Tinsel: a manythread overlay for FPGA clusters

Matthew Naylor
University of Cambridge
(Corresponding author)
matthew.naylor@cl.cam.ac.uk
https://github.com/POETSII/tinsel

Simon W. Moore
University of Cambridge

A. Theodore Markettos
University of Cambridge

David Thomas
Imperial College London

Andrey Mokhov
Newcastle University

Shane Fleming
Microsoft Research, UK

Andrew Brown
University of Southampton

***Abstract*—Commodity FPGA boards with advanced networking facilities have great potential in the construction of high-performance compute clusters that scale. However, low-level design tools and long synthesis times are major barriers to productivity for application developers. In this paper, we explore the potential of a distributed soft-processor overlay, programmed in software at a high-level of abstraction, to deliver a useful level of performance for FPGA clusters. In particular, we demonstrate the use of hardware multhreading to achieve a fast, space-efficient, high-throughput overlay, and compare a 12-FPGA instance of it (12,288 RISC-V threads) against a conventional Xeon cluster on the problem of distributed graph processing.**

I. Introduction

The communication bottleneck is one of the main factors affecting the scalability of high-performance compute clusters [1]. For example, in the domain of distributed graph processing, partitioning graphs among compute nodes can result in a high proportion of cut edges, and a cost-dominating communication requirement [2]. In this paper, we are interested in the development of compute clusters that are optimised for communication, and the potential benefits to distributed applications.

Efficient communication is one of the primary strengths of FPGA technology, mainly due to an ability to process network traffic at line-rate with minimal latency overheads [3]. This strength has fed the production of commodity FPGA boards equipped with multiple state-of-the-art network interfaces. The flexibility of FPGAs also permits other standard I/O interfaces, such as SATA and PCI Express, to be repurposed for even greater inter-board communication options [4, 5]. All this, combined with a general-purpose compute fabric, makes FPGAs an attractive choice for cluster computing.

However, a major factor blocking the wider adoption of FPGA-based systems is developer productivity, and the level of knowledge that is needed to exploit them effectively. It is therefore helpful for the FPGA community to support application development through higher-level overlays that can be targeted without FPGA expertise. In this paper, we explore the extent to which a distributed soft-processor overlay, programmed in software at a high level of abstraction, can deliver a useful level of performance for FPGA clusters. Our main contributions are:

- A new FPGA-optimised hyperthreaded RISC-V core called Tinsel, which integrates inter-core communication at a deep level, and which tolerates the inherent latencies of floating-point operations and off-chip memory accesses. Tinsel trades single-thread performance for improved area, frequency, and throughput over existing soft-processor designs.
- A distributed overlay connecting multiple Tinsel cores within an FPGA, and multiple FPGAs within a cluster, enabling efficient and reliable messaging between any two threads in the cluster. Tinsel offers a rich feature set compared to existing manycore overlays: off-chip memories, data caches, FPUs, and inter-FPGA communication.
- A hardware-assisted, distributed, termination-detection primitive, which can also be used as a global synchronisation barrier, greatly simplifying the programming model for pure message-passing systems.
- A thin software layer that sits on top of the Tinsel overlay and provides a high-level vertex-centric programming API supporting both synchronous and asynchronous execution. Architectural details are completely hidden.
- A whole-system evaluation in the domain of distributed graph processing, showing linear performance scaling to 12 FPGAs (12,288 RISC-V threads), good utilisation of off-chip memory and communication resources, and an order-of-magnitude reduction in energy compared to a Xeon cluster programmed at a similar level of abstraction.

II. Design goals

The work described in this paper is part of a larger project called POETS (Partial Ordered Event Triggered Systems) looking at hardware support for an *event-driven parallel programming model* [6]. In this model, programs are expressed as graphs in which edges represent communication links along which messages may be sent, and vertices perform event-driven computation. This is similar to Google's Pregel model [7] and deLorimier's GraphStep model [8], but allows both synchronous and asynchronous styles of message-passing; while the synchronous style is deadlock-free and generally easier to program, the asynchronous style can enable greater

parallelism and scalability. All this leads to a number of important design goals for the overlay, outlined below.

G1: Exploit current hardware – While it is desirable for the overlay to be portable to various FPGA clusters, it is essential that it can, at least, exploit the main features of our current cluster, based around the DE5-Net FPGA board, and detailed in Figure 1. This means support for the three main off-chip resources on the DE5-Net: DDR3 DRAM, QDRII+ SRAM, and SFP+ interconnect.

G2: Parameterisation – As we intend to support a variety of applications, it is important that the balance between processing resources, memory resources, and communication resources can be tuned appropriately on a per-application basis. In other words, the overlay needs to be highly-parameterised.

G3: Latency tolerance – Some applications will require floating-point support, and on the DE5-Net this entails tens of cycles of latency. Further latency is introduced by off-chip memory access (to achieve goal G1) and resource sharing (to achieve goal G2). A key aim of the overlay is therefore to tolerate latency as cleanly as possible, i.e. in a way that is efficient in hardware and easy-to-use for programmers.

G4: Asynchronous messaging – The event-driven programming model requires our overlay to support efficient, reliable, and asynchronous sending and receiving of messages between any pair of cores in the cluster. This includes the detection and correction of errors on the inherently unreliable inter-FPGA links. The messaging primitives must support non-blocking operation, allowing asynchronous applications to avoid message-dependent deadlock.

G5: Soft multicasting – Some applications will involve graphs with high fan-in and fan-out, such as neural simulation and social network analysis, and should be supported by the overlay. This does not imply support for true hardware multicasting, which is expensive. However, the overlay should at least support software-based multicasting through features for fast message forwarding and forking.

G6: Termination detection – A central problem in distributed systems is detecting termination, i.e. detecting when (1) all nodes have finished sending messages, and (2) there are no undelivered messages in-flight. But this is both difficult and inefficient to do purely in software, motivating hardware support in the overlay. Such a capability is useful in asynchronous applications, for detecting convergence, but it also enables a globally-synchronous execution model by allowing a new time step to start each time the system converges.

We are not aware of any existing overlay that meets all of these goals. We discuss related work in Section VII.

III. Tinsel overlay

Motivated by the design goals set out in the previous section, we now present Tinsel, our distributed soft-processor overlay.

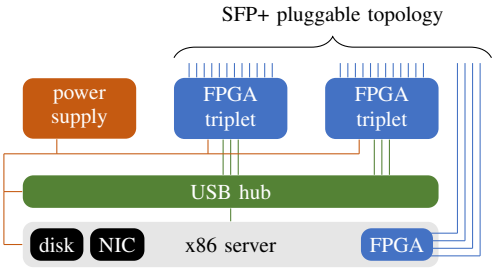

Fig. 1. A single POETS box comprising an x86 server and 7× interconnected DE5-Net FPGA boards. Each DE5-Net supports 2× 4GB DDR3 DRAMs, 4× 8MB QDRII+ SRAMs, and 4× 10G SFP+ ports. One FPGA is used as a PCIe to SFP+ bridge board, providing a fast connection between the x86 and remaining six worker FPGAs. The worker FPGAs are arranged as two triplets, each providing additional inter-FPGA communication links via a custom PCIe backplane (unused in this paper). In-house power-management boards allow per-FPGA power switching, power measurement, and fan monitoring. Multiple POETS boxes are connected together to form the full POETS cluster.

Subsystem	Parameter	Default value
Core	`ThreadsPerCore`	16
Core	`BytesPerInstrMem`	16384
Core	`CoresPerFPU`	4
Core	`CoresPerDCache`	4
Core	`CoresPerMailbox`	4
Cache	`DCachesPerDRAM`	8
Cache	`BytesPerBeat`	32
Cache	`BeatsPerLine`	1
Cache	`DCacheSetsPerThread`	4
Cache	`DCacheNumWays`	8
NoC	`MailboxMeshXLen`	4
NoC	`MailboxMeshYLen`	4
NoC	`BytesPerFlit`	16
NoC	`MaxFlitsPerMsg`	4
Mailbox	`MsgSlotsPerThread`	16

Fig. 2. A selection of parameters of the Tinsel circuit generator. The default configuration provides 1024 RISC-V threads per FPGA.

Multithreaded RISC-V core Multithreading is a powerful tool for tolerating latency (goal G3): it allows a processor to stay busy by continuing to execute some threads while others are suspended on the result of a latent (high-latency) instruction. We have developed a barrel-scheduled multithreaded core implementing a large subset of the RV32IMF instruction set. In our core, floating-point and memory instructions are examples of latent instructions, as are custom instructions that block until it is possible to send or receive a message. When a thread executes a latent instruction, it becomes *suspended*, and is automatically resumed when the instruction completes. The number of threads is controlled by a synthesis-time parameter `ThreadsPerCore`, which is 16 by default, as shown in Figure 2, and can be as high as 32. Thus, instruction latencies of tens of cycles can be tolerated. The pipeline has 6 logical stages, shown and described in Figure 3, but uses 8 physical stages to achieve an Fmax above 450MHz on a lightly-utilised DE5 ($<$ 1% util.), and above 250MHz on a heavily-utilised DE5 ($>$ 60% util.). To execute an instruction on every cycle, there must

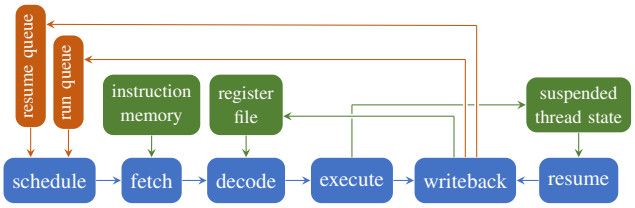

Fig. 3. Tinsel core pipeline. At most one instruction per thread is present in the pipeline at any time, eliminating all data and control hazards. This results in a small, high-frequency core with high throughput for multi-threaded workloads. Latent instructions are suspended in the *execute* stage and resumed in the *resume* stage. Two queues storing ready-to-execute threads are used to avoid a bottleneck in writeback, where writeback requests can arrive simultaneously from both *execute* and *resume*.

exist at least as many runnable threads as physical pipeline stages. This may motivate reducing pipeline depth in future, but our current focus is on high multithreaded throughput.

Instructions are stored in a dual-port block RAM of size `BytesPerInstrMem`, which can be shared by up to two cores. A single floating-point unit is shared by `CoresPerFPU` cores (default 4). FPU operations are implemented using Altera IP blocks and have latencies as high as 14 cycles at 250MHz [9].

Data cache To keep the programming model simple (goal G3), we have opted to use data caches to optimise access to off-chip memory rather than DMAing blocks into a scratchpad. This includes access to the two DDR3 DRAMs and the four QDRII+ SRAMs on each DE5.

A typical RISC workload will not access memory on every instruction, motivating the ability for a cache to be shared by multiple cores, defined by `CoresPerDCache`. The cache is a set-associative write-back cache with a pseudo-LRU replacement policy. It is partitioned by thread id, avoiding cache line aliasing and sharing between threads (message-passing is intended to be the primary communication mechanism). It is non-blocking, delivering responses out-of-order so that requests can be served at full-throughput (one per cycle).

Assuming one memory access every four instructions, a single DDR3 DRAM can satisfy a maximum of 64 cores running at 250MHz, provided that programs typically access all the words of a cache line before it is evicted. However, 100% DRAM throughput is unlikely in practice, so 32 cores per DRAM is a more realistic figure. This leads to our default values of four and eight for the `CoresPerDCache` and `DCachesPerDRAM` parameters respectively.

Mailbox The mailbox is the mechanism by which threads send and receive messages (goal G4). A single mailbox serves a group of cores, defined by `CoresPerMailbox`. Mailboxes are then connected together to form a distributed network on which any thread can send a message to any other thread. For flexibility, we support variable-length messages comprising one or more *flits* up to `MaxFlitsPerMsg` (default 4). The flit size is defined by `BytesPerFlit` (default 16).

At the heart of a mailbox lies a *memory-mapped scratchpad*

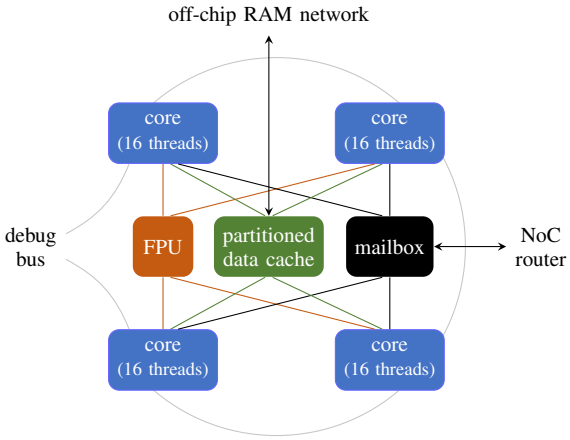

Fig. 4. Default configuration of a Tinsel tile. In general, FPUs and caches can be shared between tiles.

that stores both incoming and outgoing messages. The scratchpad is implemented using a mixed-width block RAM, with a 32-bit port on the core side and a much larger flit-sized port on the network side. The scratchpad is divided into several message *slots* per thread, defined by `MsgSlotsPerThread` (default 16). As well as holding messages, the scratchpad may be used as a thread-local general-purpose memory. Sending and receiving messages is achieved via custom RISC-V control/status registers (CSRs) local to each thread. These raw CSR accesses are abstracted by a very thin Tinsel API, which we outline below. *Each Tinsel API function corresponds to just one or two single-cycle CSR accesses.*

Sending messages To send a message residing in the scratchpad, a thread must first ensure that the network has capacity for it (to ensure deadlock-freedom – goal G4) by calling

```
bool tinselCanSend();
```

and if the result is true, the thread can call

```
void tinselSend(uint32_t dest, volatile void* msg);
```

where `dest` is a global thread identifier, and `msg` is a message-aligned address in the scratchpad. The message is not guaranteed to have left the mailbox until `tinselCanSend()` returns true again, at which point data pointed to by `msg` can safely be mutated, e.g. by writing a new message. The number of flits in the message being sent is also stored in a CSR that can be modified by a call to

```
void tinselSetLen(uint32_t numFlits);
```

Receiving messages To receive a message, a thread must first allocate a slot in the scratchpad for an incoming message to be stored. Allocating a slot can be viewed as transferring ownership of that slot from the software to the hardware. This is done by a call to

```
void tinselAlloc(volatile void* addr);
```

where `addr` is a message-aligned address in the scratchpad. Multiple slots can be allocated in this way, creating a receive buffer of the desired size. The hardware may use any one of

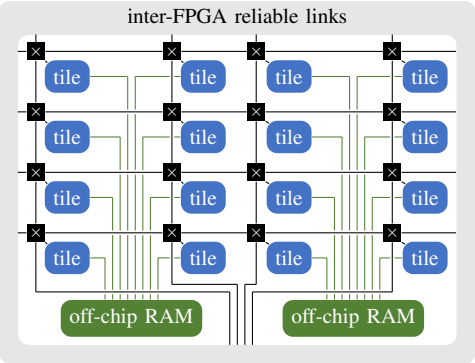

Fig. 5. Default configuration of the Tinsel NoC on a single FPGA. Tiles are connected together by dimension-ordered routers, and inter-FPGA links are connected to the NoC rim. Each off-chip RAM blob contains a DDR3 controller and two QDRII+ controllers.

Subsystem	Quantity	ALMs	% of DE5
Core	64	51,029	21.7
FPU	16	15,612	6.7
DDR3 controller	2	7,928	3.5
Data cache	16	7,522	3.2
NoC router	16	7,609	3.2
QDRII+ controller	4	5,623	2.4
10G Ethernet MAC	4	5,505	2.3
Mailbox	16	4,783	2.0
Interconnect etc.	1	37,660	16.0
Total		143,271	61.0

Fig. 6. Default Tinsel area breakdown on the DE5-Net at 250MHz.

the allocated slots to store an incoming message, but as soon as a slot is used it will be automatically deallocated. Now, when a thread wishes to receive a message it can call

```
bool tinselCanRecv();
```

to see if a message is available and, if so, receive it by calling

```
volatile void* tinselRecv();
```

which returns a pointer to a slot containing a received message. If the identity of the sender is required, it must be included in the message contents. Receiving a message can be viewed as transferring ownership of a slot from the hardware back to the software. If multiple slots contain incoming messages, `tinselRecv` is free to pick any: we deliberately avoid guarantees about message ordering, to keep optimisation opportunities open in the communication subsystem.

Soft multicasting There are two features of the mailbox that support efficient multicasting in software: (1) a message can be forwarded (received and sent) without copying and without passing through the 32-bit core, and (2) once in the scratchpad, a message can be efficiently sent multiple times (forked).

Network-on-chip Tinsel uses a 2D tiled network-on-chip (NoC), with each tile containing a single mailbox and some number of cores, FPUs, and caches. The default configurations the Tinsel tile and Noc are shown in Figures 4 and 5. We use separate on-chip networks for message-passing and off-chip memory access. While it is tempting to create a single unified NoC for both, this leads to complications. Deadlock in asynchronous message-passing systems is usually avoided by ensuring that threads are always ready-to-receive and never block on a send operation. But this is not possible if, in order to receive a message, a thread needs to access memory which would involve sending over the unified NoC. The problem could be avoided using virtual channel routing [10, 11], but at the cost of complexity. Another concern with a unified NoC is congestion: we want to support high communication bandwidth *and* high memory utilisation, which is more obviously achieved using separate networks.

Each tile's mailbox connects to a dimension-ordered router, and all routers are connected in a 2D mesh arrangement using bidirectional half-rate FIFOs. At the edges of the on-chip mesh are the inter-FPGA reliable links, extending the mailbox mesh over the entire cluster. The default configuration of Tinsel contains 1,024 RISC-V threads per FPGA, which is 12,288 threads in total in our current 2-box cluster (3×4 FPGAs).

Further to the memory and mailbox networks, there is a low-performance 8-bit debug bus connecting all the cores on an FPGA. It provides each thread with a virtual UART, i.e. non-blocking functions for putting and getting bytes. The debug bus connects to the host x86 server via USB JTAG.

Inter-FPGA links In a large cluster with many inter-FPGA links, bit errors will be common and therefore must be detected and corrected (goal G4). On top of a raw link we place a 10Gbps Ethernet MAC, which automatically detects and drops packets containing CRC errors. On top of the MAC we place our own window-based reliability layer that retransmits dropped packets. Ethernet allows us to use standard and free IP cores for inter-board communication, and as we use the links point-to-point, many Ethernet packet fields can be reused for our own purposes, resulting in little overhead on the wire.

Termination detection This feature (goal G6) allows an application to observe when all threads in the system have indicated that they no longer wish to send *and* there are no messages in flight. We refer to this as the *idle event*. It can be used to detect termination in asynchronous applications (difficult to do in software) and to advance time in synchronous applications (inefficient to do in software). The feature is entirely supported by a single hardware primitive:

```
int32_t tinselIdle(bool vote);
```

which *blocks* until either (1) a message is available to receive, or (2) all threads in the entire system are blocked on a call to `tinselIdle` and there are no undelivered messages in the system. The function returns zero in the former case and non-zero in the latter. A return value > 1 denotes that all callers voted true. The voting mechanism allows termination to be detected in *synchronous* applications, e.g. all threads in the system are stable since the last time step.

Our implementation of `tinselIdle` is based on Safra's distributed termination detection algorithm [12]. We put an

idle detection component on each worker FPGA, which has two main responsibilities: (1) maintain a 64-bit count of the number of messages sent by any thread on the FPGA, minus the number received; and (2) determine when all threads on the FPGA are in a call to `tinselIdle`. The idle detection process is instigated by the master bridge board, broadcasting a token out to all worker FPGAs. Each worker receives the token, waits until all threads on that FPGA are in a call to `tinselIdle`, and then responds with the 64-bit count. If all the counts sent back to the master sum to zero, then the idle event is detected and a barrier release phase is triggered, causing the calls to `tinselIdle` on the workers to return non-zero. Otherwise, the idle detection process is restarted.

Resource utilisation The resource requirements of the default configuration Tinsel on the DE5-Net (part `5SGXEA7N2F45C2`) are shown in Figure 6. It meets timing at 250MHz.

IV. POLITE API

To provide a truly high-level programming environment, we need abstractions that hide architectural details. The POETS project is actively exploring a graph-based event-driven programming abstraction as a solution to this problem [6]. To evaluate the suitability of Tinsel as a target for this abstraction, we present a basic, lightweight version of it, called POLite.

POLite is a thin C++ layer on top of the Tinsel API that takes care of mapping arbitrary graphs onto the overlay. Behaviours of vertices in the graph are defined by *event handlers* that update the *vertex state* when a particular event occurs, e.g. when a message arrives on an incoming edge, or the network is ready to send a new message, or termination is detected. It is similar to the vertex-centric paradigm [7, 8], but supports both synchronous and asynchronous execution.

Vertices In a POLite application, vertices are defined by inheriting from the `PVertex` class:

```
template <typename S, typename E, typename M>
  struct PVertex {
    // Vertex state
    S* s;
    PPin* readyToSend;

    // Event handlers
    void init();
    void send(M* msg);
    void recv(M* msg, E* edge);
    bool step();
    bool finish(M* msg);
  };
```

Fig. 7. Essential structure of a POLite task/vertex. It is parameterised by the task state type `S`, the edge weight type `E`, and the message type `M`.

Each vertex has access to local state `s`, and a `readyToSend` field whose value is one of:

- `No` – the vertex doesn't want to send.
- `Pin(p)` – the vertex wants to send on pin *p*.
- `HostPin` – the vertex wants to send to the host.

```
// Vertex state
struct SSSPState {
  // Is this the source vertex?
  bool isSource;
  // The shortest known distance to this vertex
  int dist;
};

// Vertex behaviour
struct SSSPVertex : PVertex<SSSPState,int,int> {
  void init() {
    *readyToSend = s->isSource ? Pin(0) : No;
  }
  void send(int* msg) {
    *msg = s->dist;
    *readyToSend = No;
  }
  void recv(int* dist, int* weight) {
    int newDist = *dist + *weight;
    if (newDist < s->dist) {
      s->dist = newDist;
      *readyToSend = Pin(0);
    }
  }
  bool step() { return false; }
  bool finish(int* msg) {
    *msg = s->dist;
    return true;
  }
};
```

Fig. 8. Asynchronous single-source shortest paths using POLite.

A pin is an array of outgoing edges, and sending a message on a pin means sending a message along all edges in the array. A vertex can have a number of pins. Vertices should initialise `*readyToSend` in the `init` handler, which runs once for every vertex when the application starts. After that, the other event handlers come into play.

Send handler Any vertex indicating that it wishes to send will eventually have its `send` handler called, unless another handler (called before the `send` handler has had chance to run) updates `*readyToSend` to `No`. When called, the `send` handler is provided with a message buffer, to which the outgoing message should be written. The destination is deduced from the value of `*readyToSend` immediately before the `send` handler is called.

Receive handler A message arriving at a vertex causes the `recv` handler of the vertex to be called with a pointer to the message and a pointer to the weight associated with the incoming edge along which the message has arrived. The edge weight is passed to the `recv` handler rather than the `send` handler because it is associated with a particular edge, not a pin capturing multiple edges. For unweighted graphs, the edge weight type can be declared as `PEmpty` and ignored.

Step handler The `step` handler is called when no vertex in the entire graph wishes to send, and there are no messages in-flight. The return value indicates whether or not the vertex wishes to continue executing. Typically, an asynchronous application will simply return `false`, while a synchronous one will do some compute, perhaps requesting to send again, and return `true` to start a new time step.

Finish handler If the conditions for calling the `step` handler are met, but the previous call of the `step` handler returned `false` at every vertex, then the `finish` handler is called. The key point here is that the `finish` handler can only be invoked when all vertices in the graph do not wish to continue. At this stage, each vertex may optionally send a message to the host by writing to the provided buffer and returning `true`.

SSSP example To illustrate the `PVertex` class, Figure 8 shows an asynchronous POLite solution to the single-source shortest paths problem. Each vertex maintains an `int` representing the shortest known path to it (initially the largest positive integer), and a read-only `bool` indicating whether or not it is the source vertex. When the application starts, only the source vertex requests to send, but this triggers further iterative sending until the shortest paths to the all vertices have been determined. Finally, when the vertex states have stabilised, the `finish` handler is called to send the results back to the host. In this example, a single pin (pin 0) on each vertex is sufficient to solve the problem.

Graph construction On the host side, i.e. the x86 servers in our cluster, POLite provides a `PGraph` type with operations for adding vertices, pins, edges, and edge weights. For example, a graph for the SSSP example is declared as:

```
PGraph<SSSPVertex, SSSPState, int, int> graph;
```

Using this, an application can prepare an arbitrary graph to be mapped onto the Tinsel overlay. The initial state of each vertex can also be specified using this data structure.

Graph mapping The POLite mapper takes a `PGraph` and decides which vertices will run on which Tinsel threads. It employs an hierarchical graph partitioning scheme using the standard METIS tool [13]: first the graph is partitioned between FPGA boards, then each FPGA's subgraph is partitioned between tiles, and finally each tile's subgraph is partitioned between threads. In each case, we ask METIS to minimise to minimise the edge cut, i.e. the number of edges that cross partitions. After mapping, POLite writes the graph into cluster memory and triggers execution. By default, vertex states are written into the off-chip QDRII+ SRAMs, and edge lists are written in the DDR3 DRAMs. Once the application is up and running, the host and the graph vertices can continue to communicate: any vertex can send messages to the host via the `HostPin` or the `finish` handler, and the host can send messages to any vertex.

Softswitch Central to our implementation of POLite is an event loop running on each Tinsel thread, which we call *the softswitch* as it effectively context-switches between vertices mapped to the same thread. The softswitch has four main responsibilities: (1) to maintain a queue of vertices wanting to send; (2) to implement multicast sends over a pin by sending over each edge associated with that pin; (3) to pass messages efficiently between vertices running on the same thread and on different threads; and (4) to invoke the vertex handlers when required, to meet the semantics of the POLite library.

Limitations One of the features of the Pregel framework [7] is the ability for vertices to add and remove vertices and edges at runtime – but currently, POLite only supports static graphs. And multicasting is currently implemented by sending directly to each destination one-at-a-time. For large fan-outs, a hierarchical multicast (where messages get forked at intermediate stages along the way to the destinations) could reduce communication costs significantly.

V. Evaluation: microbenchmarks

In this section, we use software microbenchmarks written in C++ to test the performance of the default configuration of the Tinsel overlay under basic conditions, before moving on to a more substantial POLite-based case study in Section VI.

Memory To measure the off-chip memory performance, we use a simple microbenchmark in which each thread iterates over a different array in memory and increments each array element. We vary the type of off-chip memory and the cache line size. Here are the measured results from a single DE5-Net, with 1024 threads accessing memory in parallel:

	2×DDR3 (32B lines)	2×DDR3 (64B lines)	4×QDRII+ (32B lines)
Throughput (GB/s)	7.8	10.2	13.0
Bus utilisation (%)	48.8	63.8	81.3
RAM utilisation (%)	30.3	39.9	90.0

Fig. 9. Memory performance for various memory types and cache line sizes.

Depending on the cache line size, the DDR3 throughput ranges from 30%–40%. Full throughput is not expected, due to the somewhat irregular access pattern (all threads accessing different parts of DRAM at the same time). The QDRII+ SRAMs are not sensitive to the access pattern, and a 90% throughput is achieved.

Communication For NoC performance, we use a benchmark in which threads in each tile (Figure 4) exchange messages with each of the neighbouring tiles (i.e. to the north, south, east, and west). The throughput of the NoC is limited by the mailboxes rather than the links connecting tiles together. Each mailbox scratchpad has one flit-sized port on the network side, which can either be read or written on each cycle. So the upper limit on message transmission is the flit size (16 bytes), multiplied by the Fmax (250MHz) divided by two, multiplied by the number of mailboxes (16), which is 32GB/s. Our benchmark achieves a lower throughput of 14.4GB/s (44% of the total), due to contention on the mailbox input port; up to four neighbours can be sending to the same mailbox at a time, leading to backpressure and reducing the rate of the senders.

For inter-FPGA performance, we use a similar microbenchmark, with threads on each FPGA exchanging messages with up to four neighbouring FPGAs in the 2D mesh. The upper limit this time is the bandwidth of each link (10Gbps bidirectional), multiplied by the number of links in the 3×4 mesh (17), which is 42.5GB/s. Our benchmark achieves a throughput

of 32.1GB/s (75% of the theoretical limit). Overhead can be attributed to at least two areas: the use of standard 10G Ethernet MACs and our custom reliability layer sitting on top.

Synchronisation We can measure the basic performance of the termination detection primitive by calling `tinselIdle` in a tight loop on every thread. With no messages in-flight, this will result in global synchronisation of all threads on every iteration. Running this benchmark, we see 40,000 synchronisations per second on a 3×2 FPGA mesh (6,144 threads), dropping to 26,000 on a 3×4 mesh (12,288 threads). As expected, performance does not scale due to the global synchronisation point – indeed this is the motivation for also supporting asynchronous communication in Tinsel. Nonetheless, these synchronisation rates are high enough to be useful in real applications (Section VI).

VI. Case study: distributed graph processing

A recent study [14] explores the performance of three distributed graph processing systems based on Google's vertex-centric programming model [7], including the Apache Giraph system previously used at Facebook [16]. All systems were evaluated on a conventional 128-machine cluster, and a modern system called Blogel [17] was declared best performer. In the remainder of this section, we compare the performance of Blogel (running on a Xeon cluster) against POLite (our own vertex-centric programming abstraction from Section IV running on the Tinsel overlay on top of our FPGA cluster).

Experimental setup We consider four graph processing algorithms: *PageRank* for ranking webpages [18] (with equations implemented using floating-point); *SSSP* (single-source shortest paths) for weighted graphs; *MSSP* (multiple-source shortest paths) for unweighted graphs; and *HashMin* for computing weakly-connected components. The POLite and Blogel versions of each benchmark are synchronous, and essentially the same. We use a *geometric random graph generator* to produce graphs for benchmarking purposes. This generator allows us to easily vary the amount of locality in graphs, so we can explore the limits of the communication subsystem. For the experiments below, we a use geometric random graph with 2M vertices and over 200M edges. All experimental data is available in the data package accompanying this paper [15].

Blogel versions of each application were run on an x86 cluster containing six E5-2430L Xeon servers, each with 12 threads, and 10G Ethernet. The POLite versions were run on our 12-FPGA cluster (Figure 10). The power consumption of a single busy DE5-Net board is just under 50W, which is around half that of a single busy Xeon server. So the 12 DE5-Net FPGAs use the same power as the 6 Xeon servers.

Results Figure 11 shows the scaling characteristics of the two clusters, as well as the relative performance. The relative performance of the FPGA cluster is significantly better, consuming order-of-magnitude less energy than the Xeon cluster on the same workload. One of the characteristics of distributed graph processing systems is that a large number of machines is usually needed to provide a significant advantage over a non-distributed solution to the same problem [14]. This distribution overhead does not have such a big affect on the FPGA cluster, with hardware support for the programming model, along with efficient networking.

Fig. 10. Our current FPGA cluster comprising two POETS boxes (12 worker FPGAs in total, in a 3×4 arragement), each in a standard 4U rack-mountable case. Refer to Figure 1 for further details about POETS boxes.

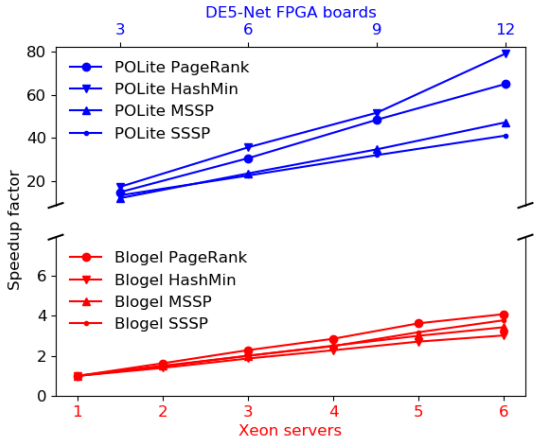

Fig. 11. Performance scaling of the FPGA cluster and the Xeon cluster. Speedup is relative to Blogel performance on a single 12-thread Xeon machine. The two X axes have been aligned w.r.t. power consumption: one DE5-Net FPGA board (50W busy) uses half the power of a single Xeon machine (100W busy); so 12× DE5-Net FPGA boards use the same power as 6× Xeon machines.

Figure 12 shows performance counters from the POLite PageRank application running on all 12 FPGAs. We see 38% utilisation of the 334GB/s total off-chip RAM bandwidth on 12 DE5s, but this limit is very unlikely to be reached in practice due to irregular memory access. Figure 12 also shows the performance of an asynchronous version of PageRank implemented using POLite. The synchronous version offers slightly better performance as it has fewer software overheads, but it will be interesting to monitor this comparison in future as the size of our cluster grows and the cost of global synchronisation increases.

Metric	Sync	GALS
Time (s)	0.49	0.59
Data cache hit rate (%)	91.5	93.9
Off-chip memory (GB/s)	125.8	127.7
CPU utilisation (%):	56.4	71.3
NoC messages (GB/s)	32.2	27.2
Inter-board messages (GB/s)	7.3	6.1

Fig. 12. Performance of synchronous and GALS (globally-asynchronous locally-synchronous) POLite implementations of PageRank on 12 FPGAs.

Feature	**Tinsel-64**	**Tinsel-128**	**μaptive** [24]
Cores	64	128	120
Threads	1024	2048	120
Off-chip DRAM	2×DDR3	2×DDR3	None
Off-chip SRAM	4×QDRII+	4×QDRII+	None
Data caches	16×64KB	16×64KB	0
Flit size (bytes)	16	16	4
NoC	2D mesh	2D mesh	Hoplite torus
FPUs	16	16	0
Inter-FPGA comms	4×10Gbps	4×10Gbps	None
Termination detection	Yes	Yes	No
Fmax (MHz)	250	210	94
Area (% of DE5)	61	88	∼100

Fig. 13. Feature set of the Tinsel overlay versus the μaptive overlay [24], including clock frequencies and area requirements on the DE5-Net board.

VII. Related work

Kumar et al. have recently developed the 120-core μaptive overlay [24] which adapts an existing 32-bit MIPS core (from the Imagination Technologies Academic Program) for the DE5-Net FPGA, and adds support for lightweight inter-core messaging over a Hoplite NoC [25]. A side-by-side comparison against Tinsel is shown in Figure 13. Overall, a 128-core Tinsel configuration uses less area, while clocking at twice the frequency and adding support for floating-point, off-chip memory, data caches, reliable inter-FPGA links, and termination detection. The μaptive overlay implements message-passing through remote-store instructions, where a sending core writes directly into to the scratchpad of a receiving core. This means that control-flow and synchronisation, if desired, must be implemented in software at some expense. The paper does not present any run-time performance results.

Another recent overlay is Gray's GRVI Phalanx [22, 23], a manycore RV32I fabric supporting message-passing via a Hoplite NoC. Gray reports that a single 3-stage GRVI core has an Fmax of 375MHz, uses 320 LUTs, and has a predicted CPI (cycles per instruction) of 1.6. These numbers can be summarised by a single figure of 0.7 MIPS/LUT. By comparison, a single 16-thread pure RV32I Tinsel core (with tightly-coupled data and instruction memories) uses 500 ALMs, clocks at 450MHz, and has a predicted CPI of 1 (there are no pipeline hazards due to multithreading), giving a figure of 0.9 MIPS/LUT. This rough comparison assumes a highly-threaded workload, and involves Fmax and LUT counts taken from different FPGA architectures (Virtex Ultrascale versus Stratix V). Unlike GRVI, Tinsel is not appropriate for single-threaded workloads.

Gray hand-maps a remarkable 1,680 GRVI cores clocking at 250MHz onto a modern, large Xilinx XCVU9P FPGA using relationally placed macros. However, the hand-mapped approach is quite fragile, and its effectiveness could be offset when introducing off-the-shelf IP into the design, e.g. DRAM/SRAM controllers, Ethernet MACs, FPUs, or custom accelerators, all of which are likely to reduce regularity. Off-chip memory access, inter-FPGA communication, and floating-point are left for future work. Gray also cites high-level programming support as an important goal for the future, which we have begun to explore in this paper.

Graph processing has been studied by the FPGA community as a topic in its own right [8, 26, 27]. GraphStep is the seminal work [8], pioneering the use of on-chip graph representations and bespoke graph processing pipelines to achieve high performance. In contrast, we have focused on the case where performance is limited by *off-chip* memory and communication bandwidth. In such cases, processing power can be traded for greater flexibility, resulting in a more widely-applicable overlay, without loss of run-time performance, assuming off-chip resources are saturated. In addition, GraphStep does not support dynamically-changing graphs or asynchronous message-passing, both of which are possible in Tinsel.

VIII. Conclusions and future work

Tinsel is a feature-rich soft-processor overlay allowing an adjustable balance between processing, memory, and communication resources, and is appropriate for distributed applications with modest compute requirements. Hardware multithreading helps tolerate the latencies of floating-point units, external memory, and shared resources while keeping Fmax high, and hardware termination detection supports both synchronous and asynchronous programming styles. We have applied a 12-FPGA version of Tinsel with 12,288 RISC-V threads to the domain of distributed graph processing, observing a significant performance improvement over a Xeon cluster programmed at a similar (vertex-centric) level of abstraction.

Future work In our current design, large numbers of autonomous threads accessing DRAM concurrently leads to a disorganised (suboptimal) access pattern, at least for small cache-line sizes. We plan to explore ways to feed threads the data they need in an order decided by a per-FPGA scheduler, possibly at the expense of some generality. Other areas for improvement include more efficient multicasting, and better transfer rates between the PCs and the FPGAs in our cluster.

Acknowledgments We thank Martijn Bakker, Jonny Beaumont, Graeme Bragg, Tom Bytheway, Brian Jones, Alex Rast, Malcolm Scott, and Mark Vousden for their suggestions and assistance. This work was supported by EPSRC grant EP/N031768/1 (POETS project).

Open access Research data supporting this paper can be obtained online [15]. Meanwhile, this work continues to be developed at https://github.com/POETSII/tinsel.

REFERENCES

[1] S. H. Mirsadeghi, *Improving Communication Performance through Topology and Congestion Awareness in HPC Systems*, PhD thesis, Queen's University, Ontario, 2017.

[2] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, *An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms*, PACT 2015.

[3] M. Attig and G. Brebner, *400 Gb/s Programmable Packet Parsing on a Single FPGA*, ANCS 2011.

[4] P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar, *Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation*, FCCM 2012.

[5] A. T. Markettos, P. J. Fox, S. W. Moore, and A. W. Moore, *Interconnect for commodity FPGA clusters: Standardized or customized?*, FPL 2014.

[6] POETS project website. Online, accessed: 21 May 2019. Available: https://poets-project.org/.

[7] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: A System for Large-scale Graph Processing*, SIGMOD 2010.

[8] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., and A. DeHon, *GraphStep: A System Architecture for Sparse-Graph Algorithms*, FCCM 2006.

[9] *Altera Floating-Point Megafunctions User Guide*, Online, accessed: 3 Jan 2019. Available: https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/ug/ug_altfp_mfug.pdf.

[10] W. J. Dally. *Wire-Efficient VLSI Multiprocessor Communication Networks*, in proceedings of the Stanford Conference on Advanced Research in VLSI, 1987.

[11] R. Mullins, A. West, and S. W. Moore, *Low-Latency Virtual-Channel Routers for On-Chip Networks*, ISCA 2004.

[12] E. W. Dijkstra, *Shmuel Safra's version of termination detection*, EWD998, Online, accessed: 6 Dec 2018. Available: https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF.

[13] G. Karypis, et al., *METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering*. Online, accessed: 20 Dec 2018. Available: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[14] K. Ammar and M. Tamer Özsu, *Experimental Analysis of Distributed Graph Systems*, CoRR 2018.

[15] M. Naylor, Research data supporting "Tinsel: a manythread overlay for FPGA clusters". Available: https://doi.org/10.17863/CAM.40123.

[16] The Apache Software Foundation, *Apache Giraph*, Online, accessed: 15 Mar 2019. Available: http://giraph.apache.org/.

[17] D. Yan, J. Cheng, Y. Lu, and W. Ng, *Blogel: A Block-centric Framework for Distributed Computation on Real-world Graphs*, VLDB Endowment, 2014.

[18] S. Brin and L. Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, 7th International Conference on the World Wide Web, 1998.

[19] S. A. Hauck, *Multi-FPGA Systems*, PhD Thesis, University of Washington, 1995.

[20] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P. Droz, *RAMP Blue: A Message-Passing Manycore System in FPGAs*, FPL 2007.

[21] O. Mencer, et al., *Cube: A 512-FPGA cluster*, SPL 2009.

[22] J. Gray, *A 1680-core, 26 MB Parallel Processor Overlay for Xilinx UltraScale+ VU9P*, Hot Chips 29, 2017.

[23] J. Gray, *GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator*, FCCM 2016.

[24] C. Kumar H B, P. Ravi, G. Modi, and N. Kapre, *120 core microAptiv MIPS Overlay for the Terasic DE5-NET FPGA Board*, FPGA 2017.

[25] N. Kapre and J. Gray, *Hoplite: Building austere overlay NoCs for FPGAs*, FCCM 2015.

[26] E. Nurvitadhi, et al., *GraphGen: An FPGA Framework for Vertex-Centric Graph Computation*, FCCM 2014.

[27] N. Kapre, *Custom FPGA-based soft-processors for sparse graph acceleration*, ASAP 2015.