Chapter 17. Target

Authors: *Soonhoi Ha*

Other Contributors: *John S. Davis II*

Target has a clear meaning in code generation domains, a model of the target machine for which code will be generated. Class CGTarget is the base class for all code generation targets whether it is a single processor target or a multiprocessor target. Class MultiTarget, derived from class CGTarget, serves as the base target for all multiprocessor targets. For single processor targets, we have AsmTarget and HLLTarget to distinguish assembly code generation targets and high level language generation targets. If we generate assembly code for a target, the target will be derived from class AsmTarget. If we generate a high level language code, the target will be derived from HLLTarget. For detailed discussion for Target hierarchy, refer to [4] in References .

In this chapter, we will describe class CGTarget and some base multiprocessor targets since we focus on multiprocessor code generation. Refer to other sources for AsmTarget and other high level language targets.

17.1 Class CGTarget

Class CGTarget is derived from class Target. It has a four-argument constructor.

```
CGTarget(const char* name, const char* starclass,
      const char* desc, char sep);
```

The first argument is the name of the target and the second argument is the star class that this target can support. The third argument is the description of this target. The last one is a separator character for unique symbol generation.

There are two protected states in the CGTarget:

```
StringState destDirectory;
IntState loopingLevel;
```

The first state indicates where to put the code file. The second state determines which scheduler is used in case this target is a single processor target. By default, *loopingLevel* = 0 and we do not try looping. If *loopingLevel* = 1, we select Joe's loop scheduler. Otherwise, we use the most complicated loop scheduler.

At the top level, three methods of the Target class are called in sequence: `setup, run,` and `wrapup.`

```
void setup();
```

In this method, we do the following tasks:

(1) Initialize `myCode` and `procedure` code stream.

(2) Select a scheduler if no scheduler is selected yet.

At this stage, we check whether the galaxy is assigned or not. In multiprocessor targets, a child target is not assigned a galaxy until a sub-univers is created. If the galaxy is not assigned, return.

(3) Reset the symbol lists.

(4) If we are the top-level target, initialize the galaxy (which performs preinitialization and initialization of the member blocks, including HOF star expansion and porthole type resolution). Then modify the galaxy if necessary by calling *modifyGalaxy*. The base class implementation of modifyGalaxy splices in type conversion stars where needed, if the domain has supplied a table of type conversion stars. (If there is no table, type conversion is presumed not needed. If there is a table, and a type mismatch is found that the table has no entry to fix, then an error is reported.) Some derived domains redefine *modifyGalaxy* to perform additional transformations. For example, in AsmTarget we insert some stars (CircToLin, LinToCirc) at loop boundaries to change the type of buffer addressing in case a loop scheduling is performed.

```
virtual int modifyGalaxy();
```
Is a protected method.

(5) If it is a child target, the schedule was already made at this stage from a parallel scheduler of the parent multiprocessor target. Otherwise, we initialize and schedule the graph.

(6) If it is a child target or it is not inside a wormhole, return. Otherwise, we first adjust the sample rate of the wormhole portholes (`adjustSampleRates`). Then, we generate and download code: `generateCode` and `wormLoadCode`.

```
void adjustSampleRates();
```
This method is a protected method to be called when this target is inside a wormhole. After scheduling is performed, we need to multiply the sample rate of wormhole portholes by the repetition count of the stars inside the wormhole connected to the porthole.

```
virtual void generateCode();
```
This method guides the overall procedure to generate code for single processor targets. The procedure is as follows:

(1) If this target is a child target, call `setup` to initialize the variables. Copy the symbol counter (`symbolCounter`) of the parent target to the symbol counter of this target to achieve a unique symbol in the system scope.

(2) We compute buffer sizes, allocate memory, etc: `allocateMemory`.

```
virtual int allocateMemory();
```
This method is protected. It does nothing and returns TRUE in this base class.

(3) Call the method `generateCodeStreams()`. This method will be described later.

(4) Organize the CodeStreams into a single code stream and save the result to the `myCode`

stream: `frameCode.`

```
virtual void frameCode();
```

This method is a protected method. It does nothing in this base class.

(5) If this target is not a child target, write the generated code to a file: `writeCode.`

```
virtual void writeCode(const char* name = NULL);
```

This is a public method to write the `myCode` stream to the argument file. If no argument is given, use "code.output" as the default file name.

(6) If it is a child target, copy the symbol counter to that of the parent target.

The methods described above for code generation are all virtual methods. They will be redefined in the derived targets.

The method

```
void CGTarget::generateCodeStreams();
```

does the following things:

(1) Write initial code.

```
virtual void headerCode();
```

In this base class, this protected method writes the header comment to the `myCode` CodeStream.

```
virtual StringList headerComment(const char* begin = NULL,
        const char* end = "", const char* cont = NULL);
```

This method is a public virtual method to generate the header comment in the code. In this base class, the head comments include the user id, code creation date, target name, and the galaxy name. The arguments are passed to the `comment` method.

```
virtual StringList comment(const char* cmt, const char* begin =
NULL, const char* end = "", const char* cont = NULL);
```

This public method generates a comment from a specified string *`cmt.`* We prepend *`begin`* and append *`end`* to the string. If *`begin`* is NULL, we prepend '#' as a shell-stype comment. If *`cont`* is specified, multi-line comments are supported.

(2) We do initialization for code generation: for example, compute offsets of portholes and call `initCode` methods of stars: `codeGenInit`.

```
virtual int codeGenInit();
```

is a protected method. It does nothing and returns TRUE in this base class.

(3) Generate the code for the main loop: `mainLoopCode.`

```
virtual void mainLoopCode();
```

In this method we first compute the number of iterations. If this target is inside a wormhole, the number is -1 indicating an infinite loop. Otherwise, the `stopTime` of the scheduler determines the number of iterations. In this base class, we call the following five methods sequentially: `beginIteration, wormInputCode` if inside a wormhole, `com-`

pileRun, wormOutputCode if inside a wormhole, and endIteration. In the derived class, this sequence may be changed .

```
void beginIteration(int numiter, int depth);
void endIteration(int numiter, int depth);
```

These public methods form the head or ending of the main loop. The arguments of both methods are the number of iteration and the depth of the loop. In the main loop, the depth is set 0.

```
virtual void wormInputCode();
virtual void wormOutputCode();
virtual void wormInputCode(PortHole& p);
virtual void wormOutputCode(PortHole& p);
```

The above methods are all public. They generate code at the wormhole boundary if the target resides in a wormhole. The last two methods generate code for the argument porthole that is at the wormhole boundary. In this base class, put comments in myCode CodeStream indicating that the methods are successfully executed. They should be redefined in the derived classes to be useful. The first two methods traverse all portholes at the wormhole boundary to use the last two methods.

```
virtual void compileRun(SDFScheduler* sched);
```

This protected method calls compileRun of the argument scheduler. By default, this method calls go methods of all stars in the scheduled order to generate code in myCode CodeStream.

(4) Call wrapup methods of stars to generate code after the main loop, but still inside the main function.

(5) Add more code if necessary: trailerCode

```
virtual void trailerCode();
```

This protected method does nothing in this base class.

The method

```
virtual int wormLoadCode();
```

is protected. It downloads code to the target machine and starts executing it if the target resides in a wormhole. In this base class, we just display the code.

Now, we discuss the run method.

```
int run();
```

If the target is not inside a wormhole, it generates code by calling generateCode as explained above. Otherwise, we do the transfer of data to and from the target since this method will be called when the wormhole is executed: sendWormData and receiveWormData in sequence.

```
virtual int sendWormData();
virtual int receiveWormData();
virtual int sendWormData(PortHole& p);
virtual int receiveWormData(PortHole& p);
```

The above methods are all protected. They send and receive samples to this target when run inside a wormhole. The argument is the porthole of the interior star at the wormhole boundary. If no argument is given, send and receive for all the appropriate portholes. In this base class, we generate comments to indicate that these methods are successfully called.

`void wrapup();`
In derived classes, wrapup will generate code to finalize, download, and run the code. This CGTarget class just displays the code.

So far, we have explained the three top level methods of the CGTarget class. Methods related to the CodeStream and unique symbol generations can be found in the previous chapter. We will describe the remaining members.

17.1.1 Other CGTarget protected members

`char* schedFileName;`
The name of the log file in case a loop scheduling is taken. By default, the name is set to "`schedule.log`".

`int noSchedule;`
This is a flag to be set to TRUE if scheduling is not needed in the setup stage. This flag will be set when the schedule is copied from `copySchedule` method in parallel code generation. By default, this flag is set FALSE.

`StringList indent(int` *`depth`*`);`
This method returns a list of spaces for indenting. The number of spaces is 4 per each *`depth.`*

`void switchCodeStream(Block*` *`b`*`, CodeStream*` *`s`*`);`
This method is set to the current `myCode` pointer of the argument block *`b`* to *`s`* CodeStream. If *`b`* is a galaxy, perform this for all component stars.

17.1.2 Other CGTarget public members

`static int haltRequested();`
Returns TRUE if error is signaled while Ptolemy is running.

`int inWormhole();`
`int isA(const char*` *`class`*`);`
Is a standard `isA` method for type identification.

Returns TRUE or FALSE, based on whether the target is inside a wormhole or not.

`Block* makeNew() const;`
Create a new, identical CGTarget. Internal variables are not copied.

`virtual int incrementalAdd(CGStar*` *`s`*`, int` *`flag`* `= 1);`
This method is called when we add code for the argument star *`s`* incrementally. If *`flag`* is 1 (default), we allocate memory for the star, and call `setup, initCode, go,` and `wra-`

pup of the star. If *flag* is 0, we just call go method of that star. In this base class, generate an error message.

`virtual int insertGalaxyCode(galaxy* g, SDFScheduler* sched);`
This method inserts the code for the argument galaxy *g* incrementally. We have to allocate resources and generate initialization, main loop, and wrapup code. It is used to generate code for the galaxy inside a dynamic construct. A dynamic construct is a wormhole in the code generation domain. When we call the go method of the wormhole, we generate code for the inside galaxy.

```
virtual int compileCode();
virtual int loadCode();
virtual int runCode();
```
These methods compile and load the code, and run the target. The base class, generates error messages.

`void writeFiring(Star& s, int depth);`
This method generates code for a firing of the argument star. The base class simply executes run of the star.

```
void genLoopInit(Star& s, int reps);
void genLoopEnd(Star& s);
```
In case loop scheduling is taken, we may want to perform loop initialization routines for stars inside each loop. These methods call beginLoop and endLoop methods of the argument star.

`void copySchedule(SDFSchedule& sched);`
If this is a child target, the schedule is inherited from the parallel scheduling of the parent target. This method copies the argument schedule to the schedule of this target and set noSchedule flag.

`virtual int systemCall(const char* cmd, const char* error = NULL, const char* host ="localhost");`
This method makes a system call using rshSystem utility function. If *error* is specified and the system call is unsuccessful, display the error message.

`void amInherited();`
This method declares that this target is inherited from other targets.

`virtual int support(Star* s);`
Returns TRUE if this target allows the argument star; returns FALSE otherwise.

`virtual int execTime(DataFlowStar* s, CGTarget* t = 0);`
We return the execution time of the argument star *s* in the argument target *t.* In a heterogeneous system, execution time of a given star may vary depending on which target executes the star. In this base class, we just call myExecTime method of the star.

17.1.3 Class HLLTarget

Class HLLTarget, derived from CGTarget class, is a base class of all high level language code generation targets. There is AsmTarget class for the base target of all assembly code generation targets. Since we will illustrate the C code generation target, we will explain the HLLTarget class only in this subsection.

HLLTarget class has a constructor with three arguments as CGTarget class. In this base class, we provide some methods to generate C++ code. The following three protected methods are defined to create a C++ identifier, derived from the actual name.

```
StringList sanitize(const char* s) const;
StringList sanitizedName(const NamedObj& b) const;
virtual StringList sanitizedFullName(const NamedObj& b) const;
```

The first method takes a string argument and modifies it with a valid C++ identifier. If the string contains a non-alphanumeric character, it will replace it with '_'. If the string starts with a number, it prepends 'x' at the beginning. The second method calls the first method with the name of the argument object. The third method generates an identifier for the argument object that will be placed in `struct` data structure. Therefore, we put '.' between the object name and its parent name.

Some public methods are defined.

```
void beginIteration(int repetitions, int depth);
void endIteration(int repetitions, int depth);
```

If the *repetitions* is negative, we print a `while` loop with infinite repetition. Otherwise, we generate a `for` loop. The second argument *depth* determines the amount of indent we put in front of the code.

```
void wrapup();
```

Saves the generated code to "code.output" file name.

Since this target is not an actual target, it has a pure virtual method: `makeNew`.

17.2 Multiprocessor Targets

There are two base multiprocessor targets: MultiTarget and CGMultiTarget. Class MultiTarget, derived from class CGTarget, serves a base multiprocessor target for CG domain. On the other hand, CGMultiTarget class is the base multiprocessor target for CG domain, thus derived from MultiTarget class. Since the MultiTarget class is a pure virtual class, the derived classes should redefine the pure virtual methods of the class.

Some members only meaningful for CG domain are split to MultiTarget class and the CGMultiTarget class. If they are accessed from the parallel scheduler, some members are placed in MultiTarget class. Otherwise, they are placed in CGMultiTarget class (Note that this is the organization issue). Refer to the CGMultiTarget class for detailed descriptions.

17.2.1 Class MultiTarget

Class MultiTarget, derived from CGTarget, has a constructor with three arguments.

`MultiTarget(const char* name, const char* starclass, const char* desc);`
The arguments are the name of the target, the star class it supports, and the description text. The constructor hides `loopingLevel` parameter inherited from the CGTarget class since the parallel scheduler does no looping as of now.

`IntState nprocs;`
This protected variable (or state) represents the number of processors. We can set this state, and also change the initial value, via the following public method:

`void setTargets(int num);`
After child targets are created, the number of child targets is stored in the following protected member:

`int nChildrenAlloc;`
There are three states, which are all protected, to choose a scheduling option.

```
IntState manualAssignment;
IntState oneStarOneProc;
IntState adjustSchedule;
```
If the first state is set to YES, we assign stars manually by setting `procId` state of all stars. If `oneStarOneProc` is set to YES, the parallel scheduler puts all invocations of a star into the same processor. Note that if manual scheduling is chosen, `oneStarOneProc` is automatically set YES. The last state, `adjustSchedule,` will be used to override the scheduling result manually. This feature has not been implemented yet. There are some public methods related to these states:

```
int assignManually();
int getOSOPreq();
int overrideSchedule();
void setOSOPreq(int i);
```
The first three methods query the current value of the states. The last method sets the current value of the `oneStarOneProc` state to the argument value.

There are two other states that are protected:

```
IntState sendTime;
IntState inheritProcessors;
```
The first state indicates the communication cost to send a unit sample between nearest neighbor processors. If `inheritProcessors` is set to YES, we inherit the child targets from somewhere else by the following method.

`int inheritChildTargets(Target* mtarget);`
This is a public method to inherit child targets from the argument target. If the number of processors is greater than the number of child targets of *`mtarget,`* this method returns FALSE with error message. Otherwise, it copies the pointer to the child targets of *`mtarget`* as its child targets. If the number of processors is 1, we can use a single processor tar-

get as the argument. In this case, the argument target becomes the child target of this target.

```
void enforceInheritance();
int inherited();
```
The first method sets the initial value of the `inheritProcessors` state while the second method gets the current value of the state.

```
void initState();
```
Is a redefined public method to initialize the state and implements the precedence relation between states.

Other MultiTarget public members

```
virtual DataFlowStar* createSpread() = 0;
virtual DataFlowStar* createCollect() = 0;
virtual DataFlowStar* createReceive(int from, int to, int num) = 0;
virtual DataFlowStar* createSend(int from, int to, int num) = 0;
```
These methods are pure virtual methods to create Spread, Collect, Receive, and Send stars that are required for sub-universe generation. The last two method need three arguments to tell the source and the destination processors as well as the sample rate.

```
virtual void pairSendReceive(DataFlowStar* snd, DataFlowStar* rcv);
```
This method pairs a Send, *snd,* and a Receive, *rcv,* stars. In this base class, it does nothing.

```
virtual IntArray* candidateProcs(ParProcessors* procs, DataFlowStar* s);
```
This method returns the array of candidate processors which can schedule the star *s.* The first argument is the current ParProcessors that tries to schedule the star . This class does nothing and returns NULL.

```
virtual Profile* manualSchedule(int count);
```
This method is used when this target is inside a wormhole. This method determines the processor assignments of the Profile manually. The argument indicates the number of invocations of the wormhole.

```
virtual void saveCommPattern();
virtual void restoreCommPattern();
virtual void clearCommPattern();
```
These methods are used to manage the communication resources. This base class does nothing. The first method saves the current resource schedule, while the second method restores the saved schedule. The last method clears the resource schedule.

```
virtual int scheduleComm(ParNode* node, int when, int limit = 0);
```
This method schedules the argument communication node, *node,* available at *when.* If the target can not schedule the node until *limit,* return -1. If it can, return the schedule time. In this base class, just return the second argument, *when,* indicating that the node is scheduled immediately after it is available to model a fully-connected interconnection of processors.

`virtual ParNode* backComm(ParNode*` *node*);
For a given communication node, find a communication node scheduled just before the argument node on the same communication resource. In this base class, return NULL.

```
virtual void prepareSchedule();
virtual void prepareCodeGen();
```
These two methods are called just before scheduling starts, and just before code generation starts, to do necessary tasks in the target class. They do nothing in this base class.

17.2.2 Class CGMultiTarget

While class CGMultiTarget is the base multiprocessor target for all code generation domains, either homogeneous or heterogeneous, it models a fully-connected multiprocessor target. In the target list in pigi, "FullyConnected" target refers to this target. It is defined in `$PTOLEMY/src/domains/cg/targets` directory. It has a constructor with three argument like its base class, MultiTarget.

To specify child targets, this class has the following three states.

```
StringArrayState childType;
StringArrayState resources;
IntArrayState relTimeScales;
```
The above states are all protected. The first state, `childType,` specifies the names of the child targets as a list of strings separated by a space. If the number of strings is fewer than the number of processors specified by `nproc` parameter, the last entry of `childType` is extended to the remaining processors. For example, if we set `nproc` equal to 4 and `childType` to be "default-CG56[2] default-CG96", then the first two child targets become "default-CG56" and the next two child targets become "default-CG96".

The second state, `resources,` specifies special resources for child targets. If we say "0 XXX ; 3 YYY", the first child target (index 0) has XXX resource and the fourth child (index 3) has YYY resource. Here ';' is a delimeter. If a child target (index 0) has a `resources` state already, XXX resource is appended to the state at the end. Note that we can not edit the states of child targets in the current pigi. If a star needs a special resource, the star designer should define `resources` StringArrayState in the definition of the star. For example, a star S is created with `resources` = YYY. Then, the star will be scheduled to the fourth child. One special resource is the target index. If `resources` state of a star is set to "2", the star is scheduled to the third target (index 2).

The third state indicates the relative computing speed of the processors. The number of entries in this state should be equal to the number of entries in `childType.` Since we specify the execution of a star with the number of cycles in the target for which the star is defined, we have to compensate the relative cycle time of processors in case of a heterogeneous target environment.

Once we specify the child targets, we select a scheduler with appropriate options. States inherited from class MultiTarget are used to select the appropriate scheduling options. In the CGMultiTarget class, we have the following three states, all protected, to choose a

scheduler unless the manual scheduling option is taken.

`IntState ignoreIPC;`
`IntState overlapComm;`
`IntState useCluster;`
The first state indicates whether we want to ignore communication overhead in scheduling or not. If it says YES, we select the Hu's Level Scheduler . If it says NO, we use the next state, `overlapComm.` If this state says YES, we use the dynamic level scheduler . If it says No, we use the last state, `useCluster.` If it says YES, we use the declustering algorithm . If it says NO, we again use the dynamic level scheduler. By default, we use the dynamic level scheduler by setting all states NO. Currently, we do not allow communication to be overlapped with computation. If more scheduling algorithms are implemented, we may need to introduce more parameters to choose those algorithms.

There are other states that are also protected.

`StringState filePrefix;`
Indicates the prefix of the file name generated for each processor. By default, it is set to "code_proc", thus creating code_proc0, code_proc1, etc for code files of child targets.

`IntState ganttChart;`
If this state says YES (default), we display the Gantt chart of the scheduling result.

`StringState logFile;`
Specifies the log file.

`IntState amortizedComm;`
If this state is set to YES, we provide the necessary facilities to packetize samples for communication to reduce the communication overhead. These have not been used nor tested yet.

Now, we discuss the three basic methods: `setup,` run, wrapup.

`void setup();`

(1) Based on the states, we create child targets and set them up: `prepareChildren.`
`virtual void prepareChildren();`
This method is protected. If the children are inherited, it does nothing. Otherwise, it clears the list of current child targets if they exist. Then, it creates new child targets by `createChild` method and give them a unique name using `filePrefix` followed by the target index. This method also adjusts the `resources` parameter of child targets with the `resources` specified in this target: `resourceInfo.` Finally, it initializes all child targets.

`virtual Target* createChild(int` *`index`*`);`
This protected method creates a child target, determined by `childTypes,` by *`index.`*

`virtual void resourceInfo();`
This method parses the `resources` state of this class and adjusts the `resources` param-

eter of child targets. If no `resources` parameter exists in a child target, it creates one.

(2) Choose a scheduler based on the states: `chooseScheduler`.

```
virtual void chooseScheduler();
```

This is a protected method to choose a scheduler based on the states related to scheduling algorithms.

(3) If it is a heterogeneous target, we flatten the wormholes: `flattenWorm`. To represent a universe for heterogeneous targets, we manually partition the stars using wormholes: which stars are assigned to which target.

```
void flattenWorm();
```

This method flattens wormholes recursively if the wormholes have a code generation domain inside.

(4) Set up the scheduler object. Clear `myCode` stream.

(5) Initialize the flattened galaxy, and perform the parallel scheduling: `Target::setup`.

(6) If the child targets are not inherited, display the Gantt chart if requested: `writeSchedule`.

```
void writeSchedule();
```

This public method displays a Gantt chart.

(7) If this target is inside a wormhole, it adjusts the sample rate of the wormhole ports (`CGTarget::adjustSampleRates`), generates code (`generateCode`), and downloads and runs code in the target (`CGTarget::wormLoadCode`).

```
void generateCode();
```

This is a redefined public method. If the number or processors is 1, just call `generateCode` of the child target and return. Otherwise, we first set the stop time, or the number of iteration, for child targets (`beginIteration`). If the target is inside a wormhole, the stop time becomes -1 indicating it is an infinite loop. The next step is to generate wormhole interface code (`wormInputCode`, wormOutCode if the target is inside a wormhole. Finally, we generate code for all child targets (`ParScheduler::compileRun`). Note that we generate wormhole interface code before generating code for child targets since we can not intervene the code generation procedure of each child target once started.

```
void beginIteration(int repetitions, int depth);
void endIteration(int repetitions, int depth);
```

These are redefined protected methods. In the first method, we call `setStopTime` to set up the stop time of child targets. We do nothing in the second method.

```
void setStopTime(double val);
```

This method sets the stop time of the current target. If the child targets are not inherited, it also sets the stop time of the child targets.

```
void wormInputCode();
void wormOutputCode();
void wormInputCode(PortHole& p);
```

`void wormOutputCode(PortHole& p);`
These are all redefined public methods. The first two methods traverse the portholes of wormholes in the original graph, find out all portholes in sub-universes matched to each wormhole porthole, and generate wormhole interface code for the portholes. The complicated thing is that more than one ParNode is associated with a star and these ParNodes may be assigned to several processors. The last two methods are used when the number of processors is 1 since we then use `CGTarget::wormInputCode,wormOutputCode` instead of the first two methods.

`int run();`
If this target does not lie in a wormhole or it has only one processor, we just use `CGTarget::run` to generate code. Otherwise, we transfer data samples to and from the target: `sendWormData` and `receiveWormData.`

`int sendWormData();`
`int receiveWormData();`
These are redefined protected methods. They send data samples to the current target and receive data samples from the current target. We traverse the wormhole portholes to identify all portholes in the sub-universes corresponding to them, and call `sendWormData,` receiveWormData for them.

`void wrapup();`
In this base class, we write code for each processor to a file.

Other CGMultiTarget protected members

`ParProcessors* parProcs;`
This is a pointer to the actual scheduling object associated with the current parallel scheduler.

`IntArray canProcs;`
This is an integer array to be used in `candidateProcs` to contain the list of processor indices.

`virtual void resetResources();`
This method clears the resources this target maintains such as communication resources.

`void updataRM(int from, int to);`
This method updates a reachability matrix for communication amortization. A reachability matrix is created if `amortizedComm` is set to YES. We can packetize communication samples only when packetizing does not introduce deadlock of the graph. To detect the deadlock condition, we conceptually cluster the nodes assigned to the same processors. If the resulting graph is acyclic, we can packetize communication samples. Instead of clustering the graph, we set up the reachability matrix and update it in all send nodes. If there is a cycle of send nodes, we can see the deadlock possibility.

Other CGMultiTarget public members

The destructor deletes the child targets, scheduler, and reachability matrix if they exist. There

is an `isA` method defined for type identification.

`Block* makeNew() const;`
Creates an object of CGMultiTarget class.

`int execTime(DataFlowStar* s, CGTarget* t);`
This method returns the execution time of a star *s* if scheduled on the given target *t.* If the target does not support the star, a value of -1 is returned. If it is a heterogeneous target, we consider the relative time scale of processors. If the second argument is NULL or it is a homogeneous multiprocessor target, just return the execution time of the star in its definition.

`IntArray* candidateProcs(ParProcessors* par, DataFlowStar* s);`
This method returns a pointer to an integer array of processor indices. We search the processors that can schedule the argument star *s* by checking the star type and the resource requirements. We include at most one idle processor.

`int commTime(int from, int to, int nSamples, int type);`
This method returns the expected communication overhead when transferring *nSamples* data from *from* processor to *to* processor. If *type* = 2, this method returns the sum of receiving and sending overhead.

`int scheduleComm(ParNode* comm, int when, int limit = 0);`
Since it models a fully-connected multiprocessor, we can schedule a communication star anytime without resource conflict that returns the second argument *when.*

`ParNode* backComm(ParNode* rcv);`
This method returns the corresponding send node paired with the argument receive node, *rcv.* If the argument node is not a receive node, return NULL.

`int amortize(int from, int to);`
This method returns TRUE or FALSE, based on whether communication can be amortized between two argument processors.

17.2.3 Class CGSharedBus

Class CGSharedBus, derived from class CGMultiTarget, is a base class for shared bus multiprocessor targets. It has the same kind of constructor as its base class.

This class has an object to model the shared bus.

```
UniProcessor bus;
UniProcessor bestBus;
```
These are two protected members to save the current bus schedule and the best bus schedule obtained so far. The `bus` and `bestBus` are copied to each other by the following public methods.

`void saveCommPattern();`

```
void restoreCommPattern();
clearCommPattern();
void resetResources()
```

The first method is a public method to clear `bus` schedule, while the second is a protected method to clear both `bus` and `bestBus.`

This classes redefines the following two public methods.

`int scheduleComm(ParNode*` *`node`*`, int` *`when`*`, int` *`limit`* `= 0);`

This method schedules the argument node available at *`when`* on `bus.` If we can schedule the node before *`limit,`* we schedule the node and return the schedule time. Otherwise, we return -1. If *`limit`* = 0, there is no limit on when to schedule the node.

`ParNode* backComm(ParNode*` *`node`*`);`

For a given communication node, find another node scheduled just before the argument node on `bus.`

17.3 Heterogeneous Support

In this section, we summarize the special routines to support heterogeneous targets. They are already explained in earlier chapters.

1. To specify the component targets , we first set `childTypes` state of the target class that must be derived from class CGMultiTarget. We may add special resources to the processors by setting `resources` state, a list of items separated by ';'. An item starts with the target index followed by a list of strings identifying resources. The relative computing speed of processors are specified by `relTimeScales` state.

2. An application program for a heterogeneous target uses wormholes. In pigi, all stars in a universe should be in the same domain. To overcome this restriction, we use wormhole representation to distinguish stars for different targets, or domains, but still in the same universe. Once the graph is read into the Ptolemy kernel, all wormholes of code generation domain are flattened to make a single universe: `flattenWorm` method of CGMultiTarget class. Currently, we manually partition the stars to different kinds of processors. For example, if we have three "default-CG96" targets and one "default-CG56" target, we partition the stars to two kinds: CG96 or CG56. This partitioning is based on the original wormhole representation. If we ignore this partitioning, we can apply an automatic scheduling with the flattened graph. This feature has not been tested yet even though no significant change is required in the current code.

3. When we schedule a star in the scheduling phase, we first obtain the list of processors that can schedule the star: `candidateProcs` method of `CGMultiTarget` class. The execution time of the star to a processor is computed in `execTime` method of `CGMultiTarget` class considering the relative speed of processors.

4. After scheduling is performed, we create sub-universes for child targets. In case manual partitioning is performed, we just clone the stars from the original graph in the sub-universes. In case we use automatic partitioning, we need to create a star in the current target with the same name as the corresponding star in the original graph: `cloneStar` private method of `UniProcessor` class. We assume that we use the same name for a star in all domains.