Exploiting irregular parallelism to accelerate FPGA routing

by

Alan Y. Zhu

S.B. Writing and Computer Science and Engineering
Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

Authored by:	Alan Y. Zhu Department of Electrical Engineering and Computer Science August 30, 2024
Certified by:	Daniel Sanchez Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Exploiting irregular parallelism to accelerate FPGA routing

by

Alan Y. Zhu

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

In the era of hardware specialization, field-programmable gate arrays (FPGAs) provide a promising platform for computer architects, combining the programmability of software with the speed and performance of hardware. Despite this, compiling hardware programs onto FPGAs can be incredibly time-consuming, making it hard to develop and iterate on complex FPGA programs. Of particular relevance is the routing phase, which takes a circuit's technology-mapped netlist and routes its signals using the switches and wires present on a given FPGA architecture, often with a target of minimizing critical path delay. This optimization problem is known to be NP-hard, and existing algorithms for approximating it exhibit very little regular parallelism.

This thesis accelerates the routing phase of VTR 8.0, a commonly used, open-source research tool for FPGA CAD flow. We show that despite the lack of regular parallelism, routing still exhibits significant irregular parallelism. This parallelism can be exploited on parallel architectures that provide hardware support for ordered tasks and fine-grained speculation, such as the Swarm architecture. Using Swarm, we exploit the parallelism present at the core of VTR's algorithm, achieving a 35.9x speedup on a single routing iteration of a large benchmark (`cholesky_mc`) on 256 cores.

Thesis supervisor: Daniel Sanchez
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I am very grateful to have had the support of many people during the completion of this thesis project, who I will do my best to thank accordingly:

With regards to the thesis itself, I am incredibly thankful to Professor Daniel Sanchez for introducing me to the FPGA routing problem and the Swarm architecture; I have learned so much about the world of computer architecture and irregular parallelism in the process, and he has helped me point my focus in the right direction when there were so many more places to become distracted or lost. I am also very grateful to Fares Elsabbagh, who survived my unceasing barrage of questions, helped me navigate the innumerable challenges which cropped up during this project, and has taught me about everything from dataflow architectures to the details of how, exactly, to navigate all the infrastructure the group has.

With regards to the arc of my academic interests, I am especially grateful for Silvina Hanono Wachman and Professor Mengjia Yan, whose teaching in 6.004 first piqued my interest in computer systems and architecture and led me down the path to where I am today. I am additionally thankful to Silvina, Professor Christina Delimitrou, and Joe Steinmeyer for having me as a teaching assistant for 6.191 and 6.190; serving as a TA for their classes was incredibly rewarding and helped keep my skills sharp, contributing to a number of specific tricks used in this thesis.

I'm thankful to many of the MIT staff and faculty who have taken an interest in my progress and given me great advice, academic or otherwise. I'm especially thankful to my undergraduate advisor, Nick Montfort; my boss at MIT Admissions, Chris Peterson; and my academic administrators, Shannon Larkin and Katrina LaCurts, all of whom have helped me through one crisis or another with great care and thoughtfulness.

I am incredibly grateful for the support of all of my friends, who often believed in me even when I did not. I am particularly grateful to my former roommates, Tong Zhao, Matthew Ho, and Krit Boonsiriseth, as well as my eternal first reader, Fatima Abbasi. Shuli Jones, Paolo Adajar, Jonathan Huang, and Tristan Shin have always been great sources of advice and comfort, particularly when it comes to confronting the existential question of "what comes next?" I also could not have made it through this year and summer without the friendship of Katie Kitzinger, Alice Le, Silu Shen, Kelsey Glover, and so many others.

Last, but certainly not least, I owe a world of gratitude to my sister, Megan. I'm so thankful to have had a year here with you around, so proud of all you've done already, and so excited to see what you do next.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Field-programmable gate arrays (FPGAs) have become increasingly popular as industrial demand for hardware accelerators has risen. In contrast to application-specific integrated circuits (ASICs), FPGAs can be reconfigured by the end user, promising the performance of hardware with reprogrammability of software. This has allowed programmers to accelerate applications which require significant flexibility over time, such as software-defined networking [1] or search engine ranking [2], and the widespread availability of FPGAs has allowed for innovation by individuals and companies which would otherwise not have access to specialized hardware platforms.

Unfortunately, as modern systems have gotten larger, compiling programs onto FPGAs has become incredibly complex and time-consuming. FPGA programs are typically specified using hardware description languages (HDLs) and compiled down to hardware primitives over a number of steps, often using device-specific, commercial tools such as Vivado [3] or Quartus [4]. For a two-FPGA design with 128 processing elements, compilation can take 13 hours [5]; for even larger designs, the process may take days or even weeks, making iteration difficult.

In this thesis, we parallelize the routing phase of FPGA compilation. Routing takes the logic elements of a given design, which have already been placed onto the board, and connects

them using the reconfigurable switches available on the FPGA. Algorithms for routing have been hard to accelerate due to their irregular parallelism: like most graph algorithms, access patterns are not known a priori, making independent threads of computation hard to identify and exploit.

Prior work has addressed this difficulty through different software techniques. Some parallel routers perform routing without concern for potential dependences; these routers are non-deterministic, making them impractical to use in situations where repeatability is crucial [6]. Other routers pre-partition the problem into independent subproblems which must be scheduled [7, 8], which requires software synchronization and is limited in parallelism by the number of potentially independent subproblems. Finally, other routers exploit fine-grained parallelism in the graph search at the core of FPGA routing using a conventional multicore, but this produces poor results given the overheads of CPU threads with small tasks [9].

Given these challenges, hardware support for ordered, fine-grained parallelism is necessary to fully exploit the parallelism present in FPGA routing. We accelerate the routing phase for a commonly used, open-source research tool for computer-aided design (CAD) of FPGAs, Verilog-to-Routing (VTR) 8.0 [10], using the Swarm hardware architecture [11]. Swarm is a tiled multicore which uses tiny, ordered tasks, speculative execution, and selective aborts to exploit irregular parallelism that is otherwise unavailable on conventional multicores. Using a number of different techniques, we explore the parallelism present in the FPGA routing phase and use it to provide a significant speedup for an important problem.

1.1 Contributions

We make the following contributions:

1. We implement FPGA routing in Swarm. We demonstrate that Swarm's task-based execution model is sufficient to capture the irregular parallelism present in a complex application (VTR) without significant changes to its approach. We also introduce a

number of strategies for using the Swarm execution model which could be useful in accelerating other applications.

2. We evaluate our implementation in the Swarm simulator and find that it produces a 35.9x speedup over VTR on realistically sized benchmarks with 256 cores with minimal losses in routing quality. We also identify some barriers to further acceleration, and propose potential solutions for them.

Chapter 2

Background

2.1 The FPGA Routing Problem

Traditional FPGAs consist of an array of configurable logic blocks (CLBs) and a network of wires; the connections between these wires pass through routing switches, which are programmable and can connect different wires to each other. Modern FPGAs also have specialized hardware blocks which serve common use cases, including digital signal processing blocks (DSPs), memories, and even full ARM cores. Many of these blocks are reconfigurable; for example, CLBs contain lookup tables (LUTs) which implement simple Boolean functions, taking in a fixed number of inputs and producing a configurable output.

Once a circuit has been synthesized for a particular FPGA's hardware primitives, FPGA CAD tools compile that circuit onto the FPGA through two distinct phases: placement and routing. Placement takes these hardware primitives and places them on the FPGA; for example, it assigns circuit logic to the specific CLBs available on the FPGA based on some optimization objective, such as wirelength. The routing phase's task is to connect these already configured circuit elements to each other using the actual wires and switches available in the FPGA's interconnect, such that the output of every element (e.g., a logic gate) connects only to the corresponding inputs it should be wired to (e.g., the input of a

DSP). These outputs and inputs are known as pins.

Our work focuses on the routing phase, which we define more formally as follows:

> Let a particular FPGA architecture's *routing resource graph* (RRG) be a directed graph $G = (V, E)$ where the vertices V correspond to the architecture's pins and wires, and the edges correspond to connections between them (e.g., a routing switch which connects different wire segments). Then, for a given placement of an FPGA program, let a net N_i be the combination of a signal, s_i, and its sinks $T_i = \{t_{i1}, t_{i2}, \ldots, t_{im}\}$, and say a net N_i is routed by some route tree RT_i if RT_i is a tree in G with root s_i and $T_i \subset RT_i$. Find a set of route trees RT_i such that every net is routed and every route tree is disjoint.

2.1.1 Pathfinder

FPGA routing is, in general, NP-complete [9, 12]. As such, heuristic algorithms which iteratively converge to an approximate solution are necessary to perform routing. Most FPGA routing algorithms, both commercial and academic, are ultimately based on PathFinder [13], which iteratively re-routes every net in a fixed order using Dijkstra's algorithm repeatedly until no nets share routing resources (Algorithm 1).

While routing a given sink t_{ij}, the cost of a node in PathFinder, c_n, is the combination of two factors: the intrinsic delay of a given node and its congestion (i.e., the number of other routes using this node). The relative weighting of these two terms is given by the *criticality* of a node to the routing, which is defined as D_{ij}/D_{max}, where D_{ij} is the delay of the longest path containing the route from s_i to t_{ij} and D_{max} is the delay of the longest path overall; i.e., the critical path. The congestion cost of nodes already on the route tree is set to zero. PathFinder routes sinks in order from most to least critical.

There are a few important characteristics of this algorithm: first, the route tree for each signal-sink connection depends on the routing of the previous connections within the same net. This is ideal because it allows for reuse of routing resources for sinks which are near each

Algorithm 1 The original PathFinder algorithm [13]

```
while shared resources exist do
    for each net N_i do
        Rip up routing tree RT_i
        RT_i ← s_i
        for each signal t_ij ∈ N_i do
            Initialize priority queue PQ to RT_i
            while PQ is non-empty do                          ▷ Dijkstra's algorithm
                Remove lowest cost node m from PQ, with path cost P_im
                if m is t_ij then
                    break
                end if
                for each child n of node m do
                    Add n to PQ at cost c_n + P_im
                end for
            end while
            for each node n in path from s_i to t_ij do       ▷ backtrace
                Update c_n
                Add n to RT_i
            end for
        end for
    end for
end while
```

other, but creates a data dependence between the routing of those connections. Second, its performance is dependent on the order in which nets are routed: different orders can lead to significant differences in PathFinder performance [14, 15]. Third, all nets are fully re-routed in each iteration, which can be a significant amount of work on modern circuits with many nets.

2.1.2 Related Work

Work on improving FPGA routing tends to have two focuses: providing improvements to serial algorithms for routing, or improving runtime by looking for traditional forms of parallelism in existing algorithms, usually on multicore general-purpose processor systems. Notably, since the FPGA routing problem is NP-complete, a tradeoff also exists between runtime of the algorithm and quality of the routing [16]; there are also multiple criteria for FPGA routing quality, such as wirelength and critical-path delay.

Serial Improvements

Most work on providing better serial algorithms for routing is based off of PathFinder. Much recent work focuses on decreasing the amount of re-routing performed by the routing algorithm; many algorithms, for example, have shifted from ripping up entire nets on each iteration to only re-routing source-sink connections which contain congested nodes [16, 17]. Other optimizations include only expanding portions of the existing route tree close to the target sink for sinks which are not critical [18].

One particular area of focus is reassessing the costs from the original PathFinder algorithm, which reflected FPGA architectures at the time of writing. By adding different heuristics, algorithms have minimized the Manhattan distance of explored nodes [17], wirelengths on different architectures [16, 19], or the delay incurred along the path [16]. The introduction of these heuristics transform PathFinder's Dijkstra's-based router into an A*-based router. Other work has explored reducing the order dependence and increasing the

stability of PathFinder by changing how pressure and congestion costs are calculated over time [15].

Although our work in this thesis focuses on providing parallel speedup for a particular router implementation, our techniques are applicable to these serial algorithms because of their common, Pathfinder-based approach. In other words, the parallelism we find in our work is orthogonal to the serial speedup provided by the previous work here.

Parallel Routers

PathFinder is a difficult challenge for traditional parallelism, because its parallelism is irregular; as in most graph algorithms, the nodes that are visited (and, correspondingly, their memory accesses) are dependent on the nodes that are visited previously and cannot be predicted at compile time. Parallel routers fall into two categories - coarse-grained routers and fine-grained routers [6]. Coarse-grained routers [7, 8] work by partitioning the routing problem into independent problems which can be passed to different instances of VTR; this is done by clustering groups of nets with overlapping bounding boxes and then scheduling them in a reasonable order. Fine-grained routers, by contrast, start by parallelizing the shortest-paths solver; for example, Moctar *et. al.* [9], uses a software transactional memory to speed up A* search by allowing threads to expand multiple nodes at once, using locks to ensure accesses remain disjoint.

Works which perform hardware acceleration of FPGA routing, e.g., using the GPU or another FPGA, are more sparse in the literature. Corolla [20] combines a coarse-grained parallel router with GPU-based fine-grained acceleration of a Bellman-Ford-based shortest-paths solver, achieving an 18x speedup. Korolija and Stojilović [21] implement priority queues, wavefront expansion, and a cost-resetting DMA module on an FPGA to accelerate A* search; they achieve 2-4x speedup against their baseline and no speedup against VTR.

Beyond the quality-runtime tradeoff identified earlier, an important characteristic for parallel routers is determinism. In order to achieve consistent results and allow for reliable

testing, it is important for routers to produce repeatedly the same output, given the same inputs. Determinism is generally guaranteed by serial execution but is not always certain for parallel routers.

2.1.3 Verilog-to-Routing (VTR)

Verilog-to-Routing [10, 22] is a state-of-the-art, open-source, academic system which performs HDL compilation for FPGAs. It contains many tools for synthesizing digital circuits written in Verilog onto an FPGA, including Versatile Place and Route (VPR), which takes in a BLIF netlist describing a circuit which has been technology-mapped to a particular FPGA architecture, and produces packing, placement, and routing files for the given circuit on that architecture.

VPR's router [23] is a serial router which makes a number of improvements to the original PathFinder algorithm. Like previously described work, it only re-routes congested connections, uses A* search, and—for nets with high fanout—only expands a portion of the route tree closest to the sink it is currently routing. It additionally calculates bounding boxes for each net, which it expands dynamically if the route tree found is near the edge of the bounding box (Algorithm 2).

VPR's A* heuristic, which it calls its *router lookahead*, is calculated by profiling the routing network of the FPGA and building a "map" of the delay and congestion costs of resources on its sample routes based on wire type and orientation. It scales these costs by wirelength to decrease congestion on the long wire network. VPR's heuristic is not necessarily monotone; that is to say, the estimated final cost of a path may decrease along the path.

VPR's routing phase is primarily serial, although its timing-driven router does its once-per-iteration static timing analysis (STA) using Tatum [24], a parallel STA library which uses Thread Building Blocks (TBB). Although our focus in this work has been on accelerating the core routing algorithm, we do reimplement the TBB-based parallelization in Swarm and Fractal to ensure comparability of results.

Algorithm 2 VPR's routing algorithm [23]

```
while shared resources exist do
    for each net N_i do
        Prune subtrees of RT_i which have congested resources
        for each unrouted signal t_ij ∈ N_i do
            if N_i is high-fanout and t_ij is not critical then
                Run A* from portions of the route tree near the sink t_ij
                if A* succeeds, add the backtrace to RT_i and continue
            end if
            Run A* from RT_i, only considering routing resources inside the bounding box
for net N_i
            if A* succeeds, add the backtrace to RT_i and continue
            Run A* from RT_i, considering routing resources on the whole FPGA
            Add the backtrace to RT_i
        end for
        Expand the bounding box if RT_i borders or exits the bounding box
    end for
end while
```

2.2 Swarm and Irregular Parallelism

In order to improve performance, parallel architectures such as conventional multicores have traditionally relied on programmers to extract thread-level parallelism from existing programs; that is, they rely on the software to identify independent threads of computation which can be run on different processing elements. Thread-level parallelism, however, can be hard to exploit on conventional multicores with large runtime overheads which dominate for smaller task sizes and without hardware support for ordered algorithms [25].

Hardware architectures have attempted to extract additional parallelism using hardware transactional memory (HTM) [26] and thread-level speculation (TLS) [27], where transactions or threads run speculatively and abort when a conflict is detected. This addresses some difficulties, but data conflicts on software queues can still bottleneck performance. In this chapter, we provide an overview of Swarm [11], a hardware architecture which uses speculation and hardware queues to allow users to exploit ordered, irregular parallelism.

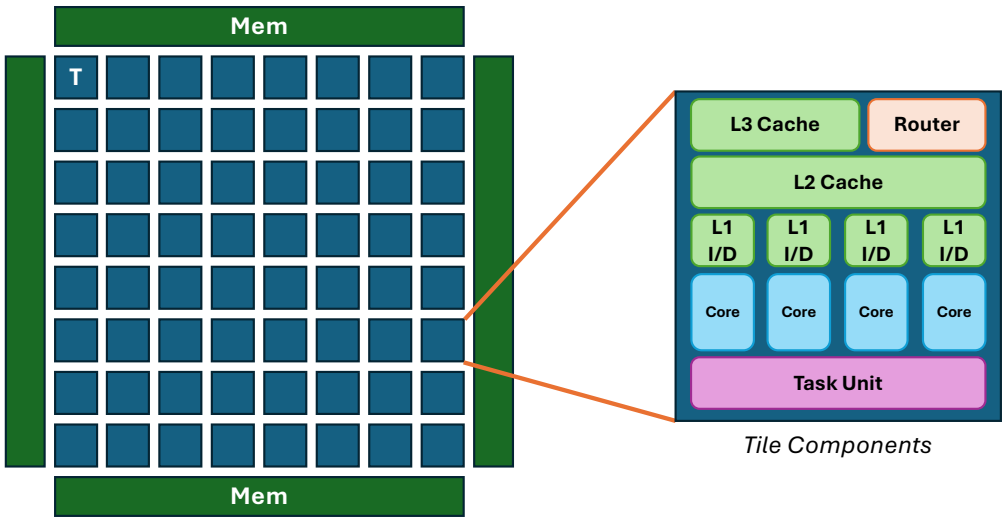

Figure 2.1: A Swarm chip with 64 tiles.

2.2.1 Swarm

The Swarm architecture [11] allows programmers to exploit irregular parallelism by combining a number of important techniques. The architecture itself is a tiled multicore, with each tile consisting of a group of four cores with private L1 caches and a shared L2 cache, along with a slice of a chip-wide L3 cache. Each tile also contains a router and a task unit, which manages the execution of Swarm programs. The design of the architecture is shown in Figure 2.1.

Swarm programs consist of small tasks with programmer-defined timestamps, which appear to execute atomically and in timestamp order. Tasks can *enqueue* children, which must have later or equal timestamps. Tasks with the same timestamp are allowed to execute in any order. For example, a Swarm task for breadth-first search (BFS) could be implemented as in Figure 2.2. In this example, our timestamps represent our distances from the start node, and therefore the task at a certain node with the lowest distance will always appear to execute first, as desired in BFS.

This execution model allows to Swarm to execute tasks speculatively and out-of-order; tasks are issued from the task unit's task queue, and completed tasks are stored in a commit

```
void bfs_task(swarm::Timestamp dist, Vertex* v, Vertex* parent) {
    if(v->visited) return;
    v->visited = true;
    v->parent = parent;
    for(Vertex* neighbor : v->neighbors) {
        swarm::enqueue(bfs_task,
                       dist + 1, // timestamp
                       NOHINT,   // spatial hint
                       neighbor, // task arguments
                       );
    }
}
```

Figure 2.2: An example Swarm task for breadth-first-search.

queue with associated information. Swarm utilizes selective aborts, meaning tasks are only aborted if their read/write sets directly conflict, based on the timestamp, or if they have a dependence on an aborted task; e.g., in Figure 2.2, later tasks which may have executed speculatively at the same node will abort due to their read of `v->visited`, as will any child tasks they spawned. A task can commit once all earlier tasks on the chip have completed. This technique, in combination with the large speculation windows provided by hardware queues, allows Swarm to effectively exploit the parallelism in the independent tasks of a given problem.

In some cases, it is preferable to run certain tasks serially, such as when tasks are likely to access the same data. Swarm provides *spatial hints* [28], which enqueues same-hint tasks to the same tile. This improves locality for task data and reduces data-based aborts from conflicting accesses. Additionally, Swarm allows users to run tasks non-speculatively or to explicitly serialize them [29], as may be necessary for tasks which generate exceptions or are expensive and may have been misspeculated. These tasks run when they are the earliest unfinished task, and do not run concurrently with any other task.

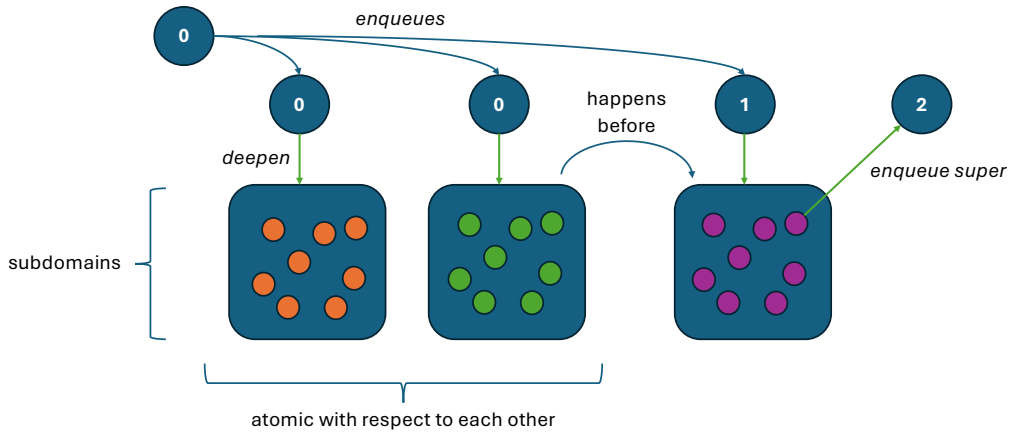

Figure 2.3: The Fractal execution model.

2.2.2 Fractal

Fractal [30] is an extension of Swarm which allows for the creation of nested *domains*. A task in Fractal can enqueue tasks within its domain, create a nested subdomain and enqueue tasks to it (also called *deepening*), or enqueue tasks to its superdomain, if one exists. At each domain level, Fractal tasks and their subdomains appear to execute completely atomically and in the order of their timestamp, as shown in Figure 2.3. Fractal retains the fine-grained parallelism of Swarm by speculatively executing small tasks from all available domains while maintaining the atomicity and global ordering required by its execution model using selective aborts.

Fractal allows for users to parallelize programs which have nested parallelism; for example, a transactional database may have coarse-grained parallelism across transactions (which must execute atomically) while also exhibiting fine-grained parallelism within them. Indeed, Fractal supports unlimited nesting, although it requires spilling tasks to memory for programs which use more than four levels of nesting.

Chapter 3

Implementation

This chapter describes our approach to accelerating VPR in Swarm.

3.1 High-level Techniques

3.1.1 Nested Loops and Continuations

We use Fractal to support the nested loops in PathFinder. Despite the complexity of PathFinder, we never go beyond four nesting levels, eliminating the need for Fractal's task spillers. Our subdomain structure is described in Algorithm 3.

Algorithm 3 VPR's structure in Fractal

```
while shared resources exist do                                   ▷ Domain 1
    Fractal deepen
    for each net N_i do                                           ▷ Domain 2
        Prune subtrees of RT_i which have congested resources
        Fractal deepen
        for each unrouted signal t_ij ∈ N_i do                    ▷ Domain 3
            Fractal deepen
            Perform A* search                                     ▷ Domain 4
        end for
        Fractal undeepen
        Expand the bounding box if RT_i borders or exits the bounding box   ▷ Domain 2
    end for
end while
```

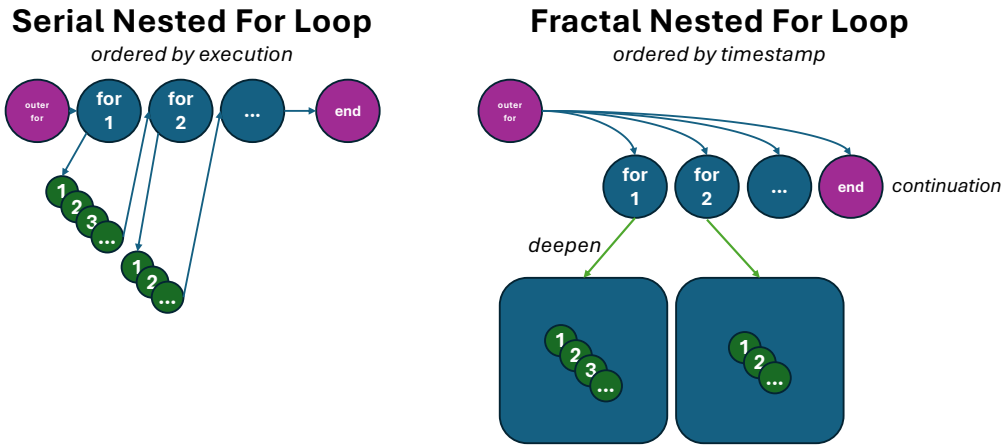

Figure 3.1: Nested for loops in Fractal.

Each "for" in Algorithm 3 is structured as a "parallel for", as shown in Figure 3.1; i.e., all tasks are enqueued at the same time. Tasks which occur at a certain Fractal level but *after* a nested for loop are enqueued to a higher timestamp as continuations, such as the bounding box expansion shown above. These continuations are non-speculative, since they are unlikely to complete without conflicts until routing finishes; indeed, the net-level continuation *requires* serialization since it creates a segmentation fault when net routing is incomplete.

In our current implementation of VPR, we additionally serialize before each net, meaning that all tasks from the previous net must commit before the following net begins routing. This means our router is a fine-grained parallel router, although our approach is theoretically compatible with coarse-grained parallelism as well. We discuss our reasons for this in Chapter 4.3.2.

3.1.2 Spawner Tasks

In general, we follow T4 [31], which uses recursive spawner trees to quickly enqueue tasks for loops with known iteration counts. When spawner trees are balanced, this approach shortens the critical path of task spawns to be logarithmic, as opposed to the linear path of serial spawners. This is possible for most loops in VPR because of the fixed structure of the FPGA routing problem; routing resources *always* have the same connections, nets *always*

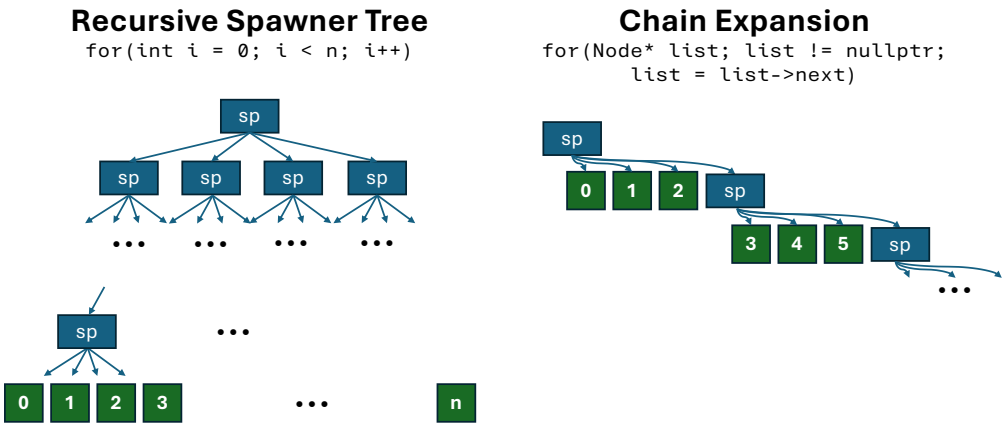

Figure 3.2: Two approaches to spawning tasks quickly: recursive spawners, for known-tripcount loops, and chain expansion, for serializing loop variables.

have the same sinks, etc.

One small exception to this occurs where VPR uses linked lists to store data, such the children of route tree nodes. In these cases, the current pointer to the linked list is *serializing*, since each iteration's value of that pointer depends unconditionally on that of the previous iteration. Here, we use chain expansion with fixed fanout—we spawn 3 tasks to process each element of the list before spawning a new task to process the next chunk of the list, allowing element processing to run off the critical path of task spawns.

We illustrate these two approaches to task spawning in Figure 3.2.

3.2 Accelerating A* Search

In order to achieve the most fine-grained parallelism, we start by accelerating the most nested level of the routing process, the A* search.

3.2.1 Approach

The key order constraint of A* search is that of the priority queue: i.e., nodes must be visited in an order such that the node with the lowest estimated total distance to the goal is always relaxed first. Thus, it is natural to divide A* search into tasks which represent node

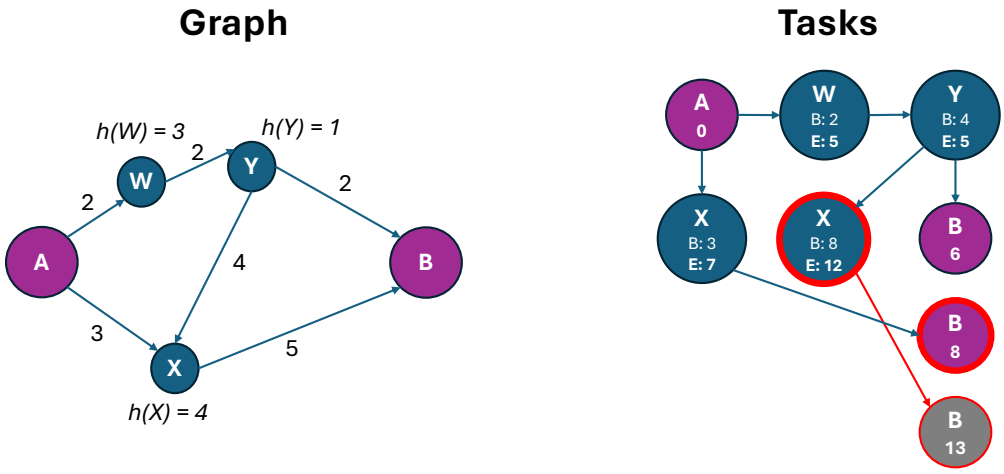

Figure 3.3: An example A* search on a graph with heuristic h, with estimated total distance as timestamps. Even though both A and Y enqueue tasks to relax X, the enqueue from A has a lower estimated cost, leading to the abort of the task from X and its children.

visits, with timestamps that represent the estimated total distance of that node, as shown in Figure 3.3. The task we use for VPR is outlined in Algorithm 4.

Algorithm 4 Psuedocode for our A* visit task.

procedure ASTAR_VISIT(node i, cost c, backwards cost c_b, parent node i_p, done d)
  if d is set, **return**
  if $c <$ recorded total cost and $c_b <$ recorded backwards cost, or i is the sink **then**
    Record the new total cost c, backwards cost c_b, and parent node i_p
    Enqueue a task to *reset* the node cost
    if i is the sink **then**
      Set the flag d
      Calculate a backtrace and update PathFinder costs
      return
    end if
    for neighbor n of i **do**
      Calculate new total cost c' and backwards cost c'_b from i
      Enqueue ASTAR_VISIT(n, c', c'_b, i, d) with timestamp c'
    end for
  end if
end procedure

Since a serial A* search terminates when the sink is found, we use a "done" flag (a `bool*`) which is passed to all tasks within a parallel search; this flag terminates all tasks with timestamps later than the timestamp of the earliest task which reaches the sink node.

Swarm's abort semantics allow us to do this safely; any later task which has already run will be aborted, since it will have read the "done" flag set by the sink node. Additionally, the calculation of new costs is independent for each neighbor n, and so can be done in parallel by enqueuing cost-calculation tasks for each node at the timestamp of the parent.

One unique usage of the Fractal execution scheme here is found in resetting node costs. Since VPR uses the same routing context to record node costs, it uses a `std::vector` to record all touched nodes for each connection and then resets them manually. Here, because we know exactly when a node is being relaxed, we can enqueue a task to reset that node specifically, eliminating the need for an additional data structure. However, because we don't necessarily know what the maximum timestamp is *within* the current domain, we instead enqueue the task to the next timestamp in the *parent domain*, which is guaranteed to run after all tasks in this domain complete. In order to ensure the reset occurs between each search, we set the parent-level timestamp of each search as two times its index, allowing us to properly place the reset tasks between searches.

3.2.2 Timestamps

Using this formulation is not quite sufficient to handle VPR's routing phase, for two reasons: first, Swarm timestamps are integers, while the distances in VPR are floating-point values. Second, VPR's lookahead is non-monotonic, meaning that the estimated total cost of a successor node, factoring in the backwards path cost and the estimated delay, may be less than the estimated total cost of the current node when it was placed on the heap. This means that it is possible for the current node to attempt to enqueue a successor node with a smaller timestamp, which violates Swarm's assumptions.

To address the first issue, we simply re-interpret the floating-point distances as unsigned integers. Because the distances in VPR are always positive, IEEE 754 guarantees us that if floats $x, y > 0$, then $x < y$ if and only if $\text{int}(x) < \text{int}(y)$. We also pass the float distance as a task argument to the child task for ease of use, although it could be omitted.

We depart from VPR's implementation in that we forcibly make VPR's relaxations monotonic by taking the maximum of the newly calculated timestamp t_{succ} and the current timestamp t_{curr}. Because this changes the routing behavior of VPR, we also modify our baseline to perform monotonic relaxations. We evaluate the performance of this change in Chapter 4.1.2.

Because our routing occurs in parallel, there is no guaranteed order for the relaxation of two connections with the same estimated distance. Instead, additional order constraints are necessary to guarantee determinism, as discussed in Section 3.4.

3.3 A* in Parallel

We run the A* searches for all the connections in a single net in parallel. To avoid false dependences, each search is assigned an individual "context", which includes the sink node and the "done" flag, as well as other reused values such as the bounding box and cost parameters. This context is allocated beforehand for each search, passed by reference between calls to ASTAR_VISIT, and deallocated afterwards.

3.3.1 Repeated Searches

In the case where VPR fails to route within the bounding box for a given net, it repeats the routing attempt with a larger bounding box. Because attempts to route the same connection on the bounding box and on the entire FPGA are likely to share resources (and thus cause aborts), and because routing on the entire FPGA is only done in rare cases where the bounding box method fails, we start the full-FPGA searches for each connection non-speculatively. These repeated searches share a context with the initial search, including the "done" flag, so that they can terminate immediately if the initial search succeeds.

VPR also contains an optimization for high-fanout nets known as *spatial lookup*, where only portions of the route tree nearest the sink are added to the initial priority queue. For the sake of simplicity, we omit this optimization in our implementation and the baseline,

although a similar approach where the first routing attempt is done speculatively and the subsequent routing attempts are non-speculative would be feasible here.

3.3.2 Route Tree Forwarding

As discussed in Chapter 2.1.1, PathFinder builds the initial route tree for a given connection from the existing routing of the previous connections within that net. This allows for connections within the same net to reuse routing resources, decreasing wirelength and overall congestion on the FPGA. However, this means that a data dependence exists between each connection; when a given node of the route tree is updated by the routing of an earlier connection, all children of that node in subsequent searches abort, even though route tree updating is only *additive*; i.e., any node that was initially enqueued will never be dequeued by a later update.

To solve this problem, we collapse all the searches for a given net into a single Fractal subdomain. We place the original sink routing order in the upper 31 bits of our 64-bit timestamps, use one additional bit for cost resetting, and leave the lower 32 bits the same (i.e., as the bits of our float). Enqueues from the initial route tree happen at timestamp 0, so that subsequent route tree updates do not abort them. Then, when a search completes its backtrace, it *forwards* any route tree nodes it adds to the tree to all subsequent searches at the appropriate timestamp, so that later searches can reuse the routing resources from earlier searches.

Cost resetting occurs after search n at timestamp $(n \ll 33)|(1 \ll 32)$. Because every task for search $n < n'$ occurs at a timestamp lower than every task for search n', Swarm guarantees that all searches appear atomic to each other as before. An illustration of this process can be seen in Figure 3.4.

We focus solely on performing the first A* search for each connection in parallel. Since the subsequent searches—i.e., the full-FPGA searches—are already serialized, we push them to the end of the routing order, and enqueue the route tree at timestamp 0 within that search.

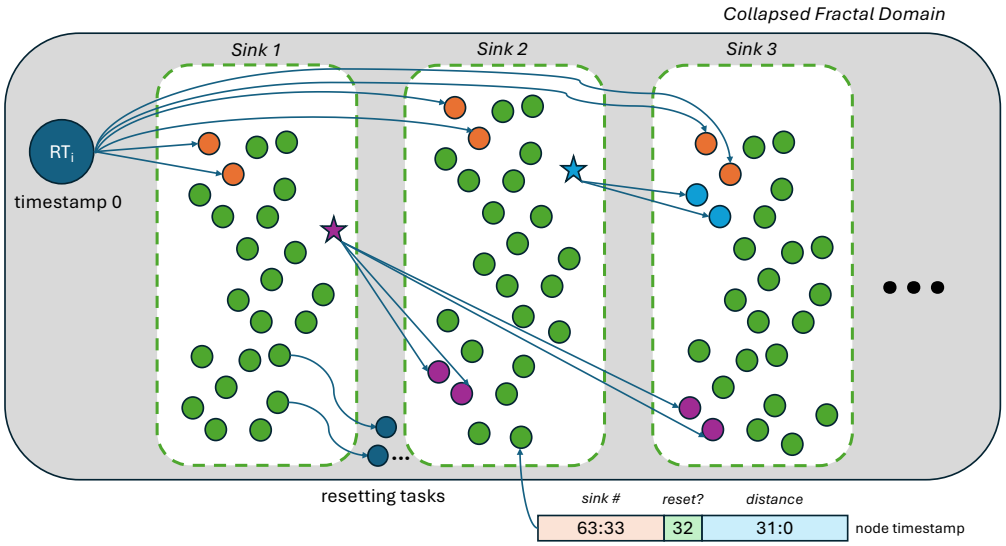

Figure 3.4: Our route tree forwarding scheme. Note that this does not include the serialized, full-FPGA searches.

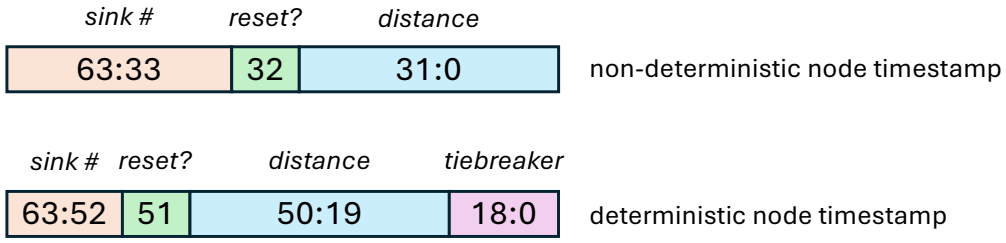

Figure 3.5: Timestamps for deterministic and non-deterministic implementations of A*.

At that point, the route tree will already be fully up-to-date when the search initializes. This change results in a slight change in behavior from VPR's default routing approach: connections which fail to route within the net bounding box are routed sequentially *after* all other connections have been routed, rather than in their initial position.

3.4 Determinism

We implement a deterministic version of our approach by providing a fixed order for routing nodes at the same distance. We accomplish this by appending some bits at the bottom of our timestamp which specify an application-specific tiebreaker, conveying additional information about the edge being relaxed, as shown in Figure 3.5.

For the purposes of our work, we use a bitwise XOR of the node IDs for the start and end of the edge being relaxed, which was sufficient to guarantee determinism on the problems we evaluated. In cases where our total distance was kept the same due to monotonic relaxations but our tiebreaker decreased, we incremented the total distance by one to maintain timestamp ordering. Unfortunately, because we are constrained to 64-bit timestamps in our version of Swarm, the size of problems we can evaluate using this technique is limited; the maximum sink number, i.e., the size of the highest fanout net, must be less than 4096, and there must be at most 2^{19} or around 520k nodes in the routing resource graph. We leave expanding Swarm timestamps or evaluating smaller tiebreakers to future work.

Chapter 4

Evaluation

4.1 Experimental Setup

4.1.1 Modeled System

We evaluate our system using the Swarm simulator, which is execution-driven using a custom RISC-V binary instrumentation tool. We take the parameters for the Swarm system (Table 4.1) from prior work [11, 31] with the exception of the cores, which are replaced with a simple RISC-V core with floating-point extensions. The simulator uses detailed timing models for caches, network, and main memory, as well as for Swarm features such as enqueues, aborts, and conflict checking. Notably, queue and cache sizes are *per-tile*, meaning that larger systems also have lower queue and cache contention.

We perform routing on benchmarks of various sizes (Table 4.2); for small benchmarks, we use the MCNC benchmarks [35] included with VPR. We use some of the medium-sized benchmarks provided by VPR [10, 22], as well as the large-scale benchmarks provided as part of the Titan benchmark suite [36]. For the VPR benchmarks, we route at a fixed channel width of 100; for the Titan benchmarks, we route on the associated timing-driven Stratix IV architecture and we route at a fixed channel width of 300. We use VPR's `map` router lookahead for all benchmarks. We measure from the beginning of the first routing iteration

Cores	4 cores/tile, up to 256 cores in 64 tiles, RISC-V ISA with F (float), D (double), and C (compressed) extensions; simple core model with 1 IPC except for misses and Swarm instructions
L1 caches	32 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	1 MB, per-tile, 8-way, inclusive, 9-cycle latency
L3 cache	64 MB, shared, static NUCA [32] (4 MB bank/tile), 16-way, inclusive, 12-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	Four $n \times n$ meshes, 192-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [33])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core, 16 commit queue entries/core
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [34] Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Commit	Tiles send updates to virtual time arbiter every 100 cycles
Spills	Spill 15 tasks when task queue is 86% full

Table 4.1: Configuration of the modeled systems.

to the completion of routing, meaning that we exclude all setup and teardown costs.

4.1.2 Baseline

For our baseline, we compare against a version of VPR which contains two modifications from VPR 8.0, as described in Chapter 3. First, we modify VPR to take only monotonic relaxations; i.e., to set the estimated total cost of a successor node to at least the estimated total cost of its parent. Second, we modify VPR to eliminate spatial lookup. We do not implement the re-ordering caused by our route tree forwarding scheme (Chapter 3.3.2); instead, we measure the effect of that change *against our baseline* in full runs (Section 4.2.1).

Because our baseline is modified from the default behavior of VPR, we compare the

Benchmark	Source	Consolidated Nets
`tseng`	MCNC	572
`alu4`	MCNC	634
`elliptic`	MCNC	1853
`blob_merge`	VPR 7	3839
`arm_core`	VPR 8	9225
`neuron`	Titan	51456
`cholesky_mc`	Titan	66664

Table 4.2: List of benchmarks used for evaluation; consolidated nets refers to the number of nets produced by VPR before routing.

	Wall-clock time (s)			**Wirelength**			**Critical path** (ns)		
	Default	Mono	Baseline	Default	Mono	Baseline	Default	Mono	Baseline
`blob_merge`	2.52	2.21	3.93	94.8k	94.6k	94.2k	14.72	14.73	14.71
`neuron`	29.12	31.08	32.48	764k	770k	768k	8.70	8.84	8.84
`cholesky_mc`	78.18	81.31	84.67	1.23m	1.24m	1.23m	7.98	8.26	8.23

Table 4.3: Routing performance of default VPR, VPR with monotonic relaxations, and our baseline on larger benchmarks. Smaller values are better.

routing performance and routing time of unmodified VPR, VPR with monotonic relaxations, and our baseline. We perform this evaluation on some of our larger benchmarks, which are more likely to suffer from the removal of spatial lookup. Since this comparison is solely algorithmic and simulation overheads are large, we use x86 for this comparison.

The results of this comparison are shown in Table 4.3. In general, our baseline produces routing results that are comparable but slightly worse than VPR with respect to runtime, and within 5% with respect to wirelength and critical path delay. We did not include `arm_core` in this comparison because its routing does not run to completion with the given architecture and channel width on any configuration. It is not entirely clear which of our changes contributes more to the worse routing times of the baseline; the lack of spatial lookup could be easily remedied by further work, while the monotonic relaxations are a necessary result of Swarm semantics.

	Cycles			Wirelength		Critical path (ns)	
	Baseline	Swarm	*Speed-up*	Baseline	Swarm	Baseline	Swarm
`tseng`	1.04T	267B	3.89x	6.38k	6.52k (+2.0%)	6.430	6.514 (+1.3%)
`alu4`	1.92T	541B	3.54x	10.8k	10.9k (+1.9%)	5.350	5.391 (+0.8%)
`elliptic`	10.8T	1.50T	7.20x	36.0k	35.0k (-2.8%)	8.759	9.314 (+6.3%)
`blob_merge`	25.4T	4.64T	5.46x	93.9k	97.7k (+4.3%)	14.73	14.93 (+1.4%)

Table 4.4: Routing performance of Baseline and Swarm (256 cores, deterministic) on full runs of smaller benchmarks.

4.2 Routing Performance

4.2.1 Full Run

We measure the performance of our four small-to-medium-sized benchmarks by running VPR to completion. This allows us to capture changes in both routing time and performance (i.e., wirelength and critical path delay) from our Swarm implementation. We instrument the baseline to track its progress throughout the routing process and run it under Swarm on a 16-core, 4-tile configuration, although it only takes up one core.

Table 4.4 summarizes the results from full runs, using our deterministic router. Our implementation achieves modest speedups across the board, with minor regressions with respect to wirelength and critical path. Differences in routing quality are caused by two factors: first, our repeated search approach (Chapter 3.3.1) changes the order in which full-FPGA searches are completed; second, our tiebreaker orders nodes which have the same estimated total distance differently from VPR. The overall differences are relatively small, and our implementation and the baseline remain comparable.

4.2.2 Single Iteration

For our largest benchmarks—i.e., `arm_core` and the two Titan benchmarks—running routing to completion in simulation is infeasible. For these benchmarks, we measure just one iteration of PathFinder, from the start of the loop to the completion of routing of all nets,

	Baseline	Swarm, non-det.		Swarm, det.	
	Cycles	Cycles	*Speedup*	Cycles	*Speedup*
`tseng`	39.3B	21.7B	1.81x	21.9B	1.80x
`alu4`	96.1B	43.5B	2.21x	43.1B	2.23x
`elliptic`	265B	107B	2.48x	106B	2.48x
`blob_merge`	756B	289B	2.61x	284B	2.66x
`arm_core`	14.9T	716B	20.8x	714B	20.8x
`neuron`	32.9T	*		1.03T	31.9x
`cholesky_mc`	92.5T	*		2.58T	35.9x

Table 4.5: Routing time of Baseline and Swarm (256 cores) on single routing iterations of each benchmark.

excluding static timing analysis (STA). This focus has the added benefit of constraining the measurement on the core improvements from our Swarm implementation. We choose the third iteration in order to avoid potential start-up costs while still retaining a significant amount of work. For the sake of comparison, we also include the results from our smaller models here. Due to timestamp constraints, the largest problem we can route with a deterministic router is `arm_core`.

Scalability

As seen in Table 4.5, our Swarm implementation achieves small improvements on the smaller circuits but provides significant improvements for our larger circuits. The reason for this difference is a dearth of parallelism on the smaller circuits; smaller circuits are less likely to have nodes that can be visited independently for a given connection or across two connections, which can be partially inferred from the routing resource graph sizes. Additionally, as circuits get larger, a smaller proportion of time is taken up by net initialization and the ceiling for parallelism increases. Both of these effects can be seen in Table 4.6.

Figure 4.1 contrasts the scalability of our smallest benchmark, `tseng`, and our two largest benchmarks, `neuron` and `cholesky_mc`. As core count increases, `tseng`'s performance actually worsens, while the Titan benchmarks continue to improve, although at a decreasing rate.

Circuit	RRG nodes	% serialized cycles
`tseng`	17.6k	9.4%
`alu4`	23.9k	7.6%
`elliptic`	52.2k	8.2%
`blob_merge`	130k	9.2%
`arm_core`	278k	1.3%
`neuron`	5.15M	0.7%
`cholesky_mc`	4.82M	0.5%

Table 4.6: Routing resource graph nodes and proportion of committed cycles occupied by net initialization for each benchmark.

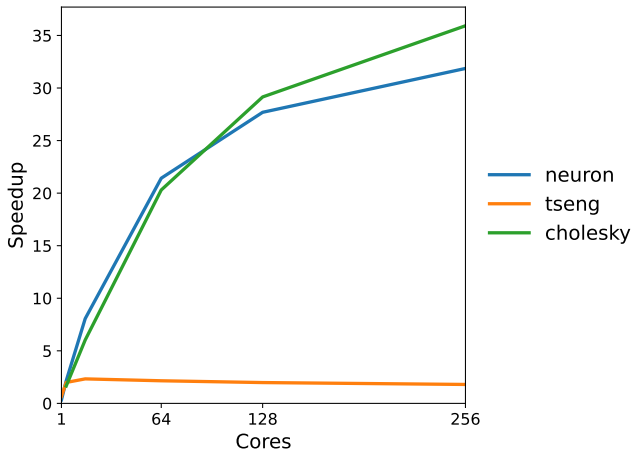

Figure 4.1: Speedup vs. core count for `tseng`, `neuron`, and `cholesky_mc`.

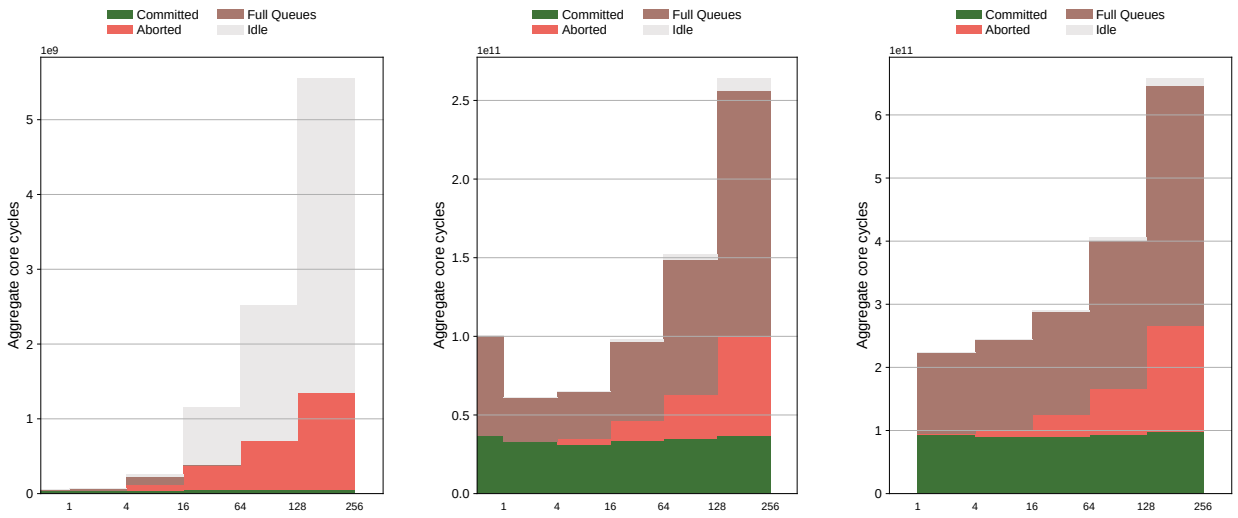

Figure 4.2: Cycle breakdowns for `tseng`, `neuron`, and `cholesky_mc`, from left to right. Note that the 1 core configuration did not complete on `cholesky_mc`.

Figure 4.2 shows the scalability bottlenecks from a hardware perspective: `tseng` spends the vast majority of its cycles empty at higher core counts - as discussed earlier, there simply is not enough work to fill the system, and so a lack of speedup is inevitable. By contrast, `neuron` and `cholesky_mc` both have very few idle cycles and a much higher proportion of committed cycles at higher core counts. However, many cycles are spent waiting on full queues: higher core counts (and hence larger queues) are not always enough to stave off queue pressures, indicating a potential place for future optimization.

Still, even with the large queue pressure seen on `neuron` and `cholesky_mc`, parallelism remains a bottleneck for scalability on our largest circuits. Given a simulation with infinite queues, `neuron`'s runtime only decreases by 8.3% on 256 cores, to give a speedup of 34.5x. Indeed, as shown in Figure 4.3, idle cycles dominate in such a simulation. The significantly increased aborts on higher core counts likely indicate that Swarm has exhausted all the parallelism present in the implementation; thus, the commit queue pressure seen earlier is a result of those aborting tasks, rather than a direct constraint on our implementation's performance.

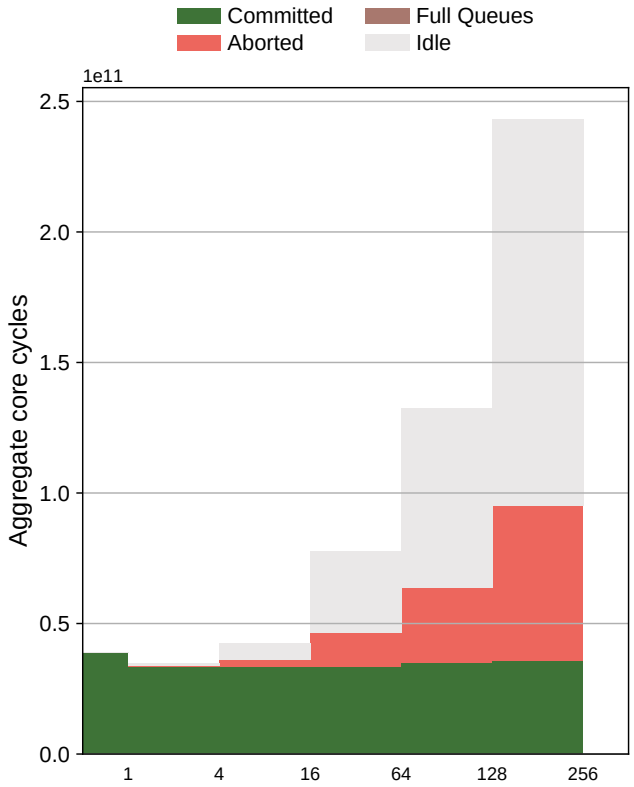

Figure 4.3: Cycle breakdowns for `neuron` with infinite queues.

Discussion

Across the benchmarks it runs, our deterministic router gives us speedup which is comparable to (within 3% of) our non-deterministic router—in other words, the addition of order constraints does not significantly change the parallelism we find in the problem. This means that our deterministic router is also likely to scale well to larger models, given that the problem of tiebreaker size is addressed. Since having consistent results is critical to reliable testing, it is likely that the deterministic router would be preferred in real-world contexts.

Counterintuitively, speedups for a single iteration of the smaller circuits have *lower speed-up* than the full runs. Two phenomena at least partially explain this difference. First, the naive Swarm implementation of STA accounts for a significant portion of cycles and bolsters our numbers; for example, on `elliptic`, STA occupies at least 35% of committed cycles and achieves 10.2x parallelism on 256 cores. Second, the third routing iteration is not entirely representative of routing time—a random sample of three other routing iterations (out of

30) all found higher speedups (between 3.09x and 3.22x). Although more analysis needs to be done to fully explain this phenomenon, it is promising for the potential speedups of VPR for full runs on larger models.

4.3 Roadblocks

4.3.1 Amdahl's Law

Amdahl's Law states that the maximum speedup achievable in a program whose serial portion takes up a proportion p of its overall time is $\frac{1}{p}$. This means that our choice to serialize before starting to route a net necessarily caps our performance at $\frac{1}{p}$, where p is the proportion of cycles spent in net initialization. For our smaller benchmarks, $p \geq 0.076$, limiting speedups for those benchmarks to 13.1x *maximally*.

In general, this means that breaking serialized tasks down into smaller tasks is critical to improving parallelism, and we have manually divided some larger tasks in order to achieve this. Unfortunately, some graph operations - particularly depth-first-search (DFS) - are inherently difficult to accelerate, as they require traversal of a graph in a particular sequence, which is necessarily serializing. Relatedly, these graph operations are also more difficult to express in Swarm, since the timestamps needed for the all the nodes in a parent's subtree are not explicitly known at the time it runs. During net initialization, VPR does multiple conversions between two related data structures which convey routing information, both of which involve a DFS, making increased parallelism hard to find.

4.3.2 False Dependences

In a system with nested parallelism, one alternative to decreasing the size of serialized tasks at the most fine-grain level is to increase parallelism at a coarser level. In the case of VPR, this would entail running multiple nets in parallel. Although we attempted to enable this

functionality, we faced significant engineering challenges due to the structure of VPR, which reuses a number of routing structures between nets, likely to save memory. These shared routing structures resulted in significant increases in aborts and no increase in parallelism, even though, in principle, the routes could have been routed in parallel. Ultimately, it was not possible to achieve any significant improvements from removing serialization between nets in our timeframe.

Both of these roadblocks indicate the importance of parallel-friendly data structures and algorithms when porting complex applications to Swarm; although some programs can be naturally "Swarm-ified", additional work must be done to eliminate false sharing. Compiler-driven tools such as T4 [31] can help to automate or guide this work, but may fall short when confronted with existing codebases with complex global state or uncooperative data structures (e.g., `std::vector`). In these cases, it may be necessary to use data structures designed explicitly for Swarm semantics [37], and to carefully look at traces to distinguish true and false dependences. Finally, for sufficiently memory-intensive programs, there may be a tradeoff between memory usage and data dependences; initializing scratchpads for every subproblem can be expensive, and it may be preferable to initialize a fixed number and hash problems to scratchpads instead.

Chapter 5

Conclusion

We have presented an approach which extracts the irregular parallelism present inside FPGA routing using Swarm, a tiled multicore which uses an execution model based on small, timestamped tasks. Our approach uses a combination of techniques to accelerate routing at two levels: first, within the A* search used to route single connections, and second, in routing multiple connections within a single net at once. Our implementation achieves up to 36x speedup on large benchmarks, and we show that the primary constraint on additional performance is parallelism.

5.1 Future Work

This work opens many interesting avenues for further work. First and foremost, the primary bottleneck for our implementation is a lack of parallelism. Fortunately, there is likely more parallelism to be extracted in the FPGA routing problem. Shen *et. al.*, for example, show that it is possible to achieve 19.13x speedup with a net-based parallel router while maintaining serial equivalency [8]; those gains should be composable with the fine-grained parallelism that we exploit in this thesis and would yield significant improvements if realized.

Enabling determinism on larger test cases could yield interesting results. Although expanding timestamps beyond 64 bits should be sufficient to replicate our approach, the cur-

rent choice of tiebreaker is somewhat arbitrary, and it is not immediately clear whether it is necessary or sufficient to guarantee determinism at all model sizes. Additionally, narrower tiebreaker representations may be needed to route significantly larger (e.g., multi-FPGA) models depending on timestamp size, which could introduce a three-way tradeoff between tiebreaker representations, routing time, and determinism.

Additional use of information about the FPGA architecture could likely further improve performance. Although the exact order of relaxations for a given routing iteration is not known *a priori*, FPGAs have very regular structure, and the routing resource graph and netlist *are* fixed before routing starts. Use of that information to partition or guide the routing workload, such as through the use of Swarm's spatial hints, could likely improve performance if parallelism is sufficiently high.

Finally, our evaluation shows that circuit size alone does not determine the parallelism achievable by our system. Characterizing FPGA problems that might be more or less parallelizable could help us understand more about what makes certain routing problems hard. That work could also have implications for how logic blocks are packed and placed on the FPGA before routing, which would improve the overall performance of FPGA compilation beyond routing alone.

References

[1] D. Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[2] A. Putnam et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *IEEE Micro* 35.3 (2015), pp. 10–22. DOI: 10.1109/MM.2015.42.

[3] Advanced Micro Devices (AMD). *Vivado Design Suite*. 2023. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[4] Intel Corporation. *Intel Quartus Prime Design Software*. URL: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html.

[5] F. Elsabbagh, S. Sheikhha, V. A. Ying, Q. M. Nguyen, J. S. Emer, and D. Sanchez. "Accelerating RTL Simulation with Hardware-Software Co-Design". In: *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2023, 14 pages. DOI: 10.1145/3613424.3614257.

[6] M. Stojilović. "Parallel FPGA routing: Survey and challenges". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056782.

[7] M. Gort and J. H. Anderson. "Accelerating FPGA Routing Through Parallelization and Engineering Enhancements Special Section on PAR-CAD 2010". In: *IEEE Transactions*

on Computer-Aided Design of Integrated Circuits and Systems 31.1 (2012), pp. 61–74. DOI: 10.1109/TCAD.2011.2165715.

[8] M. Shen, W. Zhang, G. Luo, and N. Xiao. "Serial-Equivalent Static and Dynamic Parallel Routing for FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.2 (2020), pp. 411–423. DOI: 10.1109/TCAD.2018.2887050.

[9] Y. O. M. Moctar and P. Brisk. "Parallel FPGA routing based on the operator formulation". In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2014, pp. 1–6. DOI: 10.1145/2593069.2593177.

[10] K. E. Murray et al. "VTR 8: High-Performance CAD and Customizable FPGA Architecture Modelling". In: *ACM Trans. Reconfigurable Technol. Syst.* 13.2 (June 2020). ISSN: 1936-7406. DOI: 10.1145/3388617. URL: https://doi.org/10.1145/3388617.

[11] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. "A scalable architecture for ordered parallelism". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 228–241. DOI: 10.1145/2830772.2830777.

[12] Y.-L. Wu and D. Chang. "On the NP-Completeness of Regular 2-D FPGA Routing Architectures and a Novel Solution". In: *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '94. San Jose, California, USA: IEEE Computer Society Press, 1994, pp. 362–366. ISBN: 0897916905.

[13] L. McMurchie and C. Ebeling. "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs". In: *Third International ACM Symposium on Field-Programmable Gate Arrays*. 1995, pp. 111–117. DOI: 10.1109/FPGA.1995.242049.

[14] R. Y. Rubin and A. M. DeHon. "Timing-driven pathfinder pathology and remediation: quantifying and reducing delay noise in VPR-pathfinder". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA

'11. Monterey, CA, USA: Association for Computing Machinery, 2011, pp. 173–176. ISBN: 9781450305549. DOI: 10.1145/1950413.1950447. URL: https://doi.org/10.1145/1950413.1950447.

[15] Y. Zha and J. Li. "Revisiting PathFinder Routing Algorithm". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 24–34. ISBN: 9781450391498. DOI: 10.1145/3490422.3502356. URL: https://doi.org/10.1145/3490422.3502356.

[16] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. "CRoute: A Fast High-Quality Timing-Driven Connection-Based FPGA Router". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 53–60. DOI: 10.1109/FCCM.2019.00017.

[17] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. "A connection-based router for FPGAs". In: *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 326–329. DOI: 10.1109/FPT.2013.6718378.

[18] D. Wang, Z. Duan, C. Tian, B. Huang, and N. Zhang. "A Runtime Optimization Approach for FPGA Routing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.8 (2018), pp. 1706–1710. DOI: 10.1109/TCAD.2017.2768416.

[19] Y. Zhou, P. Maidee, C. Lavin, A. Kaviani, and D. Stroobandt. "RWRoute: An Open-Source Timing-Driven Router for Commercial FPGAs". In: *ACM Trans. Reconfigurable Technol. Syst.* 15.1 (Nov. 2021). ISSN: 1936-7406. DOI: 10.1145/3491236. URL: https://doi.org/10.1145/3491236.

[20] M. Shen and G. Luo. "Corolla: GPU-Accelerated FPGA Routing Based on Subgraph Dynamic Expansion". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA:

Association for Computing Machinery, 2017, pp. 105–114. ISBN: 9781450343541. DOI: 10.1145/3020078.3021732. URL: https://doi.org/10.1145/3020078.3021732.

[21] D. Korolija and M. Stojilović. "FPGA-Assisted Deterministic Routing for FPGAs". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019, pp. 155–162. DOI: 10.1109/IPDPSW.2019.00034.

[22] J. Luu et al. "VTR 7.0: Next Generation Architecture and CAD System for FPGAs". In: *ACM Trans. Reconfigurable Technol. Syst.* 7.2 (July 2014). ISSN: 1936-7406. DOI: 10.1145/2617593. URL: https://doi.org/10.1145/2617593.

[23] K. E. Murray, S. Zhong, and V. Betz. "AIR: A Fast but Lazy Timing-Driven FPGA Router". In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020, pp. 338–344. DOI: 10.1109/ASP-DAC47756.2020.9045175.

[24] K. E. Murray and V. Betz. "Tatum: Parallel Timing Analysis for Faster Design Cycles and Improved Optimization". In: *IEEE International Conference on Field-Programmable Technology (FPT)*. 2018.

[25] M. A. Hassaan, M. Burtscher, and K. Pingali. "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms". In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: Association for Computing Machinery, 2011, pp. 3–12. ISBN: 9781450301190. DOI: 10.1145/1941553.1941557. URL: https://doi.org/10.1145/1941553.1941557.

[26] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. 2nd. Morgan and Claypool Publishers, 2010. ISBN: 1608452352.

[27] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano. "A Survey on Thread-Level Speculation Techniques". In: *ACM Comput. Surv.* 49.2 (June 2016). ISSN: 0360-0300. DOI: 10.1145/2938369. URL: https://doi.org/10.1145/2938369.

[28] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez. "Data-centric execution of speculative parallel programs". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783708.

[29] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez. "Harmonizing speculative and non-speculative execution in architectures for ordered parallelism". In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-51. Fukuoka, Japan: IEEE Press, 2018, pp. 217–230. ISBN: 9781538662403. DOI: 10.1109/MICRO.2018.00026. URL: https://doi.org/10.1109/MICRO.2018.00026.

[30] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez. "Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 587–599. ISBN: 9781450348928. DOI: 10.1145/3079856.3080218. URL: https://doi.org/10.1145/3079856.3080218.

[31] V. A. Ying, M. C. Jeffrey, and D. Sanchez. "T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 159–172. DOI: 10.1109/ISCA45697.2020.00024.

[32] C. Kim, D. Burger, and S. W. Keckler. "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: Association for Computing Machinery, 2002, pp. 211–222. ISBN: 1581135742. DOI: 10.1145/605397.605420. URL: https://doi.org/10.1145/605397.605420.

[33] 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". In: *IEEE Micro* 27.5 (2007), pp. 15–31. DOI: 10.1109/MM.2007.4378780.

[34] J. L. Carter and M. N. Wegman. "Universal classes of hash functions (Extended Abstract)". In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC '77. Boulder, Colorado, USA: Association for Computing Machinery, 1977, pp. 106–112. ISBN: 9781450374095. DOI: 10.1145/800105.803400. URL: https://doi.org/10.1145/800105.803400.

[35] S. Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina, 1991.

[36] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. "Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap between Academic and Commercial CAD". In: *ACM Trans. Reconfigurable Technol. Syst.* 8.2 (Mar. 2015). ISSN: 1936-7406. DOI: 10.1145/2629579. URL: https://doi.org/10.1145/2629579.

[37] V. A. Ying. "Compiler-Hardware Co-Design for Pervasive Parallelization". Available at https://people.csail.mit.edu/sanchez/theses/2023.ying.phd.pdf. PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, Sept. 2023.