Center for Embedded and Cyber-Physical Systems
University of California, Irvine

A Synthetic Graph Toolbox for Benchmarking the Grid of Processing Cells

Anson Fong, Yutong Wang, Rainer Dömer

Technical Report CECS-24-03
June 5, 2024

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

ansonhf@uci.edu
http://www.cecs.uci.edu

A Synthetic Graph Toolbox for Benchmarking the Grid of Processing Cells

Anson Fong, Yutong Wang, Rainer Dömer

Technical Report CECS-24-03
June 5, 2024

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

ansonhf@uci.edu
http://www.cecs.uci.edu

Abstract

Classical computer architectures are based on the concept of utilizing a large system memory to support communication between computational cores. To remedy the memory access bottleneck that limits optimization of such systems, the Grid of Processing Cells (GPC) has been proposed as a highly scalable multi-core alternative that maximizes the benefits of parallelization. This work introduces a toolbox that aims to immensely improve benchmarking productivity for the GPC model. The included tools generate random task graphs based on configurable constraints, as well as their corresponding SystemC timing simulations, C++ executables, and abstract syntax trees. The synthetic examples can be used to test the feasibility and optimization of mapping data flow applications to the GPC architecture.

Contents

List of Figures

A Synthetic Graph Toolbox for Benchmarking the Grid of Processing Cells

Anson Fong, Yutong Wang, Rainer Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
ansonhf@uci.edu
http://www.cecs.uci.edu

Abstract

Classical computer architectures are based on the concept of utilizing a large system memory to support communication between computational cores. To remedy the memory access bottleneck that limits optimization of such systems, the Grid of Processing Cells (GPC) has been proposed as a highly scalable multicore alternative that maximizes the benefits of parallelization. This work introduces a toolbox that aims to immensely improve benchmarking productivity for the GPC model. The included tools generate random task graphs based on configurable constraints, as well as their corresponding SystemC timing simulations, C++ executables, and abstract syntax trees. The synthetic examples can be used to test the feasibility and optimization of mapping data flow applications to the GPC architecture.

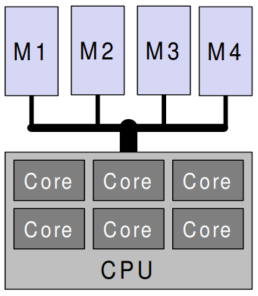


Figure 1: Architecture of SM Designs [1]

1 Introduction

In most computer systems, strategies for hardware optimization have been centered around increasing clock speeds, processor counts, and cache sizes. Despite the development of sophisticated memory hierarchies, parallel processing, and other specialized solutions, most modern designs still rely on large, slow central memory modules to support its processor cores, as illustrated in Figure 1. The ever increasing memory access complexity and contention forms the memory access bottleneck, which becomes the limiting factor in optimization.

The *Grid of Processing Cells (GPC)* [3] has been proposed as an alternative design to maximize parallelization benefits by mitigating the memory access bottleneck. The GPC architecture uses a "checkerboard" pattern tiled by alternating processor and memory modules, as shown in Figure 2. The design allows adjacent cores to directly access neighboring memory modules, reducing the risk of memory access contention. Optimization of parallelization on such an architecture becomes limited only by the mapping of tasks onto the checkerboard.

1.1 Problem Definition

To take full advantage of this architecture, it becomes necessary to develop a pipeline for mapping applications to the GPC. Various projects in the past have manually mapped applications to the GPC architecture. These projects include an APNG encoder [4], a JPEG encoder [1] (Figure 3), a Mandelbrot Set Visualizer [5] (Figure 4), and a Canny Edge Detec-

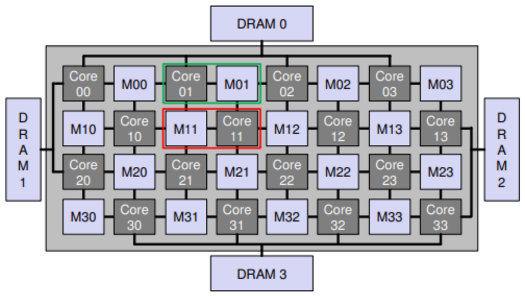


Figure 2: 4x4 GPC Checkerboard [3]

tor [2]. Despite observing speedups when evaluating the execution performance of applications mapped onto the GPC, each project commented on the impracticality of manually mapping applications onto the GPC checkerboard. Thus, in the interest of accelerating benchmarking productivity and extending the architecture's use cases, an automatic *GPC Compiler (GPCC)* must handle the mapping of applications onto the GPC.

The tools developed in this work seek to contribute toward the development of the GPCC by performing three tasks: synthetic graph generation, outputting SystemC simulation source code, and producing formatted inputs for the GPCC. This toolbox will help validate the feasibility of mapping configurable application structures, as well as offer a baseline for simulation comparison. In this report, we will refer to the collection of included tools as the *Synthetic Graph Toolbox (SGT)*.

At the time of this report, the GPCC (Figure 6) requires input C++ applications to conform to a specific set of syntax and structure rules, which we will call *Guided C++ (GCPP)*. This includes requiring that all functions be contained within a class called a **Module**. Inter-function communication must be completed by interacting through **Channels**. In this form, the GPC parser will be able to generate a GPC mapping for the application using SystemC, directly utilizing the modules and channels to represent hardware. The GPCC also requires an *Application Task Graph* (*ATG*, a modified variant of the abstract syntax tree data structure) for mapping input applications. The ATG contains information describing the control flow and dependencies within the application, as well as the local variable names and implemented functionality within its functions.

1.2 Toolbox Overview

The SGT (Figure 5) includes six C++ files: the synthetic graph generator, the SystemC TLM1 generator, the SystemC TLM2 generator, the GCPP generator, the ATG extraction generator, and a graphic generator. The graph generator processes the user configuration file and generates a text file containing the graph's nodes and edges. This graph is used by the other files to generate outputs. The TLM1 and TLM2 writers produce SystemC model source code, with user-configurable timing settings. The GCPP writer produces the aforementioned guided C++ file from the generated graph, and the graphic generator generates a Python script that creates a visualization of the graph. The ATG extraction generator generates another Python file, parsed from the GCPP file, that extracts the application control flow, dependencies, and implemented functionality.

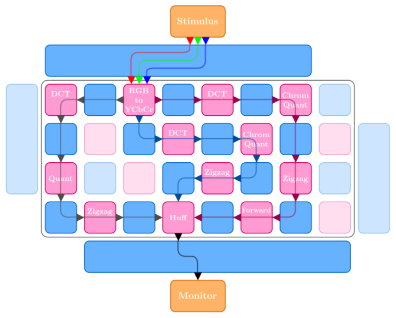


Figure 3: GPC Mapping: JPEG Encoder [1]

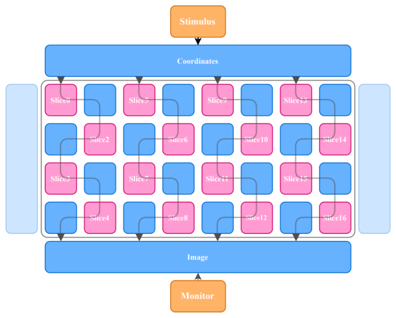


Figure 4: GPC Mapping: Mandelbrot Set [5]

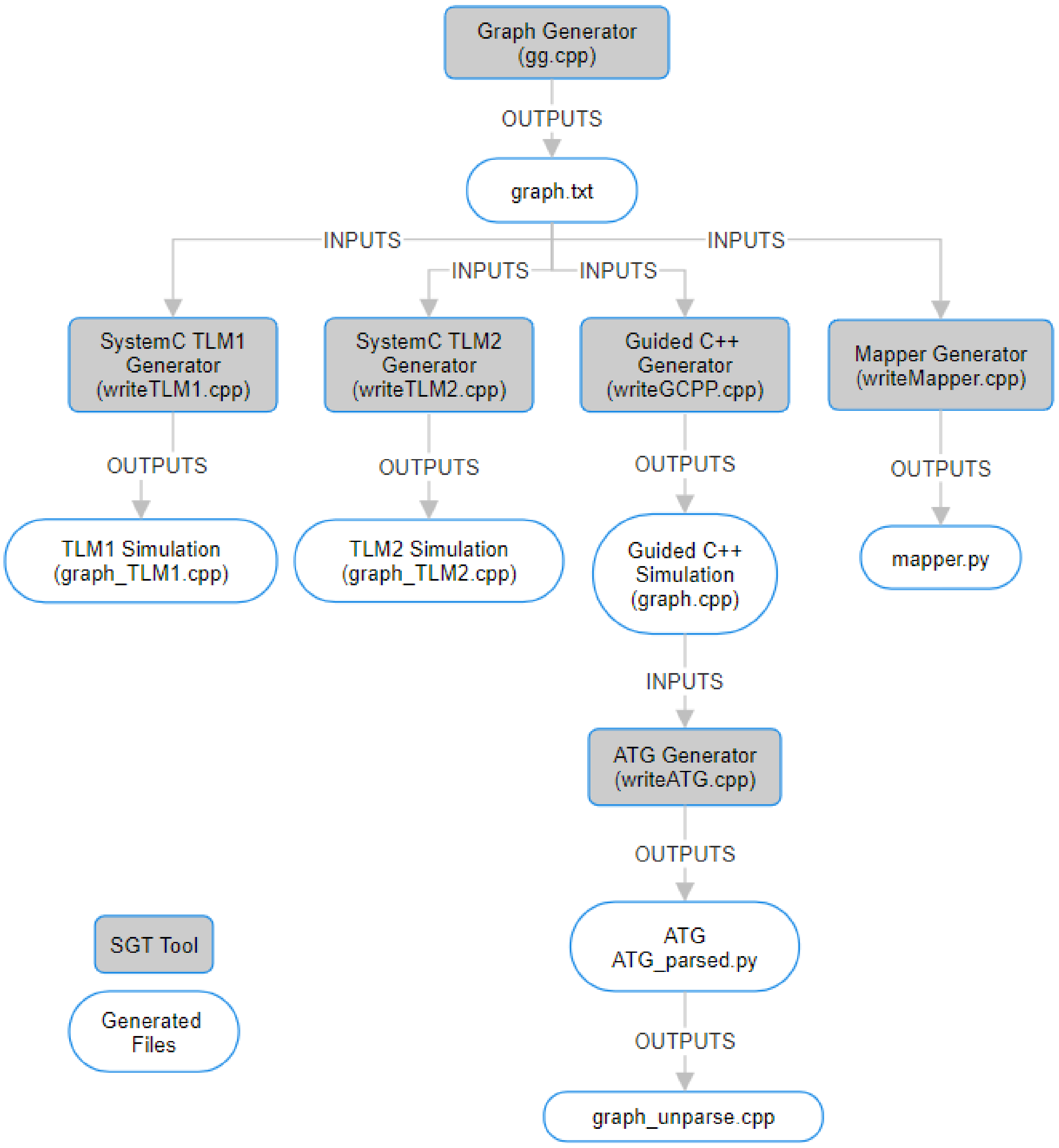

Graph Generator (gg.cpp)
OUTPUTS
graph.txt
INPUTS
INPUTS
INPUTS
INPUTS
SystemC TLM1 Generator (writeTLM1.cpp)
SystemC TLM2 Generator (writeTLM2.cpp)
Guided C++ Generator (writeGCPP.cpp)
Mapper Generator (writeMapper.cpp)
OUTPUTS
OUTPUTS
OUTPUTS
OUTPUTS
TLM1 Simulation (graph_TLM1.cpp)
TLM2 Simulation (graph_TLM2.cpp)
Guided C++ Simulation (graph.cpp)
mapper.py
INPUTS
ATG Generator (writeATG.cpp)
OUTPUTS
ATG ATG_parsed.py
OUTPUTS
graph_unparse.cpp
SGT Tool
Generated Files

Figure 5: The SGT Toolbox

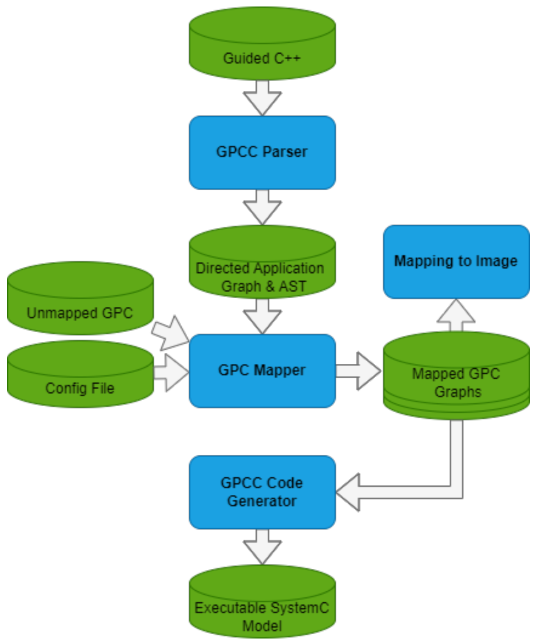


Figure 6: GPCC Flowchart

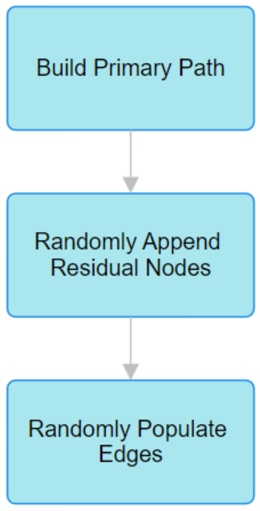


Figure 7: Graph Generator Stages

2 The SGT Executables

In the following sections, we will discuss the implementation and operation of each tool in the SGT, as well as the applications of their generated outputs.

2.1 Graph Generator

The graph generator **gg.cpp** randomly constructs directed acyclic graphs based on constraints specified in the user configuration file. The graph is represented using the vector and set data structures, storing nodes and edges respectively. Along with the ID of the nodes, we also keep track of the depth and number of outgoing edges, properties that will be useful when constructing SystemC models later. The generator constructs the graph in three stages, as shown in Figure 7.

2.1.1 Generator Algorithm

First, a primary path is created by initializing H nodes and connecting them to form a singly linked list, where H is the height of the graph specified in the configuration file. Next, nodes are appended randomly to the non-terminal nodes along the primary path, until the graph contains all N user specified nodes. The generator is capable of generating graphs with singular and non-singular outputs, meaning the graph may terminate at one node or contain multiple terminal nodes. Should the user specify for singular output, the generator will connect appended nodes to the terminal primary node to avoid hanging nodes.

In the final stage, additional edges are randomly formed between random nodes. The generator iterates through all nodes, where it may randomly form an edge between it and a random destination node. Each node has a $3o\%$ chance of forming this additional edge, by default. This probability setting can be configured by editing the threshold variable within the graph generator's source code.

These steps ensure a random graph is generated within the constraints specified within the configuration file. The completed graph is written to a text file, which will be used by the other tools within the SGT. The user should note that the graph generator will fail if the constraints specified in the configuration file do not describe a plausible graph.

2.1.2 Input/Output

The configuration file, **gg.config**, allows for selecting a node count, a height (maximum length from source node to terminal node or nodes), maximum edge count per node, and singular/non-singular output. A minimum of 2 nodes is required to produce output from the graph generator, given the height and maximum edge parameters always describe a plausible graph. The singular output setting is active high; setting it to 0 will yield multiple terminal nodes and setting it to 1 will yield a single terminal node.

The output text file will always specify the number of nodes in the graph on the first line, and then list the connections at each node. Each node is denoted as **V*x*[d]**, where *x* is the node's ID and *d* is the node's depth. A node's depth is defined by the number of edges away it is from the source node, **V0**.

2.1.3 Example Configuration

```
// CONFIGURATION FILE (for gg.cpp)
Nodes (min 2): 5
Height: 3
Maximum edges: 3
Singular output [0/1]: 1
```

Figure 8: Graph Configuration File

Figure 8 shows an example configuration in the configuration file, describing a graph with five nodes, maximum height of three, maximum of three edges per node, and a single terminal node. The graph generator first connects nodes **V0**, **V1**, and **V2** in a singly linked list, completing the primary path of the graph. Next, in this example, nodes **V3** and **V4** are appended to the primary path at random. This example shows both nodes connecting to V0. Since the configuration file specifies for singular output, all hanging nodes are connected to the terminal node **V2**. In this example, no additional edges are added. Figure 9 shows the resulting output text file, describing a graph that satisfies all constraints specified in the configuration file.

```
Nodes: 5
V0[0]: V1[1],V3[1],V4[1],
V1[1]: V2[2],
V2[2]:
V3[1]: V2[2],
V4[1]: V2[2],
```

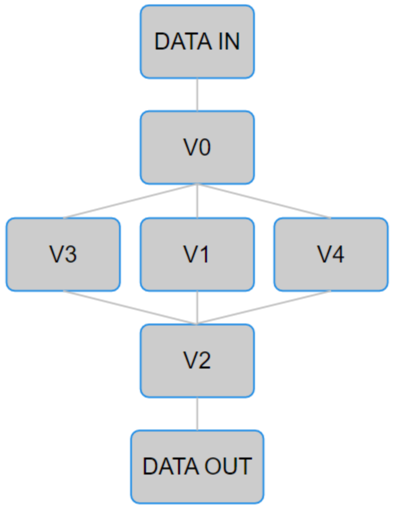
Figure 9: Graph Output Example

2.2 SystemC Simulation Generators

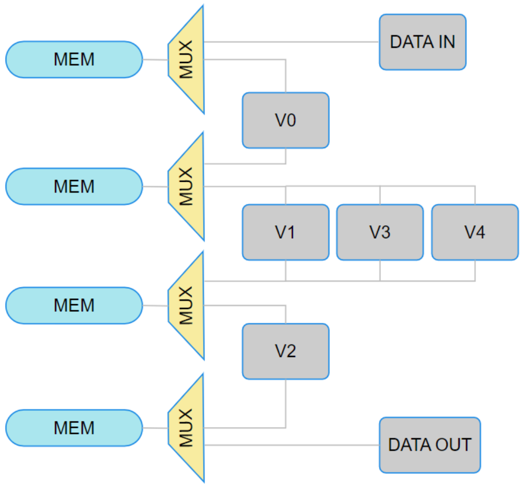
The SGT contains two SystemC generators, **writeTLM1.cpp** and **writeTLM2.cpp**, which process the text file produced by the graph generator to generate SystemC simulation models. Each node in the generated task graph represents a function or subtask from some application, which will be assigned to a processor core. Upon conversion to a SystemC simulation, each node becomes a module within a *Design-Under-Test (DUT)*. The two generators differ in how they handle inter-module (inter-function) communication. Both create DUTs and simulate their behavior when connected to Stimulus and Monitor modules.

The TLM1 model (Figure 10(a)) uses channels to represent communication between modules. These channels are implemented using FIFO buses. Each node is represented by a module and has dedicated channels for incoming and outgoing edges, where data is read from and written into. Since the SGT graph generator doesn't assign any functionality to the nodes within its task graph, a configurable delay is imposed at each node to represent some computation time. This setting can be configured in the TLM configuration file, **TLM.config**, by changing the value next to **Delay Factor** under **TLM1**, which is measured in milliseconds. The generator adds code into each module to calculate the distance of the longest path from source to terminal node(s). This output is observable at the Monitor during simulation, by compiling and running the generated **graph_TLM1.cpp**.

The TLM2 model (Figure 10(b)) enables a more rigorous model for inter-module communication, requiring modules to read and write data from and into memory modules, with control signals and multiplexers to facilitate parallel access. The implementation for memory modules and muxes can be found in the **lib/TLM2** folder. Nodes must register I/O sockets with memory modules, specifying access signals and addresses. The TLM2 simulation also calculates the distance of the longest path from source to terminal node(s), but utilizes more configurable delay values than TLM1. In the TLM configuration file, **TLM.config**, the memory size, read delay, and write delay of on and off chip memories can be configured. Memory size is measured in bytes, and delays are measured in nanoseconds. Similarly to TLM1, the artificial computation time can be configured by changing the value next to **Delay Factor** under **TLM2**, which is measured in milliseconds. After compiling and running the generated **graph_TLM2.cpp** simulation, the Monitor will once again display the calculated longest path(s).



(a) TLM1 Model



(b) TLM2 Model

Figure 10: SystemC Model Examples

2.2.1 Example SystemC Model Configuration

The TLM generators both parse the text file generated by the graph generator and produce C++ simulation files. Although both TLM1 and TLM2 will support simulation with multiple terminal nodes, only TLM2 utilizes multithreading to read outputs concurrently. Thus, only the TLM2 simulation will report accurate timings at the Monitor.

Using the graph generated in Figure 9 and the configuration file settings in Figure 11, we obtain the simulations shown in Figure 12 and Figure 13. Due to the additional timing settings present in the TLM2 simulation, as well as initialization delays hidden within the source code, the stimulus takes slightly longer to propagate through the DUT and reach the Monitor.

```
// CONFIGURATION FILE (for TLM models)
[TLM1]
-- Delay Factor: 1

[TLM2]
-- Off-Chip Memory Size: 8*1024*1024
-- Off-Chip Read Delay: 100
-- Off-Chip Write Delay: 110
-- On-Chip Memory Size: 4*1024*1024
-- On-Chip Read Delay: 5
-- On-Chip Write Delay: 6
-- Delay Factor: 1
```

Figure 11: TLM Configuration File

2.3 Guided C++ Code Generator

To map applications to the GPC using the GPCC format, the input application must be translated into a form that the GPCC can parse. As previously stated, this form is called guided C++, and requires compartmentalizing functions into **Modules** and performing communication through **Channels**. Each **Module** creates a dedicated thread and communicates to other **Modules** by sending data through queues (channels). To perform this translation, the GCPP generator **writeGCPP.cpp** also takes the text file generated by the graph generator as input. The resulting C++ file, **graph.cpp**, contains the necessary functionality to perform the same longest-path calculation that the TLM1 and TLM2 simulations perform, as observed in Figure 14, which also uses the graph in Figure 9. The structure of the outputted **graph.cpp** closely resembles that of the TLM1 simulation (Figure 10(a)). However, the GCPP simulation is stripped of the timing information that SystemC provides.

```
        SystemC 2.3.3-Accellera --- May 16 2023 21:54:37
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
<>       0 s: Stimulus sent.
>>      3 ms: Monitor received output [3] from V2 after      3 ms delay.
```

Figure 12: Example TLM1 Simulation Output

```
        SystemC 2.3.3-Accellera --- May 16 2023 21:54:37
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
<>       0 s: Stimulus sent.
>> 4001770 ns: Monitor received output [3] from V2 after 4001770 ns delay.
```

Figure 13: Example TLM2 Simulation Output

```
<> Stimulus sent [0].
>> Monitor received [3] from V2.
<> Monitor exits simulation.
```

Figure 14: Example GCPP Output

2.4 ATG Extraction Generator

The GPCC also requires the control flow, dependencies, and functionality of an application's functions to be extracted from the GCPP file. This data is stored in an *Application Task Graph (ATG)*, a modified variant of an abstract syntax tree. To extract the ATG from the GCPP file generated by the GCPP generator, the ATG extraction generator **writeATG.cpp** parses the GCPP generator's output and produces a python script, **ATG_parsed.py**. This python script populates an ATG data structure, recording the channels that each **Module** utilizes and the functional code within each **Module**. The script prints this information to the terminal when run, as seen in Figure 15 and Figure 16, which uses the GCPP code based on the graph in Figure 9. The GPCC may use the **graph.cpp** file and the data structure created by running **ATG_parsed.py** to attempt creating a GPC mapping for the task graph generated by the graph generator.

The script also generates **graph_unparse.cpp**, which is reconstructed from the ATG, and when run, yields the same output as the **graph.cpp** file generated by the GCPP generator.

2.5 Mapper Generator

The mapper generator **writerMapper.cpp** generates another python script, **mapper.py**, which is responsible for producing a visualization of the generated task graph. The mapper generator also takes the text file generated by the graph generator as input. The generated graphic for the graph in Figure 9 is shown in Figure 17.

```
<mapperLib.AppTaskGraph object at 0x7f92ff2b6b90>
SOURCE: graph.cpp
HEADER: 5 lines
GPCC:   True
MODULES: 12
    MODULE V0
        PORT QUEUE_IN<int> input
        PORT QUEUE_OUT<int> V1
        PORT QUEUE_OUT<int> V3
        PORT QUEUE_OUT<int> V4
        THREAD main (detached)
        Code: 11 lines
    MODULE V1
        PORT QUEUE_IN<int> V0
        PORT QUEUE_OUT<int> V2
        THREAD main (detached)
        Code: 11 lines
    MODULE V2
        PORT QUEUE_IN<int> V1
        PORT QUEUE_IN<int> V3
        PORT QUEUE_IN<int> V4
        PORT QUEUE_OUT<int> output_V2
        THREAD main (detached)
        Code: 17 lines
    MODULE V3
        PORT QUEUE_IN<int> V0
        PORT QUEUE_OUT<int> V2
        THREAD main (detached)
        Code: 11 lines
    MODULE V4
        PORT QUEUE_IN<int> V0
        PORT QUEUE_OUT<int> V2
        THREAD main (detached)
        Code: 11 lines
    MODULE DUT
        PORT QUEUE_IN<int> DataIn
        PORT QUEUE_OUT<int> V2_out
        CHANNEL_INSTANCE QUEUE<int> V0_V1
        CHANNEL_INSTANCE QUEUE<int> V0_V3
        CHANNEL_INSTANCE QUEUE<int> V0_V4
        CHANNEL_INSTANCE QUEUE<int> V1_V2
        CHANNEL_INSTANCE QUEUE<int> V3_V2
        CHANNEL_INSTANCE QUEUE<int> V4_V2
        MODULE_INSTANCE V0 v0(DataIn,V0_V1,V0_V3,V0_V4)
        MODULE_INSTANCE V1 v1(V0_V1,V1_V2)
        MODULE_INSTANCE V2 v2(V1_V2,V3_V2,V4_V2,V2_out)
        MODULE_INSTANCE V3 v3(V0_V3,V3_V2)
        MODULE_INSTANCE V4 v4(V0_V4,V4_V2)
        Code: 2 lines
```

Figure 15: ATG Extraction Output [Part 1/2]

```
    MODULE DataIn
        PORT QUEUE_IN<int> input
        PORT QUEUE_OUT<int> output
        THREAD main (detached)
        Code: 9 lines
    MODULE DataOut
        PORT QUEUE_IN<int> input
        PORT QUEUE_OUT<int> output
        THREAD main (detached)
        Code: 9 lines
    MODULE Platform
        PORT QUEUE_IN<int> input
        PORT QUEUE_OUT<int> V2_out
        CHANNEL_INSTANCE QUEUE<int> din_dut(1)
        CHANNEL_INSTANCE QUEUE<int> dut_V2(1)
        MODULE_INSTANCE DataIn din(input,din_dut)
        MODULE_INSTANCE DUT dut(din_dut,dut_V2)
        MODULE_INSTANCE DataOut V2(dut_V2,V2_out)
        Code: 2 lines
    MODULE Stimulus
        PORT QUEUE_OUT<int> output
        THREAD main (joinable)
        Code: 7 lines
    MODULE Monitor
        PORT QUEUE_IN<int> input_V2
        THREAD main (detached)
        Code: 8 lines
    MODULE Top
        CHANNEL_INSTANCE QUEUE<int> stim_plat(1)
        CHANNEL_INSTANCE QUEUE<int> plat_mon_V2(1)
        MODULE_INSTANCE Stimulus stim(stim_plat)
        MODULE_INSTANCE Platform plat(stim_plat,plat_mon_V2)
        MODULE_INSTANCE Monitor mon(plat_mon_V2)
        Code: 2 lines
INSTANCE TREE:
        MODULE_INSTANCE Top top()
FOOTER: 6 lines
```

Figure 16: ATG Extraction Output [Part 2/2]

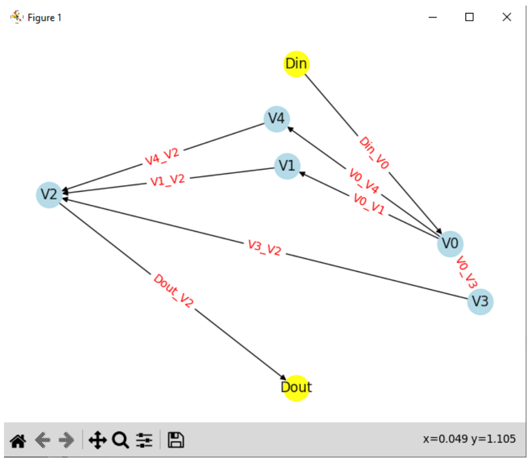


Figure 17: Example Graph Visualization

3 Experiments and Results

To test the SGT, each of its tools was tested on a variety of input graphs. Using the TLM configuration from Figure 11, graphs with 5, 10, 25, 50, and 100 nodes were tested, at various heights. Figure 1 displays each graph's input attributes, the TLM1 output and timing, and the TLM2 output and timing. Figure 2 displays the GPCC simulation output, and the ATG's reverse engineered simulation output. All of the simulations – TLM1, TLM2, GPCC, and the reverse engineered ATG – retain the input graph's attributes. Thus, the simulation models may be used as an accurate baseline, and the GPCC and ATG can be used as test cases for the GPCC.

Nodes	Height	TLM1	TLM2
5	3	3 at 3ms	3 at 4001770ns
10	8	8 at 8ms	8 at 9001986ns
25	20	20 at 20ms	20 at 21002638ns
50	43	43 at 43ms	43 at 44003723ns
100	94	94 at 94ms	94 at 95005803ns

Table 1: SystemC Simulation Results [Part 1/2]

Nodes	Height	GPCC	ATG Reversed
5	3	3	3
10	8	8	8
25	20	20	20
50	43	43	43
100	94	94	94

Table 2: SystemC Simulation Results [Part 2/2]

Makefile Command	Function
clean	Remove all generated files
gg	Runs graph generator
testTLM1	Produces TLM1 model
testTLM2	Produces TLM2 model
testGCPP	Produces GCPP file
mapper	Produces visual
ATG	Extracts ATG

Table 3: Makefile Commands

4 Conclusion

Various application have been manually mapped onto the GPC architecture in an effort to benchmark its design. Despite the significant speedups observed in these experiments, manually mapping applications onto the GPC checkerboard is impractical. This has led to the need to develop a compiler, GPCC, for automatically mapping arbitrary data flow applications onto the GPC. To quickly evaluate the feasibility of mapping various application structures onto the GPC using the GPCC, the SGT can be used to generate graphs, produce baseline simulations, and format GPCC inputs.

Furthermore, the SGT can assist the GPCC in becoming operational by automatically producing large quantities of test cases, streamlining testing an debugging. With a functioning GPCC, applications can begin taking full advantage of the highly parallel GPC architecture.

References

[1] Rainer Dömer Arya Daroui. A Loosely-Timed TLM-2.0 Model of a JPEG Encoder on a Checkerboard GPC. Technical Report CECS-TR-22-04, Center for Embedded Computer Systems, University of California, Irvine, October 2022.

[2] John Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[3] Rainer Dömer. A Grid of Processing Cells (GPC) with Local Memories. Technical Report CECS-TR-22-01, Center for Embedded Computer Systems, University of California, Irvine, April 2022.

[4] Vivek Govindasamy and Rainer Dömer. Mapping of an APNG Encoder to the Grid of Processing Cells Architecture. Technical Report CECS-TR-22-02, Center for Embedded Computer Systems, University of California, Irvine, September 2022.

[5] Rainer Dömer Yutong Wang. A Scalable SystemC Model of a Checkerboard Grid of Processing Cells. Technical Report CECS-TR-22-03, Center for Embedded Computer Systems, University of California, Irvine, September 2022.