Data Mapping in Agglomeration of Virtual Processors*

Nai-Wei Lin

Department of Computer Science and Information Engineering
National Chung Cheng University
MinHsiung, Chiayi 621, Taiwan, ROC
naiwei@cs.ccu.edu.tw

Abstract

High-level data parallel programming languages usually support virtual processors to relieve the programmers from the size limitation of the underlying machine. For such programming languages, more efficient target code can be generated if their compiler can agglomerate virtual processors to fit the number of physical processors on the underlying machine. This paper describes a source-level program transformation approach to agglomerating virtual processors. This approach allows us to separate the module for the agglomeration of virtual processors and the module for the final code generation. This separation can greatly improve the retargetability and maintainability of the compiler.

1 Introduction

High-level data parallel programming languages usually support virtual processors to relieve the programmers from the size limitation of the underlying machine. For such programming languages, more efficient target code can be generated if their compiler can agglomerate virtual processors to fit the number of physical processors on the underlying machine. This agglomeration can significantly reduce the overhead incurred in task switching and task communication.

This paper will describe a source-level program transformation approach to agglomerating virtual processors. This approach allows us to separate the module for the agglomeration of virtual processors and the module for the final code generation. This separation can greatly improve the retargetability and maintainability of the compiler.

This paper will describe this approach based on the data parallel programming language CCC [1]; however, this approach is applicable to other data parallel programming languages as well. The agglomeration of virtual processors includes the replication of the data allocated and the code executed on the virtual processors. The description about the replication of the code is given in an earlier paper [1]. This paper will focus on the description about the replication of the data.

The remainder of this paper is organized as follows. Section 2 introduces the language CCC and its salient features. Section 3 gives the structure of our compiler for CCC. Section 4 describes a distribution model used in our analysis. Section 5 illustrates the distribution functions supported in the CCC. Section 6 elaborates the program transformations used to achieve the agglomeration of virtual processors. Finally, Section 7 concludes this paper.

2 The Parallel Programming Language CCC

The programming language CCC is an extension of the programming language C. CCC supports the ANSI C for sequential programming and a number of new features for data parallel programming. CCC programs are executed on a collection of virtual processors. These processors are composed of a front end processor and an array of back end processors. A CCC program may consist of both sequential and data parallel computation. The front end processor executes sequential computation, while the back end processors execute data parallel computation. A CCC programmer can use back end processor arrays of arbitrary dimension and size in his programs. This feature frees the programmer from dealing with the limitation on the size of the physical processors.

Data parallel computation of a CCC program is delegated from the front end processor to the back end processors and is *synchronously* executed on the back

*This work was supported in part by the National Science Council of ROC under grant number NSC-87-2213-E-194-006.

end processors; that is, every back end processor executes the same operation at the same time. Hence, the semantics of a CCC program is *deterministic*. This feature makes CCC programs easier to code, understand, and maintain than programs written in asynchronous data parallel programming languages. CCC also supports global name space for data parallel computation. Each back end processor can directly access data structures distributed on other back end processors via variable references. This feature allows users avoid using low-level message passing primitives to access distributed data structures, thereby focusing on the design of high-level parallel algorithms.

Environments for performing data parallel computation are specified via *domain* declarations. A domain declaration specifies the dimension and size of an array of back end processors, the data structures distributed on each processor, and the parallel operations executed on each processor. As an example, the following domain declaration declares an environment for performing matrix multiplication in parallel.

```
#define  DIM  16

domain par_matrix_op[DIM] {
  int a[DIM], b[DIM], c[DIM];
  multiply(distribute in int [DIM:block][DIM],
         distribute in int [DIM][DIM:block],
         distribute out int [DIM:block][DIM]);
};

par_matrix_op::multiply(A, B, C) {
distribute in int [DIM:block][DIM] A;
distribute in int [DIM][DIM:block] B;
distribute out int [DIM:block][DIM] C;
{
  int i, j, t[DIM];

  a := A;
  b := B;
  for (i = 0; i < DIM; i++) {
    c[i] = 0;
    t := par_matrix_op[i].b;
    for (j = 0; j < DIM; j++) {
      c[i] += a[j] * t[j];
    }
  }
  C := c;
}
```

This example declares a one-dimensional domain, consisting of 16 back end processors, called `par_matrix_op`. There are three integer arrays `a[16]`, `b[16]`, and `c[16]` on each processor. A parallel function `multiply` is defined to perform matrix multiplication on the domain. The keywords `in`, `out`, and `inout` are used to specify the direction of argument data movements between the front end and beck end processors when parallel functions are entered and exited. The keyword `distribute` in an argument declaration is used to specify whether the corresponding argument is distributed to (collected from) back end processors at the entry (exit) point of an invocation. The keyword `block` specifies a data distribution function. Currently, CCC supports four commonly used data distribution functions `compress`, `block`, `cyclic`, and `block-cyclic`. A domain name and variables declared in the domain are visible in the parallel functions declared in the domain. Hence, in the function `multiply` we can use `par_matrix_op[i].b[j]` to *directly* access the variable `b[j]` on the processor `par_matrix_op[i]`.

Domain instances can be created by the declaration of domain variables. The parallel functions for a domain can be invoked via the domain variables. For example, the following program performs a matrix multiplication in parallel by calling the function `multiply` defined in the domain `par_matrix_op`.

```
#define  DIM  16

main()
{
  int A[DIM][DIM],B[DIM][DIM],C[DIM][DIM];
  domain par_matrix_op m;

  read_array(A);
  read_array(B);
  m.multiply(A, B, C);
  print_array(C);
}
```

A domain variable `m`, an instance of domain `par_matrix_op`, is first created. Variable `m` can then be used to invoke the parallel function `multiply`.

3 A Compiler for CCC

Our compiler for CCC currently consists of three parts: the front end, the program transformer, and the code generator. The structure of the compiler is shown in Figure 1. The front end performs the lexical, syntax, and semantic analyses. It reads a CCC program and generates an abstract syntax tree as an internal representation of the program. The program

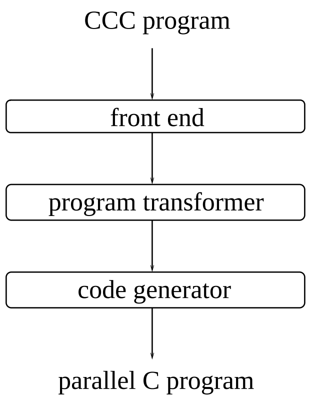

Figure 1: Structure of the CCC compiler

transformer performs the agglomeration of virtual processors by transforming the abstract syntax tree so that the number of virtual processors required by each domain can fit the number of physical processors on the underlying machine. Hence, we may need to agglomerate a number of virtual processors into one virtual processor. This can be achieved by appropriately mapping the data and replicating the code for each parallel function defined in the domains of the program. The code generator generates a parallel C program for the underlying machine from the abstract syntax tree. The parallel C program will include instructions about how programs on the underlying machine are loaded and terminated, how programs on the processors are synchronized, and how data are communicated between processors. Both the program transformer and the code generator will use the information provided by data distribution functions.

4 The Distribution Model

Data parallel computation requires the distribution of large data structures onto virtual processors and the distribution of virtual processors onto physical processors. These two distribution tasks share similar properties and can be discussed in the same fashion. This section describes a model for distributing points in a space onto points in another space. We will call the first space the *source space* and the second the *target space*. It is often that more than one point in the source space may be distributed onto a point in the target space. To be able to accommodate multiple points, each point in the target space would itself be a space, called an *offset space*. A *distribution* is a function that maps each point s in the source space into a pair $\langle t, o\rangle$, where t is a point in the target space and o is a point in the offset space.

Example 4.1 Suppose we would distribute the points in a one-dimensional source space S of 64 points onto the points in a one-dimensional target space T of 8 points. If the points in S are cyclically distributed onto the points in T as contiguous blocks of size 4, then each point in T will be distributed 8 points from S. Hence we will regard each point in T as an offset space O of 8 points to accommodate the 8 points from S. In this distribution, the points in S with coordinate 0 to 3 will be distributed onto the point with coordinate 0 in T, and these points will also be distributed, respectively, onto the points with coordinate 0 to 3 in O corresponding to the point with coordinate 0 in T. The first 32 points in S will be distributed onto T and corresponding O in the similar way. The distribution of these 32 points consists of the first cycle of this distribution. The second cycle of this distribution will distribute the last 32 points onto the points in T and the points with coordinate 4 to 7 in each corresponding O. $\square$

In this paper we will assume that the three types of spaces in a distribution have the same number of dimensions, though the size in each dimension may be different. We will also assume that the offset spaces in a distribution have the same size; in other words, the offset spaces are regarded as the same. In fact, the size of the smallest space that can contain any of these offset spaces will be taken as the size of *the* offset space. It will become clear in the following discussion that these assumptions will not restrict the applicability of the model in any way, while they can simplify the discussion significantly.

Definition 4.1 Let S, T, and O be respectively the source, target, and offset spaces. A *target function* Θ is a function

$$\Theta : S \rightarrow T$$

such that for each point $s \in S$,

$$\Theta(s) = t,$$

for some $t \in T$; An *offset function* Ω is a function

$$\Omega : S \rightarrow T$$

such that for each point $s \in S$,

$$\Omega(s) = o,$$

for some $o \in O$; A *distribution function* Δ is a function

$$\Delta : S \rightarrow T \times O$$

such that for each point $s \in S$,

$$\Delta(s) = \langle \Theta(s), \Omega(s) \rangle = \langle t, o \rangle,$$

for some $t \in T$ and $o \in O$. ∎

In Definition 4.1, if the spaces S, D, and O are n-dimensional, the functions Δ, Θ, and Ω can be refined as n-tuples of functions. The ith function in an n-tuple is the function for computing the coordinate in the ith dimension. More specifically, we can express the function Θ as

$$\begin{aligned}
& \Theta(\langle s_0, s_1, \ldots, s_{n-1} \rangle) \\
= \; & \Theta(s) \\
= \; & \langle \theta_0, \theta_1, \ldots, \theta_{n-1} \rangle (s) \\
= \; & \langle \theta_0(s), \theta_1(s), \ldots, \theta_{n-1}(s) \rangle \\
= \; & \langle t_0, t_1, \ldots, t_{n-1} \rangle \\
= \; & t,
\end{aligned}$$

where $\theta_i : S \rightarrow T$ gives the coordinate in the ith dimension of t. Similarly, we can express the function Ω as

$$\Omega(s) = \langle \omega_0(s), \omega_1(s), \ldots, \omega_{n-1}(s) \rangle,$$

where $\omega_i : S \rightarrow O$ gives the coordinate in the ith dimension of o. The function Δ can be expressed as

$$\begin{aligned}
& \Delta(s) \\
= \; & \langle \Theta(s), \Omega(s) \rangle \\
= \; & \langle \langle \theta_0(s), \theta_1(s), \ldots, \theta_{n-1}(s) \rangle, \\
& \; \langle \omega_0(s), \omega_1(s), \ldots, \omega_{n-1}(s) \rangle \rangle
\end{aligned}$$

For the convenience of expressing each dimension separately, the expression above can be rewritten as

$$\begin{aligned}
& \langle \langle \theta_0(s), \theta_1(s), \ldots, \theta_{n-1}(s) \rangle, \\
& \; \langle \omega_0(s), \omega_1(s), \ldots, \omega_{n-1}(s) \rangle \rangle \\
= \; & \langle \langle \theta_0(s), \omega_0(s) \rangle, \langle \theta_1(s), \omega_1(s) \rangle, \ldots, \\
& \; \langle \theta_{n-1}(s), \omega_{n-1}(s) \rangle \rangle \\
= \; & \langle \delta_0(s), \delta_1(s), \ldots, \delta_{n-1}(s) \rangle \\
= \; & \langle \delta_0, \delta_1, \ldots, \delta_{n-1} \rangle (s),
\end{aligned}$$

where $\delta_i : S \rightarrow T \times O$ gives the coordinates in the ith dimension of t and o as a pair. According to this model, to completely specify a distribution from a space onto another space, we need to specify a target function and an offset function for each dimension of the spaces.

Example 4.2 The distribution in Example 4.1 can be expressed as $\Delta(s) = \langle \Theta(s), \Omega(s) \rangle$, where

$$\Theta(s) = s \; \texttt{div} \; 4 \; \texttt{mod} \; 8,$$

and

$$\Omega(s) = s \; \texttt{div} \; (4 \times 8) \times 4 + s \; \texttt{mod} \; 4,$$

2 is the size of a block and 8 is the size of the target space. Hence, the point with coordinate 57 in S will be distributed onto the point with coordinate

$$\Theta(57) = 57 \; \texttt{div} \; 4 \; \texttt{mod} \; 8 = 6$$

in T, and the point with coordinate

$$\Omega(57) = 57 \; \texttt{div} \; (4 \times 8) \times 4 + 57 \; \texttt{mod} \; 4 = 5$$

in O; namely, $\Delta(57) = \langle \Theta(57), \Omega(57) \rangle = \langle 6, 5 \rangle$. □

5 Distribution Functions

In CCC programs, data distribution functions are specified in the declarations for the arguments of parallel functions. Currently, CCC only supports four commonly used functions for data distribution: `compress`, `block`, `cyclic`, and `block-cyclic`(b). For one-dimensional spaces, the class of `block-cyclic`(b) functions can be defined as $\Delta(s) = \langle \Theta(s), \Omega(s) \rangle$, where

$$\Theta(s) = s \; \texttt{div} \; b \; \texttt{mod} \; p, \qquad (1)$$

and

$$\Omega(s) = s \; \texttt{div} \; (b \times p) \times b + s \; \texttt{mod} \; b, \qquad (2)$$

b is the size of a block, and p is the size of the target space or the number of virtual processors. The distribution in Example 4.2 is a distribution with the function `block-cyclic`(4).

The first three functions are special cases of the class of `block-cyclic` functions. The `compress` function is a `block-cyclic` function with block size b equal to the number of processors p. The `block` function is a `block-cyclic` function with block size b equal to $\lceil n/p \rceil$, where n is the size of the source space. The `cyclic` function is a `block-cyclic` function with block size b equal to 1. Therefore, in the rest of the paper, we will only consider the class of `block-cyclic` functions, and Δ, Θ, and Ω will be used to mention this class of functions. Since these functions distribute the points in one dimension of the source space to the corresponding dimension in the target and offset spaces, we will only consider one-dimensional spaces from now on.

To discuss the overall effects of both the distribution from data structures to virtual processors and the distribution from virtual processors to physical processors, we need to introduce two notions. The first notion is the *multiplication* of one space with another space.

Definition 5.1 Let S and T be two one-dimensional spaces. The multiplication of S with T, denoted by $S \otimes T$, can be characterized by the function $\mu : S \times T \rightarrow S \otimes T$ defined as

$$\mu(s,t) = s \times |T| + t, \tag{3}$$

for $s \in S$ and $t \in T$, where $|S \otimes T| = |S| \times |T|$. ∎

Notice that the operator $\otimes$ is not commutative. The second notion is the *cascade* of two distribution functions.

Definition 5.2 Let $\Delta_1 : S \rightarrow U \times V$ defined as $\Delta_1(s) = \langle \Theta_1(s), \Omega_1(s) \rangle$, for every $s \in S$, and $\Delta_2 : U \rightarrow T \times W$ defined as $\Delta_2(u) = \langle \Theta_2(u), \Omega_2(u) \rangle$, for every $u \in U$, be two distribution functions, and let $O = W \otimes U$. The cascade of Δ_1 and Δ_2 is the function $\Delta_1 \diamond \Delta_2 : S \rightarrow T \times O$ defined as

$$\begin{aligned} & \Delta_1 \diamond \Delta_2(s) \\ = \;& \langle \Theta_2(\Theta_1(s)),\ \mu(\Omega_2(\Theta_1(s)),\ \Omega_1(s)) \rangle \\ = \;& \langle t, o \rangle, \end{aligned} \tag{4}$$

for every $s \in S$, and some $t \in T$ and $o \in O$. ∎

6 Program Transformation

Based on the distribution model introduced in previous sections, this section describes the transformation from a program that is independent of the number of physical processors into a program that has the same number of virtual processors as the physical processors. In other words, in the cases where the number of virtual processors is larger than the number of physical processors, we need to agglomerate virtual processors so that the number of resultant virtual processors equals to the number of physical processors. In this paper, we will only discuss the transformation about data mapping; namely, how data are mapped between the original virtual processors and the resultant virtual processors. The transformation about code replication is discussed in another paper.

The objective of this transformation is to use a distribution function to capture the overall effects of the distribution from data structures to virtual processors and the distribution from virtual processors to physical processors. Let these two distributions be expressed as `block-cyclic`(b_1) and `block-cyclic`(b_2). Then the distribution function that can correctly capture the overall effects of these two functions is `block-cyclic`$(b_1) \diamond$ `block-cyclic`(b_2). Unfortunately, the cascade of two `block-cyclic` functions is not necessary a `block-cyclic` function. However, to make this program transformation possible, we need to find a `block-cyclic` function that is very close to the function `block-cyclic`$(b_1) \diamond$ `block-cyclic`(b_2). The function we use is `block-cyclic`$(b_1 \times b_2)$. The reason for choosing this function is due to the following property. We will use the notation $a \mid b$ to denote that integer a *divides* integer b.

Theorem 6.1 *Let* `block-cyclic`*(b_1) : $S \rightarrow U \times V$ and* `block-cyclic`*(b_2) : $U \rightarrow T \times W$ be two distribution functions, and $O = W \otimes V$. If $|U| \mid (|T| \times b_2)$, and for each $s \in S$,* `block-cyclic`*$(b_1) \diamond$* `block-cyclic`*(b_2) $(s) = \langle t_\diamond, o_\diamond \rangle$, for some $t_\diamond \in T$ and $o_\diamond \in O$, and* `block-cyclic`*$(b_1 \times b_2)$ $(s) = \langle t_\times, o_\times \rangle$, for some $t_\times \in T$ and $o_\times \in O$, then $t_\diamond = t_\times$.*

Proof We first consider the function `block-cyclic`$(b_1) \diamond$ `block-cyclic`(b_2) by referring to Equation (4) and (1). For each $s \in S$, let

$$\begin{aligned} s &= m_1 \times b_1 + r_1, \\ m_1 &= m_2 \times |U| + r_2, \\ r_2 &= m_3 \times b_2 + r_3, \\ m_3 &= m_4 \times |T| + r_4. \end{aligned}$$

Then these equations yield

$$\begin{aligned} s &= m_2 \times |U| \times b_1 + r_2 \times b_1 + r_1, \\ t_1 &= s \texttt{ div } b_1 \texttt{ mod } |U| = r_2, \\ r_2 &= m_4 \times |T| \times b_2 + r_4 \times b_2 + r_3, \\ t_2 &= r_2 \texttt{ div } b_2 \texttt{ mod } |T| = r_4. \end{aligned}$$

and

$$t_\diamond = t_2 = r_4. \tag{5}$$

Let $|U| = c_2 \times |T| \times b_2$. We can also obtain

$$\begin{aligned} s \;=\;& m_2 \times |U| \times b_1 + m_4 \times |T| \times b_1 \times b_2 + \\ & r_4 \times b_1 \times b_2 + r_3 \times b_1 + r_1, \end{aligned}$$

or equivalently,

$$\begin{aligned} s \;=\;& ((m_2 \times c_2 + m_4) \times |T| + r_4) \times (b_1 \times b_2) + \\ & r_3 \times b_1 + r_1. \end{aligned} \tag{6}$$

We next consider the function `block-cyclic`$(b_1 \times b_2)$ by referring to Equation (1). For each $s \in S$, let

$$\begin{aligned} s &= n \times (b_1 \times b_2) + r, \\ n &= m \times |T| + t. \end{aligned}$$

Then these equations yield

$$s = (m \times |T| + t) \times (b_1 \times b_2) + r, \qquad (7)$$

and

$$t_\times = s \texttt{ div } (b_1 \times b_2) \texttt{ mod } |T| = t. \qquad (8)$$

From Equation (6) and (7), we can infer that

$$m = m_2 \times c_2 + m_4,$$
$$t = r_4,$$
$$r = r_3 \times b_1 + r_1.$$

From Equation (5) and (8), we have $t_\diamond = r_4 = t = t_\times$. □

This property allows the function `block-cyclic`($b_1 \times b_2$) to work just the same as the function `block-cyclic`(b_1) $\diamond$ `block-cyclic`(b_2) if we can redeploy the points in the offset space at the entry and exit points of a parallel function. This redeployment can be performed as follows. We will redeploy each `in` or `inout` array A into another array B at the entry point of a parallel function via the following `map` function. Let $b1$ and $b2$ be the size of a block in the distribution functions `block-cyclic`(b_1) and `block-cyclic`(b_2), and $c1$ and $c2$ be the number of cycles in the respective distribution functions. Each element $A[i]$ in A will be mapped to the element $B[\texttt{map}(i, b1, c1, b2, c2)]$ in B.

```
map(i, b1, c1, b2, c2)
{
  for (m2 = 0; m2 < c1; m2++) {
    for (m4 = 0; m4 < c2; m4++) {
      for (r3 = 0; r3 < b2; r3++) {
        for (r1 = 0; r1 < b1; r1++) {
          if (i-- == 0) {
            t1 = m4*b2*b1*c1;
            t2 = r3*b1*c1;
            t3 = m2*b1;
            return t1+t2+t3+r1;
          }
        }
      }
    }
  }
}
```

Similarly, we will redeploy each `out` or `inout` array A into another array B at the exit point of a parallel function via the following `unmap` function.

```
unmap(i, b1, c1, b2, c2)
{
  for (m4 = 0; m4 < c2; m4++) {
    for (r3 = 0; r3 < b2; r3++) {
      for (m2 = 0; m2 < c1; m2++) {
        for (r1 = 0; r1 < b1; r1++) {
          if (i-- == 0) {
            t1 = m2*b1*b2*c2;
            t2 = m4*b1*c2;
            t3 = r3*b1;
            return t1+t2+t3+r1;
          }
        }
      }
    }
  }
}
```

Each element $A[i]$ in A will be mapped to the element $B[\texttt{unmap}(i, b1, c1, b2, c2)]$ in B. The correctness of this redeployment is given by the following theorem.

Theorem 6.2 *Let* `block-cyclic`*(b_1)* $: S \rightarrow U \times V$ *and* `block-cyclic`*(b_2)* $: U \rightarrow T \times W$ *be two distribution functions, and* $O = W \otimes V$. *If* $|U| \mid (|T| \times b_2)$, *and* `block-cyclic`*($b_1$)* $\diamond$ `block-cyclic`*(b_2) (s)* $= \langle t_\diamond, o_\diamond \rangle$, *for some* $t_\diamond \in T$ *and* $o_\diamond \in O$, *and* `block-cyclic`*($b_1 \times b_2$) (s)* $= \langle t_\times, o_\times \rangle$, *for some* $t_\times \in T$ *and* $o_\times \in O$, *then* $o_\diamond = \texttt{map}(o_\times)$ *and* $o_\times = \texttt{unmap}(o_\diamond)$.

Proof We first consider the function `block-cyclic`(b_1) $\diamond$ `block-cyclic`(b_2) by referring to Equation (4), (3), (2), and (1). For each $s \in S$, let

$$\begin{aligned} s &= m_1 \times b_1 + r_1, \\ m_1 &= m_2 \times |U| + r_2, \\ r_2 &= m_3 \times b_2 + r_3, \\ m_3 &= m_4 \times |T| + r_4. \end{aligned}$$

Then these equations yield

$$\begin{aligned} s &= m_2 \times |U| \times b_1 + r_2 \times b_1 + r_1, \\ o_1 &= s \texttt{ div } (b_1 \times |U|) \times b_1 + s \texttt{ mod } b_1 \\ &= m_2 \times b_1 + r_1, \\ r_2 &= m_4 \times |T| \times b_2 + r_4 \times b_2 + r_3, \\ o_2 &= r_2 \texttt{ div } (b_2 \times |T|) \times b_2 + s \texttt{ mod } b_2 \\ &= m_4 \times b_2 + r_3, \\ o_\diamond &= \mu(o_2, o_1) \\ &= o_2 \times (b_1 \times c_1) + o_1 \\ &= m_4 \times b2 \times b1 \times c1 + r_3 \times b_1 \times c_1 + \\ &\quad m_2 \times b_1 + r_1. \end{aligned}$$

The last equation can be rewritten into

$$o_\diamond = (((m_4 \times b_2) + r_3) \times c_1) + m_2) \times b_1 + r_1. \qquad (9)$$

We next consider the function `block-cyclic`($b_1 \times b_2$) by referring to Equation (2). For each $s \in S$, let

$$\begin{aligned} s &= n \times (b_1 \times b_2) + r, \\ n &= m \times |T| + t. \end{aligned}$$

Then these equations yield

$$\begin{aligned} s &= m \times |T| \times b_1 \times b_2 + t \times b_1 \times b_2 + r, \\ o_\times &= s \texttt{ div } (|T| \times b_1 \times b_2) \times (b_1 \times b_2) + \\ &\quad s \texttt{ mod } (b_1 \times b_2) \\ &= m \times b_1 \times b_2 + r \\ &= (m_2 \times c_2 + m_4) \times b_1 \times b_2 + (r_3 \times b_1 + r_1) \\ &= m_2 \times b_1 \times b_2 \times c_2 + m_4 \times b_1 \times b_2 + \\ &\quad r_3 \times b_1 + r_1. \end{aligned}$$

The last equation can be rewritten into

$$o_\times = (((m_2 \times c_2) + m_4) \times b_2) + r_3) \times b_1 + r_1. \quad (10)$$

From Equation (9) and (10), we have $o_\diamond = \texttt{map}(o_\times)$ and $o_\times = \texttt{unmap}(o_\diamond)$. $\square$

7 Conclusions

This paper has described a source-level program transformation approach to agglomerating virtual processors. This approach allows us to separate the module for the agglomeration of virtual processors and the module for the final code generation. This separation can greatly improve the retargetability and maintainability of the compiler. The agglomeration of virtual processors includes the replication of the data allocated and the code executed on the virtual processors. This paper focuses on the description about the replication of the data.

This paper has described this approach based on the data parallel programming language CCC; however, this approach is also applicable to other data parallel programming languages that supports the class of block-cyclic distribution functions. This paper has shown that the cascade of two block-cyclic distribution functions can be transformed into another block-cyclic distribution function by appropriately redeploying the local data on each virtual processor.

References

[1] C.-W. Chiang, T.-C. Liu, and N.-W. Lin, "Design and Implementation of a Data Parallel Language," *Proc. of the Second Workshop on Compiler Techniques for High-Performance Computing*, March 1996, pp. 9-16.