330–339, Washington, DC, November 1994.

[61] 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.

[62] J. Subhlok and G. Vondran. Optimal mapping of sequences of data parallel tasks. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 134–143, Santa Barbara, California, July 1995.

[63] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[64] 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.

[65] S.-H. Teng. *Points, Spheres, and Separators: A Unified Geometric Approach to Graph Partitioning*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1991.

[66] 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.

[67] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 191–208, Columbus, Ohio, August 1995. Springer Verlag.

[68] 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.

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

[70] C. Verbrugge, P. Co, and L. Hendren. Generalized constant propagation: A study in C. In Tibor Gyimóthy, editor, *Proceeding of the Sixth International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 74–90, Linköping, Sweden, April 1996. Springer Verlag.

[71] L. Wang, J. Stichnoth, and S. Chatterjee. Runtime performance of parallel array assignment: An empirical study. In *Proceedings of Supercomputing '96*, Pittsburgh, Pennsylvania, November 1996.

[72] 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.

[73] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

[74] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–754, June 1995.

[75] 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.

Appendix A

External Functions in Catacomb

This appendix lists and describes the external functions available in Catacomb. Section A.1 covers Catacomb's core set of external functions, and Section A.2 covers the additional set provided by the distributed array assignment module in Catacomb/Fx. See Section 4.4.9 for a description of what a Catacomb external function is.

A.1 Core set of external functions

The following external functions are provided in the core implementation of Catacomb.

A.1.1 Numerical functions

- `max`(a, b): Returns the maximum of the two inputs.
- `min`(a, b): Returns the minimum of the two inputs.
- `pmod`(a, b): Returns the "positive modulo"; i.e., the positive remainder of a/b. Both a and b must be integers, and b must be positive. The result is between 0 and $b - 1$, inclusive.
- `fldiv`(a, b): Returns $\lfloor a/b \rfloor$ (i.e., "floor div"), where a and b are integers.
- `ceildiv`(a, b): Returns $\lceil a/b \rceil$ (i.e., "ceiling div"), where a and b are integers.
- `FLOOR`(x): Returns $\lfloor x \rfloor$.
- `CEIL`(x): Returns $\lceil x \rceil$.

Note that using the external functions `FLOOR` and `CEIL` is preferable to using the analogous `floor` and `ceil` functions in the C math library, because Catacomb will try to use compile-time information to simplify the result.

A.1.2 Structure queries

The following functions query the structure of the input

- `EQUIV`(e_1, e_2): Returns 1 if the two input expressions are structurally equivalent, 0 otherwise. This is similar to the `equal` function in Lisp.

- CONSTANT(c): Returns 1 if the input expression is a constant, 0 otherwise.
- IS_ARRAYREF(c): Returns 1 if the input expression is an array reference, 0 otherwise.
- IS_EXPR(c): Returns 1 if the input expression is a unary or binary expression, 0 otherwise.
- IS_TERNARY(c): Returns 1 if the input expression is a ternary expression, 0 otherwise.
- IS_SLICE(c): Returns 1 if the input expression is a "slice" (i.e., a subscript triplet), 0 otherwise.
- IS_SYM(c): Returns 1 if the input expression is a "symbol" (i.e., a variable), 0 otherwise.
- NDIM(a): Returns the number of array dimensions in the array or array reference a. The input expression may only be a symbol or an array reference. It returns 0 if a is a scalar variable.
- RANK(e): Returns the "rank" of an expression, which essentially describes the number of subscripts in an array reference. Formally, the rank of an expression is defined as follows:

  > The rank of a variable or constant is 0. The rank of an array reference is the number of dimensions that contain a subscript triplet. The rank of a subscript triplet is 1. To compute the rank of an expression, we compare the ranks of both terms. If the ranks are equal, the result is that rank. If one rank is 0, the result is the other rank. If the ranks are unequal and nonzero, it is an error. If an array subscript rank is not 0 or 1, it is an error.

  Upon an error, -1 is returned.
- SAME_TYPE(e_1, e_2): Returns 1 if the two expressions have the same type, 0 otherwise. For structs and unions, it checks whether the actual fields are the same, and not just whether the struct or union names are the same.
- TYPENAME(e): Returns a string constant that identifies the type of the input expression. The string is typically used to construct the same of a special function in the runtime support library that is specialized to arguments of that type.

A.1.3 Structure dissection

Note: For the functions that take an array dimension number as input, the dimension numbering is 0-based, left-to-right.

- GET_ARRAY(a): Given an array reference a, return just the array portion (stripping off the subscripts).
- GET_SUB(a, d): Given an array reference a and a dimension number d, returns the subscript for that dimension.
- GET_DIMSIZE(a, d): Given an array or array reference a and a dimension number d, returns the declared dimension size for the given dimension.
- GET_LEFT(e): Given a binary expression, return the left subexpression.
- GET_RIGHT(e): Given a binary expression, return the right subexpression.
- GET_OP(e): Given a binary expression, return an integer denoting the operator. This value is useful only when used in the EXPR external function.

- SLICE_FIRST(s): Given a subscript triplet s, return the “first” or “lower bound” value (i.e., the first of the three components).

- SLICE_LAST(s): Given a subscript triplet s, return the “last or “upper bound” value (i.e., the second of the three components).

- SLICE_STRIDE(s): Given a subscript triplet s, return the “stride” value (i.e., the third of the three components).

- GET_FIRST(a, d): Given an array reference a and a dimension number d, returns the “first” component of the slice in the subscript of that dimension. If that component is missing, then the default value of 0 is returned.

- GET_LAST(a, d): Given an array reference a and a dimension number d, returns the “last” component of the slice in the subscript of that dimension. If that component is missing, then the default value, one less than that array dimension’s size, is returned.

- GET_STRIDE(a, d): Given an array reference a and a dimension number d, returns the “stride” component of the slice in the subscript of that dimension. If that component is missing, then the default value of 1 is returned.

A.1.4 Expression construction

These external functions are used to create expressions that cannot be expressed syntactically in the template files.

- EXPR(op, e_1, e_2): Creates an expression from the given operator and subexpressions. The operator is an integer, returned from GET_OP.

- SLICE(ℓ, h, s): Creates a subscript triplet with the given three components. This function is needed because there is no Catacomb template syntax for specifying a triplet.

A.1.5 List manipulation

- MAKE_LIST($size, init$): Creates a list of length $size$, with each element initialized to the value $init$.

- GET_LIST_ITEM(l, n): Returns item n from list l. Numbering of list elements is 0-based.

- REPLACE_LIST_ITEM(l, n, val): Destructively replaces item n of list l with val, returning the list. Numbering of list elements is 0-based.

- COPY_LIST(l): Returns a copy of the list. If the input l is an array reference, it returns a list of the array subscripts.

A.1.6 Index permutations

Index permutation is a concept related to the array assignment statement, used for executing transpose-like operations (see Section 3.1). For the purposes of such transposes, Catacomb allows array references to be tagged with an index permutation. The default, naturally, is the identity permutation. The following external functions allow manipulation of the index permutation.

- APPLY_IDXPERM(a, p): Creates a new array reference, by taking a copy of array reference a and tagging it with the index permutation from array reference p.
- GET_DIM(a, s): Given an array reference a and a slice number s, returns the dimension number associated with the slice, taking the index permutation into account. Note that scalar subscripts in the array reference are ignored when determining the result.

A.1.7 Miscellaneous

- PRINT$(\ldots)$: Prints its arguments to standard output, followed by a newline. Used for diagnostic purposes.
- EXIT$()$: Terminates execution of Catacomb and the compiler to which it is linked by calling the `exit` function.
- SET_LBOUND(v, c): Manually sets the lower bound of the "bounds" information (see Section 4.5) of code variable v to the constant c.
- SET_UBOUND(v, c): Manually sets the upper bound of the "bounds" information of code variable v to the constant c.

A.2 External functions for the distributed array assignment in Catacomb/Fx

The following external functions are provided in Catacomb/Fx, the distributed array assignment add-on to Catacomb.

A.2.1 Distribution information

- IS_DIST(a): Takes an array or array reference as input, and returns whether the array is distributed.
- IS_DIMDIST(a, d): Takes an array or array reference as input, and returns whether dimension d of the array is distributed.
- GET_BSIZE(a, d): Takes an array or array reference as input, and returns the declared block size of dimension d of the array.
- GET_NUMPROC(a, d): Takes an array or array reference as input, and returns the number of processors over which dimension d of the array is distributed.
- GET_DISTTYPE(a, d): Takes an array or array reference as input, and returns the distribution type of dimension d of the array (0 if it is known to be block, 1 if it is known to be cyclic, and 2 otherwise to denote the general block-cyclic).
- GET_OLEFT(a, d): Takes an array or array reference as input, and returns the "left overlap" parameter of the alignment for the array in dimension d.
- GET_ORIGHT(a, d): Takes an array or array reference as input, and returns the "right overlap" parameter of the alignment for the array in dimension d.
- GET_AOFFSET(a, d): Takes an array or array reference as input, and returns the alignment offset for the array in dimension d.

- GET_ARRAY_BLKSIZE(a, d): Takes an array or array reference as input, and returns the "true" block size of the distribution of the array in dimension d. The "true" block size is the declared block size with the alignment overlap added in.
- GET_ARRAY_STRIDE(a, d): Takes an array or array reference as input, and returns the "stride" of the distribution of the array in dimension d. The distribution stride is defined as the declared block size times the number of processors of the distribution.
- GET_TEMPLATE_NDIM(a): Takes an array or array reference as input, and returns number of dimensions in the template to which the array is aligned.
- GET_ALIGNDESC_PERM(a, d): Takes an array or array reference as input, and returns the array index that is aligned with template dimensions d. A -1 return value indicates alignment with a constant template element (e.g., "align A[i] with T[3]").
- GET_DESC(a): Takes an array or array reference as input, and returns the array used to hold the runtime distribution descriptor.
- COPY_DISTINFO(new, old): Assigns array new the distributions information belonging to old. This is typically used in conjunction with creating distributed temporary arrays.

A.2.2 Access to compiler flags

The following external functions test flags in the compilation environment, all but NUMPROC returning either 0 or 1.

- DOCOMM(): Whether to perform any actual communication. If not set, the templates should only pack and unpack the communication buffers, but not actually do any communication or synchronization.
- DEPOSIT(): Whether the deposit model of communication is desired (i.e., computes the destination addresses on the receiver's behalf).
- COMM_BLOCK(): If the deposit model of communication is used, this option specifies whether to send compressed address-data blocks rather than the default address-data pairs.
- NUMPROC(): Returns the number of processors on which the program will be executing.