GPCC: Grid of Processing Cells Compiler with Pareto-Optimal Design Space Exploration

Yutong Wang, Yanda Li, Rainer Dömer
Center for Embedded and Cyber-Physical Systems, University of California, Irvine, CA, USA
{yutongw5,yandal5,doemer}@uci.edu

Abstract

Programming grid–based system-on-chip architectures remains challenging due to the need for efficient task placement and localized communication. This paper presents the Grid of Processing Cells Compiler (GPCC), an automated compilation flow that maps a task graph extracted from a hierarchical data flow specification onto a two-dimensional grid of processing elements with local memories and neighbor-only communication. GPCC analyzes the application graph, performs task and data partitioning, assigns tasks to grid locations, and generates the needed on-chip and off-chip communication, producing an executable SystemC TLM-2.0 model for simulation and evaluation. GPCC further supports automatic design space exploration of checkerboard grid configurations by varying dimensions and identifying feasible mappings, allowing it to derive Pareto-optimal target architectures. Experimental results on embedded applications and synthetic benchmarks show that GPCC reliably transforms dataflow applications into grid-accurate SystemC models and supports efficient architectural design space exploration.

1 Introduction and Motivation

For high performance, parallel and pipelined applications have taken advantage of two-dimensional grid structures in a variety of configurations. From systolic arrays with only basic operators [1] via course grain reconfigurable arrays (CGRAs) with simple processing elements [2, 3], such architectures have grown the compute capacity of their nodes to network-on-chip (NOC) architectures with fully programmable processor cells [4]. At the same time, processing *in* or *near memory* [5] has recently become an active area of research that overcomes the classic von-Neumann bottleneck [6] by co-locating compute and memory units.

Aligned with these advances, we target in this work a *Grid of Processing Cells (GPC)* architecture, namely R-type checkerboard, where pairs of processors with local memories are arranged in a 2D array on a chip [7], as illustrated in Fig. 1.

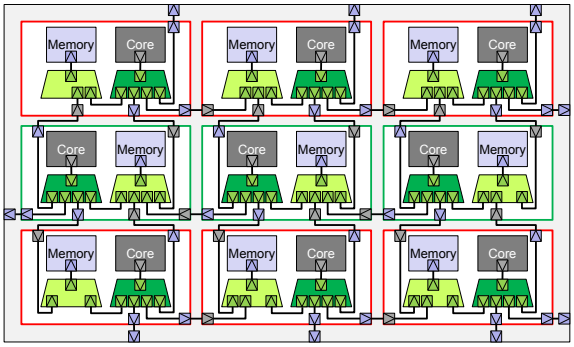


Figure 1: R-type Checkerboard (CBR) in 3x3 size

The main advantage of array architectures is their inherent *scalability* where the height and width of the array can be freely chosen to fit the application at hand. For performance, however, the distance of data movements between cells must be considered. For example, mapping algorithms for NOCs typically minimize the Manhattan distance between communicating cells. Since cells can pass data to any other cell in the network, the number of router hops generally scales linearly ($O(n)$).

In contrast to NoCs, we focus in this work on a *truly scalable* architecture with *constant* communication cost ($O(1)$). We intentionally limit the target checkerboard architecture to communicate only locally. A processor can only reach its own memory and the memories of its immediate neighbors. Specifically, we concentrate on a restricted GPC where processors and memories are arranged in alternating fashion, as shown in Fig. 1. We refer to this target as a R-type checkerboard (CBR), since the cell in the top-left corner has the core on the right side of the memory.

Note that the *true scalability* is an intentional limitation of the architecture and makes the application to architecture mapping harder, but is at the same time a significant performance advantage with a minimal constant 1-hop communication.

Programming a GPC remains a challenge because applications must be partitioned into tasks, mapped onto the grid, and connected through memory channels and I/O ports. Manual mapping is slow and error-prone, while existing heuristic approaches are often rigid and do not generalize well.

The *Grid of Processing Cells Compiler (GPCC)* addresses these challenges by automating the process. GPCC accepts task graph applications specified in C++ or Python and builds an internal representation of modules, ports, channels, and tasks. The task graph model is represented both as a structured object and a NetworkX [8] graph for inspection. The mapping proceeds in two stages. Level-1 mapping places tasks onto cores of a user-specified grid using a recursive search that respects task dependencies and I/O placement. Level-2 mapping adds memory and off-chip I/O assignments and assures the consistency of the result. The mapped system is then translated to SystemC TLM-2.0

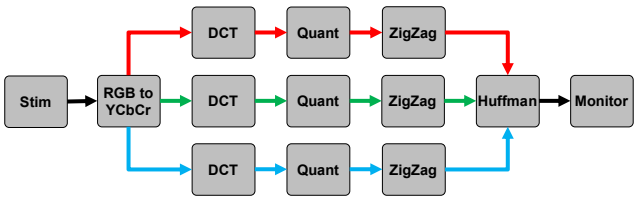


Figure 2: Task Graph of a JPEG Encoder with RGB color channels

[9] where communication channels are realized as memory queues and task operations are refined into SystemC semantics. GPCC also generates a Makefile for direct compilation and, if requested, exports the mapped grid to Graphviz [10] or HTML for easy inspection.

By combining parsing, task graph extraction, mapping, and model generation into a single flow (see Fig. 4), GPCC enables rapid design space exploration (DSE) of checkerboard architectures. Users can adjust grid size, I/O sides, or configuration parameters and immediately obtain a simulatable TLM-2.0 model. In this work, we present the design and implementation of GPCC, explain its mapping algorithms, and demonstrate its effectiveness on both embedded applications and synthetic benchmarks.

1.1 Problem Definition

The checkerboard GPC as shown in Fig. 1, provides a truly scalable platform for parallel computing by arranging processing cores with local memories in a two-dimensional grid with efficient neighbor communication [7]. However, mapping real applications onto a GPC remains a difficult problem. Developers must decide how to partition the application into tasks, assign tasks to specific cells, and route communication through local memories and off-chip I/O. Poor mappings can lead to congestion, load imbalance, or even infeasible designs.

Fig. 2 shows a simple example, the JPEG encoder algorithm, where the application is modeled as a directed acyclic graph (DAG) of image-processing tasks on the three RGB color channels. Each task must be mapped to a core in the checkerboard, and each edge must be assigned to a memory, so that data dependencies are preserved and communication remains efficient. A manual mapping of JPEG encoding onto the CBR grid is shown in Fig. 3, which illustrates the difficulty of balancing tasks, respecting I/O placement, and avoiding communication bottlenecks.

This example illustrates the broader challenge: mapping applications onto a GPC by hand is error-prone and does not scale. Existing heuristic-based methods help, but they remain limited in flexibility and often fail to produce an executable model that can be simulated at the system level. What is needed is an automated approach that extracts the task graph, maps tasks and channels across the grid, and quickly generates a simulatable model for evaluation.

The "forward" task shown in Fig. 3 is not part of the original input task graph. It is introduced only in the manual mapping to simplify the placement process and make the mapping more intuitive for a human designer. The automatic GPCC mapping flow does not insert such auxiliary

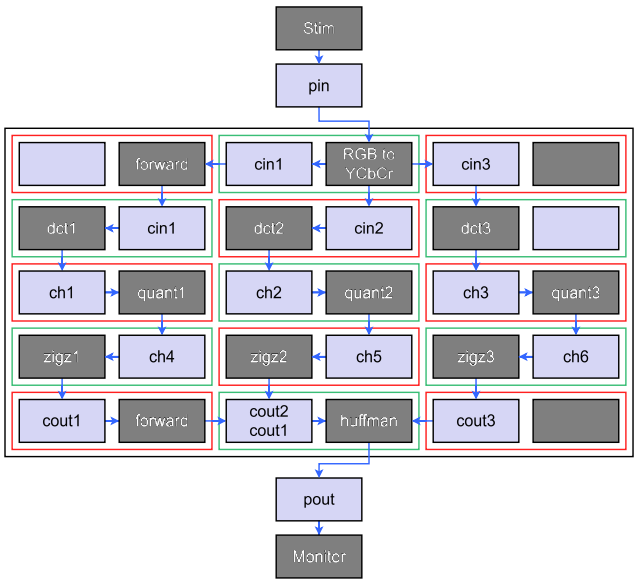


Figure 3: JPEG Encoder manually mapped to a R-type Checkerboard GPC

tasks and operates strictly on the extracted application DAG under a one-task-per-processing-element model. Supporting task-level transformations or allowing multiple tasks to share a processing element remains future work.

1.2 Background and Related Work

Organizing processing elements in regular two-dimensional layouts has been widely explored, for example in coarse-grain reconfigurable arrays (CGRAs) [2, 3], in exploration frameworks such as Canal [11] and CGRA-ME [12], and in tile-based multiprocessor designs such as the Tile Processor Architecture [13] and TILE64 [14]. Grid-like structures also appear in Network-on-Chip (NoC) platforms [4], where routers provide packet-switched communication, e.g., Intel Polaris [15]. While NoC architectures share similarities with our grid layout, they rely on global routing structures, distributed caches, or multi-hop networks. As such, they do not focus on purely local memory access nor strictly neighbor-to-neighbor communication.

In contrast, the Grid of Processing Cells (GPC) [7, 16] enforces a much more constrained and predictable execution substrate: each cell contains its own private local memory, and all communication—both data movement and synchronization—occurs exclusively through direct neighbor cells. There is no global interconnect, shared memory, or long-range routing fabric. This strict locality property is what enables true scalability, since adding more cells does not increase global contention or communication distance.

Existing GPC implementations and studies have so far relied on manual or ad-hoc heuristics tailored to specific applications. Prior works have demonstrated *manual* mappings for Mandelbrot visualization [17], Canny edge detection [18], APNG encoding [19], JPEG encoding [20], and GoogLeNet [21], as well as RTL prototypes and FPGA implementations with RISC-V–based tiles [22, 23, 24]. However, these studies are not compilers in the classic sense:

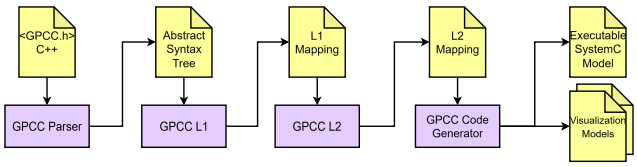


Figure 4: GPCC workflow

each mapping required significant manual effort to balance tasks, assign I/O, choose intermediate memories, and validate communication legality. Moreover, none of these flows automatically produce an executable SystemC TLM-2.0 model following the mapping.

Because of these differences, GPCC is not directly comparable to existing CGRA mappers, NoC compilers, or static placement tools. Those tools typically assume flexible routing, multi-hop communication, or globally shared memory architectures — assumptions that the GPC explicitly does not permit. In this paper, our GPCC instead targets the much stricter R-type checkerboard model, automatically bridging the gap between an untimed task graph and a runnable, grid-accurate TLM-2.0 architecture model. This includes task-graph extraction, task placement, memory and I/O assignment, and generation of complete SystemC code realizing the checkerboard-style neighbor communication semantics.

2 GPC Compiler

GPCC is a command-line compiler that automatically maps an application to a GPC. As shown in Fig. 4, GPCC reads a C++ model and constructs the application's Abstract Syntax Tree (AST). From this input, GPCC extracts the task graph, places tasks on a R-type checkerboard with user-defined dimensions $H \times W$, and generates a corresponding simulatable SystemC TLM-2.0 model that reflects the mapping.

In this section, we first describe how applications are specified, then explain our 2-level mapping algorithm, and finally present the output generation.

2.1 Input Specification

The input application for GPCC is specified as a hierarchical composition of modules (computation) connected by channels (communication). When flattened, the application forms a task graph, as shown in Fig. 2. The system designer can specify the module hierarchy in a C++ file (using a lightweight header `<GPCC.h>`) which the GPCC frontend parses, or by use of a Python API that directly constructs the modules and connecting channels as an internal representation.

Fig. 5 illustrates the GPCC Python API for building the JPEG encoder application. The construction starts with an Application object named `JPEG_Encoder`. Next, a `DCT` component is constructed as a Module object with two ports PortIn and PortOut. The functionality of the `DCT` module is specified as a `main` function that, in an infinite loop, reads a block of image pixels, transforms it using a ChenDCT function, and pushes the result to the output port. Finally, a

```
import GPCC_App
def Construct(AppName="JPEG_Encoder"):
    App = GPCC_App.Application(AppName)
    ...
    m = GPCC_App.Module("DCT")
    m.Ports.append(GPCC_App.Port("PortIn", "QUEUE_IN", "Block"))
    m.Ports.append(GPCC_App.Port("PortOut", "QUEUE_OUT", "Block"))
    m.Main.extend(r"""
        void main(void)
        {   Block in_Block, out_Block;
            while (1)
            {   PortIn.POP(in_Block);
                out_Block = ChenDCT(in_Block)
                PortOut.PUSH(out_Block);
            }
        }""".splitlines ())
    m.Threads.append(GPCC_App.Thread("main", detached=True))
    App.Modules.append(m)
    ...
    m = GPCC_App.Module("Encoder")
    m.Ports.append(GPCC_App.Port("Port1", "QUEUE_IN", "Block"))
    m.Ports.append(GPCC_App.Port("Port2", "QUEUE_OUT", "Block"))
    m.ChannelInsts.append(GPCC_App.ChannelInst("ch1","QUEUE","Block"))
    m.ChannelInsts.append(GPCC_App.ChannelInst("ch2","QUEUE","Block"))
    App.AddModuleInst(m, "dct", "DCT")
    App.AddModuleInst(m, "quan", "Quantize")
    App.AddModuleInst(m, "zigz", "Zigzag")
    m.AddChannelParams("ch1", [FIFO_SIZE])
    m.AddChannelParams("ch2", [FIFO_SIZE])
    m.AddPortMap("dct", ["Port1", "ch1"])
    m.AddPortMap("quan", ["ch1", "ch2"])
    m.AddPortMap("zigz", ["ch2", "Port2"])
    App.Modules.append(m)
    ...
    return App
```

Figure 5: Constructing the JPEG Encoder application with GPCC Python API

Thread is created in DCT to execute the `main` function.

Next, Fig. 5 shows the construction of a hierarchical module `Encoder` that contains an instance of the `DCT` module and two other modules `Quantize` and `Zigzag`. The modules process blocks of image pixels in a pipeline with the help of two channel instances `ch1` and `ch2`.

The construction of the `JPEG_Encoder` application is completed when all modules are instantiated in the App object.

2.2 Level 1 Mapping Algorithm

GPCC applies a two-level mapping strategy. At Level-1, tasks from the application graph are recursively placed onto the cells of the target checkerboard (Alg. 1). For each task, the set of candidate positions is the intersection of the neighbors of already mapped predecessors and successors. If no candidate exists, the search backtracks. In this way, the placement preserves task dependencies while keeping related tasks close together in the grid.

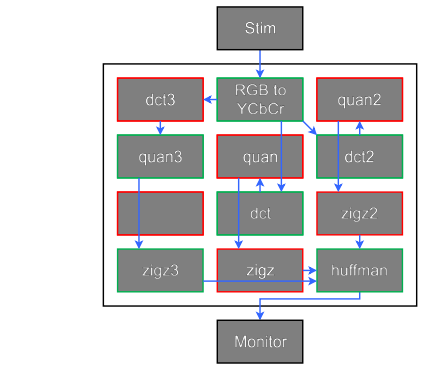
As an example, we illustrate Alg. 1 on the JPEG Encoder application. Fig. 2 shows the task graph with the three color channels, and Fig. 6a shows the result of Level-1 mapping.

The mapping process begins by assigning off-chip modules, such as Stim and Mon, to the off-chip I/O units which serve as source and sink of the input task graph. The DCT successors of the source node, in this case the RGB-to-YCbCr color convertor, are then added to the queue. When RGB-to-YCbCr is dequeued, its candidate positions are computed as the neighbors of the cell assigned to Stim. A valid cell is chosen and the mapping is extended. The same process is repeated for the remaining nodes, with each placement de-

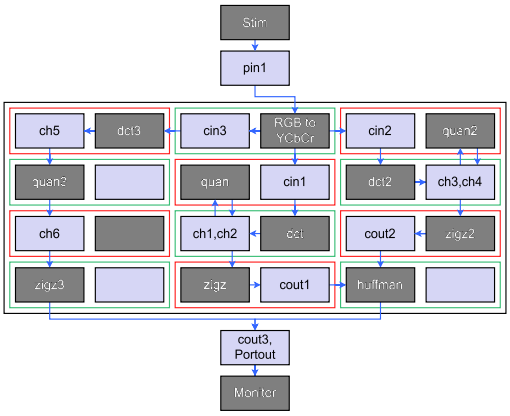
Algorithm 1: Level-1 Mapping with recursive search and user-defined premapping

Input : Application graph $A(T, D)$, with start task t_{start}
Input : Dimensions $H \times W$ of the checkerboard $G(C, N)$
Input : Premap relation $P \subseteq T \times C$
Output: Mapping m from tasks to grid cells (or $\emptyset$ if none)

Function *Mapper_L1*(A, G, t_{start}, P) **begin**
  $q := [t_{start}]$
  $v := \emptyset$, $m := \emptyset$
  return RecursiveMap(G, A, q, v, m, P)
end
Function *RecursiveMap*(G, A, q, v, m, P) **begin**
  if $q = \emptyset$ **then**
    return m
  end
  $t := q$.dequeue()
  $N := predecessors(t) \cup successors(t)$
  for $n \in N$ **do**
    if n *not mapped* **and** $n \notin q$ **then**
      q.enqueue(n)
    end
  end
  if t.targets $= \emptyset$ **then**
    if $\exists c \in C : (t, c) \in P$ **then**
      t.targets $:= \{ c \in C \mid (t, c) \in P \}$
    end
    else
      t.targets $:= \bigcap_{n \in N}$ neighbors$(m(n))$
    end
  end
  for $c \in t$.targets **do**
    return RecursiveMap$(G, A, q, v \cup \{t\}, m \cup \{(t, c)\}, P)$
  end
  return $\emptyset$
end



(a) Level-1 mapping



(b) Level-2 mapping

Figure 6: 2-Level Mapping of JPEG Encoder to a R-type Checkerboard

termined by

$$t.\text{targets} := \bigcap_{n \in N} \text{neighbors}(m(n))$$

where $m(n)$ returns the cell mapped to node n, and neighbors(c) returns all accessible neighbors of cell c. If no candidates remain, the algorithm backtracks and explores alternative choices. At the end of the recursion, all tasks from the JPEG graph are assigned to cells in the GPC (Fig. 6a), ensuring that data dependencies are respected and communicating tasks are placed next to each other.

Given that graph mapping is generally known as NP-complete, our recursive search strategy with backtracking is of the same complexity. For many real-world examples, however, Alg. 1 completes in reasonable time, as we will show in Sec. 4. If needed, the designer may also apply a timeout to each GPCC call during DSE.

2.3 User-Defined Premapping

In addition to fully automatic mapping, GPCC supports *user-defined premapping* to constrain the placement of selected tasks onto specific grid cells. Premapping allows designers to incorporate prior architectural knowledge, I/O constraints, or manually optimized placements into the otherwise automated mapping process. Rather than prescribing a complete assignment, premapping specifies only a subset of task-to-cell bindings, leaving the remaining tasks to be placed automatically by the compiler.

Formally, premapping is expressed as a partial relation $P \subseteq T \times C$, where T denotes the set of application tasks and C the set of available grid cells. Each pair $(t, c) \in P$ restricts task t to be placed on cell c. Tasks not contained in P remain unconstrained and are handled by the recursive Level-1 mapping algorithm. By modeling premapping as a relation rather than a complete function, GPCC naturally supports partial and incremental placement specifications.

Premapping is integrated into Level-1 mapping by constraining the candidate cell set of affected tasks. During recursive search, when a task t is selected for placement, its candidate targets t.targets are first checked against the premapping relation. If t appears in P, the candidate set is restricted to the user-specified cells. Otherwise, the original neighbor-intersection rule is applied, preserving locality constraints imposed by already placed predecessor and successor tasks. In this way, premapping narrows the search space without altering the fundamental structure of the recursive mapping algorithm.

This mechanism enables a range of practical use cases. For example, computationally intensive tasks can be anchored to central regions of the grid, or previously validated placements can be reused across design space exploration runs. At the same time, GPCC ensures that premapped constraints are respected consistently throughout mapping and exploration, and infeasible premaps are detected automati-

cally when no valid completion of the mapping exists.

Algorithm 2: Level-2 Mapping of data channels to memories

```
   Input  : Task graph A(T, D), Level-1 mapping m on grid
            G(C, M)
   Output: Extended mapping m including data channels
1  Function Mapper_L2(A, G, m) begin
2      foreach edge (s, d) ∈ D do
3          c_s := m(s), c_d := m(d)
4          E := neighbors(c_s) ∩ neighbors(c_d)
5          if E ≠ ∅ then
6              ∀m_e ∈ E: m := m ∪ {((s, d), m_e)}
7          else
8              return ∅
9          end
10     end
11     return m
12 end
```

2.4 Level 2 Mapping Algorithm

Level-2 mapping then assigns the communication edges of the task graph to memory cells, as shown in Alg. 2. For each edge (s, d) in the graph, the algorithm finds a memory cell that is adjacent to both the source and destination cells. If multiple candidates exist, the one with lowest usage is selected for load balancing; if no candidate is available, the placement is considered infeasible. This step ensures that every channel has a realizable path through memory, and that off-chip communication flows through the correct boundary ports of the grid. The result for the JPEG encoder is shown in Fig. 6b, where memory cells are assigned to connect the previously mapped tasks.

The complexity of Alg. 2 is linear with respect to the number of edges in the graph, $O(e)$. Together, the two levels produce a complete mapping, which is then translated into a simulatable SystemC TLM-2.0 model by the GPCC code generator.

Beyond producing valid assignments, GPCC mapper also evaluates the mapping using several metrics, including the grid size, on-chip utilization, memory usage, and communication distance. These measures provide a quick assessment of mapping efficiency and guide the explorer in selecting Pareto-optimal GPC dimensions.

3 GPCC Exploration

Given that GPCC automatically maps an application graph to the target checkerboard, the next problem is to identify suitable dimensions. As indicated above, there is a tradeoff between the height H and width W of the target GPC, which determines its number of cells and thus the area cost $H \times W$, and the need to map the unstructured application graph to the regular grid.

We define the grid selection problem as finding the smallest R-type checkerboard that can accommodate the application. Also, we prefer square grids.

Formally, minimize $H \times W$, subject to fitting the given application such that GPCC successfully finds a mapping. Secondary, minimize $H + W$ for squareness.

We note that the dimensions of the grid are independent axes, thus, this is a multi-objective optimization problem where Pareto principles apply. We design a GPC Explorer `gpc_explore` which automatically explores the design space to identify Pareto-optimal grid dimensions. Based on the number of tasks specified in the application graph, `gpc_explore` generates and visualizes an exploration map of the design space with GPC targets tabulated by size, as illustrated at the top row of Fig. 7. Each entry in the two-dimensional map reflects the height and width of a potential target checkerboard.

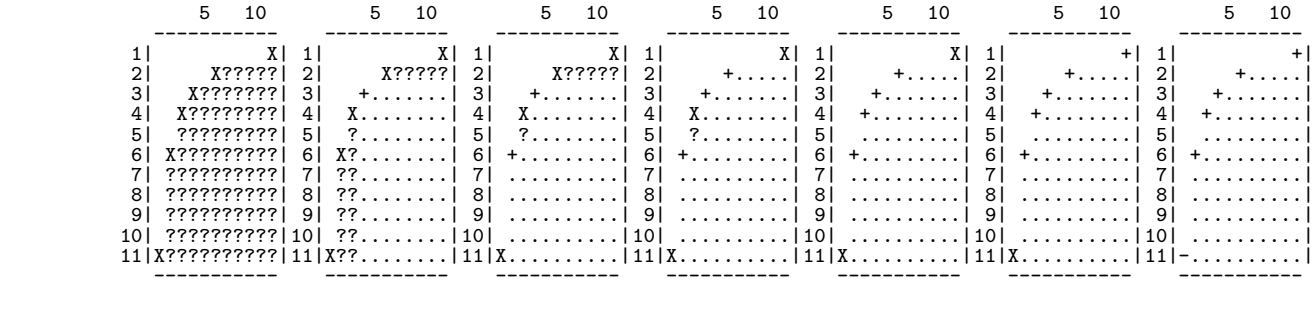
`gpc_explore` distinguishes different GPC dimensions based on their suitability for the given application and produces a Pareto-optimal front which separates feasible from infeasible solutions. Grid sizes too small for the application are identified as *infeasible* and visualized as empty space. `gpc_explore` then computes an initial front of potentially Pareto-optimal solutions that are termed *explorable* targets and shows them as `X` characters. Alg. 3 shows how this initial front is computed (complexity $O(n)$). Based on the given number of nodes n, the algorithm enumerates a list X of pairs (h, w) where no element dominates any other element. Beyond this initial Pareto front, all other solutions are marked as *unknown* and indicated by question marks (`?` character), as shown in Fig. 7a.

Algorithm 3: Compute the initial Pareto front of explorable solutions in X

```
   Function InitialParetoFront(n) ↦ X begin
       Input  : n number of nodes to be mapped
       Output: X initial front to be explored
1      l := [ ]
2      for w := 1 to n do
3          h := ⌈n/w⌉
4          if w > h then
5              break
6          end
7          l.append((h, w))
8      end
9      X := [ ]
10     for (h, w) ∈ l do
11         if h ≠ w then
12             X.append((w, h))
13         end
14     end
15     X.extend(reverse(l))
16     return (X)
   end
```

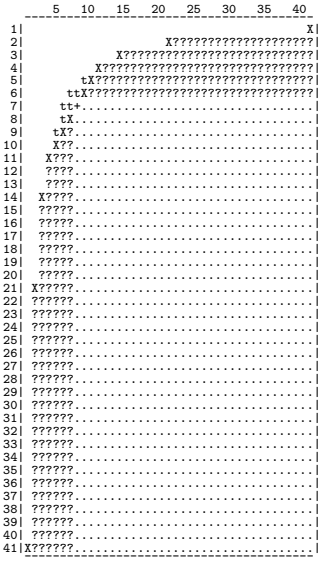
`gpc_explore` then systematically calls GPCC to determine the actual state of explorable and unknown solutions. If GPCC successfully finds a mapping, the particular solution is termed *realizable*, marked with a `+` character, and all its dominated solutions are marked as *suboptimal* with a dot (`.` character), since they are of no interest anymore and can be pruned from further exploration. This progress is illustrated in Fig. 7b through Fig. 7g for the JPEG Encoder application. Here, the Pareto-optimal exploration finds a realizable mapping for five of the six explorable options. Only the last one, an extremely thin 1×11 CBR grid, is found *unrealizable* and is marked with a `-` character.

So many realizable solutions, however, are not usually



	(a) Start		(b) Step 1		(c) Step 2		(d) Step 3		(e) Step 4		(f) Step 5		(g) Step 6	
Realizable:	0	(0.0%)	1	(0.8%)	2	(1.7%)	3	(2.5%)	4	(3.3%)	5	(4.1%)	5	(4.1%)
Suboptimal:	0	(0.0%)	71	(58.7%)	82	(67.8%)	87	(71.9%)	88	(72.7%)	88	(72.7%)	88	(72.7%)
Infeasible:	27	(22.3%)	27	(22.3%)	27	(22.3%)	27	(22.3%)	27	(22.3%)	27	(22.3%)	27	(22.3%)
Unrealizable:	0	(0.0%)	0	(0.0%)	0	(0.0%)	0	(0.0%)	0	(0.0%)	0	(0.0%)	1	(0.8%)
Explorable:	6	(5.0%)	5	(4.1%)	4	(3.3%)	3	(2.5%)	2	(1.7%)	1	(0.8%)	0	(0.0%)
Unknown:	88	(72.7%)	17	(14.0%)	6	(5.0%)	1	(0.8%)	0	(0.0%)	0	(0.0%)	0	(0.0%)
Total:	121	(100.0%)	121	(100.0%)	121	(100.0%)	121	(100.0%)	121	(100.0%)	121	(100.0%)	121	(100.0%)

Figure 7: Pareto-optimal design space exploration of the JPEG Encoder example on R-type checkerboard.



Realizable:	1	(0.1%)
Suboptimal:	1189	(70.7%)
Infeasible:	158	(9.4%)
Timed-out:	7	(0.4%)
Unrealizable:	0	(0.0%)
Explorable:	13	(0.8%)
Unknown:	313	(18.6%)
Total:	1681	(100.0%)

Figure 8: Pareto-optimal search for a first realizable solution.

the case. Due to the intentionally limited connectivity, successful mappings of complex task graphs may not be found by GPCC. These cases are distinguished as *unrealizable* (indicated as `-`) or *timeout* (recorded as `t`), and the Pareto front is adjusted accordingly with added explorable solutions to the right and below the current candidate.

Fig. 8 shows an initial exploration of a synthetic example (called "large_4" below) where the system designer is interested in quickly finding a first realizable solution. Here gpc_explore is called with a timeout of 10 seconds for each GPCC run and stops as soon as the first realizable model is found. This happens in step 8 after 72 seconds of exploration time. As illustrated in Fig. 8, 7 attempts timed out, but the 8th run successfully found a mapping on a 7×8 checkerboard.

Alg. 4 shows the gpc_explore sequential design space exploration algorithm in pseudo code. Given the explorable front X determined by Alg. 3, gpc_explore repeatedly calls the GPC compiler to realize the candidates until the explorable set X is empty or an early termination condition, e.g. s_{max} steps or t_{max} runtime, is reached.

Algorithm 4: Design space exploration along the Pareto-optimal front in X (sequential approach)

Function *Explore(X)* $\mapsto$ R **begin**
- **Input :** X initial front to be explored
- **Input :** s_{max} maximum number of exploration steps
- **Input :** t_{max} maximum exploration time
- **Output:** R Pareto-optimal front of realizable solutions
- **Output:** U list of unrealizable solutions
- **Output:** T list of timed-out solutions

1 $R := [\,]$
2 $U := [\,]$
3 $T := [\,]$
4 $s := 0$
5 **while** X **do**
6 $x :=$ pick(X)
7 $r :=$ run_gpcc(x)
8 **if** $r =$ *success* **then**
9 R.insert(x)
10 **else**
11 **if** $r =$ *timeout* **then**
12 T.append(x)
13 **else**
14 U.append(x)
15 **end**
16 $(h, w) := x$
17 **for** $x_{new} \in \{(h+1, w), (h, w+1)\}$ **do**
18 **if** $\neg$ *dominates*$(R \cup X,\ x_{new})$ **then**
19 X.insert(x_{new})
20 **end**
21 **end**
22 **if** $s \geq s_{max}$ *or time()* $\geq t_{max}$ **then**
23 break
24 **end**
25 $s := s + 1$
26 **end**
27 **return** $((R, U, T))$

end

While the order of picking $x \in X$ does not matter in a complete DSE due to the Pareto optimization, gpc_explore picks first solutions that have maximum impact (most dominated points that are unknown), so that an exploration with early termination has the most known solutions. The complexity of Alg. 4 is $O(n)$ in the best case (when the initial Pareto front is realizable) and $O(n^2)$ in the worst case (when no realizable options are found).

4 Experiments and Results

We now evaluate our GPCC approach on a few embedded applications and a benchmark set of synthetic examples.

Table 1: Exploration Results for Image Processing Applications

Name[config] (#V, #E)	Exploration CPU time	Realized Mappings
JPEG Encoder (11,14)	0:02 sec	5: 3x4 4x3 2x6...
Mandelbrot[3x2] (8,12)	0:02 sec	5: 3x3 2x4 4x2...
Mandelbrot[3x4] (14,24)	0:06 sec	5: 3x5 4x4 2x7...
Mandelbrot[5x2] (12,20)	0:02 sec	6: 3x4 4x3 2x6...
Mandelbrot[5x4] (22,40)	0:38 sec	7: 4x6 3x8 5x5...
Mandelbrot[5x5] (27,50)	3:07 min	8: 4x7 5x6 3x9...
APNG Encoder (18,32)	0:01 sec	0: None

4.1 Embedded Application Examples

For an evaluation using real-world benchmarks, we let GPCC map and explore three image processing applications, namely the JPEG image encoder (see Fig. 2, Fig. 3, Fig. 6, and Fig. 7), a configurable Mandelbrot renderer [17], and an APNG video encoder [19]. These applications vary in their data flow and complexity, with task graphs between 8 and 27 nodes, and between 12 and 50 edges.

Tab. 1 shows our exploration results. For the JPEG encoder, exploration takes only 2 seconds time and produces the 5 models depicted in Fig. 7 which all simulate correctly.

The Mandelbrot renderer can be configured in the number of vertical and horizontal slices which compute in parallel. The small configurations are explored within a few seconds, whereas the larger 5x4 and 5x5 configurations take less than a few minutes. Again, all generated TLM2 models simulate correctly.

Only the APNG encoder cannot be mapped to our checkerboard due to a large fanout of its RGB node with 12 successors. Our GPCC explorer detects this quickly, as the CBR target does not provide a sufficient number of immediate neighbors.

4.2 Synthetic Applications

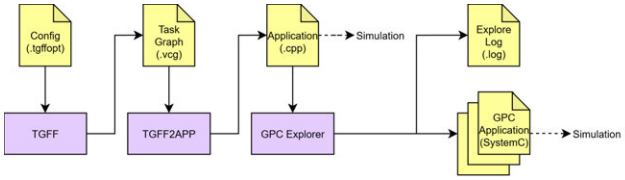


Figure 9: Synthetic Application Generation Process

Table 2: Exploration Results for Synthetic Applications

Name (#V, #E)	Exploration CPU time	Realized Mappings
small_1 (6, 5)	0:08 sec	3: 2x4 3x3 1x6
small_2 (7, 6)	0:05 sec	4: 3x3 2x5 4x2...
small_3 (11, 10)	0:04 sec	5: 3x4 4x3 2x6...
small_4 (11, 12)	0:04 sec	5: 3x4 4x3 6x2...
medium_1 (18, 18)	7:36 min	4: 2x9 3x6 5x4...
medium_2 (20, 20)	8:07 min	5: 3x7 4x5 5x4...
medium_3 (21, 22)	38:37 min	2: 3x7 1x21
medium_4 (21, 23)	0:29 sec	7: 5x5 4x6 6x4...
large_1 (22, 32)	24:02 min	4: 4x6 3x9 2x11...
large_2 (30, 30)	31:10 min	6: 5x6 4x8 7x5...
large_3 (31, 32)	32:53 min	7: 5x7 6x6 4x10...
large_4 (41, 44)	89:08 min	5: 7x8 3x14 4x13...

4.2.1 Synthetic Application Generation

Fig. 9 shows the synthetic application generation process. We use TGFF [25] to generate pseudo-random task graphs, and develop a TGFF-to-application converter (tgff2app) [26] to process a TGFF task graph and produce a lightweight C++ application suitable for GPCC. We then let GPC Explorer identify suitable checkerboard grids.

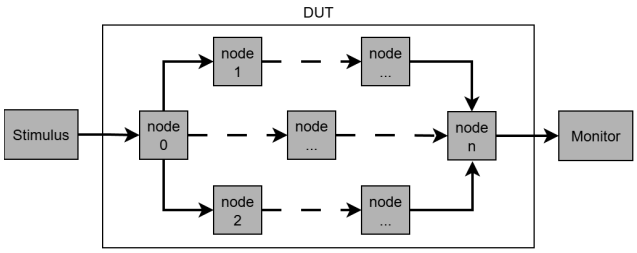
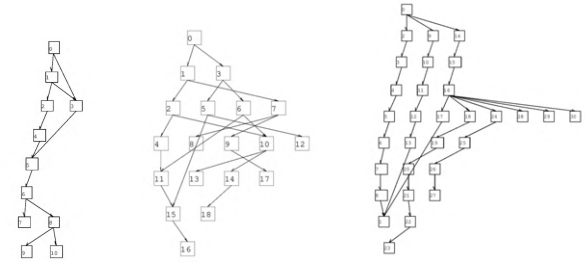


Figure 10: Generic Structure of Synthetic Applications

Fig. 10 shows the general structure of synthetic application: The Design Under Test (DUT) is derived from the task graph: each vertex becomes a submodule and each edge becomes a channel for data transfer. Stimulus sends the input data to the DUT, which processes it and forwards the results to the monitor for observation and validation.



(a) small_4 (b) medium_2 (c) large_3

Figure 11: Sample TGFF Task Graphs of Generated Applications

GPCC is evaluated on 12 synthetic applications generated from TGFF's example configurations [27]. Fig. 11 shows the task graphs of three representative applications.

4.2.2 Results

Tab. 2 summarizes our experimental results. GPCC successfully maps all 12 applications on the CBR. These applications are grouped into small, medium, and large based the on number of vertices and edges (#V, #E) in their task graphs. For all cases, the generated SystemC TLM-2.0 models produce simulation results identical to those of the lightweight applications generated by `tgff2app`, confirming functional correctness. The DSE completes quickly relative to scale. The exploration time for application `medium_4` is notably short among its group because pareto optimal solutions are found early, reducing the need to explore additional candidates. In contrast, application `medium_3` requires substantially more exploration time, as viable solutions are found only after exploring most of the possible dimensions. In addition to GPC exploration, other stages in the flow of synthetic application generation (TGFF, TGFF2APP, and Simulation) finish in a second. Overall, the results demonstrate the effectiveness and robustness of our GPCC tools for applications of different complexity.

5 Conclusion

This work introduced an automated compilation and exploration framework for the Grid of Processing Cells R-type checkerboard architecture. By combining the GPCC compiler with the GPC Explorer, the proposed approach transforms directed acyclic task graphs directly into runnable SystemC TLM-2.0 models while systematically exploring feasible R-type checkerboard configurations. The toolchain replaces manual, error-prone mapping with an end-to-end flow that encompasses task placement, memory assignment, and communication realization. Experimental results on embedded workloads and TGFF-generated synthetic applications confirm that the framework consistently produces correct mappings and efficiently characterizes the design space, revealing Pareto-optimal grid configurations and clearly separating feasible from infeasible architectures.

Future work will extend the framework with improved load-balancing strategies and timing-annotated models to enable performance-aware evaluation. In addition, parallelizing the exploration process will further reduce design-space traversal time and improve scalability for even larger and more complex applications.

References

[1] H. T. Kung and C. E. Leiserson, “Systolic arrays (for vlsi),” in *Sparse Matrix Proceedings 1978*, vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979, pp. 256–282.

[2] B. De Sutter, P. Raghavan, and A. Lambrechts, *Coarse-Grained Reconfigurable Array Architectures*, 07 2010, pp. 449–484.

[3] M. Wijtvliet, L. Waeijen, and H. Corporaal, “Coarse grained reconfigurable architectures in the past 25 years: Overview and classification,” in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016, pp. 235–244.

[4] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, “A network on chip architecture and design methodology,” in *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, 2002, pp. 117–124.

[5] A. A. Khan, J. P. C. de Lima, H. Farzaneh, and J. Castrillón, “The landscape of compute-near-memory and compute-in-memory: A research and commercial overview,” *ArXiv*, vol. abs/2401.14428, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267301099

[6] J. von Neumann, “First draft of a report on the EDVAC,” University of Pennsylvania, Tech. Rep., Jun. 1945.

[7] R. Dömer, “A Grid of Processing Cells (GPC) with Local Memories,” Center for Embedded and Cyber-physical Systems, UCI, Tech. Rep. CECS-TR-22-01, Apr. 2022.

[8] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring network structure, dynamics, and function using networkx,” in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[9] “IEEE Standard for Standard SystemC Language Reference Manual,” *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, 2012.

[10] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, “Graphviz— open source graph drawing tools,” in *Graph Drawing*, P. Mutzel, M. Jünger, and S. Leipert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 483–484.

[11] J. Melchert, K. Zhang, Y. Mei, M. Horowitz, C. Torng, and P. Raina, “Canal: A flexible interconnect generator for coarse-grained reconfigurable arrays,” 2022. [Online]. Available: https://arxiv.org/abs/2211.17207

[12] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, “Cgra-me: A unified framework for cgra modelling and exploration,” in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 184–189.

[13] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, “On-chip interconnection architecture of the tile processor,” *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[14] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008, pp. 88–598.

[15] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28 tflops network-on-chip in 65nm cmos," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, 2007, pp. 98–589.

[16] L. Luchterhandt, V. Govindasamy, Y. Wang, C. Scheytt, W. Mueller, and R. Dömer, "A quantitative guide to navigate speed/accuracy tradeoffs in system level design of RISC-V processor grids," in *2025 Forum on Specification and Design Languages (FDL)*, 2025, pp. 1–9.

[17] Y. Wang, A. Daroui, and R. Dömer, "Demonstrating Scalability of the Checkerboard GPC with SystemC TLM-2.0," in *Proceedings of the International Embedded Systems Symposium (IESS)*. Lippstadt, Germany: Springer, Nov. 2022.

[18] V. Govindasamy and R. Dömer, "Mixed-level modeling and evaluation of a cache-less grid of processing cells," *ACM Transactions on Embedded Computing Systems*, Dec. 2024. [Online]. Available: https://doi.org/10.1145/3708988

[19] V. Govindasamy, E. Arasteh, and R. Dömer, "Minimizing Memory Contention in an APNG Encoder using a Grid of Processing Cells," in *Proceedings of the International Embedded Systems Symposium (IESS)*. Lippstadt, Germany: Springer, Nov. 2022.

[20] A. Daroui and R. Dömer, "A Loosely-Timed TLM-2.0 Model of a JPEG Encoder on a Checkerboard GPC," Center for Embedded and Cyber-physical Systems, UCI, Tech. Rep. CECS-TR-22-04, Oct. 2022.

[21] C. Raccomandato, E. M. Arasteh, and R. Dömer, "MapGL: Interactive application mapping and profiling on a grid of processing cells," in *Proceedings of the Design and Verification Conference*, Munich, Germany, Nov. 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:266492571

[22] L. Luchterhandt, T. Nellius, R. Beck, R. Dömer, P. Kneuper, W. Mueller, and B. Sadiye, "Implementation of different communication structures for a rocket chip based RISC-V grid of processing cells," in *MBMV 2024; 27. Workshop*, 2024, pp. 79–89.

[23] L. Luchterhandt, V. Govindasamy, Y. Wang, R. Dömer, W. Mueller, and C. Scheytt, "Case study on combining open-source tool flows for grids of processing cells," in *Open Source Solutions for Massively Parallel Integrated Circuits (OSSMPIC) Workshop at the Design, Automation and Test in Europe (DATE) Conference*, Lyon, France, Apr. 2025, pp. 1–4.

[24] ——, "Truly scalable grids of RISC-V processors with local memories," in *MBMV 2026; 29. Workshop*, 2026.

[25] R. Dick, D. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE'98)*, 1998, pp. 97–101.

[26] Y. Li, Y. Wang, and R. Dömer, "Generating synthetic data flow models in systemc TLM based on TGFF," Center for Embedded and Cyber-physical Systems, UCI, Tech. Rep. CECS-TR-24-02, May 2024.

[27] K. Vallerio, *Task Graphs for Free (TGFF v3.0)*, 2008, accessed 2025-08-31. [Online]. Available: https://robertdick.org/projects/tgff/