International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 2, pages 307–320, July 2014

Pronto: A Low Overhead Message Passing System for High Performance Many-Core Processors

Sumeet S. Kumar, Mitzi Tjin-A-Djie, Rene van Leuken
Circuits and Systems Group, Faculty of EEMCS,
Delft University of Technology,
The Netherlands
{*s.s.kumar, t.g.r.m.vanleuken*}@*tudelft.nl*

Received: February 14, 2014
Revised: April 28, 2014
Accepted: June 2, 2014
Communicated by Koji Nakano

Abstract

Many-core processors provide the raw computation power required by modern high-performance multimedia and signal processing workloads. The conversion of this computation power into execution performance is often constrained by the overheads of communication between concurrent tasks. This paper presents Pronto, a low overhead message passing system which simplifies the semantics of data movement between communicating tasks by performing buffer management, message synchronization and address translation directly in hardware. The integration of these functions into hardware results in transfer latencies upto 30% shorter than state of the art MPI derivatives. The overheads for communication with Pronto in an 18-core processor array are under 5% for 64-word burst transfers, and less than 0.5% of total execution time using workloads such as the JPEG decoder and FIR filter. Furthermore, this paper also studies the effect of task mapping and interconnect traffic on the predictability of data block arrival times, and provides insight on where interconnect contention can be tolerated.

Keywords: multiprocessing, network-on-chip, message passing

1 Introduction

The evolution of microprocessors from single-core towards the present-day many-core is a consequence of the improved integration density achievable through modern semiconductor processes. The raw processing power of many-core processors is a key element in supporting a number of computationally heavy applications, especially in the multimedia domain. Efficiently translating this raw processing power into actual execution performance remains a challenge, hinged on the underlying hardware and software architectures.

The *dataflow* model is an effective means of harnessing the processing power of many-core arrays [6]. Applications developed using the model are described as a set of communicating tasks, with well defined input and output dependencies. Communicating tasks run asynchronously on separate processing elements and exchange data over point-to-point links. Tasks fire once their inputs become valid, and may further be synchronized using explicit barriers. Message passing is a popular

paradigm used in dataflow machines for communication between concurrently executing tasks. Since data is exchanged through explicit messages between distributed memories, unlike shared memory architectures, no cache coherence mechanism is required. While this is widely believed to be more scalable a solution for future multiprocessors, the latency of message transfers seriously limits the performance levels that can be achieved with the model.

Existing message passing implementations rely largely on feature-rich software libraries to manage the transfer of messages between *processing elements (PE)*. Thus, in addition to specifying what data must be moved between executing tasks, the programmer must also manage the actual transfer and any corresponding resource reservations. This is detrimental for two reasons. Firstly, it results in communication related tasks being managed through the processor, thus increasing execution time as well as communication latency. Secondly, it results in the implementation aspects of the underlying message passing communications architecture to be exposed to the programmer, thereby increasing complexity.

In this paper we present *Pronto*, a low overhead message passing system for many-core processors. Data transfers with Pronto are initiated using a compact set of simple yet highly effective functions that provide a layer of abstraction separating the programmer's view of inter-task communication, and its actual implementation in the underlying hardware architecture. Operations such as address translation, synchronization of transfers and resource management are handled entirely in hardware, thus simplifying the programming model and minimizing the time spent by processing elements in executing non-task related operations. The significant contributions of this paper are:

- A low-overhead message passing system that achieves atleast 30% lower end-to-end transfer latencies than state-of-the-art *Direct Memory Access (DMA)* based schemes, with an average setup overhead of under 5% for a 64 word burst, using reservation based message flow control.
- Scalable throughput and execution speedup on a many-core accelerator with the Pronto message passing system using real *JPEG* decoder and *FIR* filter workloads.
- Insight into the effects of task mapping and extraneous interconnect traffic flows on the arrival time jitter experienced in workload outputs. Significantly, this paper highlights where interconnect contention can be tolerated, and where it results in arrival time variations.

This paper is organized as follows: Section 2 provides an overview of related many-core array processors and message passing schemes. Section 3 describes the NagaM many-core accelerator within which the Pronto message passing system is tested. Section 4 explains the architecture of Pronto, and its automated buffer management, address translation and ordering of messages. The Pronto Application Programming Interface (API) is also introduced in Section 4. Both Pronto and NagaM are evaluated in Section 5 to determine end-to-end message transfer latency, application performance and the impact of extraneous interconnect traffic on output jitter. Concluding remarks for the paper are provided in Section 6.

2 Related Work and Motivation

A number of many-core processors, both in academia as well as the industry, implement message passing for inter-task communication. For instance, the 430-core *picoArray* uses basic message passing *put* and *get* functions to transfer data between concurrently executing tasks. During compilation, tasks are mapped onto cores and their communication flows converted into interconnect schedules. Since interconnect arbitration and resource reservations are performed at compile-time, communications do not incur any additional latency penalties related to these operations are runtime. The dataflow based *Ambric Massively Parallel Processor Array* [4] implements a similar methodology although with a hierarchical interconnect structure. The *Intel SCC* [7] on the other hand performs all required reservations at runtime rather than statically. Message passing is implemented through a global shared address space accessible through each PEs *Message Passing Buffer (MPB)*. Tasks executing on PEs share data through virtual connections established by dynamically allocating common memory objects within this space, using functions from the *RCCE* library. Synchronization,

ordering of messages and shared accesses must be managed through a programmer-enforced protocol in software.

Apart from these implementations, there also exist individual message passing proposals based on the *MPI* standard [1] often with specific objectives. For instance, *QoS-ocMPI* adds *Quality of Service (QoS)* support into a subset of MPI functions, specifically for NOC based multiprocessors [5], thus allowing critical transfers to occur through a reserved channel, i.e. *Guaranteed Throughput (GT)*. Another proposal, *TMD-MPI*, adapts MPI towards supporting message passing between processors across multiple *Field Programmable Gate Arrays (FPGA)*. It essentially abstracts the complexity of inter-chip communication, instead providing the programmer with a homogeneous view of the system. Despite their merits, these proposals are largely based on the original MPI standard, which itself is intended for large distributed memory systems. This objective of the standard reflects in the overheads incurred due to its use in resource constrained multi-/many-core processors. Psota and Agarwal noted this in their proposal *rMPI*, indicating the need for a simple message passing *Application Programming Interface (API)* with a small memory footprint to replace MPI in chip multiprocessors [9].

The drawbacks of heavy software libraries reflect primarily in the latency of data transfers. Proposals without static scheduling and resource reservations often require the MPB and synchronization of data transfers to be explicitly managed by the programmer. These operations are performed through functions of the software library executed on the PE. Consequently, the latency incurred to setup and manage transfers is higher than if the same were managed in hardware. Therefore, by removing the need for explicit management of the MPB and synchronization of data transfers through function calls, the latency of transfers could be greatly reduced. Furthermore, this would also serve to abstract the implementation of the message passing system from the programmer, and simplify the semantics of inter-task communication.

3 NagaM Many-Core Accelerator

The NagaM project aims to develop a many-core array processor to accelerate dataflow workloads, especially in the multimedia and signal processing domains. This will eventually be integrated into a cache-based shared memory array processor as a special purpose programmable accelerator. NagaM uses ρ-VEX *Very Long Instruction Word (VLIW)* cores [11] as PEs, which provide a performance benefit by exploiting inherent *Instruction Level Parallelism (ILP)* within executing tasks. PEs are placed within tiles containing private data and instruction memories, and a message passing communication interface. Tiles are connected over an *R3* router based mesh-topology *Network-on-Chip (NOC)* [10]. The *R3* architecture enables vertical scaling of the mesh, thus allowing NagaM to be implemented as a 3D stacked array. The best-effort nature of the R3 however implies that transfer latencies will be affected by extraneous traffic and consequent congestion in the interconnect. The

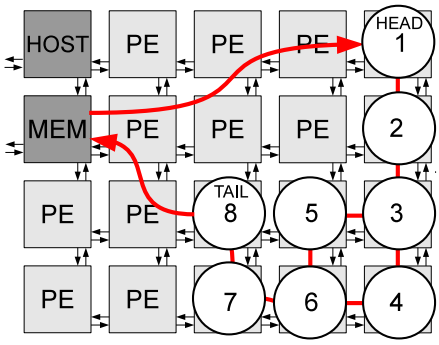


Figure 1: NagaM accelerator with host processor (HOST), and dual ported memory buffer (MEM). Only the Head and Tail tasks of the mapped task graph read from and write to the memory buffer respectively.

actual effect of extraneous traffic on latencies is examined later in this paper.

NagaM is intended to accelerate dataflow workloads specified as *task graphs* or *Kahn Process Networks (KPN)*. Tasks are spawned and pinned onto PEs according to the task graph, with communicating nodes mapped as close to one another as possible, by a runtime mapper at the host processor. Each task executes asynchronously on a ρ-VEX PE upon its input data becoming ready, and produces data that similarly triggers the next task in the process network. Fast dual-ported memories are placed along the periphery of the array to store the input data to, and output data from the head and tail of the task graph respectively. These essentially serve as data IO for the accelerator. In this paper, we assume only a single dual-ported memory in the array for simplicity. The NagaM accelerator is illustrated in Figure 1.

4 The Pronto Message Passing System

The performance gains of many-cores over sequential implementations are quickly lost as communication overheads approach task execution times [9]. In order to maximize throughput of the many-core array, it is important that message transfer latencies be kept low. Message transfers managed directly in software require processor intervention for data movement and synchronization. By implementing these functions in hardware, PEs are released from having to explicitly oversee data transfers, thereby allowing them to perform useful work instead. Pronto uses a DMA engine based message passing system for data transfers. Data blocks are moved between tile-local memories using hardware managed *Message Passing Buffers (MPB)* over the R3 network-on-chip interconnect. Figure 2 illustrates the architecture of a NagaM PE tile with the Pronto message passing interface.

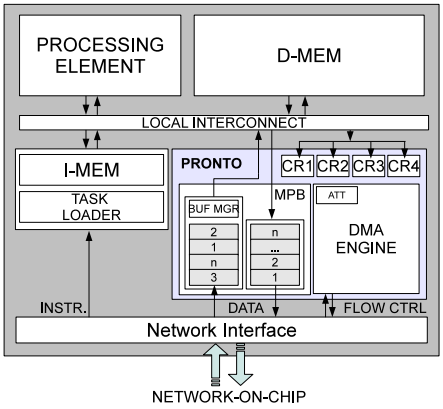


Figure 2: NagaM tile containing a ρ-VEX processing element, local memories, Pronto message passing interface and a network interface

4.1 Pronto API

Executing tasks communicate through calls to four simple message passing functions of the Pronto API, as listed in Table 1. These functions are essentially shells that set Pronto's hardware registers with the parameters of the message transfer. In contrast to the heavy send and receive primitives of existing message passing libraries, our API's functions are extremely light-weight, consisting only of a few writes to memory mapped control registers.

The *MP_send* and *MP_receive* functions are always called in pairs between communicating tasks, with the calls specifying only the size of the message, its location in the tile's local memory, and the sender/recipient's task ID. This provides a high level of abstraction, hiding details such as the actual physical PE onto which tasks are mapped. Each argument of the function calls maps to a particular control register of the Pronto interface, as listed in Table 2. The Pronto architecture allows programmers to extend the software API by defining multiple message types through the *CR4*

control register. During message transfers, the contents of this register are encoded into the message header (also known as *message envelope*), enabling control signaling between executing tasks.

Table 1: Pronto Message Passing API for NagaM

Function	Arguments
MP_send()	destination task id, length, local memory address of data
MP_receive()	source task id, length, local memory address for data
MP_mread()	local memory address for data, length, MEM address
MP_mwrite()	local memory address of data, length, MEM address

Table 2: Control Register Mappings

	CR1	**CR2**	**CR3**	**CR4**
MP_send()	Local memory address	Length	Dest. Task ID	Type (DAT)
MP_mwrite()	Local memory address	Length	MEM address	Type (MWR)
MP_receive()	Source Task ID	Length	Local memory address	Type (DAT)
MP_mread()	MEM address	Length	Local memory address	Type (MRD)

4.2 Hardware Architecture

The control registers together with the software API act as an interface between the executing application code and the Pronto hardware. Rather than actually performing the transfer through software, the message passing functions of our API only configure Pronto's control registers to initiate transfers between communicating tasks. The actual transfer is performed and managed by the hardware architecture itself. The following sections examine Pronto's management of the MPB, flow control and synchronization of messages, and its abstraction of physical addressing from the programmer.

4.2.1 Address Translation

As previously mentioned, the *MP_send* and *MP_receive* functions specify message transfers using task IDs of the recipient and source respectively, instead of their physical PE addresses. This is enabled by a per-tile *Address Translation Table (ATT)* programmed during task mapping, which translates programmer specified task IDs into the physical network addresses of the corresponding PEs. Consequently, the communication semantics for Pronto completely abstract details such as the physical address of PEs, and allow all inter-task communications to be specified at the task level itself. In addition to reducing the complexity of programming using message passing, this abstraction also permits task mappings to be adapted at runtime without requiring the software to be recompiled since physical addresses of PEs are not specified anywhere in the code.

4.2.2 Buffer Management

Before any data can actually be transmitted, it is essential for the sending node's message passing interface to determine whether sufficient free space exists in the downstream MPB. This is achieved through the use of a *message envelope* containing the source node's physical address, the amount of MPB space requested and the type of the message. Envelopes are handled at the downstream node on a first-come-first-served basis, with accepted envelopes resulting in the MPB reserving the requested chunk of memory for the impending message. The *buffer manager* actively tracks the utilization of the MPB through a status table, as shown in Figure 3. Upon arrival of each message, the buffer manager translates the source node address into its corresponding task ID, and places this information together with the MPB memory address at which the message is located into a free tuple of the status table in a circular FIFO-like manner. A pointer indicates the oldest waiting

message entry in the table, illustrated as an emboldened tuple in Figure 3. A successful reservation results in an acknowledgement to the upstream node indicating that the transfer may commence. In the event of insufficient MPB space, the corresponding envelope is buffered until the requested space becomes available. Therefore, no negative acknowledgements are returned, preventing repeated envelope transmissions from the stalled sender. Since only a single envelope is required per message regardless of its size, the overhead it poses remains fixed, and is quickly amortized during burst transfers.

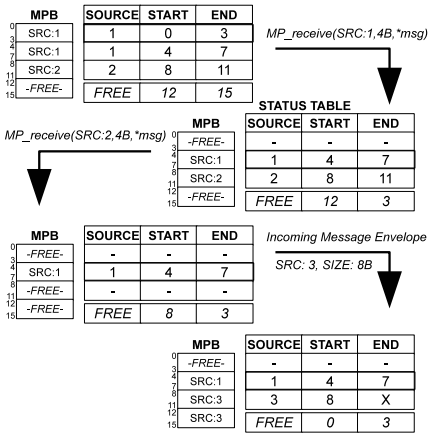


Figure 3: Illustration of buffer management and message ordering in the Message Passing Buffer (MPB)

Envelopes are generated automatically once an *MP_send()* call moves a complete block of data into the MPB. Therefore, destination MPB reservations are handled automatically by the DMA engine rather than explicitly by the programmer. The motivation for using a message envelope is two fold:

1. The NOC used in the NagaM enforces a protocol allowing for a maximum payload of 64B (16 words) per packet. Larger payloads are split into multiple packets, each of which is arbitrated separately by the R3 router's round robin arbiter. Multiple tasks communicating concurrently with a downstream task would result in the latter's MPB being inundated with only parts of messages, necessitating a buffer of a larger capacity. On the other hand, the use of message envelopes and the reservation based message flow control system ensures that received messages can always be stored as a whole, and that transfers commence only upon reservation of sufficient storage in the MPB. Furthermore, the mechanism simplifies buffer management by allocating memory on a per-message basis rather than per-source.

2. The message envelope and reservation based message flow control further ensure that packets belonging to messages in flight do not end up blocked in router FIFOs due to a full downstream MPB. Given the NOC's best effort nature, this would lead to blocked links, and give rise to the possibility of network deadlocks due to the absence of time-outs and packet dropping in the R3 router. Our mechanism therefore separates flow-control and buffering for the message passing system from that of the NOC.

Multiple requests from different upstream nodes to a single MPB are handled sequentially, although once accepted, transfers may proceed concurrently. This is possible since the buffer manager allocates disjoint blocks of memory to each transfer, allowing received words to be placed in their appropriate MPB memory locations simply based on their source. The *MP_send* function does not specify the destination memory address for any transfer. Where this data is placed in the receiving node's local memory is determined by the arguments of the *MP_receive* call at the destination, essentially simplifying the semantics of data movement in the system. Needless to say, each node may only hold one request (both active and pending) to any particular downstream node at any given

point in time. Furthermore, words constituting a message must form a contiguous block in memory, i.e. they must be located at sequential memory addresses.

4.2.3 Ordering of Messages at Destination

The buffer manager preserves the entry order of incoming data blocks using the status table, ensuring that the oldest received block is popped from the buffer when requested by the executing task. In the case of concurrent tasks with uneven loads where the upstream task generates multiple data blocks during a single run of the downstream task, this mechanism guarantees that blocks are consumed in the same order as they are generated. Received blocks are moved into the local data memory of the PE once the *MP_receive* function with the corresponding source task ID is called.

In case a task produces more than one type of output data, a programmer defined protocol must be enforced to order the *MP_send* and corresponding *MP_receive* calls. This is because the functions do not include any details of the destination memory address for the remote task, thus making it hard to determine which data block the message contains. Given the nature of dataflow based programs and their definite input-output dependencies, this ordering is trivial to enforce. Therefore, if a task generates two outputs and sends them in one order, the downstream task must call *MP_receive* in this exact same order. This is illustrated in Figure 3 which shows the MPB of a destination node receiving messages from a number of nodes, even before older messages already waiting in the buffer are consumed. When *MP_receive* is called, the waiting messages from the requested source task ID are returned to the PE in the order in which they arrived.

The *MP_receive* function is blocking, and hence stalls the PE until data from the specified source is received by the MPB. The *MP_send* function, on the other hand, is non-blocking except for when the local MPB's output buffer is full in addition to the downstream MPB's input buffer. In this case, execution is stalled by clock-gating the local PE. Proper load-balancing of tasks to ensure that they incur similar execution times minimizes the occurrence of such buffer full/empty stalls. We illustrate this in the following section with the *JPEG* decoder.

5 Experimental Evaluation

We evaluate Pronto using a cycle-accurate HDL based simulation model of NagaM. The model uses 18 ρ-VEX processing elements connected over a 4x5 mesh topology network, with a single data memory buffer from which task graphs fetch their input data, and write their output to. Although a practical hardware implementation would place limitations on the size of this buffer, for the purpose of our simulations, we impose no such constraints. This does not affect the validity of the presented results since the evaluation focuses primarily on the message passing system within the array, and its consequent impact on application performance. The MPB is sized at 512B (128-words) for the input, and 256B (64-words) for the output.

Three dataflow workloads are used to analyze the performance impact and scalability of Pronto: *JPEG* decoder, *Moving Average FIR* filter and a custom test workload. The *JPEG* decoder from the *MiBench* benchmark suite [3] implements the decoding of JPEG images into the Bitmap format. The conversion process involves three stages, namely *Huffman decoding*, *Inverse Discrete Cosine Transform (IDCT)* and *colour conversion*. The original sequential implementation of the JPEG decoder from the benchmark suite was parallelized manually by converting each of its three stages into concurrently executable tasks, with the Pronto API functions used for data transfer. After initial experiments, a more effective four-stage JPEG decoder was developed to overcome inefficiencies noted in our three-stage implementation. The two versions are identified as JPEG-3 and JPEG-4, with the suffix signifying the number of concurrent stages in their task graphs. The input data set for these workload consists of a 512x512 pixel JPEG encoded image. The *Moving Average FIR filter* workload is used in signal processing applications to remove unwanted noise in signals. The filter essentially implements the equation listed in (1), where x and y represent the input and output signals respectively. The nature of this algorithm allows it to be partitioned into multiple concurrent tasks, each with a similar computational load. However, partitioning may only be beneficial upto a certain point, after which communication latencies become comparable to the execution time of tasks

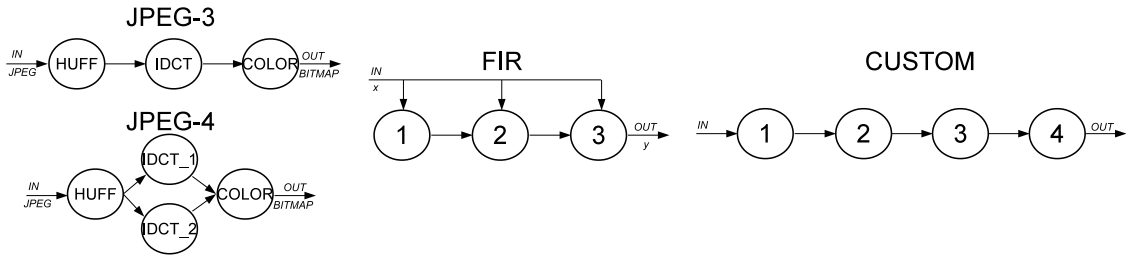


Figure 4: Task graphs for the *JPEG-3*, *JPEG-4*, *FIR* and *CUSTOM* workloads.

themselves, thus limiting further performance gains. The *Custom* application represents an ideal dataflow workload with identical concurrent tasks. Such partitioning can be expected to minimize execution stalls. Figure 4 illustrates the task graphs for the JPEG-3, JPEG-4, Moving Average FIR filter and Custom workloads.

$$y[i] = \frac{1}{N}x[i] + \frac{1}{N}x[i-1] + \frac{1}{N}x[i-2] + ... + \frac{1}{N}x[i-N-1] \quad (1)$$

The evaluation of Pronto consists of five separate experiments that determine:

1. End-to-end message transfer latency
2. Communication overheads
3. Application performance with Pronto
4. Impact of input dataset size
5. Impact of extraneous interconnect traffic on output jitter

The following subsections describe each of these experiments, and provide an overview of the obtained results.

5.1 End-to-end Message Transfer Latency

The performance of Pronto is first evaluated in terms of its message transfer latency per hop, i.e. the latency incurred in transferring a message between two adjacent nodes. For this, two tasks are pinned onto neighbouring PEs in the NagaM array. The first task generates a burst of 64 data words and transfers these using an *MP_send* call to the second task which then receives the burst using *MP_receive*. In order to accurately estimate the transfer latency, these measurements are performed without any extraneous interconnect traffic (zero network load). The obtained latencies are listed in Table 3. The same table also includes the transfer latency for similar sized bursts from literature - Francesco's Shared Queue and Scratch Queue DMA [2], and the MPI derivative for multiprocessors with on-chip interconnects - ocMPI [5]. Pronto is observed to have a transfer latency 30% lower than the closest distributed memory based proposal, Scratch Queue DMA [2]. Note that the use of a larger burst size of 256 words works in favour of ocMPI since the overheads of transfer setup are better amortized by large bursts. Despite this, the overall per word transfer latency of ocMPI is observed to be significantly larger than that of Pronto, indicating the higher transfer overheads of MPI-based systems.

Table 3: Comparison of average transfer latency per word

	Latency per word (cycles)	Burst Size (words)
ocMPI	32.9	256
[2]-Shared Queue	20	64
[2]-Scratch Queue DMA	9	64
Pronto	**6.48**	64

The transfer of the message envelope and the downstream node's acknowledgement of buffer reservation impose a one-time latency overhead for each message. While message envelopes indicate the source node and quantum of MPB space required at the destination, the former is already included into the packet header according to the R3's protocol. Therefore, the message envelope in NagaM is a 2-flit packet consisting of the header and a single flit containing an integer value of the required MPB space. The envelope length remains the same regardless of message size. A 64 word message on the R3 NOC is sent in 4 packets, or 68 flits in total. A single message envelope and the corresponding downstream MPB acknowledgement result in 3 additional flits (2 for the envelope and 1 for the acknowledgement) being exchanged between the nodes. This constitutes an overhead of under 5% for a 64 word message.

5.2 Communication Overheads

In order to determine the transfer overhead for messages in terms of total execution time, we mapped single instances of workload task graphs onto the array with zero network load. Figure 5 illustrates the fraction of total execution time spent in execution, stalls due to a full/empty MPB and in communication across different workloads. The same figure also indicates the overhead imposed by message envelopes as a fraction of total execution time. As previously mentioned, only one envelope and its corresponding acknowledgement are generated for each message transfer. Consequently, the number of envelopes and acknowledgements exchanged over the interconnect depends only on the number of messages transferred, and not their size. In general, the transfer overhead of the message envelopes constitutes less than 0.5% of the total execution time across all workloads. The time spent stalled due to a full/empty buffer is primarily caused by imbalances between the tasks, and this can be reduced by precise partitioning and load-balancing. Frameworks such as Daedalus [8] enable such analysis and help in precise partitioning of workloads for high performance and scalability.

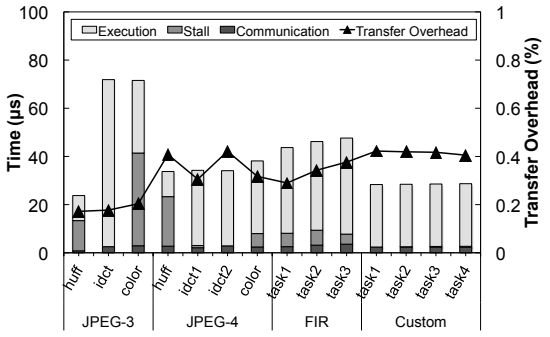


Figure 5: Breakdown of task execution as a fraction of its total execution time. The *transfer overhead* reflects the overhead imposed by message envelopes as a percentage of total execution time.

The consequences of inefficient partitioning in the case of the JPEG-3 workload are also illustrated in Figure 5. Initial runs of the workload on the NagaM array revealed an imbalance in the runtime of its three constituent stages. The IDCT stage in particular was observed to run close to six times as long as the Huffman decoding stage, resulting in repeated execution stalls for the latter. The IDCT stage was subsequently partitioned further into two concurrent tasks to address the imbalance in task loads, as shown earlier in Figure 4. The resulting implementation reduced buffer-related execution stalls by 38% and reduced execution time by 45% as compared to the three-stage version. Although the number of message transfers in both implementations is identical, Pronto's transfer overhead appears higher in the case of JPEG-4 due to the reduced execution time.

5.3 Application performance with Pronto

Execution performance can be improved in two ways - by increasing the number of concurrently executing tasks through fine grained partitioning, and by increasing the number of instances of the task graph executing in parallel. We observe from Figure 6 that the former does not always yield significant returns. In the case of the FIR filter for instance, the speedup obtained through fine-grained partitioning tends to flatten out beyond 6 tasks for a 6400 sample input size as a consequence of the reduction in computational load per task to a level where communication latencies become significant.

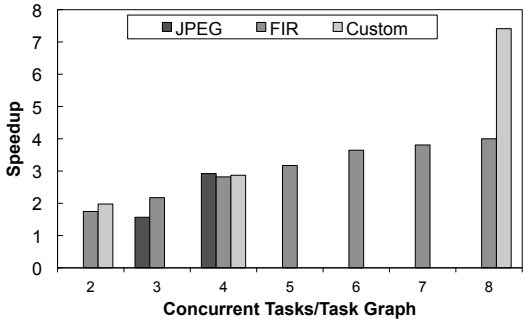
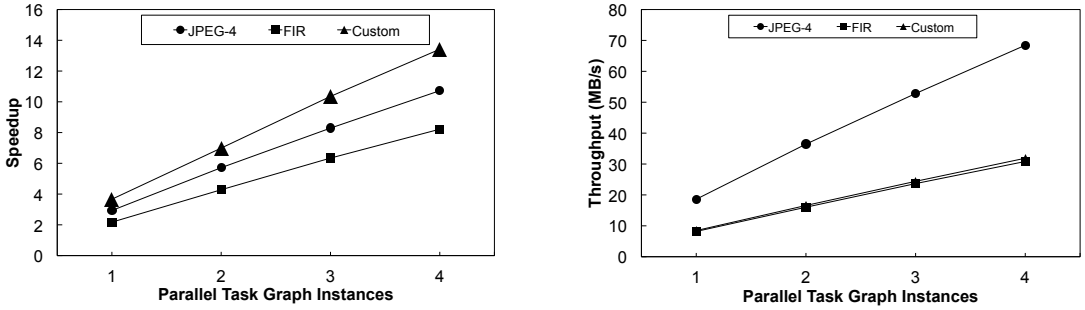


Figure 6: Performance improvements obtained with fine-grained partitioning

Instantiation of multiple instances of the task graph on the other hand allows for exploitation of data-level parallelism, thus achieving greater speedup and higher throughput. Figure 7(a) reports the execution speedup for the workloads with a varying number of parallel instances of the task graph, over sequential execution on the host PE. A linear improvement in speedup is observed as the number of parallel instances executing on the NagaM array is increased. The corresponding throughputs for these workloads considering a 200MHz clock frequency are reported in Figure 7(b). Note that since the FIR workload generates output data blocks of size 376B as against 256B for the Custom workload, the two yield very similar throughputs despite their significantly different speedups. In comparison, the JPEG decoder generates larger output data blocks of size 768B, thus explaining its prominently higher throughput.



(a) (b)

Figure 7: (a) Execution speedup relative to sequential execution on the host PE (sequential exection times - *JPEG-4*: 105.3μs, *FIR*: 115.8μs and *Custom*: 106.1μs) (b) Throughput at 200MHz

5.4 Impact of input dataset size

The runtime of all workloads is influenced by the size of their input datasets. When the number of concurrent tasks per task graph as well as the number of parallel instances of the task graph are fixed, the runtime can be expected to increase as the input dataset size is increased. A longer runtime can however be beneficial as it tends to amortize the impact of communication and configuration overheads. Most significantly, a longer runtime softens the impact of ATT configuration that occurs when the task graph is spawned on the NagaM array. To estimate the performance impact of such overheads, we varied the input dataset size for workloads, effectively changing their runtime. Figure 8 reports the speedup obtained over sequential execution across different dataset sizes.

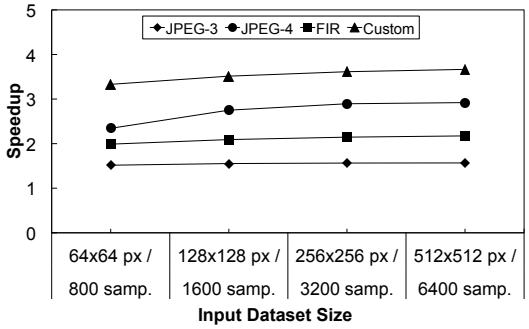


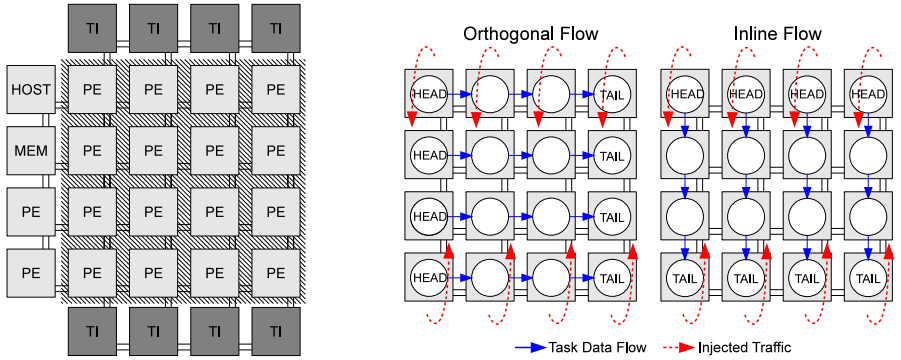
Figure 8: Speedup with varied input dataset sizes

The figure indicates small improvements in speedup with increasing dataset size. Note that with the 64x64/800 samples dataset, the overall execution time for most workloads is low enough for the the ATT configuration operation to constitute a moderate overhead. With larger dataset sizes on the other hand, speedup improvements are less pronounced, since the configuration operation no longer forms an appreciable fraction of total execution time. These results suggest that overheads of Pronto within the NagaM architecture are sufficiently low so as to yield similar speedups across a range of input dataset sizes.

5.5 Impact of extraneous interconnect traffic on output jitter

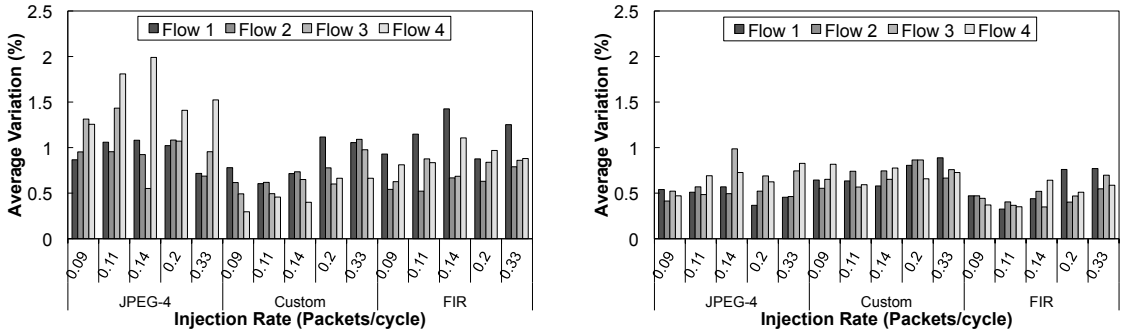
Given the best-effort nature of NagaM's R3 NOC, it is prudent to evaluate the impact of interconnect traffic on the variation in arrival rate of data blocks. For this purpose a set of *Traffic Injectors (TI)* are placed at the North and South edges of the array. Injectors at the northern edge emulate cache miss and write-back requests directed towards those on the southern edge. These requests vary in size from 4B (cache miss) to 64B (cache line write-back) at various injection rates, emulating extremely pessimistic miss rates. The injectors on the southern edge respond with appropriately sized packets to the requesting injector, as illustrated in Figure 9. Multiple parallel instances of a task graph are mapped onto PEs of the array, with task data blocks moving in a direction orthogonal to the injected synthetic traffic. The output jitter is measured as the variation from expected arrival time for data blocks at the memory buffer *(MEM)* averaged over the entire execution of the workload for a given input dataset.

The average variation in arrival time for workloads at different injection rates for the case when task data blocks and synthetic traffic flow in orthogonal directions is reported in Figure 10(a). In order to provide a comparison, we adapted the traffic injectors and task mapping such that the injected traffic and task data blocks flow inline with one another as shown in Figure 9(b). The average variation across workloads and injection rates is observed to drop from the earlier peak of 2% to under 1% with this new mapping. This can be seen in Figure 10(b). Rather than the injection rate, it is the relative direction of interconnect traffic that significantly influences arrival time variations for data blocks.



(a) (b)

Figure 9: (a) NagaM array with traffic injectors simulating cache traffic. Tasks are mapped to PEs within the highlighted region of the array. (b) Illustration of relative directions of task data and injected traffic - orthogonal and inline.



(a) (b)

Figure 10: Average arrival time variation for: (a) orthogonal flows and (b) inline flows

In the first case, the XY routing algorithm of the network results in increased contention in the North-South network links on account of their utilization by both injected traffic as well as data blocks moving to and from the memory buffer. As a consequence, the head stages of all graph instances remain stalled until their requested data blocks arrive, resulting in accumulation of the delay at all subsequent stages. In the second case, due to the location of the head task for each task graph instance, input data blocks are routed in a direction orthogonal to the injected traffic. Consequently, input data blocks encounter little contention in their path, and therefore do not delay task execution. Output data blocks from the tail moving towards the memory buffer similarly incur minimal delays. Contention at the memory buffer itself, on the other hand, contributes to the variations observed for workloads in Figure 10(b).

Although interconnect traffic affects the transfer latencies for intermediate data blocks moving between stages of a task graph, the actual impact is minimized due to the relatively smaller hop count for such transfers as compared to those directed at the memory buffer. This is a consequence of the mapping algorithm's placement of communicating tasks on neighbouring cores, often resulting in single hop transfers. Even in the case of JPEG-4 where the *IDCT_1* and *IDCT_2* tasks communicate with the colour conversion and Huffman stages over a multi-hop path, interconnect contention is seen to have a smaller impact on output jitter. Interestingly, it is also observed that due to imbalances in execution times of workload tasks, arrival time variations are evened out by buffer full stalls, resulting in decreased jitter.

These results indicate that contention is better tolerated by data blocks moving between intermediate stages of the task graph, than by those moving between the head/tail tasks and the memory

buffer. Contention in the path of data blocks moving to and from the memory buffer are observed to cause high arrival time variations. Mapping strategies that result in such critical data flows following orthogonal paths to extraneous interconnect traffic can significantly reduce output jitter, as observed in Figure 10(b).

6 Conclusion

This paper presented the Pronto low overhead message passing system for many-core processors. By implementing functions to manage message transfer, synchronization, address translation and buffer management directly in hardware, we reduced message transfer latencies by upto 30% in comparison to state-of-art DMA-based proposals. The reservation based message flow control system implemented through the use of message envelopes imposed an overhead of under 5% for 64 word burst transfers, constituting less than 0.5% of the total execution time of workloads such as the JPEG decoder and FIR filter. In addition to the low latency, the presented system simplified the semantics of data movement by abstracting the implementation details of the communications architecture from the programmer, enabling data transfers to be specified at the task level. The speedup over sequential execution obtained with Pronto in an 18-core NagaM array was found to scale linearly with the number of parallel task graph instances, with similar performance across a range of input dataset sizes. An analysis of the impact of the relative direction of task data flows and extraneous interconnect traffic on the arrival time for output data revealed that blocks moving between the memory buffer and the head/tail of task graphs resulted in the highest arrival time variations when delayed due to contention. These effects were found to be mitigated by adapting the task mapping to ensure that such performance critical data move in a direction orthogonal to extraneous interconnect traffic.

7 Acknowledgements

This work was supported in part by the CATRENE programme under the Computing Fabric for High Performance Applications (COBRA) project (CA104).

References

[1] Message P Forum. Mpi: A message-passing interface standard. Technical report, 1994.

[2] Poletti Francesco, Poggiali Antonio, and Paul Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 736–741, 2005.

[3] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[4] B. Hutchings, B. Nelson, S. West, and R. Curtis. Optical flow on the ambric massively parallel processor array (mppa). In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, pages 141–148, 2009.

[5] J Joven, F Angiolini, D Castells-Rufas, and J Carrabina. Qos-ocmpi: Qos-aware on-chip message passing library for noc. In *Workshop on Programming Models for Emerging Architectures*, 2010.

[6] Edward A. Lee and Thomas Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[7] Timothy G. Mattson et al. The 48-core scc processor: the programmer's view. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[8] H. Nikolov et al. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the Design Automation Conference*, pages 574–579, 2008.

[9] James Psota and Anant Agarwal. rmpi: message passing on multicore processors with on-chip interconnect. In *Proceedings of the International Conference on High performance embedded architectures and compilers*, pages 22–37, 2008.

[10] S. S. Kumar and R. Van Leuken. A 3d network-on-chip for stacked-die transactional chip multiprocessors using through silicon vias. In *Proceedings of the International Conference on the Design Technology of Integrated Systems in Nanoscale Era*, pages 1–6, 2011.

[11] S. Wong, T. van As, and G. Brown. p-vex: A reconfigurable and extensible softcore vliw processor. In *Proceedings of the International Conference on Field Programmable Technology*, pages 369–372, 2008.