General hardware multicasting for fine-grained message-passing architectures

Matthew Naylor[1], Simon W. Moore[1], David Thomas[2], Jonathan R. Beaumont[2], Shane Fleming[3], Mark Vousden[4], A. Theodore Markettos[1], Thomas Bytheway[1], and Andrew Brown[4]
[1]University of Cambridge, [2]Imperial College London, [3]Swansea University, [4]University of Southampton

***Abstract*—Manycore architectures are increasingly favouring message-passing or partitioned global address spaces (PGAS) over cache coherency for reasons of power efficiency and scalability. However, in the absence of cache coherency, there can be a lack of hardware support for one-to-many communication patterns, which are prevalent in some application domains. To address this, we present new hardware primitives for multicast communication in rack-scale manycore systems. These primitives guarantee delivery to both colocated and distributed destinations, and can capture large unstructured communication patterns precisely. As a result, reliable multicast transfers among any number of software tasks, connected in any topology, can be fully offloaded to hardware. We implement the new primitives in a research platform consisting of 50K RISC-V threads distributed over 48 FPGAs, and demonstrate significant performance benefits on a range of applications expressed using a high-level vertex-centric programming model.**

I. Introduction

Today's general-purpose processors rely on elaborate hardware features such as superscalar execution and cache coherency to automatically *infer* parallelism and communication from general workloads. However, for inherently parallel workloads with explicit communication patterns, which are common in HPC domains, these costly hardware features become unnecessary. Instead, processors consisting of larger numbers of far simpler cores, communicating by message-passing or PGAS, can achieve more performance from a single chip, and scale more easily to large numbers of chips. This is the premise behind a number of recently developed manycore designs [1, 2, 3, 4, 5, 6, 7].

One of the strengths of cache-coherent architectures is hardware support for one-to-many communication patterns. Data from a single writer is automatically propagated to multiple readers through a hierarchy of caches, which avoid repeated fetching of the same data from a central point. In contrast, message-passing/PGAS architectures commonly provide hardware unicast primitives that are limited to sending/storing to a *single* destination core/address at a time. As a result, multicasting is typically implemented in software by repeated unicast, which can easily lead to congestion on the manycore interconnect.

Congestion is the main threat to the scalability of manycore architectures, and is exacerbated by a lack of communication locality in some HPC domains, or by the cost of identifying locality. Long-range interactions typically involve a greater number of less efficient links, leading to bottlenecks. For example, in distributed graph processing, small-world graphs are common and contain inherently distributed fanouts where neighbouring vertices cannot all be physically colocated. Even in cases where locality of communication is high, it can be expensive to identify in very large data sets, to the point where the cost may outweigh the benefit of manycore execution in the first place. Either way, efficient support for distributed communication becomes key.

The most general approach to hardware-accelerated multicasting in manycore architectures that we are aware of comes from the SpiNNaker machine [2] developed at the University of Manchester. SpiNNaker is a one million ARM core machine featuring a torus of chip-multiprocessors (CMPs) connected together via *programmable routers*. Each router allows an incoming packet to be delivered simultaneously to any subset of the 18 cores in the CMP (*local multicasting*), and it allows bifurcation of packets in up to six directions as they travel from chip to chip through the torus (*distributed multicasting*). This is a powerful combination of features, but some aspects of the design are unsatisfying outside SpiNNaker's target domain of spiking neural network simulation:

- Each programmable router makes routing decisions using a 1024-entry content-addressable memory (CAM), which is too small to capture large unstructured communication patterns precisely. Instead, packets can be delivered approximately to a superset of the desired destinations, and software at the receivers can decide whether or not to discard them. Naturally, this leads to more traffic on the network than is necessary. It also means that multicast communication requires software assistance, and software disposal of unwanted messages is a significant overhead.
- The hardware does not provide guaranteed delivery. There is no hardware-enforced flow control and packets are dropped when the communication fabric is overwhelmed. Dropped packets are "retained in a buffer for software examination" [2], but software retransmission schemes are complex and will only lead to more bookkeeping on the cores and more traffic on the network.

In this paper, inspired by the SpiNNaker design, we explore new features for hardware multicasting that are *precise*, *reliable*, and *generally applicable* to a range of HPC domains. Our contributions are as follows.

- We describe the drawbacks of implementing one-to-many communication patterns in software, i.e. *software multicasting*, especially while guaranteeing delivery.
- We present new techniques for local and distributed hardware multicasting, implemented on top of a many-

core message-passing machine consisting of 50K RISC-V threads spread over 48 FPGAs. These techniques preserve an event-driven API with software-exposed flow control, and are highly general: multicast transfers among any number of software tasks, connected in any topology, can be fully offloaded to hardware.

- We extend a high-level vertex-centric software API, inspired by Google's Pregel model [8], to utilise the new multicast features. The API completely abstracts over architectural details, providing a rapid, portable and scalable deployment path for application developers.
- We apply the resulting hardware and software to a range of HPC problems from graph processing, spiking neural networks, and dissipative particle dynamics. We evaluate the impact of hardware multicasting on network traffic and execution time as we vary the amount of locality in the data sets. We also evaluate the cost of the new hardware features in terms of logic area and clock frequency.

II. Background

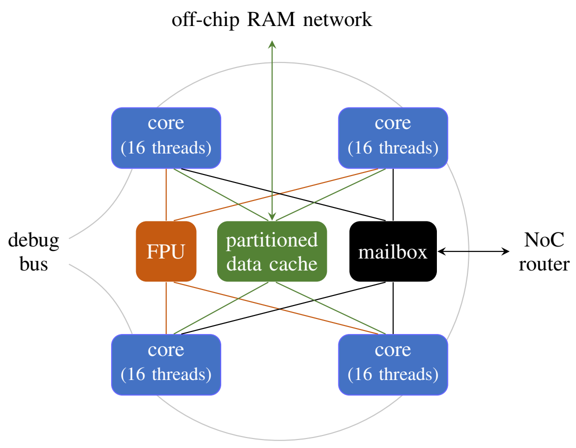
As part of a larger project, we have constructed a research platform consisting of a 48-FPGA cluster and a manycore RISC-V overlay programmed on top [7, 9]. As well as providing a reconfigurable compute fabric, FPGAs also support a high degree of scalability due to advanced inter-chip networking capabilities. The research platform therefore serves both as a rapid prototyping environment for computer architecture research and, for distributed applications, a genuine hardware accelerator. Below, we outline the design of the platform, and its unicast communication primitives, before presenting our multicast extensions in subsequent sections.

A. *Research platform*

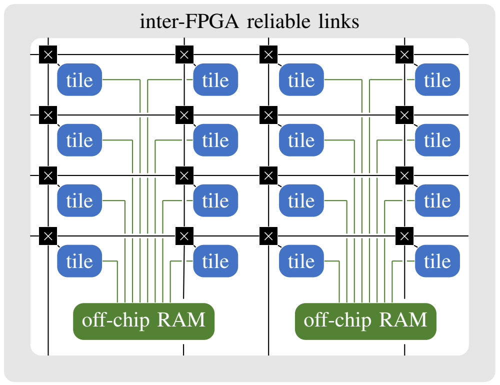
Our RISC-V **overlay** (Tinsel) has regular structure, consisting of a scalable grid of **tiles** connected by a reliable communication fabric that extends both within each FPGA and throughout the FPGA cluster. By default, a tile consists of four RV32IMF cores sharing an FPU, a data cache, and a mailbox, as shown in Figure 1a.

The **core** is 16-way multithreaded by default, capable of tolerating tens of cycles of latency arising, for example, from deeply-pipelined FPGA floating-point operations, or cache misses that lead to off-chip memory accesses. Threads are barrel-scheduled (a context switch is performed on every clock cycle) so pipeline hazards are easily avoided, leading to a small, fast, high-throughput design.

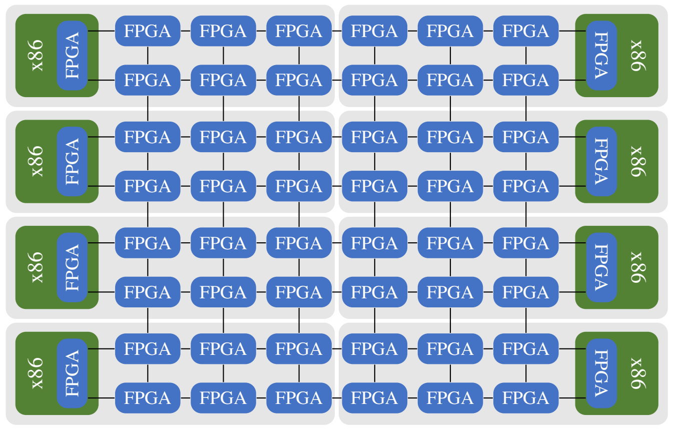
The **mailbox** contains a memory-mapped scratchpad storing up to 64KB of incoming and outgoing messages, which can also be used as a small general-purpose local memory. Messages are variable-length, containing up to four flits, with each flit holding 128 bits of payload. The mailbox allows threads to trigger transmission of outgoing messages, to allocate space for incoming messages, and to consume those messages when they arrive, all via custom RISC-V CSRs (control/status registers). The mailbox features are detailed in Section II-B.



(a) Default configuration of a 64-thread tile.



(b) Default configuration of our overlay on a single DE5-Net (Stratix V) FPGA board. Mailboxes in tiles are connected together using dimension-ordered routers to form a NoC. Inter-FPGA reliable links are connected to the NoC rim. A separate network is used to connect caches in tiles to off-chip memories. Each off-chip RAM component contains a 2GB DDR3 DRAM and two 8MB QDRII+ SRAMs.



(c) Our FPGA cluster is composed of eight *boxes* (shown in light gray). Each box contains an x86 server and 7× DE5-Net FPGA boards. One FPGA board in each box serves as a bridge between the x86 server (PCI Express) and the FPGA network (10G SFP+). The x86 servers provide command-and-control facilities, such as data injection and extraction. The full cluster contains a 2×4 mesh of boxes, and a 6×8 mesh of worker FPGAs.

Fig. 1. Manycore research platform supporting fine-grained message-passing.

The **cache** is a non-blocking 16-way set-associative write-back cache that optimises access to the large off-chip memories available on each FPGA board. It is 128KB in size and is partitioned by thread id so that cache lines are not shared between threads. This means there are no hazards in the cache pipeline, which leads to a small, fast design. Caches are not coherent: message-passing is expected to be the primary means of communication.

The **FPU** supports IEEE single-precision floating-point operations. On the Stratix V FPGAs we are using, these are expensive in both area and latency, which we mitigate through sharing and multithreading respectively.

A single-FPGA view of the overlay is depicted in Figure 1b. On the DE5-Net FPGA board, the overlay contains 64 RV32IMF cores (1,024 hardware threads), clocks at 240MHz, and utilises 67% of the FPGA. The FPGA **cluster** comprises a 6×8 grid of DE5-Net boards connected using 10G reliable links, as shown in Figure 1c. The overlay distributes over this cluster to yield a 3,072 core system (49,152 hardware threads), where any thread can send messages to any other thread.

B. Hardware unicasting

Threads send and receive messages using custom RISC-V control/status registers (CSRs). These raw CSR accesses are abstracted by a very thin software API, outlined below.

To send a message residing in the mailbox scratchpad, a thread first ensures that the network has capacity by calling

```
bool tinselCanSend();
```

and if the result is true, the thread can call

```
void tinselSend(uint32_t dest, 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.

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

```
void tinselAlloc(void* msg);
```

where `msg` is a message-aligned address in the scratchpad. Space for multiple messages can be allocated in this way, creating a receive buffer of the desired size. 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

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

which returns a pointer to the received message. Receiving a message can be viewed as transferring ownership of the space it occupies from the hardware back to the software. Once the software is finished with the received message, it can call `tinselAlloc` again to pass ownership of the memory back to the hardware, replenishing the receive buffer.

To further support event-driven computing, the hardware provides thread suspension and resumption on can-receive and/or can-send events [11]. It also provides a form of barrier synchronisation based on termination detection, which is well suited to fine-grained message-passing systems [9].

C. Guaranteed delivery and deadlock avoidance

Message delivery is guaranteed by the hardware provided that all threads eventually consume the messages available to them. This is complicated by the fact that, in general, a thread is free to send a message to any destination thread, including one which it may receive from, potentially forming a cyclic dependency with both sides waiting for the other to receive. To avoid this message-dependent deadlock in general, a thread must not block indefinitely on a send operation without considering the possibility of receiving. This is a common requirement in lossless on-chip networks, sometimes referred to as the *consumption assumption* [10].

D. Software multicasting

The aim of multicasting is to send the same message to multiple destinations while minimising traffic on the network. It can potentially be achieved *in software*, on top of hardware unicast primitives, by sending via a tree of intermediate cores. Each node in the tree can receive the message and forward it to its children, with the leaves of the tree being the final destinations. The closer to the destinations that a message is forked/copied, the less space it consumes on the network.

Some properties of our mailbox design are potentially helpful for software multicasting. First, the mailbox scratchpad stores both incoming and outgoing messages, so a message can be efficiently forwarded (received and sent) by software without being copied. Second, once a message is in the scratchpad, it can be efficiently sent multiple times by calling a single-cycle instruction for each child.

However, there are major challenges facing software multicasting. The consumption assumption requires that software is always willing to receive. If the rate at which messages arrive at a thread exceeds the rate at which they can be forwarded on, the messages must be buffered. Buffering is expensive if it requires software to copy messages from the on-chip mailbox to the off-chip memory and back. Worse, in the general case, there is no useful upper bound on the size of the buffer; the consumption rates of threads are application-dependent and hard to predict. Furthermore, there is both a latency penalty and compute penalty as messages pass through a software stack at each fork-point in the multicast tree.

Hardware routing primitives in our research platform do not suffer from the unbounded buffering problem. They follow a more disciplined form of addressing than software (dimension-ordered routing), which excludes cyclic dependencies, and they can safely exert backpressure on the network without the risk of deadlock. Combining this with performance benefits, hardware multicasting becomes an attractive possibility.

III. Local hardware multicasting

Each tile in our research platform contains 64 threads sharing a mailbox, i.e. a memory-mapped scratchpad holding incoming and outgoing messages. This raises the possibility of sending a single message to multiple threads on same tile simultaneously: a message can be transferred over the network once, stored in the destination mailbox once, and made available to any specified subset of the 64 threads sharing the mailbox. We refer to this as *local multicasting* as the destinations must be colocated on the same tile.

A. API modifications

We extend the hardware with a primitive for sending to multiple destination threads on a given tile:

```
// Send message to multiple threads on given tile
void tinselMulticast(
  uint32_t tileDest,      // Destination tile id
  uint32_t destMaskHigh, // Thread mask (high bits)
  uint32_t destMaskLow,  // Thread mask (low bits)
  void* msg);            // Message pointer
```

Like `tinselSend`, this primitive should only be called when `tinselCanSend` returns true. Unlike `tinselSend`, it takes a 64-bit mask (provided as two 32-bit parameters) denoting which of the 64 threads in the destination tile will receive the message.

On the receiver side, the API works as before: when a thread is finished with a received message, it informs the hardware via a call to `tinselAlloc`, and the hardware can reuse the memory for a new message. However, since a single message may be received by multiple threads, the hardware now must wait until *all* receivers have called `tinselAlloc` on the message before reusing the memory; but this is transparent to software.

B. NoC modifications

One of the main costs introduced by the `tinselMulticast` primitive is that the destination address of a message increases in size by 58 bits. For unicast, only 6 bits are required to identify the receiving thread on the destination tile; for multicast, this increases to 64 bits. In our current design, where every flit contains the destination address in addition to a 128 bit payload, the increased address size leads to larger flits, and wider buses to carry them through the NoC.

There are alternatives, which would reduce the address size. One option would be to use a compressed thread mask where, for example, 30 bits are used to hold 5 $\times$ 6-bit thread ids. Another possibility would be to require that the destination tile is the same as the tile of the thread that is sending, meaning that flits on the NoC never carry the mask. But both of these alternatives limit the range of the multicast, and we wish to explore the full potential of the approach.

C. Mailbox modifications

The majority of the modifications needed are in the mailbox design. The hardware needs to ensure that each message is stored once but made available to multiple receivers, and to ensure that space occupied by the message can be garbage collected when all receivers have finished with it. This is achieved using the following mailbox structures, as shown in Figure 2.

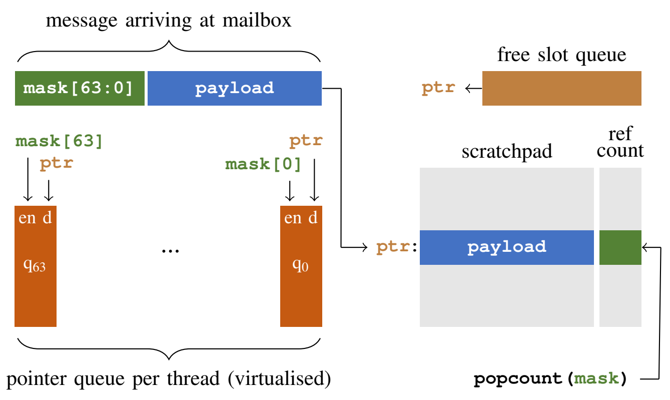


Fig. 2. Mailbox features supporting local hardware multicasting. The message payload is written into the memory-mapped scratchpad at an address indicated by the free slot queue. This address is simultaneously written into the pointer queue for each receiver as indicated by the message's destination mask. The population count of the mask is used to start a reference count for the message.

- A memory-mapped *scratchpad* storing incoming and outgoing messages, accessible to all threads in the tile. This is much the same as in the unicast design.
- A *free slot queue*, which contains a set of message-aligned addresses in the scratchpad that can be used to store incoming messages.
- A *pointer queue* for each thread, containing pointers to messages in the scratchpad that are to be received by that thread. These queues are partially virtualised to reduce on-chip memory resources, as discussed below.
- A *reference count* for each message-aligned address in the scratchpad, maintained in a separate block RAM. This count is the number of receiving threads that have yet to indicate that they are finished with the message.

The operation of the mailbox is as follows.

- When a message containing a *payload* and a destination thread *mask* arrives at the mailbox, the hardware tries to extract a scratchpad pointer *ptr* from the free slot queue. If the free slot queue is empty then the mailbox exerts backpressure on the NoC.
- The payload is written to the scratchpad at the location pointed-to by *ptr*, and the *ptr* is inserted into each thread's pointer queue as indicated by the mask. If any of the pointer queues are full then the mailbox exerts backpressure on the NoC.
- The number of high bits in the mask is computed using a population count and written in to the reference count RAM for the message at location *ptr*.
- When a thread calls `tinselRecv`, its pointer queue is dequeued. The empty status of the queue can be obtained by a call to `tinselCanRecv`.
- When a thread calls `tinselAlloc` to indicate that it has finished with the message, the reference count of that message is decremented. When the reference count reaches zero, the message address is inserted back into the free slot queue for future use.

Creating a physically separate pointer queue for each of the 64 threads in a tile allows all the queues to be accessed in parallel, but is potentially very expensive in terms of on-chip memory resources. Furthermore, this parallelism would be disproportionate in the sense the scratchpad is still shared by all 64 threads and this is likely to be a more limiting factor on the rate of message consumption. Therefore we implement the pointer queues as four physically separate sets of 16 virtual queues. Each set of virtual queues is implemented using a single block RAM for queue data and a smaller LUT-based RAM for queue metadata (front and back pointers).

IV. Distributed hardware multicasting

For applications with limited communication locality, or where the size of data sets makes identifying locality prohibitively expensive, local multicasting alone is not sufficient. In this section, we present a complementary approach to multicasting, useful for distributed fan-outs.

A. Overview

To support distributed multicasting, we generalise the destination address of a message so that it can either be a global thread id (as in the existing unicast design), or a *routing key*. Provided `tinselCanSend` returns true, a thread can send a *key-addressed* message using a new primitive:

```
// Send message using routing key
void tinselKeySend(uint32_t key, void* msg);
```

The resulting message is routed by the NoC towards a *programmable router* on the same FPGA, as shown in Figure 3. Once the message reaches the programmable router, its key is used as an index into a routing table stored in off-chip RAM, yielding a contiguous array of *routing records*. Each record contains a new destination for the message and, for each record, the router emits a copy of the message with a modified destination. A new destination may be single thread on the local NoC, multiple threads on the same tile on the local NoC, or a new routing key to be sent along one of the inter-FPGA links. In this way, the programmable routers can propagate messages unassisted throughout the system, minimising inter-FPGA bandwidth usage. By storing routing tables in off-chip RAMs, large unstructured communication graphs can be captured precisely. Contiguous arrays of routing records can be efficiently fetched using burst transfers, making effective use of DRAM. Caching is not employed due to lack of locality, but could be useful when routing tables are small.

B. Routing keys and records

A routing key is a 32-bit value consisting of an address and a size. The address is always aligned to a 32-byte DRAM beat, leaving five bits to specify the number of beats pointed to. Storing the size in the key itself allows the hardware to issue a burst read of the appropriate size. Each 256-bit beat contains several routing records, along with the number of records in the beat. The five-bit size in the key requires that the routing records fit in less than 32 beats, but a mechanism (discussed below) exists to workaround this.

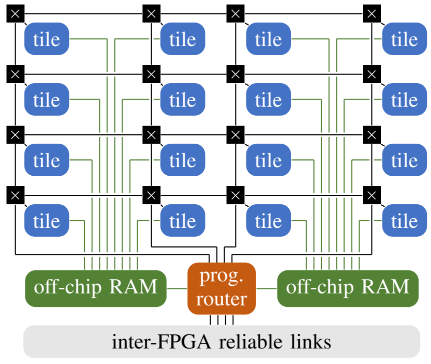


Fig. 3. New NoC structure to support distributed hardware multicasting. Comparing to Figure 1b, we introduce a *programmable router* that routes messages between and within FPGAs based on routing tables stored in large, off-chip memories.

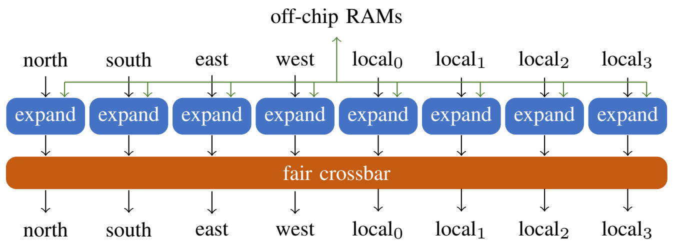


Fig. 4. Internal structure of the programmable router. Key-addressed messages from inter-FPGA links and the local NoC are expanded using routing records fetched from off-chip memory at locations specified by the keys. The resulting messages are then routed through a crossbar for the next stage of their journey.

Routing records range in size from 48 bits to 96 bits, and there are four main kinds of record understood by the router:

- A *router-to-thread* record specifies a thread id on the local NoC to which the message should be sent.
- A *router-to-tile* record specifies the coordinates of a tile on the local NoC, along with a thread mask for that tile, to which the message should be sent, exploiting local hardware multicasting.
- A *router-to-router* record specifies that the message is to be forwarded to the north, south, east, or west inter-FPGA links. It also specifies a replacement routing key for the message. This simplifies the task of mapping, as discussed in Section V.
- An *indirection* record specifies that the message is to be fed back into the current programmable router, with a new routing key. This eliminates the limitation on the number of routing records associated with a key.

Variants of the router-to-thread and router-to-tile records are provided that allow the first word or half-world of the message payload to be overwritten with data from the record. Again, this simplifies mapping, as discussed in Section V.

C. Programmable router design

The programmable router routes messages between the four inter-FPGA links and the four links on the bottom side of the NoC using an 8x8 crossbar, as shown in Figure 4. Before messages are fed into the crossbar, they pass through an *expander* for each link. If a message is key-addressed then the expander forks it according to routing records its fetches from off-chip RAM, otherwise it passes straight through unmodified. The RAM latency is masked by a 32 element flit queue inside each expander, allowing RAM requests to be pipelined. Each expander has its own connection to RAM, and operates independently of the others to avoid deadlock, i.e. ensuring that the ability of one expander to make progress is not affected by that of another. The decision to use only one programmable router per FPGA, and to connect to only one side of the NoC, is motivated by the high logic cost of the expanders and the crossbar, as well as the bandwidth limit of the off-chip RAM.

V. High-level API

Developing software applications directly on top of our RISC-V overlay is time-consuming, requiring manual mapping of abstract application-level concepts onto the fixed hardware structures. To address this, we have developed a high-level API, inspired by Google's Pregel model [8], in which an application is expressed as an *abstract task graph*. The API completely hides architectural details but, at the same time, plays to the strengths of the architecture by exposing concepts such as message-passing and multicasting. In this section, we briefly introduce task graphs and discuss how they are mapped onto our research platform, focussing on how the new hardware multicast features are exploited. Further details on the API can be found in our research platform's manual [11].

A. Task graphs

A task graph is a set of interconnected state machines (tasks) that can send messages to each other via edges. Task behaviour is defined by set of software *event handlers* (written in C++) that update the *task state* when a particular event occurs, e.g. when a message arrives on an incoming edge, or the network is ready for a new message to be sent, or when global synchronisation occurs. Tasks send messages via *pins*. A pin is a set of outgoing edges, and sending a message on a pin means sending a message along all edges in the set, i.e. a multicast send. A task can have any number of pins.

Task graphs are prepared offline using a C++ class with methods for adding tasks, pins, and edges, and for initialising task and edge states. The number of tasks and the topology of the graph are unrestricted. Deciding which tasks will run on which threads is the job of our *mapper*, which employs a hierarchical partitioning scheme: 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. A variety of partitioning algorithms are available, most of which aim to minimise the edge-cut between partitions. The default is METIS [12], which is the slowest but gives the highest quality results. For large graphs, multiple tasks end up being mapped onto each thread. To handle this, each thread runs an event-loop called the *softswitch*, which context-switches between tasks and invokes event handlers when appropriate.

In the remainder of this section we discuss how abstract pins are implemented using unicast, local multicast, and distributed multicast features respectively.

B. Using unicast primitives

Sending a message on a pin that refers to multiple destination tasks clearly may require multiple unicast send operations. However, multiple tasks may map to the same thread, so only one unicast send is needed per destination thread, not necessarily one per destination task. To exploit this fact, we implement a pin as an array of thread/key pairs on the sender side. To send a message on a pin, we iterate over the array and, for each thread/key pair, we assign the key to part of the message payload and perform a unicast send to the thread. On the receiver side, the key in the payload is used as an offset into a *receiver table*, yielding an array of local task-id/edge-state pairs. The softswitch then invokes the receive handler for each task in the array, passing it the message and the appropriate edge state.

C. Using local multicast primitives

With local hardware multicasting, we only need to send a message once per destination tile, rather than once per destination thread. On the sender side, we adapt the array of key/thread pairs representing a pin's destinations to an array of key/tile/thread-mask triples. This is a straightforward change in itself, but introduces a new problem: all receiving threads on the same tile will receive the *same payload* and hence the *same key*. Therefore, the receiver array for each receiving thread in the tile needs to stored at the same offset in each thread's receiver table. This leads a non-trivial optimisation problem: we want to pack the receiver arrays into each thread's memory, minimising the amount of space used, while ensuring that some arrays, on different threads, are mapped to the same offset. We refer to this as the *receiver packing problem*. We solve it using a greedy bitmap-based data structure that allows rapid searching for locations that are free amongst all the receivers' tables.

D. Using distributed multicast primitives

With distributed hardware multicasting, we only need to send a message once per pin, regardless of how many destinations it represents. The entire sender-side pin array can replaced by an array of routing records at the programmable router. But now the receiver packing problem is exacerbated by the fact that multicast destinations can be distributed over the *entire system* rather than being limited to a single tile. Part of the aim of distributed multicasting is to avoid the need for expensive mapping algorithms, and the system-wide packing problem risks becoming costly. Therefore, the programmable router provides two features to avoid it: (1) router-to-thread and router-to-tile records can specify that the key in the payload is replaced with a new key before forwarding the message to the local NoC; and (2) router-to-router records can specify that the routing key is replaced, in a similar way. One of the drawbacks of routing every message via

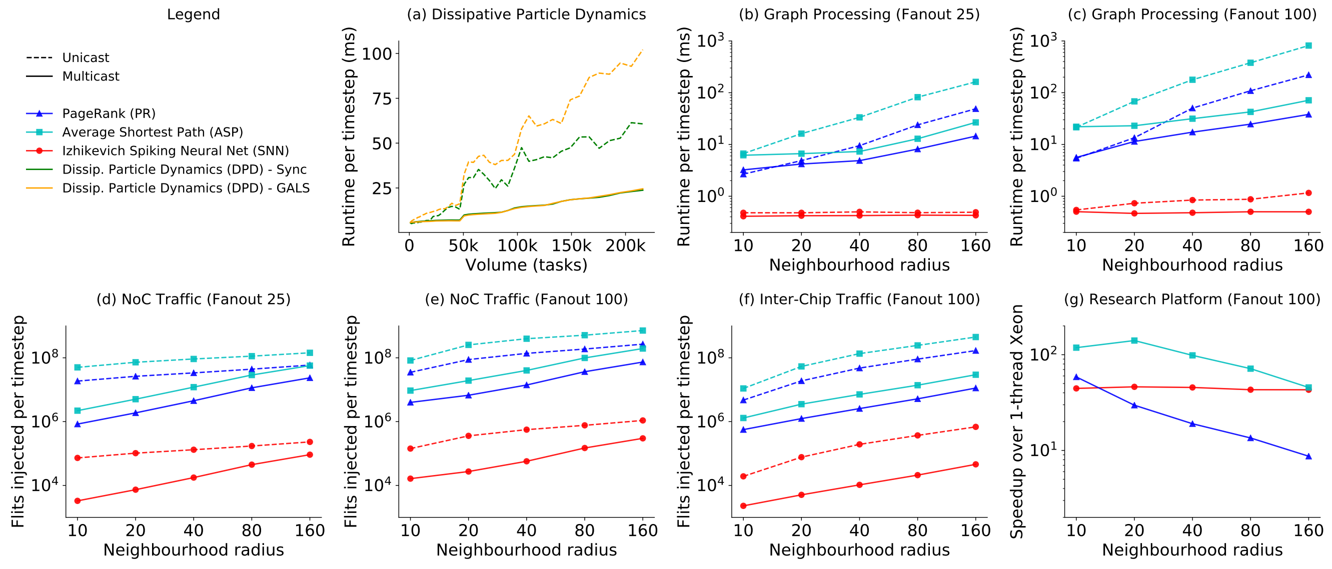


Fig. 5. Performance comparison of unicast and multicast approaches on a variety of applications.

the programmable router is that it becomes a bottleneck as it connects to only one side of the NoC. For this reason, we take a hybrid approach: we only use programmable routers for pins containing at least one destination task that does not reside on the same FPGA as the sender.

VI. Evaluation

In this section, we present performance results from a range of applications, implemented using our high-level API, and running on our research platform. We compare the use of unicast primitives and multicast primitives to implement pins, while varying the amount of locality in the task graphs. We also evaluate the cost of hardware multicasting in terms of FPGA area and clock frequency.

A. Experimental setup

We present results for four applications exhibiting a range of computational characteristics. The first is a Dissipative Particle Dynamics (**DPD**) simulator for modelling particle interactions in computational chemistry [13], requiring only local communication patterns. Space is divided into a *regular 3D grid* of tasks, with each task communicating on every timestep with its 26 neighbours. The remaining three applications all operate over *unstructured* task graphs: an implementation of the PageRank (**PR**) algorithm for ranking webpages [14]; a solver for the average shortest path (**ASP**) problem, which performs a multiple-source breadth-first search; and a spiking neural network (**SNN**) simulator using the Izhikevich model [15].

To generate unstructured task graphs for benchmarking purposes, we use a *geometric random graph generator*. Tasks are laid out in a 2D space, and each task is connected to a random set of other tasks within a specified *neighbourhood radius*. This gives us control over the amount of locality in the graphs. Each generated graph contains 1M tasks, with a fanout varying from 25 to 100. The graphs and applications (except DPD) are available in an open-access data package [16], along with the entire source code of our RISC-V overlay and APIs.

B. Performance results

Figure 5a shows that hardware multicasting has a big impact on two separate implementations of DPD. Unicast performance is hindered by NoC congestion issues, especially in the GALS version which involves more communication than the synchronous version. The Figure also shows that multicast DPD performance scales very well up to 50K tasks, the point at which we run out of hardware threads.

Figures 5b–5c show that hardware multicasting introduces resilience to reduction in communication locality. As locality decreases, the performance improvement of multicasting approaches an order-of-magnitude for a fanout of 25, and two orders for a fanout of 100. Multicasting is more beneficial in ASP, which uses four-flit messages, compared to PR and SNN, which use single-flit messages. The benefit of multicasting is generally much lower for SNN due to relatively infrequent communication (standard spiking rates), but is still noticeable for large fanouts. The improvement due to hardware multicasting is largely explained by the reduction in network traffic, as shown in Figures 5d–5f. In particular, the reduction in inter-chip traffic, where bottlenecks are most likely, closely mimics the improvement in runtime performance for larger radii.

Finally, Figure 5g shows that our research platform and high-level API can achieve a useful level of performance compared to optimised single-threaded code running on a conventional Xeon E5-2667 processor.

C. Synthesis results

The performance improvements due to hardware multicasting come at a cost. Local multicasting results in an increased flit size, while distributed multicasting introduces a programmable router with eight independent flit buffers and

	Area (ALMs)	**Fmax**
Unicast baseline	155,935 (66%)	241MHz
+ local multicast	173,718 (74%)	230MHz
+ distributed multicast	197,445 (84%)	215MHz

Fig. 6. Synthesis results for our RISC-V overlay with various extensions on the Stratix-V-based DE5-Net FPGA board. These are the best results from a batch of 20 synthesis runs using Quartus Design Space Explorer.

an 8x8 crossbar. Figure 6 shows the impact on FPGA area and clock frequency. The reduction in clock frequency is expected as a natural consequence of higher FPGA utilisation, leading to a more difficult mapping problem for the synthesis tools.

VII. Related work

Manycore architectures that omit cache-coherency features fall into two main camps: message-passing [1, 2, 4, 7] and PGAS [3, 5, 6]. In PGAS, cores communicate by performing remote loads and stores to local memories of other cores. If multiple cores share a local memory then local multicasting is possible on top of PGAS without any further hardware extensions. However, PGAS architectures do not provide notification or flow control to software, which leads to a more challenging programming model. As a result, PGAS architectures often rely on global barrier synchronisation for flow control but this has the limitation that all communicated data for a timestep must fit into local memories.

Multicasting techniques are becoming increasingly popular in the field of networks-on-chip [17, 18, 19, 20]. These techniques mainly focus on the concise encoding of multiple destinations in a message header, and are not suitable for capturing large unstructured communication patterns. This body of work also tends to focus on NoCs in isolation, rather than within the context of a manycore system. For example, the pragmatics of delivering a message to multiple cores/threads sharing a mailbox is out of scope, and this is the main issue addressed by our local multicasting scheme.

The use of programmable routers in manycore architectures, such as SpiNNaker [2], is a less studied topic. As discussed in Section I, SpiNNaker's programmable routers have some major limitations: message delivery is not guaranteed by hardware, and routing tables are stored in space-limited content-addressable memories (CAMs). More generally, the use of programmable routers has been criticised for increased latency and power requirements due to storing routing tables in memory [20, 21]. However, large unstructured topologies must be stored somewhere with sufficient capacity, and the cost of accessing memory must be paid whether it is accessed by software running on the cores or by programmable routers.

VIII. Conclusion

Multicasting is a valuable technique to reduce network congestion in manycore systems, for a range of applications. In some cases, it can be implemented in software running on the cores, but this is fraught with difficulty in a message-passing architecture with guaranteed delivery. The consumption assumption, needed by software to avoid message-dependant deadlock, leads to a buffering requirement that is effectively unbounded. In contrast, multicast-enabled hardware routers are free to exert backpressure on the network without the risk of deadlock. Hardware multicasting is also more efficient, offloading work from the cores.

In this work, we have designed, implemented, and evaluated new techniques for hardware multicasting that support both colocated and distributed destinations. These techniques preserve an event-driven API with software-exposed flow control – two main features of the message-passing paradigm. To our knowledge, they are the first such techniques capable of capturing large unstructured communication patterns precisely. All this has been done in a whole-system context, from low-level microarchitecture to high-level architecture-agnostic application development, and has been demonstrated on range of realistic applications. We hope these experiences will serve the future development of manycore architectures, in an era where power efficiency and scalability become evermore important.

Acknowledgments This work was supported by UK EPSRC grant EP/N031768/1 (POETS project).

References

[1] A. Gara, J. Moreira *IBM Blue Gene supercomputer*, IBM Research Report, 2011.
[2] C. Patterson, J. Garside, E. Painkras, S. Temple, L. A.Plana, J. Navaridas, T.Sharp, S. Furber. *Scalable communications for a million-core neural processing architecture*, Journal of Parallel and Distributed Computing, Elsevier, 72:11, 2012.
[3] L. Gwennap. *Adapteva: More flops, less watts*, Microprocessor Report, June 2011.
[4] B. Bohnenstiehl et al. *KiloCore: A 32-nm 1000-Processor Computational Array*, IEEE Journal of Solid-State Circuits, 52:4, 2017.
[5] S. Davidson et al. *The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips*", IEEE Micro, 38:2, 2018.
[6] J. Gray, *A 1680-core, 26 MB Parallel Processor Overlay for Xilinx UltraScale+ VU9P*, Hot Chips 29, 2017.
[7] M. Naylor, S. W. Moore, D. Thomas. *Tinsel: a manythread overlay for FPGA clusters*, FPL 2019.
[8] G. Malewicz et al. *Pregel: A System for Large-scale Graph Processing*, ACM SIGMOD 2010.
[9] M. Naylor et al. *Termination detection for fine-grained message-passing architectures*, ASAP 2020.
[10] A. Hansson, et al. *Avoiding Message-Dependent Deadlock in Network-Based Systems on Chip*, Journal of VLSI Design, April 2007.
[11] M. Naylor et al. *Tinsel 0.8.3 Manual*, Online, accessed: 3 Jan 2020. Available: https://github.com/POETSII/tinsel.
[12] G. Karypis, et al., *METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering*. Online, accessed: 8 Oct 2020. Available: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.
[13] P. J Hoogerbrugge and J. M. V. A Koelman, *Simulating Microscopic Hydrodynamic Phenomena with Dissipative Particle Dynamics*, Europhysics Letters, 19:3, 1992.
[14] S. Brin and L. Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, 7th International Conference on the WWW, 1998.
[15] E. Izhikevich, *Simple model of spiking neurons*, IEEE Transactions on Neural Networks, 14(6), 2003.
[16] M. Naylor, Research data supporting this paper. Available: https://doi.org/10.17863/CAM.62761.
[17] M. Ebrahimi et al. *An efficent dynamic multicast routing protocol for distributing traffic in NOCs*, DATE 2009.
[18] L Wang, Y. Jin, H. Kim and E. J. Kim, *Recursive partitioning multicast: A bandwidth-efficient routing for Networks-on-Chip*, NOCS 2009.
[19] L. Wang et al. *Efficient lookahead routing and header compression for multicasting in networks-on-chip*, ANCS 2010.
[20] X. Wang, et al. *Efficient multicast schemes for 3-D Networks-on-Chip*. Journal of Systems Architecture 59:9, 2013.
[21] S. Rodrigo, J. Flich, J. Duato and M. Hummel, *Efficient unicast and multicast support for CMPs*, MICRO 2008.