CG-DDF Domain

Authors:

Soonhoi Ha

1. Introduction

Warning: this code is experimental and unfinished; at this point demos of interesting parallel scheduling schemes for graphs with dynamic behavior are supplied, with little else.

All code generation domains included in this release assume that the dataflow graph is synchronous (or, SDF), that the number of tokens consumed and produced for each star is not varying at run time. In an SDF graph, therefore, we can determine the execution order of blocks at compile-time and minimize the runtime overhead of scheduling them. If we target a multiprocessor architecture, we furthermore assume that the relative execution times of blocks are specified. We do not allow blocks with dynamic behavior such as case construct, data-dependent iteration, and recursions. The **`CG-DDF`** domain is the code generation version of the **`DDF`** domain (so, called CG-DDF), to overcome this limitation. Using the CG-DDF domain inside a code generation domain, the user may want to use some predetermined dynamic constructs in the application graph. The CG-DDF domain always resides in a code generation domain as a wormhole: **`CGWormhole`**. Moreover, the target of the code generation domain should be a multiprocessor target.

The dynamic constructs we support in the CG-DDF domain are *case, for, do-while,* and *recursion*. The *case* construct is a generalization of the more familiar *if-then-else* construct. Unlike the simulation DDF domain, the graphical topologies of these dynamic constructs are enforced. In the simulation DDF domain, all connected SDF stars are collected to make a DDF wormhole automatically to reduce the runtime overhead by *quasi-static* scheduling. In the CG-DDF domain, however, the user should create CG-DDF wormholes explicitly, so that the graph in the CG-DDF domain consists of only CG-DDF stars and CG-DDF wormholes. The topology of the graph is identified with a predetermined topology of dynamic constructs supported by the **Ptolemy**.

2. Target

There is only one target in the CG-DDF domain, **`CGDDFTarget`**. The CGDDFTarget is derived from the **`MultiTarget`** class. Note that the CG-DDF domain does not change the physical target of the outside code generation domain. Also, the code generation domain inside the CG-DDF wormholes do not, either. Hence, we set a target parameter, **`inheritProcessors`**, **YES** meaning that the target inside a wormhole uses the same physical architecture (or child targets) as the outside domain. The targets of all code generation domains inside CG-DDF wormholes should be set this parameter to YES. If the parameter is set YES, the **`nprocs`** target parameter is ignored. Here is the reason why the outside domain of the CG-DDF domain should have a multiprocessor target: the target of the CG-DDF domain inherits the child targets of the outside target. A single processor target does not have any child target.

The CG-DDFTarget has a string parameter **`constructType`** to specify which dynamic construct the graph represents. The user has to specify one of the dynamic constructs supported: case, do-while, for, recursion. We examine the first character (type-insensitive) of the given string to decide the construct. By default it is set to "case". After the graph in the CG-DDF domain is confirmed to the topology of the specified construct, the scheduler is selected automatically.

3. Scheduler

Existence of dynamic constructs in a program graph keeps the user from using the static scheduling algorithms. Some of the static scheduling algorithms are implemented in the **Ptolemy**, and described in the scheduler section in the CG domain document. If an application has dynamic behavior, the conventional solution is to discard static scheduling and incur the substantial cost of dynamic scheduling. But, dynamic scheduling is not a must in most signal processing algorithms; thus a much simpler approach based on *quasi-static* scheduling is proposed. In quasi-static scheduling, most of the scheduling decisions are made at compile-time. Some scheduling decisions are made at run time, but only when absolutely necessary. Refer to [1] for detailed discussion of the scheduling scheme.

The scheduling idea is as follows; We first treat each dynamic construct as a special SDF star and use static scheduling algorithm. The SDF star from the dynamic construct is special in the sense that it may require more than one processors to be mapped onto, and the execution time on the assigned processors are varying at runtime (assumed fixed when we compute the schedule). Most conventional scheduling algorithms assume that a block is assigned to a processor. Therefore, we had to modify the scheduling algorithms to support the case when some blocks can be scheduled onto more than one processors. The scheduling results decide the assignment and ordering of blocks on the processors (child targets). At run time, we do not achieve the expected performance from the compile-time scheduling since dynamic constructs will behave differently from what we assumed at compile time. Sometimes, the dynamic construct will finish its execution earlier than expected, and sometimes later. Our goal is to minimize the expected makespan of the program graph at run time.

We assume that the run-time behavior of each dynamic construct is known or can be approximated with a certain probability distribution. For example, the number of iteration for *for* or *do-while* constructs is the variable. And the recursion depth is a variable of *recursion* construct. This information should be entered by the following CGDDF target parameters.

`paramType`: type of the distribution. Currently, we support "geometric" distribution, "uniform" distribution, and other "general" distribution specified by a table. By default, the geometric distribution is chosen.

`paramGeo`: geometric constant of a geometric distribution. Its value is effective only when the geometric distribution is selected as the **`paramType`**. If the construct is a *case* construct, this parameter indicates the probability of branch 1 being taken. The branch number stars from 0. Therefore, there are only two branches, the parameter indicates the probability of the "TRUE" branch being taken. In case there are more than two branches, we have to use **`paramFile`** parameter to specify the probabilities of taking each branch.

`paramMin`: minimum value for a uniform distribution. It is effective only when the uniform distribution is chosen.

paramMax: maximum value for a uniform distribution. It is effective only when the uniform distribution is chosen.

paramFile: file name that contains the information on the distribution. If the construct is a *case* construct, each line contains the value for the probability of taking a branch numbered from 0. Otherwise, each line contains the integer index value and the probability for that index. The indices should be increasing order. It is effective only when a *general* distribution is selected.

Based on the specified distribution of the run-time behavior, we determine the compile-time *profile*, shortly profile, of each dynamic construct. The profile of a dynamic construct consists of the number of processors assigned to the construct and the (assumed) execution times of the construct on the assigned processors. Suppose we have a *for* construct. If the loop body is scheduled with one processor, it takes 6 time units. With two processors, the loop body takes 3 and 4 time units respectively. Moreover, each iteration cycle can be paralleled with 1 time unit skewed. There are four processors. Then, we have to determine how many processors to be assigned for the construct, and how many times the loop body will be scheduled at compile time. Assign two processors to the loop body and parallelize two iteration cycles, thus taking all 4 processors? Or, assign one processor to the loop body and parallelize three iteration cycles, thus taking 3 processors as a whole? We have developed a systematic approach to answer these tricky scheduling problems based on the distribution information [1]. We can manually determine the number of assigned processors by setting **fixedNum** parameter of the CG-DDF target. Note that we still have to decide how to schedule the dynamic construct with the given number processors.

Since the gantt chart program currently implemented can not show the schedule inside the CG-DDF wormhole, we just show the "profile" of the dynamic construct. The outside code generation domain uses that profile information of each dynamic construct (or CGWormhole) for overall static scheduling.

4. CGDDF Stars

The CGDDF stars are the key for identifying dynamic constructs. For example, the **Case** and the **EndCase** stars are used in the *case*, *do-while*, or *recursion* construct, which differ each other by the connection topology of these CGDDF Stars and CGDDFWormholes. Therefore, if the user wants to use one of these dynamic constructs, no need for writing a star exists.

The *for* construct consists of a pair of an **UpSample** type star and a **DownSample** type star, where UpSample and DownSample are not the star name but the type of some stars. If a star produces more than consumes, it is called an UpSample star. In the preprocessor file, we define a method **readTypeName**, as shown below:

```
method {
        name { readTypeName }
        access { public }
        type { "const char *" }
        code { return "UpSample"; }
}
```

The examples of UpSample type stars are **Repeater** and **DownCounter**. The Repeater star has two inputs. One input receives a control value specifying how many times the star repeats the value of the other input to the output. The DownCounter star receives a positive integer from the input and produces down-counted values to the output. The number of tokens produced from both stars is data-dependent. On the other hand, we can design a DownSample star that has the following method:

```
method {
        name { readTypeName }
        access { public }
        type { "const char *" }
        code { return "DownSample"; }
}
```

One example of DownSample type star is **LastOfN**. The LastOfN star has two inputs of which the control input reads the value N. And, the star receives N inputs and send the last input received to the output. Thus, the number of tokens consumed is data-dependent.

As explained above, all customized CG-DDF stars will be either UpSample type or DownSample type. And, we do not expect that a casual user need to write a new CG-DDF star if we provide some representative UpSample and DownSample stars. Currently, we haven't finished code generation part in this domain. So, the CG-DDF stars are comment generators in this release.

5. Status

In this release, we include demos showing only the scheduling result, not the generated code. Code generation part of this domain has not been completed yet.

References

[1] S. Ha, "Compile-Time Scheduling of Dataflow Program Graphs with Dynamic Constructs", Ph.D. dissertation, U.C.Berkeley, 1992.

CG-DDF Stars

All standard CG-DDF stars do not generate codes in this release. The source code for all standard CGDDF stars is in $PTOLEMY/src/domains/cg-ddf/stars. By "standard" we mean the stars that are owned by **Ptolemy**.

NAME: `Case`

This star routes an "input" token to one "output" depending on the "control" token.

LOCATION: CGDDF demo library

DOMAIN: `CGDDF` **(DERIVED FROM:** `CGDDFStar`)

VERSION: 1.3 (12/10/92)

AUTHOR: *Soonhoi Ha*

INPUTS: *input* (ANYTYPE)
control (int)

OUTPUTS: *output* (multiple), (ANYTYPE)

NAME: `DownCounter`

A down counter from the input value to zero.

LOCATION: CGDDF demo library

DOMAIN: `CGDDF` **(DERIVED FROM:** `CGDDFStar`)

VERSION: 1.3 (12/10/92)

AUTHOR: *Soonhoi Ha*

INPUTS: *input* (int)

OUTPUTS: *output* (int)

DESCRIPTION:

This star generates (int-1), (int-2), ... 1,0 samples for (int) input.

NAME: `EndCase`

Depending on the "control" input, route an "input" to the "output".

LOCATION: CGDDF demo library

DOMAIN: `CGDDF` **(DERIVED FROM:** `CGDDFStar`)

VERSION: 1.3 (12/10/92)

AUTHOR: *Soonhoi Ha*

INPUTS: *input* (multiple), (ANYTYPE)
control (int)

OUTPUTS: *output* (ANYTYPE)

NAME: **`Fork`**

Copies input particles to each output.

LOCATION: CGDDF demo library

DOMAIN: **`CGDDF`** **(DERIVED FROM: `CGDDFStar`)**

VERSION: 1.3 (12/10/92)

AUTHOR: *Soonhoi Ha*

INPUTS: *input* (ANYTYPE)

OUTPUTS: *output* (multiple), (ANYTYPE)

DESCRIPTION:

This star is generally used to connect a single output port to multiple input ports. It will be automatically inserted when multiple inputs are connected to the same output using the graphical interface, or when the "nodeconnect" command is used in the interpreter. However, there are times when automatically inserted Fork stars are not desirable. For instance, when there is a delay on one of the arcs, then the Fork must be inserted by the user explicitly to avoid ambiguity about the location of the delay. Also, when multi-portHoles are used, auto-forking can cause problems. In this situation, one may get, for example, two outputs and several inputs on the same net. There is currently no way to automatically decipher what the user intends. Hence, the Fork star should be inserted explicitly.

NAME: **`LastOfN`**

Outputs the last token of N input tokens, where N is the control input.

LOCATION: CGDDF demo library

DOMAIN: **`CGDDF`** **(DERIVED FROM: `CGDDFStar`)**

VERSION: 1.3 (12/10/92)

AUTHOR: *Soonhoi Ha*

INPUTS: *input* (ANYTYPE)
control (int)

OUTPUTS: *output* (ANYTYPE)

NAME: **`Repeater`**

This star uses the "control" input value to specify how many times to repeat the "input" value on "output".

LOCATION: CGDDF demo library

DOMAIN: **`CGDDF`** **(DERIVED FROM: `CGDDFStar`)**

VERSION: 1.3 (12/10/92)

AUTHOR: *Soonhoi Ha*

INPUTS: *input* (ANYTYPE)
control (int)

OUTPUTS: *output* (ANYTYPE)

NAME: **`Self`**

Realizes recursion.

LOCATION: CGDDF demo library

DOMAIN: **`CGDDF`** **(DERIVED FROM: `CGDDFStar`)**

VERSION: 1.3 (12/10/92)

AUTHOR: *Soonhoi Ha*

INPUTS: *input* (multiple), (ANYTYPE)

OUTPUTS: *output* (multiple), (ANYTYPE)

DESCRIPTION:

This is a star for recursion. At runtime, it appears to contain a clone of the galaxy whose name is given by "recurGal". That galaxy is supplied with the input particles and executed. That galaxy may be one within which this instance of Self resides, hence realizing recursion. At compile time, this star appears to just be an atomic star.

SEE ALSO: fibonnacci.

CG-DDF Demos

NAME: **`demoCase1`**

Demonstration of the quasi-static scheduler for a graph with a "case" construct.

VERSION: 1.1 (2/6/92)

AUTHOR: *S. Ha*

LOCATION: ˜ptolemy/src/domains/cg-ddf/demo

DESCRIPTION:

This demo displays a schedule result applying a quasi-static scheduling technique to a program graph with a *case* construct. Since there are two inputs to the **`Case`** star, this *case* construct is nothing but an *if-then-else* construct. By setting the target parameter of the CG-DDF domain inside the CGWormhole (green block at the top level), we indicate that it is a *case* construct (**`constructType`** = case) with the probability 0.5 to select the "TRUE" branch (**`paramType`** = geometric, and **`paramGeo`** = 0.5). Since the **`fixedNum`** state is 0, we let the scheduler decide the number of processors to assign to the case construct.

The CG-DDF galaxy contains two wormholes which contain CG domains inside to represent the "TRUE" and the "FALSE" branches. The outside CG domain and these inner CG domains share the same child targets by setting **`inheritProcessors`** target parameter "YES" in the inner targets. The quasi-static scheduling idea is that we fix the assignment and the execution order of stars in the CG targets at the compile time. At the top level, the "case" construct is regarded as an atomic block. The blocks inside the "TRUE" and the "FALSE" branches are also scheduled (assigned and ordered) at compile-time. At run time, based on the control value to the **`Case`** star, we either execute the TRUE branch or the FALSE branch.

The core of the scheduling technique is to determine the assumed execution profile of the "if-then-else" construct at compile-time, based on the probability of the "TRUE" branch being taken, In the displayed Gantt chart we can find out what is the estimated profile of the construct. The construct is assigned to three processors among four processors. Schedules of the "TRUE" and "FALSE" branches are not displayed. Note that we pipeline the program graph by inserting delays on a cutset manually to increase the possibility of parallelism. In this release we do not make code generation working yet.

NAME: **`demoDo1`**

Demonstration of the quasi-static scheduler for a graph with a "do-while" construct.

VERSION: 1.1 (2/6/92)

AUTHOR: *S. Ha*

LOCATION: ˜ptolemy/src/domains/cg-ddf/demo

DESCRIPTION:

This demo displays a schedule result applying a quasi-static scheduling technique to a program graph with a *do-while* construct. A *do-while* construct, which resides in a CGWormhole, consists of a **`EndCase`** star at the front, **`Case`** star at the back, and a wormhole that represents a loop-body. The control value to the **`Case`** star is generated from the loop-body, which decides whether to execute the loop-body once more or quit. The number of iteration cycles of the *do-while* loop is, therefore, data-dependent

By setting the target parameter of the CG-DDF domain inside the CGWormhole (green block at the top level), we indicate that it is a *do-while* construct (**`constructType`** = dowhile) and the number of iteration cycles is distributed geometrically (**`paramType`** = geometric) with a constant 0.7 (**`paramGeo`** = 0.7). The probability of doing one more iteration is 0.7 after finishing the current iteration cycle. Since the **`fixedNum`** state is 0, we let the scheduler decide the number of processors to assign to the for construct.

The CG-DDF galaxy contains a wormhole of CG domain to represent the body of the loop. The outside CG domain and this inner CG domain share the same child targets by setting **`inheritProcessors`** target parameter "YES" in the inner target. The quasi-static scheduling idea is that we fix the assignment and the execution order of stars in the CG targets at the compile time. At the top level, the "do-while" construct is regarded as an atomic block. The blocks inside the loop-body are also scheduled (assigned and ordered) at compile-time. At run time, we execute the loop-body as many times as the termination condition fails.

The core of the scheduling technique is to determine the assumed execution profile of the *do-while* construct at compile-time, based on the distribution of the number of iteration cycles. In the displayed Gantt chart, we can find out what is the estimated profile of the construct. The construct is assigned to 2 processors among 4. Unlike the "for" construct, in the "do-while" loop, intercycle parallelism does not exist. The assumed number of iteration cycles is 2 in this case. Note that we pipeline the program graph by inserting delays on a cutset manually to increase the possibility of parallelism. In this release we do not make code generation working properly yet.

NAME: **`demoFor1`**

Demonstration of the quasi-static scheduler for a graph with a "for" construct.

VERSION: 1.1 (2/6/92)

AUTHOR: *S. Ha*

LOCATION: ˜ptolemy/src/domains/cg-ddf/demo

DESCRIPTION:

This demo displays a schedule result applying a quasi-static scheduling technique to a program graph with a *for* construct. A *for* construct, which resides in a CGWormhole, consists of a **`Repeater`** star at the front, **`LastOfN`** star at the back, and a wormhole that represents a loop-body. The number of iteration cycles of the *for* loop depends on the control input to the **`Repeater`** star. By setting the target parameter of the CG-DDF domain inside the CGWormhole (green block at the top level), we indicate that it is a *for* construct (

constructType = for) and the number of iteration cycles is distributed uniformly (**paramType** = uniform) from 1 to 15 (**paramMin** = 1, **paramMax** = 15). Since the **fixedNum** state is 0, we let the scheduler decide the number of processors to assign to the for construct.

The CG-DDF galaxy contains a wormhole of CG domain to represent the body of the loop. The outside CG domain and this inner CG domain share the same child targets by setting **inheritProcessors** target parameter "YES" in the inner target. The quasi-static scheduling idea is that we fix the assignment and the execution order of stars in the CG targets at the compile time. At the top level, the "for" construct is regarded as an atomic block. The blocks inside the loop-body are also scheduled (assigned and ordered) at compile-time. At run time, we execute the loop-body as many times as the control value that is fed to the **Repeater** star.

The core of the scheduling technique is to determine the assumed execution profile of the *for* construct at compile-time, based on the distribution of the number of iteration cycles. In the displayed Gantt chart, we can find out what is the estimated profile of the construct. The construct is assigned to all four processors. Each loop-body is executed in a single processor, and four loops are parallelized since the loop-body does not possess any intercycle dependency. The assumed execution times on the processors are not the same on the assigned processors since the assumed number of iteration cycles is 3 (or 7) in this case. Note that we pipeline the program graph by inserting delays on a cutset manually to increase the possibility of parallelism. In this release we do not make code generation working properly yet.

NAME: **demoRecur1**

Demonstration of the quasi-static scheduler for a graph with a "recursion" construct.

VERSION: 1.1 (2/6/92)

AUTHOR: *S. Ha*

LOCATION: ˜ptolemy/src/domains/cg-ddf/demo

DESCRIPTION:

This demo displays a schedule result applying a quasi-static scheduling technique to a program graph with a *recursion* construct. A *recursion* construct, which resides in a CGWormhole, contains two **Self** stars, which will be expanded as the same recursion construct if executed. The control input to the **Case** star decides whether to execute **Self** stars. If the **Self** stars are not executed, the recursion construct at that level is terminated. The recursion body resides in the CGDDF wormholes, 5 and 6. Since there are two Self stars, the number of execution of the recursion body grows exponentially as the depth of the recursion. Therefore, the recursion construct is highly parallelizable.

By setting the target parameter of the CG-DDF domain inside the CGWormhole (green block at the top level), we indicate that it is a *recursion* construct (**constructType** = recur) and the depth of recursion is distributed uniformly (**paramType** = uniform) from 1 to 4 (**paramMin** = 1, **paramMax** = 4). By regarding the depth of recursion as the parameter for the distribution of the run-time behavior, we approximate the situation that all recursion branches are identically distributed, or all synchronized (have the same termination condition). Since the **fixedNum** state is 0, we let the scheduler decide the number of processors to assign to the recursion construct.

The CG-DDF galaxy contains wormholes of CG domain that share the same child targets as the outside CG domain by setting **`inheritProcessors`** target parameter "YES". The quasi-static scheduling idea is that we fix the assignment and the execution order of stars in the CG domains at the compile time. At the top level, the "recursion" construct is regarded as an atomic block. The blocks inside the CGDDF wormholes of CG domain are also scheduled (assigned and ordered) at compile-time. At run time, we determine which wormholes to execute depending on the control value fed to the **`Case`** star.

The core of the scheduling technique is to determine the assumed execution profile of the *recursion* construct at compile-time, based on the distribution of the depth of recursion. In the displayed Gantt chart, we can find out what is the estimated profile of the construct. The construct is assigned to four processors among 5 processors. Each recursion body is executed in a single processor, and four recursion branches are parallelized. Once all four recursion branches are parallelized, the corresponding recursion constructs are executed sequentially. Note that we pipeline the program graph by inserting delays on a cutset manually to increase the possibility of parallelism. In this release we do not make code generation working properly yet.