Compiling to TStreams, a New Model of Parallel Computation

Kathleen Knobe
HP Cambridge Research Lab
kath.knobe@hp.com

Carl Offner
HP Cambridge Research Lab
carl.offner@hp.com

Abstract

To get good performance on a parallel system, distribution and scheduling of data and computations are the critical issues. TStreams is a way of abstracting a program so as to represent exactly those aspects relevant to distribution and scheduling, leaving out all other details. We are interested in TStreams as a form on which to automatically produce good mappings and also on which to specify and tune mappings by hand. However, the focus of this paper is how to automatically generate TStreams.

1 Introduction and related work

TStreams is a new language for describing parallel computations. It describes what to compute and when it can be computed. However, it distinguishes the expression of the potential parallelism of the program from both

1. the lower-level serial details of the algorithm, and
2. the actual parallelism for a given target.

Precisely,

1. The user defines some high-level operations and specifies how they are used. The code for executing each such high-level operation is written in a serial language rich in appropriate control and data structures. TStreams, however, is not concerned with the implementation of the high-level operator. It is only concerned with how the operator is used.

   TStreams is not a general-purpose language. TStreams has a single data structure, a single control structure and a single operation.

2. TStreams makes no assumptions about the target architecture (shared memory/distributed memory/grid/web service/streaming) or the specifics of the configuration.

   Other languages are based on a specific model and then incorporate additional constructs or optimizations to deal with other models. For example, OpenMP (OpenMP Architecture Review Board, 2000) is focused on shared memory systems, HPF (High Performance Fortran Forum, 1997) on distributed memory systems and Brook and StreamIt (Thies et al., 2002) on streaming systems.

TStreams represents all styles of parallelism (e.g., task parallelism, data parallelism, loop parallelism, pipelined parallelism) in a uniform way. A program in TStreams form can be mapped and scheduled for any particular target.

TStreams is a language. A user can write it directly or it can be automatically generated by compilation techniques. In this paper we introduce TStreams and describe how to automatically generate it.

2 The elements of TStreams

We start by showing how to use TStreams to build a dynamic tree. This is the first stage of programs that solve n-body problems, which are important both in astrophysics and computational chemistry. These problems model how a large number of bodies, each acting on the others, evolves in time. The complexity can be reduced from $O(n^2)$ to $O(n \log n)$ by approximating a group of distant bodies as a single body. The computation starts by building a tree: the root is the 3-dimensional cube containing all the bodies. Cubes are recursively cut into 8 subcubes until no subcube has more than one body in it. Here we show how tree-building of this sort is represented in TStreams.

For simplicity, we will consider the problem in 2 dimensions rather than 3, so that each cube (square, actually) is divided into 4 subcubes—that is, the tree has 4-way branching.

The algorithm is quite simple: We start with the root cube. Each cube is *analyzed* to see if it contains more than one body. If it does, then it is *divided* into smaller cubes, and the process continues.

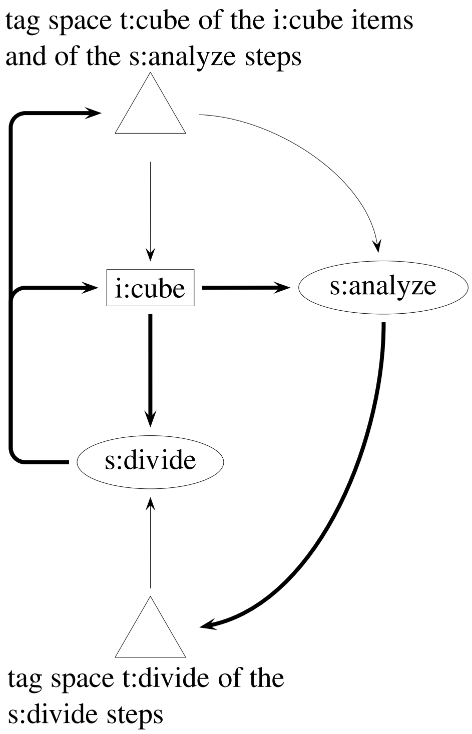

Figure 1: *Item spaces, step spaces, and tag spaces in the recursive tree building phase of an n-body problem. The (sole) item space is represented by a rectangle, the two step spaces are represented by ovals, and the two tag spaces are represented by triangles.*

There are three basic elements of a TStreams program: step spaces, item spaces, and tag spaces, which are sets of steps, items, and tags, respectively. These spaces are shown in Figure 1. The individual steps, items, and tags are shown in Figure 2, which shows how the actual program evolves in time.

Item spaces Data in the program is encapsulated into *items*. An *item space* is a collection of items. In our example there is one item space—the space of cubes, and each cube is an item.

Step spaces Computations in the program are represented as *steps*. A *step space* is a collection of steps which share the same code but apply it to different data. In our example there are two step spaces: the space of s:analyze steps (“s:” just reminds us that this is the name of a

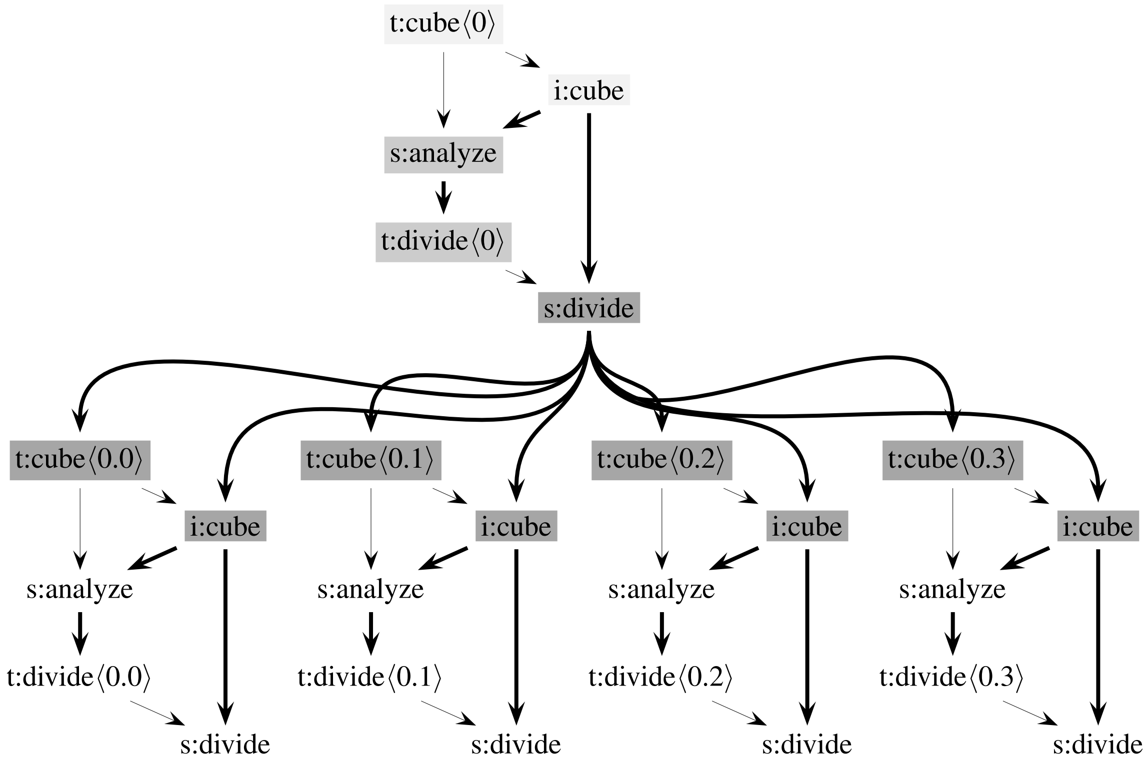

Figure 2: *Items, steps, and tags in the recursive tree building phase of an n-body problem.*

step) and the space of s:divide steps.

Tag spaces *Tags* are indices that are used to parametrize items and steps. Each item in an item space has a distinct tag, and each step in a step space has a distinct tag. So an item can be uniquely identified by specifying its name (that is, the name of its item space) and its tag; and similarly for steps. A *tag space* is a collection of tags that parametrizes one or more item spaces and/or step spaces. In our example, the tag of the root cube will be $\langle 0 \rangle$. The four child cubes will have tags $\langle 0.0 \rangle, \ldots, \langle 0.3 \rangle$, and so on.

These three elements are tied together by relations, of which there are two basic kinds:

Parametrization relations Each step space and each item space has a tag space that parametrizes it. Thus, each step and each item has a tag that identifies it uniquely. Parametrizations are represented in both Figures 1 and 2 by thin arrows.

Producer and consumer relations Steps consume items, and produce items and/or tags. Producer and consumer relations (i.e., relations between step spaces and item spaces) are represented in both Figures 1 and 2 by thick arrows.

In general, steps have access to their tags, and they take in certain items as input and produce items and/or tags as output. Other than that, they have no other information available to them, and they have no side effects other than the explicit production of tags and items.

Figure 2 shows how the recursive subdivision works. At the start of the program, we have the root cube and its tag. These two objects are shown lightly shaded at the top of the figure. The tag belongs to the t:cube tag space, which was given that name because it parametrizes the cubes.

However, each cube needs to be analyzed, so this tag space also parametrizes the s:analyze step space. So in addition to the thin arrow from the top tag to the root cube, there is also one from that tag to an s:analyze step. That step needs the root cube as input. (How does the s:analyze step know which cube to ask for? Answer: it knows its own tag, and it asks for the cube with the same tag.) Assuming the root cube needs dividing, the s:analyze step produces a tag with the same value ($\langle 0 \rangle$), putting it into the t:divide tag space. That tag space parametrizes the s:divide step space.

So the first thing that happens in the program is that the medium-shaded s:analyze step executes and produces a t:divide tag.

That t:divide tag in turn parametrizes a darkly shaded s:divide step. That step is ready to run because it needs (in addition to its tag) only the root cube as input, and that cube is of course available. (How does it know which cube it needs? The same way as before.) That step then executes.

So that step executes and produces the four child cubes, each with its own tag. These new objects are also darkly shaded.

Up to this point in the computation, the order of execution was completely determined. At this point however, there are four possible ways that execution can continue, since there are four steps that are ready to run. Note that we could schedule the execution of the steps in this program to traverse the tree being built in a depth-first manner or a breadth-first manner or in parallel, or in many other ways. The basic TStreams representation does not specify this order of execution. It simply indicates which operations are needed and for each operation shows the conditions necessary for it to be ready to run.

In this program, each step took as input an item whose tag has the same value as the step's tag. This is not the case in general. All that is required is that the step be able to compute from its own tag the tags of any items it needs as input.

3 The value of TStreams

TStreams is architecture neutral. The items represent the data consumed and produced by computations. These are inherent in the algorithm. A step is ready to run when its tag exists and its input items have been produced. How this is managed (e.g., by synchronization or communication) will vary on different architectures, and is not represented in the TStreams form itself.

TStreams expresses all types of parallelism with equal ease. Data parallelism arises from distinct step instances from the same step space. Task parallelism arises from distinct step instances of distinct step spaces. Pipelined parallelism arises from a chain of producer and consumer step spaces.

TStreams has one simple data structure, the tagged item; one simple operation, the tagged step; and one simple control structure, the tag. These are used to represent the potential parallelism of the program. We distinguish the expression of this potential parallelism from the code executed in the steps and from the distribution and scheduling of the steps.

Notice that although we have talked here about parallelism, the identical abstraction is appropriate for any reordering transformation, for example, optimizations for the memory hierarchy on uniprocessors.

TStreams is functional in the following ways:

- The contents of each item is uniquely determined by its tag.
- Each step is conceptually atomic. Either it runs to completion or the program can act as if it had never been executed.
- Each step has no side effects and always produces the same outputs from the same inputs.

Thus, re-executing a step can do no harm and so these properties support redundant execution, speculative execution and demand-driven execution. In addition, they enable a trivial and transparent checkpoint/restart capability and adaptivity to failure or variable resource availability.

A more complete discussion of TStreams and these issues can be found in Knobe and Offner (2004).

4 Mapping TStreams to a specific target

In executing a TStreams program three decisions have to be made:

- the granularity ("how big") of data and computations,
- the distribution ("where") of data and computations, and
- the schedule ("when") within each location.

The three decisions can all be made statically (i.e., compiled). This minimizes the overhead. Although the result is less rigid than existing systems, it is more rigid than necessary. Alternatively, the three decisions can all be made dynamically (i.e., interpreted). This approach is most adaptable to platform changes and data dependent computation but incurs some overhead. So at the low-tech end, it can be written by hand and interpreted immediately. At the high-tech end, it can be automatically generated and automatically optimized for a specific target.

The current TStreams prototype is a hybrid system. The program is hand written. The granularity and distribution are statically specified by the programmer. The schedule within an address space is dynamic: a step may execute when it is ready.

5 Automatic conversion of a program into TStreams form

Here is an overview of the main phases required to convert a normal serial program into a TStreams program. We assume here that we are given a single routine in a language that includes assignment statements and DO loops with arbitrary bounds and strides.

1. While normal serial code is location-based, TStreams is value-based. The first and most significant task is to transform the program program so it is value-based (Section 5.1).
2. Next, we convert this value-based serial program to a TStreams program (Section 5.2).

We will now examine each of these tasks in order.

S1: $x = \cdots$	S1: $x_1 = \cdots$
S2: $\cdots = x \cdots$	S2: $\cdots = x_1 \cdots$
if $\langle$condition$\rangle$ **then**	**if** $\langle$condition$\rangle$ **then**
S3: $x = \cdots$	S3: $x_2 = \cdots$
S4: $\cdots = x \cdots$	S4: $\cdots = x_2 \cdots$
end if	**end if**
	S5: $x_3 = \phi(x_1, x_2)$;; ϕ statement
S6: $\cdots = x \cdots$	S6: $\cdots = x \cdots$

Figure 3: *A serial code fragment and its translation into SSA form.*

5.1 Convert to a value-based form

In normal code, distinct values can be assigned to the same location. This gives rise to anti-dependences and output-dependences. These dependences unnecessarily restrict the reordering potential. We modify the program so that each location, i.e., array element, is assigned to at most once. There are two possible ways that a given location can be assigned to twice:

1. Distinct statements can modify the same location. To solve this problem we convert the program to static single assignment (SSA) form.
2. Distinct dynamic instances of a given statement can modify the same location. To solve this problem we then convert the program from SSA form into dynamic single assignment (DSA) form.

In the resulting program, each array element now holds at most a single value.

It will appear that there is significant overhead (both memory and computation) added as a result of these transformations. Much of this can be optimized away for any arbitrary ordering. The remaining overhead can be used together with parallelism and locality concerns in choosing the ordering and in choosing the granularity. In the discussion below we ignore the cost of the overhead.

5.1.1 Static Single Assignment (SSA)

In SSA form, each static assignment statement assigns to a distinct name. We show here how to transform a normal program into SSA form (Cytron et al., 1991). Consider the scalar code and its SSA form in Figure 3.

First, the left-hand side variable in each assignment is given a unique name. The two assignments to x (in statements S1 and S2) are now to variables x_1 and x_2. Some references are dominated by a single assignment (e.g., the references to x in statements S2 and S4). These references are renamed accordingly. The remaining references are not dominated by a single source assignment but rather get their values from one of several assignments (e.g., the reference in S6). At places where the control flow from two names meets, we insert a new assignment statement, assigning to a new name, using a "ϕ function" to combine the values of the two previous names. That is,

$x(:) = \cdots$	$x_1(:) = \cdots$
$\cdots = x(s_2) \cdots$	$\cdots = x_1(s_2) \cdots$
if $\langle$condition$\rangle$ **then**	**if** $\langle$condition$\rangle$ **then**
$x(s_3) = \cdots$	$x_2(s_3) = \cdots$
$\cdots = x(s_4) \cdots$	$x_3(:) = \langle$mostly values of x_1 but reflecting assignment to the element $x_2(s_3)\rangle$
	$\cdots = x_3(s_4) \cdots$
end if	**end if**
	$x_4(:) = \phi(x_1(:), x_2(:))$
$\cdots = x(s_5) \cdots$	$\cdots = x_4(s_5)$

Figure 4: *A code fragment with arrays and its translation into classic SSA form.*

the statement S5 assigns to x_3 either the value in x_1 or x_2, whichever is appropriate. Now the reference in S6 can be converted to x_3.

All references to the original variable are now dominated by exactly one assignment and this assignment assigns to a unique name.

This example contained only scalar variables. Classic SSA deals with arrays as well, but unfortunately, classic SSA form is not quite adequate for our needs for two reasons:

- Our ϕ functions may be executed. But classic ϕ functions are not honest functions. How is the ϕ function in statement S5 supposed to know which of its two arguments to return?
- Classic SSA provides no support for the kind of massive reordering of assignments to array elements required for parallelism.

Instead we rely on Array SSA (Knobe and Sarkar, 1998). First we provide some terminology. The term *local iteration space* is typically used to compare two statement instances within a loop nest. In this case, the two instances are taken from the same loop nest. In Offner and Knobe (2003) we have introduced a concept called *global iteration space*, which we use to compare two instances within the same program. Global iteration space is a single totally ordered data structure that includes for each executed statement both the lexical position of that statement in the program and its point in its local iteration space. So at any dynamic assignment, we can record the value $\langle$now$\rangle$ which is the current point in global iteration space.

Figure 5 contains the Array SSA code corresponding to the code in Figure 4.

We keep track of when each modification from the original program occurred according to the global iteration space in the original program via an assignment of $\langle$now$\rangle$ to the auxiliary variable $\tau x_2(s_3)$[1]. The ϕ function that combines the values from two assignments is now an honest function, defined as follows: The meaning of

$$x_4(:) = \phi(x_1(:), \tau x_1(:);\ x_3(:), \tau x_3(:))$$

[1]The prefix τ stands for "time". There are actually two versions of time that are important. In Offner and Knobe (2003) they are denoted by the prefixes τ_{w} and τ_{v}. The τ in these examples is τ_{w}.

is

$$\textbf{for each } j$$
$$x_4(j) = \begin{cases} x_1(j) & \text{if } \tau x_1(j) > \tau x_2(j) \\ x_2(j) & \text{otherwise} \end{cases}$$

$x_1(:) = \cdots$
$\tau x_1(:) = \langle\text{now}\rangle$
$\cdots = x_1(s_2) \cdots$
if $\langle$condition$\rangle$ **then**
  $x_2(s_3) = \cdots$
  $\tau x_2(s_3) = \langle\text{now}\rangle$
  $x_3(:) = \phi(x_2(:), \tau x_2(:);\ x_1(:), \tau x_1(:))$
  $\tau x_3(:) = \max(\tau x_2(:), \tau x_1(:))$
  $\cdots = x_3(s_4) \cdots$
end if
$x_4(:) = \phi(x_3(:), \tau x_3(:);\ x_1(:), \tau x_1(:))$
$\cdots = x_4(s_5) \cdots$

Figure 5: *The code fragment with arrays translated into Array SSA form.*

Notice that now the assignments to x_1 and x_2 can be reordered. The actual order of computation in the new program can differ from that in the original program. All we have to do is remember when in the original program each value was assigned. The τ variables are used to remember these times, and the ϕ functions use the τ variables in the reordered program to pick out the value that would have been assigned last in the original program.

5.1.2 Dynamic Single Assignment (DSA)

Array SSA eliminates overwriting of an element by two lexically distinct assignments. But overwriting can still occur by distinct instances of a given assignment. For example, an assignment to $x(i)$ within a loop on j will result in overwriting. Here we describe the algorithm we use to create DSA form (Knobe, 1997; Offner and Knobe, 2003) from array SSA form.

The basic idea is to modify the dimensionality of each object (i.e., each SSA name) to exactly the dimensions that matter. This might involve both adding and removing dimensions. We call these dimensions the *contributing dimensions* of the object.

Nothing else in the program is changed in this process. The dynamic control flow of the program is left invariant. Only the memory locations referenced may be changed.

We need another data structure in order to compute contributing dimensions for objects—the *valid iteration subspace* of each expression in the program. The valid iteration subspace of an expression is the set of loop indices of the loops containing that expression that cause that expression to vary. So if we can show that an expression is invariant with respect to some loop index, that index is not in the valid iteration subspace of that expression.

The algorithm for computing contributing dimensions can be thought of as a set of data flow equations that relate the contributing dimensions to the valid iteration subspace on both sides of each assignment statement.

We will illustrate the way the algorithm works by considering the code in Figure 6.

Initialization There are some things we know immediately.

Consider the valid iteration subspaces of the expressions on the right-hand side of S1 and S2 in Figure 6. We can determine syntactically that the first is `[]` and the second is `[`i`]`.

```
do i = 1, 10
    S1:   w_3(i) = 2
    S2:   x_7(i) = 2 * i
    S3:   y_5 = h(i)
end do

do j = 1, 10
    S4:   ··· = w_3(j) + ···
    S5:   ··· = x_7(j) + ···
    S6:   ··· = x_7(2) + ···
    S7:   ··· = y_5 + ···
end do
```

Figure 6: *Code fragment with many kinds of array references.*

Further, suppose that the array h in statement S3 is input to the routine and is not modified in the routine. We conservatively assume that the values in the array are not all identical. Therefore the dimension of h is really needed to hold all of the values in the array. So the dimension of h is contributing.

Thus we have initialized the valid iteration subspaces of the right-hand sides of S1 and S2 and the contributing dimensions of the array h in S3.

Valid iteration subspace of LHS: The valid iteration subspace of the left-hand side of an assignment statement is the same as that of the right-hand side, since both sides have equal values at equal times and therefore depend on the same set of loop indices. So once we have computed the valid iteration subspace of the right-hand side of an assignment statement, we can just propagate it automatically to the left-hand side.

In particular, the valid iteration subspace of $w_3(i)$ in S1 is `[]`, and the valid iteration subspace of $x_7(i)$ in S2 is `[`i`]`.

Contributing dimensions of LHS: Now consider the left-hand side of the assignments in the first loop.

S1: Since the valid iteration space of this expression $w_3(i)$ has just been found to be `[]`, we can see that w_3 really needs no dimensions whatsoever. That is, its value does not vary with respect to any loop indices. So it has no contributing dimensions. (Of course this example is quite simple. We could really do constant propagation here. But the same kind of reasoning would allow us to say that a 2-dimensional array really had 1 contributing dimension, for instance.)

S2: The valid iteration space of $x_7(i)$ is `[`i`]`. So the dimension of x_7 is really needed. That is, the dimension is contributing.

S3: Right now we cannot compute the set of contributing dimensions of y_5 because we don't yet know the valid iteration subspace of the corresponding right-hand sides.

However, we will see below that the valid iteration subspace of the right-hand side of S3 is `[`i`]`. When we arrive at this, we can deduce that y_5 must have a contributing dimension reflecting that it varies with i. So one has to be added.

In this particular code fragment, the addition of the new contributing dimension amounts to scalar promotion. However, the same reasoning is used to add additional contributing dimensions to arrays when needed.

Contributing dimensions of RHS references: The contributing dimensions of the right-hand side references in the second loop are taken directly from the contributing dimensions determined at their definition. This is not actually a propagation: the contributing dimensions are a property of the object itself and not of any particular reference. So we see that w_3 is a scalar. The first dimension of x_7 is contributing. For y_5, we added a dimension which is contributing.

Valid iteration subspace of RHS expressions: Consider the right-hand side expressions.

- S3: Since the dimension of h is contributing, we must assume that its value changes with its subscript. Therefore i is in the valid iteration subspace of the expression $h(i)$. (This is the information we promised above.)
- S4: Since we know that w_3 has no contributing dimensions, it does not vary with j. Therefore the valid iteration subspace of the expression $w_3(j)$ is `[]`.
- S5: The dimension of x_7 is contributing. This means that the values along that dimension may vary. As we execute different iterations of the loop on j, we are addressing different locations along that dimension. Therefore the values of the expression $x_7(j)$ vary as j varies so its valid iteration space is [j].
- S6: However, even though the values of x_7 vary along its first dimension, the reference $x_7(2)$ always retrieves the same value as j varies so the valid iteration space for $x_7(2)$ is `[]`.
- S7: Since y_5 has a contributing dimension, we have to add a dimension to its reference. That dimension has a subscript which in this case is i. We're not going to explain here how such subscript expressions are derived—this is explained in Offner and Knobe (2003). (All we will say here is that τ_v variables, mentioned in the earlier footnote, are used in constructing this subscript. In principle they could be complex.) When this is done, we do in fact find that y_5 has a valid iteration subspace of `[i]`.

The result of this transformation is what we call *weak* dynamic single assignment form: it is possible that a location will be assigned to more than once but in such a case it will always receive the same value. (The assignment S1 when changed to a scalar is still within the i loop and so is such a case.) The complete algorithm for transformation to weak DSA form is in Offner and Knobe (2003). That report addresses loop nests with multiple indices, arrays with multiple dimensions, arbitrary subscript expressions, reductions, and more. Conversion of weak dynamic single assignment to strong dynamic single assignment is fairly straightforward and we do not discuss it further.

5.2 Transform to TStreams

A program in DSA form, can then be transformed into a TStreams program. This is actually very straightforward, because such a program is already essentially in TStreams form, as follows:

First assume the granularity of computation is a single assignment, as in Figure 7. Each static assignment in the program becomes a step space and each static array name becomes an item space. Tag spaces are derived from array bounds and from loops and conditional clauses controlling execution of any basic block. Tag spaces derived from array bounds are used to parametrize the corresponding item spaces, and similarly for parametrizations of step spaces. Each consumer relation relates a step space corresponding to an assignment statement with the item spaces corresponding to the data objects on its right-hand side. Producer relations similarly deal with left-hand sides.

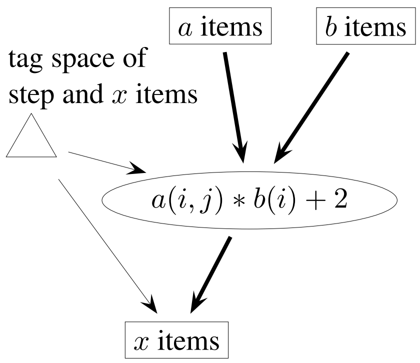

Figure 7: *A step corresponding to the assignment statement*

$$x(i,j) = a(i,j) * b(i) + 2$$

Dynamically each array element is an item, each execution of an assignment is a step and each point in an iteration space or a data space is a tag. The items referenced on the right-hand side and left-hand side of the assignment become the input and output items of the step.

Since the granularity of computation is a single assignment, each step has multiple input items, a single output item and some internal unnamed temporaries.

At this point there is a tag space for each basic block and for each item space. In fact, however, some of these tag spaces can be immediately shown to be identical; this can lead to more efficient code.

We can, for instance, relate the tag space of a step (say, an assignment within loops on `i` and `j` and controlled by a Boolean condition) with the tag space of the item it produces, say x_3. By construction, the tag space parametrizing the step space corresponding to the assignment is identical to the tag space parametrizing the item space corresponding to the array defined by the assignment. The tag spaces of the items referenced on the right-hand side may on the other hand not be the same, as we see in the assignment statement in Figure 7.

In any case, we see that the tag space created for a basic block parametrizes each of the steps (assignments) in the block and also the items defined by any step in that block.

In addition, it naively appears that each basic block will have a unique tag space. However, distinct points in the program may have identical control structures. For example, blocks before and after an IF statement will have the same tag space.

Next, suppose we consider a coarser granularity of computation: say each basic block is a single step. The step now has multiple input items and multiple output items. Further, some of the temporaries may be unnamed (that is, not visible as items at the TStreams level) if their only use is within the block.

At a still coarser grain, we can consider an innermost loop as a step space. It is parametrized by a tag space that corresponds to the loops enclosing it. We could also consider the two innermost loops as a step space; in this case its tag space would have one fewer dimension. We could also defer such decisions, leaving open the question of where the TStreams/step code boundary is to be. This decision could be made either later in compilation or at run-time. In this way, without making any target specific decisions, we can transform a program in DSA form to a hierarchical variant of TStreams. In such a program a step at one level looks like a TStreams program at the level below.

6 Future work

The current TStreams prototype implementation runs on a network (which may be heterogeneous) of Linux (or really any kind of Unix) based workstations. MPI and C++ are required; in particular, the programmer writes in C++.

We are considering a number of possible directions for future work:

- Automatic analysis of distribution and scheduling. based on array SSA and DSA form.
- Hierarchical variants of TStreams.
- Using TStreams for domain-specific language construction.
- Using TStreams as a base for adaptive systems.
- Introducing speculation and demand-driven models of execution into TStreams.

7 Acknowledgement

We thank Alex Nelson for many useful discussions as well as for most of the prototype design and implementation.

References

Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. *Efficiently computing static single assignment form and the control dependence graph*, Transactions on Programming Languages and Systems **13**, 451–490.

High Performance Fortran Forum. 1997. *High Performance Fortran Language Specification, Version 2.0*, Available from the Center for Research on Parallel Computation, Rice University, Houston, TX, and on-line at `http://www.crpc.rice.edu/HPFF`.

Knobe, Kathleen and Carl D. Offner. 2004. *TStreams: A Model of Parallel Computation (Preliminary Report)*, HP Labs Technical Report HPL-2004-78, Available at `http://www.hpl.hp.com/techreports/2004/HPL-2004-78.html`.

Knobe, Kathleen and Vivek Sarkar. 1998. *Array SSA form and its use in parallelization*, Proceedings of the 25th Acm Sigplan-Sigact Symposium on Principles of Programming Languages (Popl '98), Association for Computing Machinery.

Knobe, Kathleen B. 1997. *The subspace model: Shape-based compilation for parallel systems*, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Offner, Carl and Kathleen Knobe. 2003. *Weak Dynamic Single Assignment Form*, HP Labs Technical Report HPL-2003-169, Available at `http://www.hpl.hp.com/techreports/2003/HPL-2003-169.html`.

OpenMP Architecture Review Board. 2000. *OpenMP Fortran Application Program Interface, Version 2.0*, Available on-line at `http://www.openmp.org/specs`.

Thies, William, Michal Karczmarek, and Saman Amarasinghe. 2002. *StreamIt: A Language for Streaming Applications*, Proceedings of the 2002 International Conference on Compiler Construction, Grenoble, France, Lecture Notes in Computer Science, vol. 2304, Springer-Verlag, pp. 179–196.