Compiler Optimization of an Application-specific Runtime

Kathleen Knobe
Intel Corporation

Zoran Budimlić
Rice University

***Abstract*—Concurrent Collections (CnC) is a high level programming model designed to support parallel execution. The building blocks of CnC are *steps* (chunks of code representing units of scheduling or mapping), *items* (expressing data dependencies between steps), and *tags* (expressing control dependencies between steps).**

This paper introduces a runtime that views *attributes* (the changes in life stage of the CnC components) as events to be processed by event handlers. A major advantage of such a runtime is that it makes the state and the partial ordering of state changes explicit, raising the level at which tool support for debugging, visualising and checkpointing communicate with the programmer.

In this paper, we focus on its use as the foundation for a more advanced runtime that employs application-specific optimizations. We do this by compiling the CnC spec to generate an application-specific intermediate representation (IR) of the program. We introduce several analysis that can be done on the application-specific IR, along with program optimizations that take advantage of those analysis.

Other advantage of the attribute-based runtime include enabling the addition of now attributes at the low runtime level, for example to support speculation or demand-driven execution and enabling the programmers to define their own attributes,

I. Introduction

In the CnC programming model [4], [3], the domain expert identifies chunks of code that are the units of scheduling/mapping, and in addition identifies the ordering constraints among these chunks. There are only two types of constraints. They correspond to data dependences and control dependences. Separately, the tuning expert can indicate a tuning plan. One domain spec might be associated with several tuning specs (for different platforms or different tuning goals).

Control tags (representing control dependences) indicate which computation steps will execute (not when). The computation steps in CnC are atomic. The data items (representing data dependences) are dynamic single assignment (and represented as key/value pairs)[1]. Instances of steps, items and tags go through various life stages, each of which can be identified by an *attribute*. For example, items and tags can be *available* or *dead*. Steps may be *control ready* (they will execute), *data ready*, *ready* and *executed*. The dynamic state of a program is then simply the set of instances and their attributes together with the contents of items. We refer to this dynamic state as the *execution frontier* [11]. [2].

CnC has been implemented using a wide variety of different runtime approaches [1], [2]. The only requirement is that the control and data dependences are honored. But none of these implementations are built by explicitly managing these attributes. Because the CnC specs expose all the available inter-step parallelism, the existing implementations achieve very respectable performance [7]. However, there is room for reducing some of the runtime overheads.

This paper introduces a runtime that views the changes of life stage (attributes) as events to be processed by event handlers. A major advantage of such a runtime is that it makes the state and the partial ordering of state changes explicit. This "basic" runtime, by itself, can raise the level at which tool support for debugging, visualising and checkpointing communicate with the programmer. In this paper, we focus on its use as the foundation for a more advanced runtime that employs application-specific optimizations described below. In the basic runtime, the set of attributes and event handlers is generic and pertains to all CnC specs. The graph of state transitions can be considered as a universal Intermediate Representation for all CnC programs. The basic RT can then be viewed as an interpreter of this universal IR.

The obvious next step is compiling the CnC spec to generate an application-specific IR. For example, the universal IR contains nodes meaning "this computation step is data ready" or another meaning "this data item is available". The specific one might contain nodes for "foo is data ready" or "x is available".

Recall that the execution frontier indicate the dynamic state of a specific run as the execution proceeds. The application-specific IR allows the compiler to perform static analysis on the IR, based on propagation of a static/symbolic version of the execution frontier encapsulating what is statically known

[1] If the application adheres to these two rules the execution will be deterministic. We claim that these rules are easier to follow than "avoid race conditions".

[2] This process supports continuous, asynchronous checkpointing with no user interaction.

at each point in the graph. For example, the compiler could identify and eliminate handlers for checking whether a step is data ready by proving statically that if that particular step is control ready it must also be data ready. As another example, it might be possible to identify points in this IR where an item is dead. These transformations result in a statically optimized application-specific runtime.

In addition, this application-specific attribute-based runtime provides an opportunity for making decisions (not semantic requirements) explicit in the IR. For example, in a distributed system for a specific distribution, it might be the case that an item is available on one node but not another or that a step is data ready on one node but not on another. Making these distinctions explicit allows us to make the communication explicit. The scheduling constraints then indicate that the communication of data item x from node A to B must come after x is available on node A and before it is available on node B. Because it is explicit, we have an opportunity to statically schedule the communication itself.

One software engineering advantage of the attribute-based runtime is that it facilitates the addition of new attributes at the low runtime level, for example to support speculation or demand-driven execution. In addition to attributes at the IR level, we will also describe some advantages of allowing the application programmers to define their own attributes.

This paper describes an attribute-based application-specific runtime for CnC and the details of several optimizations on this runtime. The work described in this paper is currently in the early stages of the design and implementation as part of the DOE X-Stack project.

II. Introduction to CnC

Concurrent Collections (CnC) is a high level programming model designed to support parallel execution. CnC draws some ideas from dataflow languages but extends them by adding control flow as a first class feature, so in a sense it is more of a data and control flow language than a data flow language. In addition it draws from tuple spaces, simplifying some of the compiler analysis referred in this paper. In the CnC model, the computation is expressed through *steps*, which are functional pieces of code. Data is stored in *item collections*, which are sets of (key,value) pairs. Keys are called *tags* and can be anything (integers, tuples, strings), the only requirement is that each value has a unique tag within the item collection. The values are called *items* and follow the dynamic single assignment rule (they can be assigned only once during the execution of the program).

Like data, steps are also organized in collections, called *step collections*. Each step instance in a step collection is identified by a tag. Step collections are *prescribed* by *control collections*, which are sets of unique tags. A control collection can prescribe one or more step collections An existence of a specific tag in a control collection indicates that a step identified by that particular tag in each step collection prescribed by that control collection will execute at some point.

Steps can *put* (tag, item) pairs into item collections and tags into control collections. Puts are atomic operations that obey the dynamic single assignment rule. Steps can *get* items from item collections by providing the tag of that item. A step can get an item after it has ben put[3].

Steps can put tags into control collection. Putting a specific tag into a control collection only indicates that the steps prescribed by that tag *will* execute at some point, it does not have any bearing on *when* will those steps execute.

Figure 1 shows an example of a CnC program that detects, from a collection of images, those that contain a face. For presentation purposes, CnC programs are represented as graphs, although in practice they can be written in text form, or as C++ APIs.

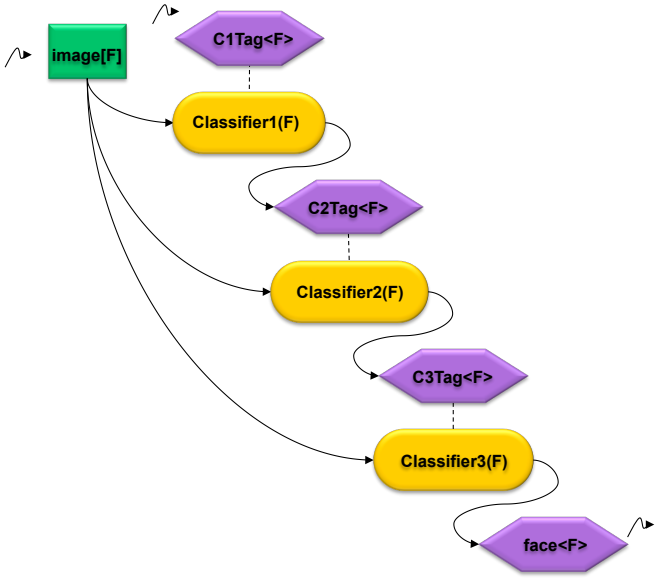

Fig. 1. Example of a face detection program written in CnC

Item collections (in our example there is only one called Image) are represented as squares. Step collections are ovals, while control collections are diamonds. Producing and consuming items and tags are shown as arrows, while prescriptions are shown as dotted lines. In the face detection example shown on Figure 1, all images are stored in the item collection `Image`, and identified by the tag `F`. The images are processed with a series of classifier steps. `Classifier1` uses a fast algorithm that can detect if an image is definitely not a face. If it detects that an image is not a face, it does nothing. If the results is inconclusive, `Classifier1` puts out a tag of the image into control collection `C2Tag`, ensuring that a `Classifer2` will run on the same image. `Classifier2` uses a different (more precise, but not as fast) algorithm that also determines if the image is definitely not a face. If the result is inconclusive, `Classifier2` puts a tag of the image into control collection `C3Tag`, which prescribes the final classifier, `Classifier3`. `Classifier3` uses a slow and precise algorithm that detects whether an image is a face or not. If the image is a face, `Classifier3` puts out a tag of that image into control collection `face`.

[3]How this is ensured is implementation-dependent. For example, the runtime can suspend the execution of a step that is trying to perform a get on a non-existing item until that item appears. Or it can delay starting the execution of a step until all the items that the step is getting are available.

The squiggly lines going into item collection `image`, control collection `C1Tag` and out of the control collection `face` represent the interaction of the CnC program with the outside world, which we call *environment*. The environment initializes the data by putting the images to be analyzed into the item collection `image`, and initializes the computation by putting the tags of all the images into the control collection `C1Tag`, which ensures that an instance of `Classifier1` will run on every image. The environment also reads the results of the computation by reading the tags from the control collection `face`. Every tag present in the control collection `face` identifies an image that has been determined to represent a face.

If the programmer follows some simple rules, the resulting CnC program is deterministic [3]. The rules are:

- The items have to obey the dynamic single assignment rule. Each tag needs to be associated with a unique item value during the execution of a program.
- The steps have no side-effects

These requirements are easier for the user to follow than the vague dont allow race conditions requirement found in most other concurrent programming models. Of course, these rules also restrict the class of applications that CnC can handle. In addition, these requirements are also much easier to dynamically check during the execution of a CnC program than checking for race conditions. Many of the existing CnC applications follow these rules. Many dont. For those that dont, CnC is used primarily for asynchronous scheduling, leaving the user with total responsibility for correctness. For applications that obey the contract, CnC shoulders some of the responsibility for correctness. One big advantage of deterministic programs is that they are much more analyzable and optimizable. This paper only considers such programs.

Another attractive feature of CnC is serializability. If the user ensures that within each step there are no puts before all the gets have completed, then steps can be serialized, that is, the runtime can execute the steps one at a time, with each step being executed from start to end. Ensuring that all the gets are performed before all puts within a step is also trivial to implement in a CnC runtime.

A typical CnC step gets some input items, computes based on the input, then puts some output items. If instead, a step would perform some puts before some gets, then it is possible to create co-routines, multiple steps that need to be active at the same time. Such co-routines are not serializable, and can also lead to deadlocks.

The granularity of the computation performed within each step might be expressed as one or more parameters, leaving the static graph looking the same. For example, one might implement a tiled matrix computation (such as Cholesky decomposition) in CnC, with the tile size being the determining factor for the amount of computation done in each step. On the other hand, one can change the granularity of the computation by taking a step and decomposing it into a subgraph with smaller steps, resulting in a very different CnC graph. Naturally, the absolute performance of the application when running on a parallel system will depend on the granularity of the CnC steps in the application. However, for a given granularity, CnC will expose all the parallelism semantically available in the application.

For a given grain, however, it is possible to express the graph more or less explicitly. For example, the face detector example described above can be expressed with each classifier as a distinct collection or with the classifier number as part of the tag [4]. If we distinguish the classifiers via collection name, it becomes explicit that only the last classifier can emit isFace¡¿. If the classifiers are all part of the same collection, this detail is lost. This decision has no impact on the granularity (parallelism) of the domain spec [5]. However, it might have a significant impact on analysis and optimization. With more actual distinctions explicit in the spec, the analyzer/optimizer have more potential to take advantage of these distinctions.

In the paper we assume the program is written by a programmer but it might be the output of compiler analysis. There is an effort underway to generate CnC from polyhedral compiler analysis.

III. The Meaning of a CnC Program

A. Abstract model

Here we describe the abstract model for a CnC program in terms of state and how that state evolves. First, in this section, we will show the abstract state and the evolution of that state. This is how we think about the meaning of the program. For an actual implementation we need a bit more, as described in Section IV. The abstract mode presented here is the basis of the implementation, the static analyses and the optimizations described in the remainder of the paper. As a CnC program executes, instances of steps, items and tags transition through various states. We identify these state transitions by attributes. For example, an item may be available and it may be dead. We write this as `x[j].available` and `x[j].dead` for example. At any point in the execution, the various instances within a collection typically have different states. There is a partial ordering among the state transitions within a single item, step or tag instance. An item must become available before it becomes dead, for example. The state of a tag or step instance is the set of attributes it has acquired. So the state monotonically increases. For available items, the state includes not only its attributes but also the contents of the item. Once an instance has acquired a set of attributes, the actual order in which these attributes were acquired becomes irrelevant. They are just sets.

Abstract attributes are those required to describe the abstract model, available for example. Table I shows the abstract attributes. Notice that dead is not one of these.

Figure 2 shows how the program state evolves and exposes the partial ordering requirements. Some tags and items are available as program inputs. An available tag will cause one or more steps to become controlReady (it will execute).

[4] In the extreme case, one may create what we call The Universal CnC graph, which consists of only a single step collection, single item collection and a single control collection, and can be used to express any kind of computation

[5] it might make a difference with tuning or with appropriate depends functions.

item	available
tag	available
step	controlReady, dataReady, ready

TABLE I. ATTRIBUTES IN THE ABSTRACT MODEL

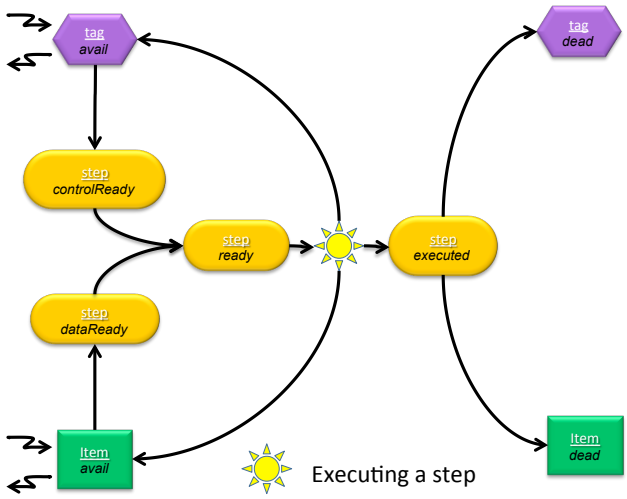

Fig. 2. Evolution of a program state

An available item may cause one or more steps to become dataReady (all its input data items are available). When a step is both controlReady and dataReady it becomes ready and it may execute. The execution of a step may make more tags and items to become available and the process continues. There are no synchronizations other than those corresponding to producer/consumer and controller/controllee so many of these events may be happening concurrently and asynchronously. There are many legal orderings of execution and many points in each legal execution. Each has a corresponding state. This is also true before any step has executed, before all the input is available, and after the program terminates.

B. Execution frontier and the CnC runtime

Here we extend the abstract model to address the limited time and space for an actual execution. For example, items and tags come into existence, are used for a while, then at some point they become irrelevant and are never accessed again. An actual execution will, of course, need to remove these. Similarly, the abstract model does not distinguish between an executed step and one that needs to be executed. This is irrelevant because the abstract model is declarative and the steps are functional. But this will not do for a real execution.

To address these practical requirements we add several more attributes as shown on the right side of Figure 2.

- *dead* as an attribute of items
- *dead* as an attribute of tags and
- *executed* as an attribute of steps.

Notice also that in the abstract model, there are an infinite set of possible instances that have empty state. This is fine for an abstract state but the execution frontier must be finite. The actual state therefore begins as empty. An instance is added to the state only when it acquires its first attribute. Maintenance of the actual state, involves adding instances to the leading edge as they acquire their first attribute. Attributes are added to the leading edge as described for the abstract model. But in addition, instances are removed from the trailing edge as they are no longer relevant.

- An item is dead and can be removed when all its uses have occurred. Removing an item from the trailing edge includes removing its contents.
- A tag is dead and can be removed when all steps it controls are executed.
- A step is executed when it completes. It can then be removed.

This process now tracks the relevant frontier of the execution. The resulting state, called the execution frontier, describes everything necessary to continue the execution [6]. Figure 2 indicates how execution frontier evolves dynamically.

CnC has been implemented in a wide variety of very different runtime approaches but this description is very close to a specification of a runtime. One can implement a fully functioning general CnC runtime this way. For an actual implementation, there are a few issues relating to management of the execution frontier that need to be considered. Examples include: how to determine when all the data is available (ddf, codelets), how to reuse memory efficiently [10] and how to determine when an item is dead [5]. The first two are important but are orthogonal to this paper. We use dead item detection as an example of an application-specific optimization in section VI.

The intermediate form described above is general and works for all CnC programs. In the next section, we show how to create an intermediate form for representing a specific application. This intermediate form is then used to implement several application-specific optimizations.

C. Termination

A program is quiescent when the following two conditions hold:

- no legal forward progress is possible. This means that:
  - all legal attribute propagation has occurred
  - there are no ready to run steps
- no step is currently executing. (Notice that an executing step many result in more possible progress.)

Valid termination occurs when the program is quiescent and all control ready steps are also executed steps. In this case some input to a controlReady step has not been produced. The assumption is that this constitutes a programmer error.

The abstract state is sufficient to understand the meaning of an application. It is almost but not quite adequate for execution on an actual plattorm.The next section acknowledges constraints imposed by an actual execution on an actual platform.

[6]This execution frontier is sufficient to restart the application [11]

IV. APLICATION-SPECIFIC RUNTIME

We will use the face detection example to illustrate the process of constructing the nodes and edges an the application-specific intermediate form using the CnC application specification and the general intermediate form.

a) Step collections: For each step collection in the application, `Classifier1()`, `Classifier2()` and `Classifier3()` for example, create explicit nodes and edges corresponding to the transitions among state elements within that collection, for example:

```
Classifier2().dataReady
Classifier2().controlReady
Classifier2().ready
Classifier2().executed

Classifier2(). dataReady,
Classifier2().controlReady →
Classifier2().ready → classifier2().executed
```

b) Item collections: For each item collection, `image[]` in our case,

1) create a node corresponding to the items available state, `image[].available` in our example
2) For each `step()` that consumes the `item[]` connect `item[].available` with `step().dataReady`
3) For each `step()` that produces the `item[]` connect the `step().ready` with `item[].available`
4) create a node corresponding to the items dead state, `image[].dead` in our example
5) For each `step()` that consumes the `item[]` connect `step().executed` with `item[].dead`

c) Tag collections: Now, for each tag collection, `C1Tag⟨⟩`, `C2Tag⟨⟩`, `C3Tag⟨⟩` and `face⟨⟩` in our case:

1) create a node corresponding to the tags available state, `tag⟨⟩.available` in our example
2) For each `step()` that produces the `tag⟨⟩` connect `tag⟨⟩.available` with `step().controlReady`
3) For each `step()` controlled by the `tag⟨⟩` connect `step().executed` with `tag⟨⟩.dead`
4) create a node corresponding to the tags dead state, `tag⟨⟩.dead` in our example

d) I/O: Add the relationships with the environment. These are the connections between the outside world and the CnC program, usually used to set up the computation, perform I/O to input/output the data into and out of the CnC programs.

e) Application-specific dataReady states: We can make this graph even more application-specific. Instead of simply indicating whether the step is `dataReady` or not, we go into more detail and distinguish between the different item collections that are read by the step. That is, if the application has `x[]` → `foo()` and `y[]` → `foo()` then, we will replace `foo().dataReady` with `foo().xReady` and `foo().yReady`. Then `foo().xReady`, `foo().yReady` → `foo(dataReady)`[7]. The relationship between `controlReady`, `dataReady` and `ready` remain as before.

[7]We stop at collection names and do not expand further to distinct gets from the same item collection.

This process creates an application-specific runtime in which each state transition of each attribute during the execution of a specific CnC program is made explicit and processed by a runtime handler. Figure 3 shows the application-specific intermediate representation for the Face Detection example.

In addition to this application-specific graph, annotations might be supplied by the user or generated from an analysis of the graph and the step code. This meta-data can improve the result of our anlayses and optimizations. Produces and consumes tag functions start with the tag of a step and generate the tag of instances it produces or consumes. Produced-by and consumed-by functions start with the tag of an item or a control tag and generate the tag of instance of step that produce or consume it.

The function relating a control tag to the step it controls does not need to be included because it is always the identity function.

Other annotations may indicate whether an edge is a must or a may edge, e.g., `foo()` may produce `barTag⟨⟩` or `bar()` must consume `z[]`. Further, anotations might indicate that one step produces exactly one of two control tags.

These all form the input to the flow analyses in the following sections.

V. ANALYSES

In this section we show how to use the application-specific intermediate representation to propagate information along the edges in this graph. In Section VI will show some examples of how this information can be used for optimizing the graph and therefore the runtime.

We follow the graph IR in Figure 3 for the our face detection example. The analysis propagates information among these nodes in the application-specific IR. The information associated with a node in the graph is a set of state elements where each element is a collection name, its static tag representation and an attribute, e.g., `foo(j, k).controlReady`. Of course, we dont include the contents of data items for these static analyses.

We will develop the ideas through a series of scenarios:

- acyclic graphs where each collection contains just a single instance
- acyclic graphs where collections contain multiple tagged instances
- cyclic graphs where each collection contains just a single instance
- cyclic graphs where collections contain multiple tagged instances

A. acyclic graphs - single instance collections

We introduce the concepts via the simplest case: acyclic graphs where each collection contains just a single instance.

We use the term state element, e.g., `foo().dataReady`, in several different ways in the following discussion. First, it refers to a node in our application-specific IR. The IR node

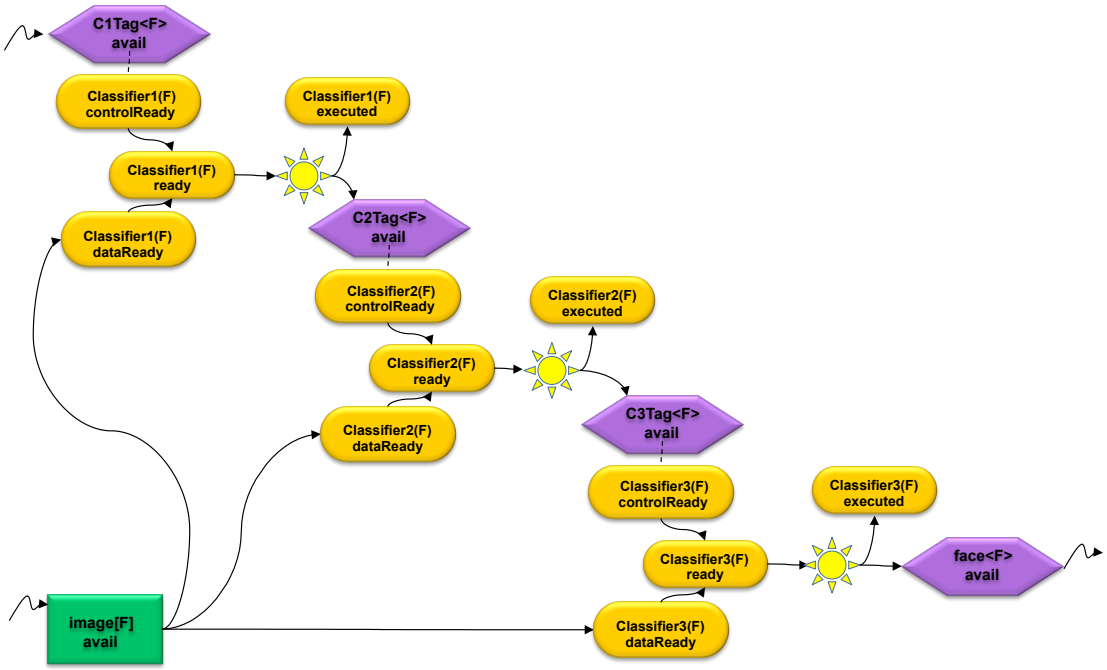

Fig. 3. The application-specific intermediate representation for the Face Detection example

stands for the runtime processing associated with that state element. Second, sets of state elements are used to identify the collection of facts, that we know when the runtime arrives at this node. Consider forward must processing. We know state element S, that is a fact, at a given node, X, exactly because the runtime, in arriving at this node X, must have passed another IR node, say Y. This node Y is, of course, the node corresponding to the state element S. For example, assume node X corresponds to `foo().dataReady`. On arrival at X we know the following (must/forward) facts: `A[].available`, `A[].dead`, `B[].available`, `bar().controlReady`. This means exactly that to get to the `foo().dataReady` IR node it was necessary to go through an IR node corresponding to `A[].dead`. So when the runtime arrives at the IR node `foo().dataReady` it knows that `X[]` is dead. So these two uses of state element are consistent.

There are two basic kinds of information to propagate: must and may. There are two directions of propagation, forward and backward. The result is four distinct analyses. In all cases we are referring to what is happening on a path. We dont consider what might be happening concurrently but elsewhere in the graph. Our optimizations based on these results only depend on path information.

The may set for a node is the set of state elements that may exist when the execution arrives at the node. For forward may propagation, consider a junction, say node A and node B join at node C. Any state element that may be true at A or B may be true at C. So the resulting may set at node C is the union of the sets at A and B together with the state element corresponding to the node C.

Backward may propagation determines for each node what state elements may occur on paths from that node. Backward may propagation also uses the union operation. If node A and node B follow from node C than facts that may follow from either A or B also may follow from C.

The must set for a node is the set of state elements that must exist when the execution arrives at the node. Must processing is more interesting than may processing.

Below we discuss some of the propagation rules. In all cases, when computing a state set for node N, it is assumed that we include the state element associated with node N itself.

These analyses do not view the IR as an arbitrary graph with anonymous nodes. Different types of IR nodes have different semantics and therefore different propagation rules. For some, the resulting state during forward propagation is the union of the states of the predecessors. For others it is the intersection. In either case after the intersection or union, we add in the state element associated with the node itself.

One aspect of our graph that we rely on for the analyses below is that `step().executed` node is not on the path from `step().ready` to `item[].available` or `tag<>.available`. Even before the step completes, the items and tags it produces may cause other steps to execute. The predecessor of `item[].available` and `tag<>.available` is therefore `step().ready` and not `step().executed`. `step().executed` is, however, the predecessor for `item[].dead` and `tag<>.dead`.

As an example, suppose we are dealing with a junction where nodes `foo().controlReady` and `foo().dataReady` join at the node associated with `foo().ready`. If some state element, say `x[].available`, is in the must set of one of the inputs to the junction, say `foo().controlReady`, then it is in the must set for `foo().ready` regardless of whether it was in the must set for the other input, `foo().dataReady` in this case.

```
must(foo().ready) =
    union (
        must(foo().dataReady)
        must(foo().controlReady)
    )
```

For other attributes, we need to compute the intersection. For example, consider an item `x[]` that might be produced by either `foo()` or `bar()`. Because of our single assignment requirement, each dynamic instance of `x[]` is produced either by one or the other but not both. `X[].available` is reached through exactly one of the `foo().ready` branch or the `bar().ready` branch. In this case a state element is in the must state of `X[].available` only if it was in the must state on both branches. At this type of junction an intersection, not a union, is required.

```
must(x[].available) =
    intersection(
        must(foo().ready)
        must(bar().ready)
    )
```

For our current scenario with only acyclic graphs we can process a node when all its inputs have been processed. Forward must processing begins with input from the environment.The only type of nodes with input from the environment are of the form `item[].available` or `tag` $\langle\rangle$ `.available`. We start with the envionment as an empty set of facts. Since, at each node, we add the nodes own state element to its set of state elements, the state at these input IR nodes includes exactly one state element corresponding to itself. We then process nodes all of whose inputs have been processed until the graph is completly processed.

We categorize nodes into the categories below and show the style of processing for each. Mostly they are distinguished by the associated attribute but in the case of available items and tags we distinguish futher between those with a single producer (possibly the envirnoment) and those with more.

Pass through junctions. Has exactly one input. (Some but not all of the `item[].available` and `tag[].available` nodes fall into this category.) Just add in current node

```
step().controlReady
step().executed
item[].available
tag[].available
```

Union junctions. These have more than one input. They all must occur.

```
step().dataReady
step().ready
item[].dead
tag<>.dead
```

For `step().dataReady` we are assuming here that all the inputs are required. Some of our implementations have this constraint. Others allow for data-dependent gets where a get may depend on the contents of a previous get. In this more general situation, we may have meta-data about which inputs will definetly occur and which are optional. We use this information to create an appropriate expression of unions and intersections to ensure correctness.

Intersection junctions. Could have multiple inputs (including the environment) but for any specific instance only one occurs.

```
item[].available
tag<>.available
```

Backward must propagation is as follows: (Note: In the backward flow case inputs to the propagation are outputs as the graph executes.)

Initialization. Have no outputs. Just initialize with this node itself

```
step().executed
item[].dead
tag<>.dead
```

Pass through junctions. Has exactly one output. Just add in current node.

```
step().controlReady
step().dataReady
```

Union junctions. These have more than one output. They all must occur.

```
tag<>.available
step().ready
item[].available
```

`tag<>.available` could have multiple outputs, each of the form `step().controlReady`. All will be taken.

`step().ready` has multiple outputs, `step().executed`, multiple `item[].available`, and multiple `tag<>.available`. Similarly, `item[].available` may have multiple outputs, one for each step that uses the item. For this diescription we assume here that we know they will all occur. But in some of our implementations this isn't a requirements. For these we would rely on user annotation or compiler analysis for better information.

Intersection junctions. Could have multiple outputs but for any specific instance only one occurs.

This might occur via annotations. For example, we might be given the fact that only of two possible control tag outputs are emitted by a particular step.

B. acyclic graphs - multi-instance collections

Now we consider acyclic graphs where collections contain multiple tagged instances. If the tags are all identical, as is the case in our face detection example, the propagation is similar to the scalar case. But more interesting cases can arise. Consider an application containing `bar(j,k)` $\rightarrow$ `y[j', k']`. In cases like this where we have no knowledge of the relationship among the tag components. We have to be conservative. But we

might have access to functions that map the tag of `bar()` to the tag of y[]. Suppose with tag functions we know that `bar(j,k)` $\rightarrow$ `y[j+1, k]`. The state at the IR node `bar(j, k).ready` includes the state element for itself, `bar(j, k).ready`. Now when we propagate the state at IR node `bar(j, k).ready` to the state at IR node `y[j', k'].available`, we want the tag components of its state elements to be functions of `[j', k']`, that is, the indices for `y[]` not those for `bar()`. So at IR node `y[j', k'].available`, we need to apply the reverse tag function as we propagate the state forward. So the state at IR node `y[j', k'].available` will contain `bar(j'-1, k').ready`

C. cyclic graphs - single instance collections

Next consider cyclic graphs where each collection contains just a single instance. Here we can not assume that when processing a node the state of its inputs are already known. The state coming from a back edge may be empty when it participates in a union or intersection. In both cases, the set might appear to be smaller than it actually is. If we iterate the process and state elements are added to back edges the sets at any given node will become monotonically larger. (of course, between one IR node and its successor the state may be smaller at the successor.) As processing proceeds any state element already in either a must or a may set is valid but some state elements that belong in such a set may not yet have been identified. We can iterate the propagation process until there are no changes.

Since we are popagating until done, we have to consider the possibility that the process does not terminate. Notice first that the sets (at both intersection style and union style nodes) will grow monotonically over the repeat-until-no-change process. This is true for must and may processing and for forward and backward processing. It is not the case that a successor has a larger set than its predecessors. Rather the claim is that the set at a given node never shrinks. We know that the number of distinct collections is fixed. The number of tag components per collection is fixed. A process of producing monotonic increasing sets where the sets have a bounded size must terminate.

D. cyclic graphs - multi-instance collections

Now we integrate the issues of cyclic graphs and the issue of collections containing multiple tagged instances. The propagation along the back edge is exactly as the propagation along any edge. This includes adjusting the perspective from the tag at the source of the back edge to the perspective of the tag of the target of this edge. We continue propagation until no changes can occur. At this point the analysis is done.

Again we have to ensure termination. The only additional possibility for this case that could lead to non-termination (beyond those of the cyclic single instance case) is that the tag component expression might keep changing forever. Here we can terminate early as long as we are careful to end with a conservative result. For must analyses (forward and backward) having too few state elements may be overly conservative but it is correct. So in fact, we could even ignore cycles altogether. For may analyses (forward and backward) having too many state elements may be overly conservative but is correct. In this case, in fact, we could use a wildcard for that tag component. We propose that for each tag component we constrain the number of possibilities to some fixed number, say one. Consider the following three:

1) A simple component name, e.g., `x[j].available`
2) A simple function of the component, e.g., `x[2*j+1].available`
3) A wild card, `x[*].available`

Because we handle at monst one functions, wwhen the process is about to modify the first function by applying another, the process simply stops. For a must analysis we leave it as was. For a may analysis we use the wildcard. This leaves us with exactly 3 possibilities for each tag component. The number of possible state elements that can be added is finite. Therefore the process terminates.

VI. Optimizations

The application-specific graph represents the work of the runtime. The previous section, provided some analyses that propagate information along the lowered application-specific IR graph. That graph specifies the work of the runtime. Here we use that information to optimize that graph, reducing the work of the runtime. We show just a few optimizations to give a flavor of what is possible.

A. Lowering step().dataReady

We first lower the dataReady attribute for each step in order to distinguish among different inputs. For example, if `foo()` has two different inputs `x` and `y`, we will replace `foo().dataReady` with `foo().xReady` and `foo().yReady` attributes. This will allow us to apply independent optimizations to these nodes.

B. Remove singletons

Any IR node that has a single input can be removed. For example, this is always be the case for `step().controlready` attribute. Using control collections to mediate between the controller and the controllee has real software engineering benefits. Because the controller and the controllee are not directly connected, controllees can be added to or removed from the graph without modifying the step code of the controller. Similarly controllers can be added to or removed from the graph without modifying the code of the controllee. However, we would like to maintain the software engineering benefits without overhead. Naively, after the application-specific put of `tag⟨⟩` emits `tag⟨⟩.available`. Then processing of `tag⟨⟩.available` emits `ControlledStep1().controlReady`, `ControlledStep2().controlready`, etc. In an application-specific implementation the put can directly emit `ControlledStep1().controlReady`, `ControlledStep2().controlready`, etc.

Some items are consumed by exactly one step. In this case the `item[].available` node has exactly one successor so it can be removed in a similar way. Similarly, some `step().executed` IR nodes might have exactly one successor.

Also, after lowering the `dataReady` attribute to distingush between, say `xReady` and `yReady`, each of these has exactly one input, `x[].avail` or `y[].avail`.

C. Redundant computations

In Figure 3 we see that `image[f].available` leads to `classifier1(F).dataReady`. This is required. But now the state element `image[f].available` is propagated forward as a must state element through `classifier1(F).Ready`, `C2Tag⟨F⟩.available` on to `Classifier2(F).controlReady`. Here we see that when the RT arrives at `Classifier2().controlReady` we know that it must be `dataReady` as well. We can remove the `Classifier2(F).dataReady` node. Now `Classifier2().controlReady` goes directly to `Classifier2().ready`. This is now a singleton and can be removed. These same two optimizations apply to all the subsequent classifiers in this example.

D. Dead computations

If a state element does not appear in any backward may set for any output to the env, then it is not necessary to compute it. This might be considered a bug, for example when an item collection should but does not get consumed by the environment. But it also might occur during application development when, for some reason, the output is intentionally restricted. Such state element need not be produced, and can be eliminated from the program.

E. Dead items

A system using DSA form has to deal effectively with dead data. We currently provide a get-count facility to track the number of gets performed on an item. The user provides a function that maps an item tag to the number of references it will have. For example, in a five point stencil, tiles in the middle will have a get count of five, those on an edge will have a get count of four, and corner tiles will have a get count of three. When an item is produced this function is invoked and the number of gets is associated with the item. When gets are executed the count is decremented. The item can be eliminated when the count reaches zero. While get counts are useful in many cases, their applicability is limited to programs where get count can be determined at the time of the put. If the get count depends on the paths taken in the program, then it cannot be used to collect dead items. For example, consider Figure 4. Here nodes A B and C are nodes in the application-specific IR. There is a path from node A to B and one from A to C. Let A represent a step that emits exactly one of 2 control tags. This is a one or the other fork. This information might be user-provided or it might come from analysis of the step code.

Let us assume that the following facts hold for this piece of the graph:

1) There is definitely a use of `x[i]` in the B path (Use in B is in the backward must set for B)
2) This use of `x[i]` always comes through A (A is in the backward must set for the use at B)
3) There is definitely not a use of `x[i]` on the C path (use is not in the backward may set of C)

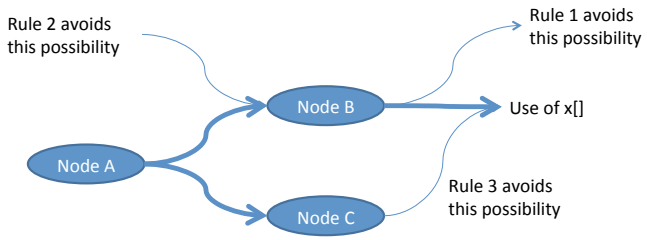

Fig. 4. Eliminating dead items

Traditional get counting does not help in this case, since the programmer does not know which path will be taken at the time the put of `x[i]` is done. But in the application-specific runtime, the get count for `x[i]` can get initialized to the conservative value (1 in our example). If the A-C path is taken, then the runtime can decrement the get count immediately. If the A-B path is taken, then the runtime will wait for the get to actually occur and decrement the count as usual. Either way, `x[i]` can get collected after the last use.

F. Cloning

If an optimization is possible on some paths through a node or region in the graph but not on all paths we can clone the region and optimize one path but not the other. Consider a variant of our face detector with a single classifier collection where the classifier number, instead of distinguishing among collections, apears as a tag component. The optimization we described above (removing dataReady) would not apply in this case because the classifier is not known to be dataReady. If we transform the classifier collection into two collections, `classifierFirst()` and `classifiersRest()`, then we can leave `classifierFirst().dataReady` but remove `classifierRest().dataReady`.

G. Node-specific primitive attributes

We can extend the idea of removing singletons. Consider an IR node corresponding to primitive attribute, put of an item processed resulting in `item[].available` node, put of a tag resulting in `tag⟨⟩.available` node, or the completion of a step resulting in `step().executed node`. As application-specific primitives, these actions could, in fact, continue the processing, for example, through `step().controlReady` and even through `step()ready`. This optimization should be applied last so that it can incorporate the results of the other transformations. There are some trade-offs to be considered here. It might not always be wise to perform this transformation.

H. Sceduling Communication

In systems with distributed memory, we can extend our model by defininng communication dependencies. For example, let's assume that : `x[]` is available on node 3 but not on node 4. We can add explicit communication operation and communication dependencies. "`x[]` is available on node 3" → "`x[]` is communicated from node 3 to node 4" → "`x[]`

is available on node 4". Because the communication is now explicit, we can decide to schedule it early or late depending maybe on memory pressure on each node. A runtime will also want to schedule communication in such a way as to balance communication and computation.

I. Speculative or Demand-Driven execution

By adding attributes such as "speculatively available" and "speculatively executed", we can support a runtime that executes a data-ready step that is not control-ready. Items produced by such a step would be speculatively available. Once the speculatively executed step becomes control-ready, it will be converted to executed step, and all the speculatively available items it produced will be converted to available.

For demand-driven execution, we can use the reverse situation. When a step becomes control-ready, but is not data-ready, we can mark the items it wants to read as "demanded". We can then use the "produced-by" tag function to find which steps should be producing those items and mark those as "demanded".

An optimized runtime system will give the highest priority to the demanded work, normal priority to the "normal" work (the steps that are both data-ready and control-ready), and lowest priority to the speculative work.

VII. Summary

A. Future work

Implementation of CnC on the Open Community Runtime, a deliverable for the DOE X-Stack project (exascale software stack) will be an generic attribute-driven runtime. On this foundation, we plan to develop the analyses and optimizations presented here. The lowered IR with explicit communication is very relevant in The X-Stack project.

That system will support a range of attribute-based tools that provide information at the level the domain expert is thinking, that is their steps, items and tags. These tools include support for visualization, performance monitoring, tracing, debugging, interactive execution, etc.

B. Related work

CnC itself draws from dataflow [8], tuple spaces [6] and task graphs [9]. Conceptually, CnC is close to iprogram dependence graphs [9] as an intermediate representation, than it is to other parallel languages. Our flow analyses and optimizations are similar in flavor to classical flow analyses and optimizations. The differences arise from the fact that we apply them to the higher level coordination graph rather than to lower level instructions.

C. Conclusions

In this paper, we introduced a runtime that views *attributes* (the changes in life stage of the CnC components) as explicit events to be processed by event handlers. This approach enables the compiler to create an application-specific intermediate representation by combining the application graph with the runtime events. This application-specific intermediate representation allows us to implement several analysis and optimization techniques, including removal of singleton events, eliminating redundant and dead computation and more precise garbage collection.

In addition to analyses and optimizations, this application-specific runtime supports user-defined attributes, checkpointing, demand-driven execution, as well as providing higher-level mechanisms for visualization, performance monitoring, debugging etc.

Acknowledgments

We would like to thank Frank Schlimbach, Vivek Sarkar, Sanjay Chatterjee and Sağnak Taşırlar for offering valuable comments and insights while we worked on this paper. This work was supported in part by the Department of Energy (Office of Science) under Award Number DE-SC0008717, and by the National Science Foundation Expedition in Computing Program, Award CCF-0926127.

References

[1] Intel concurrent collections. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc, 2012.

[2] Rice university cnc research project. https://wiki.rice.edu/confluence/display/HABANERO/CNC, 2012.

[3] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. Concurrent collections. *Scientific Programming*, 18:203–217, August 2010.

[4] Zoran Budimlić, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multicore Implementations of the Concurrent Collections Programming Model. In *CPC '09, Proceedings of the 2009 Workshop on Compilers for Parallel Computing*, 2009.

[5] Zoran Budimlić, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff N. Lowney, Vivek Sarkar, and Leo Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP'09: Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming*, pages 47–58. ACM, Jan 2009.

[6] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[7] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[8] Jack B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag.

[9] Jeanne Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[10] Dragoş Sbîrlea, Kathleen Knobe, and Vivek Sarkar. Folding of tagged single assignment values for memory-efficient parallelism. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 601–613, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Nick Vrvilo, Kathleen Knobe, and Vivek Sarkar. Execution frontiers as checkpoints in cnc. In *CnC-2012: The Fourth Annual Concurrent Collections Workshop*, 2012.