Hardware Implementation of Social-Insect-Inspired Adaptive Many-Core Task Allocation

Matthew Rowlings, Andy M. Tyrrell, Martin A. Trefzer
Intelligent Systems Group, Department of Electronics, University of York, York, YO10 5DD, UK
Email: mr589@york.ac.uk, andy.tyrrell@york.ac.uk, martin.trefzer@york.ac.uk

***Abstract*—As many-core systems scale into the hundreds of nodes, design-space exploration becomes an infeasible approach due to the many parameters that need to be optimised to produce a design that fits both the application's requirements and the device constraints. When recent hardware platform problems such as Dark Silicon, device variation and post-manufacture failures are also considered, then a classical design methodology is even harder to achieve. Instead, systems will need to continuously adapt to their operating environment and device parameters. Our previous work has shown that task allocation based on social insect colonies is an effective and efficient approach to tackling the problem of task-to-node mapping in an autonomous and adaptive fashion. In this paper we verify the approach in hardware by implementing the bio-inspired task allocation in a many-core consisting of 100 Microblaze processing nodes connected via a Network on Chip (NoC) with the distributed intelligence embedded within the NoC routers. We show that this adaptive model can be implemented in hardware with a very small hardware overhead of 3% that scales linearly despite the huge number of processing cores on the FPGA chip. Thus this work shows that social-insect inspiration is a effective way of implementing a hardware "nervous system" that will allow systems to autonomously tackle the problems that ever smaller device implementation technologies bring with them.**

I. Introduction

As the advancement of digital implementation technology starts to embrace the limitations enforced by the breakdown of Dennard Scaling [1], we see a shift towards many-core systems as a feasible solution to the problem of avoiding *Dark Silicon*[2]. The many-core paradigm divides the computing task between many processing cores across the chip, where typically each core will have varying degrees of hardware specialism and performance characteristics. These cores are interconnected within a single device using a *Network on Chip (NoC)* [3] [4]; an interconnection scheme based on conventional networking where routers and channels are provided for communication between nodes. Many node topologies, interconnect options and constraint optimisations are possible [5], giving the hardware engineer a powerful platform for implementing systems that could be made dark silicon tolerant. This flexibility comes with its own engineering caveats however: the large number of parameters will require problem and system analysis to ensure that systems implemented within NoCs fit their requirements and may necessitate the need for heuristical approaches such as [6][7][8] to optimise the design space; this is especially relevant when we consider the extra thermal and power constraints imposed on the design by dark silicon. This approach also suffers as the analysis is done at design time and so cannot be adapted should the operating conditions or properties of the chip change during operation. However such flexibility is a key requirement for supporting future many-core system design paradigms such as dynamic task allocation, in field self-repair and autonomous online optimisation [9].

Large social insect colonies also require a wide range of important tasks to be undertaken to build and maintain the colony and in most nests there are many thousands of workers available to offer their assistance to ensure the expansion and survival of the colony. However, there is a crucial equilibrium between the number of workers performing each task that must not only be maintained but must also continuously adapt to sudden changes in environment and colony need. What is most fascinating is that social insects can sustain this balance without any centralised control and with colony members that have relatively little intelligence when considered on their own. Due to this simplicity and evident scalability it would seem that social insects have evolved an interesting scalable approach to task allocation that could be applied to very large many-core systems.

We have previously shown that social-insect inspired task allocation can achieve adaptive, self-optimising task allocation in a many-core systems [10], as well as dynamic routing [11] and fault tolerance [12]. This paper takes the simulated model from this prior work and presents a hardware implementation of the task allocation model, including some scalability verification by scaling the many-core up to 100 processing cores from 36. Firstly the details of the bio-inspiration is outlined, followed by how this is implemented within the Network on Chip. Finally the results of some task allocation experiments are discussed and a comparison between the hardware implementation and the simulation model is considered.

II. Social-Insect Inspired Task Allocation

A. Many-Core Task Mapping

Deciding which task a node should be doing in a many-core system is a fundamental part of the multi-objective design space exploration involved in many-core system design. The node-to-task mapping will affect many key constraints of the system design, even for homogeneous many-cores. For example, a poor mapping may result in excessive communication overhead through longer communication paths between nodes in the data processing flow, or increased thermal load if busy nodes are clustered together and even limited system

throughput if not enough nodes are assigned to tasks on the critical path. Thus an ideal task allocation needs to optimise topologies with both a geometrical and an application graph focus. This becomes an even harder problem once adaptation is supported within the task allocation model as changing the task of one node will have both an upstream and downstream effect on other nodes in the data-flow. If classical heuristical approaches are used in an adaptive context then huge number of different task mapping scenarios must be modelled, which is clearly not very scalable and would be a limiting factor as many-cores reach the size of hundreds and thousands of nodes.

B. Emerging Scalability

To tackle this scalability problem our approach *emerges* the task allocation mapping at runtime by taking inspiration from one of Nature's self-organising complex systems, the social insects, with a specific focus on ant colonies in this paper. Indeed there are many parallels to be made; each member of the colony has to decide what task it should be undertaking at any one point and, of particular interest to building highly scalable systems, *no methods of global organisation or coordination of work exist in the colony*. In order for a colony to survive its members must work together as there is a wide range of tasks that are required to be undertaken, ranging from tasks inside the nest such as feeding and rearing of the young brood, nest expansion and maintenance tasks to scouting for and retrieving food from outside of the nest. Many models of task allocation have been explored by biologists [13] and models vary between species depending on factors such as colony size and sociability, capabilities of individuals and the typical environment of the species habitat. In our previous work [10] we highlighted the suitability of Gordan's network-based models to the task allocation problem as it abstracts away from the underlying methodologies of interactions between members of a colony, instead building a model based on the patterns of interactions between members [14]. This is more easily and efficiently translated into hardware as our biological inspiration does not need to be "ant-correct" at each node, instead we monitor the properties of packets that a node encounters (priority, rate, destination) rather than any of the actual data contained within the packet.

C. Hardware Realisation

This allows us to implement a very simple threshold based decision model at each node, resulting in a dynamic network analogous to Gordon's models presented in [15]. Each node in the many-core contains a simple 5-port router (N, E, S, W and internal node) and hardware monitors are added to each input port to track what task each packet is destined for. When the number of times a task is observed exceeds a threshold then a task switch decision is made and communicated to the node, as shown in Figure 1. This maps very efficiently to digital hardware as this is essentially several comparators and counters. The circuit shown in Figure 2 is implemented within each router for each task and thus simplicity is fundamental to allow a scalable implementation. The task suggested from this intelligence is fed into the node's processing core that uses this information to inform its current processing task, in our case it immediately switches to the suggested task but there are many other ways that this information could be used. This architecture of pulses, counters and thresholds bears similarities to the spiking neural networks seen in the autonomic nervous systems of many animals. The crayfish for example uses a very simple neuronal circuit to decide whether it needs to escape and evade predators based on the spiking input of sensory hairs on the tail of the crayfish [16]. By adding further inputs and expanding this hardware-efficient decision circuitry, we can endeavour to eventually provide each router with an analogous nervous system that makes complex decisions regarding the local aspects of the many-core's network behaviour autonomously.

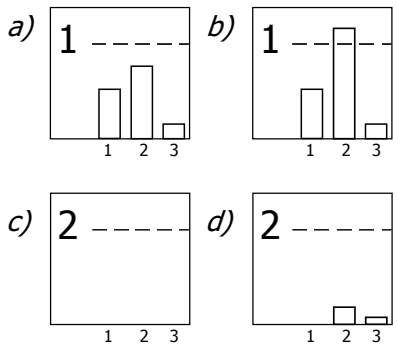

Fig. 1. Task switch decision algorithm for an example with three tasks. *a)* When a packet arrives at the router the router inspects what task the packet is destined for. The router then increments an internal counter of the destination task of the packet. *b)* If a task counter exceeds the *task switch threshold* then the decision is made to change (or maintain) to that task and the application node is informed. *c)* All of the counters are then reset *d)* The router then starts the task switch process again

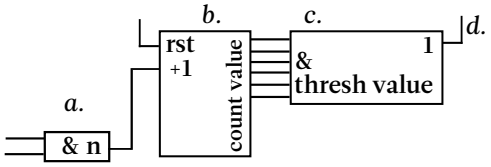

Fig. 2. The embedded hardware to initiate a task switch. First the task number is extracted from the packet header as the header passes through the router and compared to a task number (*n* in this case) (a). This is then used to increment a counter (b) of the number of packets of this task that has moved through the router, the value of which is compared to the task switch threshold (c). If the number exceeds this threshold then a flag is raised (d) which sets a status register connected to the node's processing core with the suggested task. *All* counters are then reset and this process starts again, thus if the network load changes later on in time then another task can pass the threshold and cause the task switch.

III. Network on Chip

A. NoC Design

A key interest of our investigation is the scalability of our task allocation scheme in real many-core systems. To represent the very high core count of many-core systems in the near future we have created a 10x10 many-core platform

on a Xilinx Virtex-6 LX760 FPGA. Each node in the many-core consists of a custom NoC router and a MicroBlaze Micro Controller System (MCS)[17]. The Microblaze MCS was chosen due to the general purpose nature of the processor, the C programming support and the simple to extend external IO bus. The NoC interface (I port on the router) is currently attached to this IO bus, as is the task suggestion from the intelligence but in the future hardware accelerators could also be added, allowing heterogeneity to be easily added to the NoC. The 100 processors are arranged in a grid, with an extra Microblaze attached to the N boundary input of the most north-west processor; this processor is purely for setting up the experiments and collecting results. The nodes currently simulate an application by writing and reading data packets to the NoC and waiting for a set period to simulate processing time.

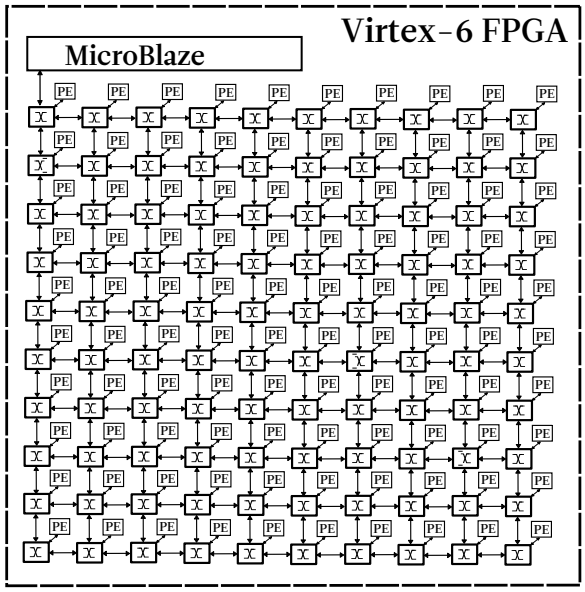

Fig. 3. Illustration of our 100 processing element (PE) many-core, showing the MicroBlaze responsible for managing the experiments and the single FPGA that the many-core is implemented within. Each PE is a MicroBlaze MCS microcontroller clocked at 100MHZ with 16KB of RAM.

Our custom NoC router is shown in Figure 4. It has five external ports, one to each cardinal neighbour and one to the associated processing node. There is also an internal configuration port that allows the nodes to change the router settings, including the routing tables and task thresholds. Indeed this port can be accessed by any node in the network with the correct packet, enabling some interesting interaction experiments in the future such as nodes disabling traffic being routed to them when they are constantly busy by directly changing their neighbours' routing tables. These ports are connected via a six-channel switch, allowing full duplex communication across all ports if such a traffic profile happens. There is also a round robin arbitrator that feeds a controller responsible for reading the routing tables and setting up the switch when a packet arrives. To keep the hardware resource requirements of each NoC router as small as possible, wormhole routing

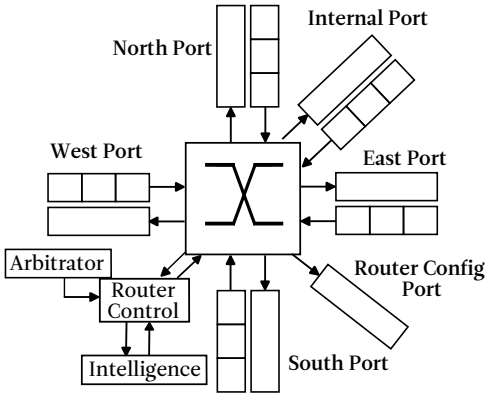

Fig. 4. Layout of the NoC router. There are five duplex routing ports and an internal configuration port. The switch supports a maximum of six data transfers simultaneously and is controlled by the controller module which is responsible for reading the routing tables and setting up the switch. The intelligence module is connected into this and currently comprises of three of the units shown in Figure 2. Each of the 5 input ports contains a three byte FIFO that allows the packet ID to be read before the packet is routed, allowing deadlock to be detected and handled by routing the packet down an alternative channel if the same ID is present on the desired output channel, as shown in Figure 5.

was used for sending data across the network, a deviation from our simulation model presented in [10]. This limits the amount of buffering required at each router and thus saves FPGA embedded memory blocks.

B. Deadlock Handling

Wormhole routing however is very susceptible to deadlock [18], especially given the dynamic nature of our system as each task switch may invalidate the routing tables. Therefore we need to detect and handle deadlocks whilst also still fulfilling our scalability requirements, meaning that we cannot use any techniques that rely on global analysis of the entire network, constrains the routing decisions we can make (such as restricting the turns that can be made at a certain node) or that use many hardware resources (such as the extra buffering required by virtual channels). Figure 5 illustrates the approach we have taken which aims to exploit the adaptivity of the many-core. Packets have a header that includes an unique, two byte ID. The input FIFOs on the router ports reads and stores this ID before passing the packet to be routed by the control logic. If this ID already exists on one of the other ports then the packet has looped around and would deadlock if routed the same way. Thus it is routed out on a different port and so shall take a different path through the network. This is repeated should the packet return as the ID is not cleared until the its tail End of Packet marker (EOP) passes through the deadlocked ports. This means that eventually the packet is (potentially incorrectly) routed to the internal port, ultimately relieving the deadlock but requiring the packet to be resent by the node that temporarily accepted the packet or for the node to perform a temporary task switch to the task of the deadlocked packet. This not only provides a decentralised, low overhead manner of handling deadlock but also provides

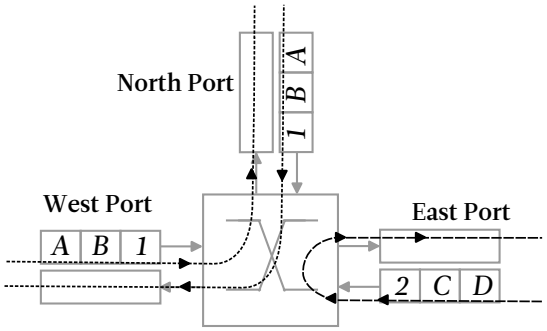

Fig. 5. Example of deadlock detection and handling, only three ports are short for clarity. There are two packets in the router, packet 1 is destined for a task 1 node and has a header of 1AB; packet 2 is for task 2 and its header is 2CD. Packet 1 has arrived at the West port and the routing table dictates that it is to be routed North. Packet 2 has arrived on the East port and the routing table says it should be routed back out on the East port. Now during packet 1's journey it ends up being routed back to this node and so arrives back on the North input. The routing table wants the packet the be routed North, but it is currently in use. This would lead to a deadlock as the packet cannot progress, however we see that the ID 1AB matches in both the North and West inputs and so we now automatically take the second option, routing out on the West port. This is free and so we route the packet that way and alleviate the deadlock. N.B. if the packet ID did not match then we would wait for the North port to be free to send.

TABLE I
HARDWARE RESOURCE REQUIREMENTS FOR THE 10X10 NOC

	Standard NoC	NoC with Intelligence	% Difference
Sequential Logic:	73,875	76,392	3.41%
Combinatorial Logic:	108,506	112,104	3.32%
Memory:	408	408	0%

us with several potential monitors and controls that a future router intelligence module could be attached into to.

C. Network Routing Setup

Finally, by allowing nodes to change their processing task any preset routing tables will be invalid as soon as a task switch happens. As we are not tackling the problem of updating the routing tables dynamically in this paper, we instead initialise the routing tables non-optimally and allow the network to adapt its task mapping to the non-optimal routing pattern. We have also re-introduced the concept of node "hunger" from our previous work. When a node is busy processing data it sets a flag that informs its immediate neighbours that it currently does not want to receive packets, this is now integrated with the deadlock avoidance mechanism to avoid routing in the direction of a busy neighbour and instead pick an alternative port by the same manner that the deadlock avoidance does.

IV. EXPERIMENTAL RESULTS AND COMPARISON

Using the hardware platform described in the previous section, we have repeated the experiments that we undertook in the simulation presented in [10]. As with these previous task allocation experiments, two application scenarios were used for the experiments: one representing a balanced application and the second an application bottleneck where the ratio of each tasks are not equal. The application graphs for these scenarios are shown in Figure 6 and 7. Packets from Task 1 nodes are generated at a fixed rate of one every 50ms, whilst Task 2 and 3 nodes do not send a packet out until they have received a packet with their task as the destination - this introduces causality into the model and is a more realistic processing stream. These and the other experiment parameters are summarised in Table II.

TABLE II
APPLICATION MODEL SETTINGS. THESE DETERMINE WHEN A PACKET IS GENERATED, AS EXPLAINED IN SECTION IV AND FOR EACH OF THE APPLICATION GRAPHS SHOWN IN FIGURE 6 AND 7

	Scenario 1			Scenario 2		
Task:	1	2	3	1	2	3
Ratio:	1	1	1	1	4	1
Rate:	50ms	0	0	50ms	0	0
Packets Required:	0	1	1	0	1	1
CPU Time:	1ms	1ms	1ms	1ms	5ms	1ms
Packet Size:	1KB	1KB	1KB	1KB	1KB	1KB

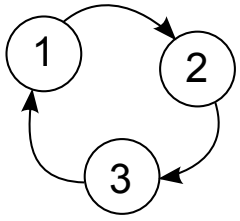

Fig. 6. Application graph for the first scenario. This represents a simple balanced processing application resulting in balanced traffic profile across the network, perturbed only by the network topology.

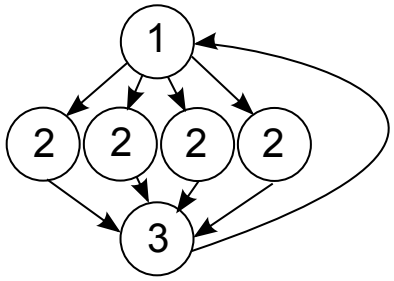

Fig. 7. Application graph for the second scenario. A data pipeline with a parallel stage is represented here whereby there are four times as many task two nodes as task one or three nodes. This can represent a typical many-core streaming application with a stage that is massively parallel, all data rates are kept the same as in the previous, balanced application graph shown in Figure 6 aside from CPU time for Task 2 nodes as shown in Table II. This scenario effectively increases the load on Task 2 nodes.

Figure 8 shows the distribution of average packet latencies over 100 runs of the first task scenario. The tasks are randomly allocated to nodes and this mapping changes between runs, but for the first two schemes we pre-load the routing tables with optimal routing patterns for each task. As task switching is not used in the first two schemes these routing tables remain valid for the entire experiment and the optimal pathways mean that no deadlock is present in the first case. The second case does

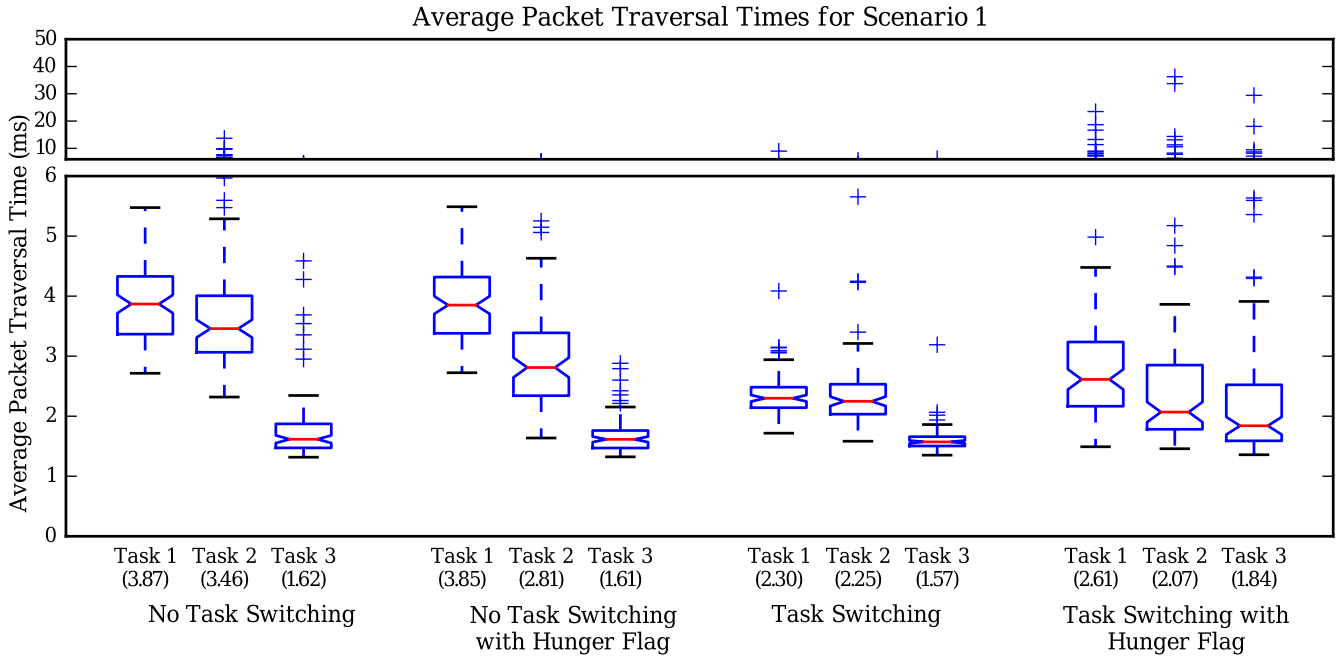

Fig. 8. Average packet traversal times of 100 runs of the first application scenario, with the medians given in brackets. The first two cases do not have any adaptive task switching but do have routing tables pre-loaded with optimal routing directions. The last two case do have adaptive task switching, but have random routing tables which can cause long paths and deadlock with wormhole routing. Despite this we see the network adapt to the random routing directions to recover a better median performance and a much tighter distribution to the first two cases. In the final case the hunger flag causes too much variation in the network for it to stabilise quickly, as discussed in Section IV. The skew of Task 3 packets is due to Task 1 nodes (their destination) being the initiator of the data-flow as the only task that sends data at a fixed rate, so the network is relatively quiet when data is sent back to it.

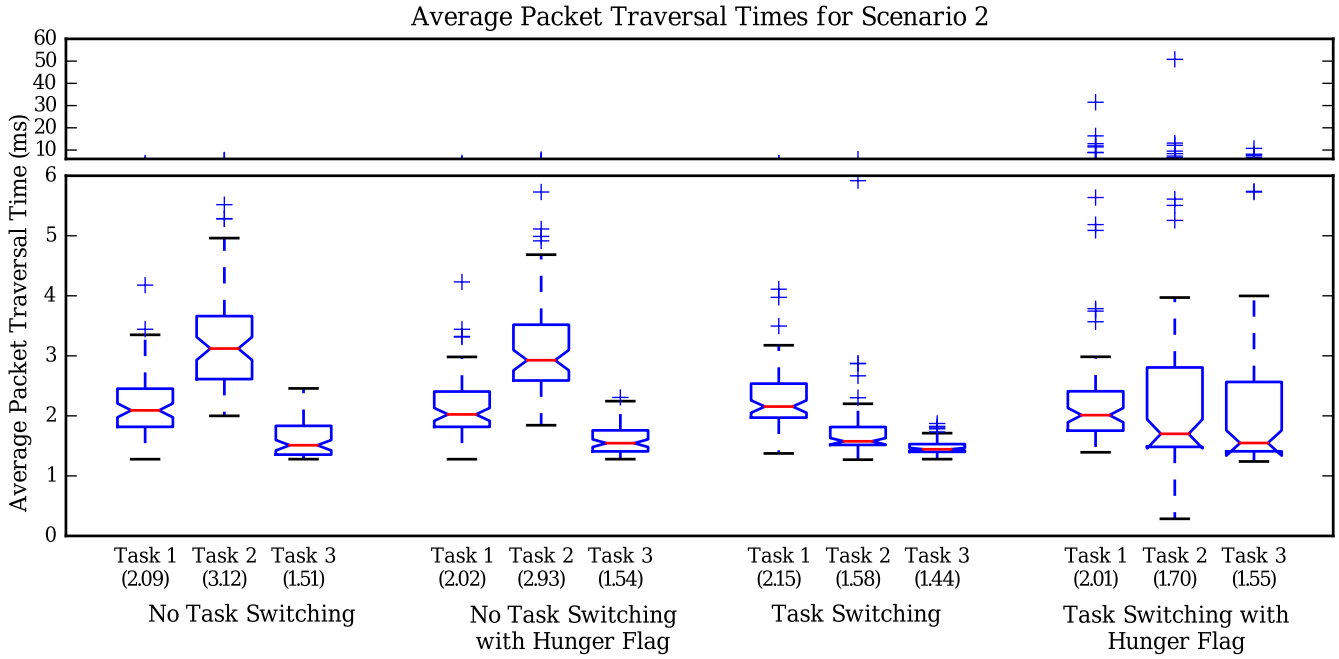

Fig. 9. Average packet traversal times of 100 runs of the second application scenario, again the medians are given in brackets. The high number of Task 2 nodes relative to Task 3 nodes mean that it is possible for Task 3 nodes to be the bottleneck (depending on the random initial task allocation), resulting in a larger distribution for packets from Task 2 nodes. The adaptive task allocation alleviates this bottleneck at a slight sacrifice of Task 1 packet latency, implying that less active Task 2 nodes may task switch to take up the Task 3 packets at the bottlenecks. Again the hunger flag causes too much variation in the network for it to stabilise quickly.

have the "hunger" flag enabled and so nodes can inform their neighbours that they are busy, causing some of the bottlenecks that cause the spread in the distribution to be alleviated. The intelligence currently only picks the next best routing choice (as preloaded into the tables) and this can cause deadlock. The deadlock recovery scheme will ultimately mean that packet is sunk at some point, but it will have a temporary impact on the network as can be seen by the fluctuation in the time domain average shown in graph (b) of Figure 10.

When we enable the task switching we see a similar improvement to the distribution as we saw in the simulation undertaken in [10]. Despite the task mapping *and* the routing tables being randomly issued for this case (optimal routing tables would be quickly rendered useless by the task switching), the network autonomously adapts its task topology to more optimal mappings as dictated by the random routing tables. This is clearly shown in graph (c) of Figure 10 as at first the average latencies are very poor (four times as long), but as the network adapts we see this improve until it is of similar performance to the schemes with pre-loaded optimal routing tables. What is interesting is that the hunger flag now introduces a huge amount of variation into the network as seen in the final case. Without the "next best choice" preloaded into the routing tables, packets are routed in a random manner and so they not only cause deadlocks as with the no task switch case, but also can cause the task mapping to adapt to the temporary diversion and thus require a far longer time for the network to "settle down" to a stable task mapping or it may even not be able to settle down at all.

The different application topography of the second scenario requires a more complex traffic pattern that the network shall have to adapt to. As Figure 9 shows, the adaptive task allocation manages to achieve this. With the first two schemes there is a delay for task 2 packets due to the high ratio of task 2 nodes trying to sink to task 3 nodes. The adaptive scheme can reduce this imbalance by possibly swapping idle task 2 nodes to task 3, indeed this is shown by the larger distribution of task 1 nodes, although the range between the medians remain much smaller than for the non-adaptive schemes. Again we see the hunger flag introduces too much of a dramatic routing change and the spread is very large; indeed given the longer CPU time of task 2 nodes this is somewhat expected as more nodes are likely to be in their "hunger" state.

V. Comparison with Simulation Model

The most fundamental difference between the hardware implementation and our previous simulation model (aside from the scaling from 36 to 100 nodes) is the routing technique: the simulation had used store and forward whilst for an efficient implementation we have moved to wormhole routing as discussed earlier in Section III. As a fundamental part of our intelligence model is observing the pattern of packets flowing through a node, wormhole routing has introduced some significant changes in network behaviour. For example, in the simulation model we showed how fault tolerance is easily achieved with the adaptive behaviour as a failed node will no longer sink packets destined for it and so an increase in packets for that node are introduced into the network that the other nodes can then react to. For wormhole routing however, it is unlikely that a packet is sent in its entirety before the head is accepted and so in our fault case we instead have many heads of packets moving around the network, making the network more vulnerable to deadlock and causing a limited response in the router as it can only accept four packets concurrently. These packets will then not be cleared due to the faulty node and the limited impact on each router's task counter threshold means that the necessary task switch will not happen until these packets completely deadlock.

Another difference introduced by wormhole routing is the lack of buffering at the node level: a packet can only be routed once it is accepted at a node and up until this time its tail is holding the routing path, preventing other packets from using this path. Our previous packet switching used some small buffers at each router, meaning that instead it took a number of packets to block a routing path and so a larger proportion network dynamics are limited to neighbour-neighbour interactions which our intelligence model is highly optimised for. This can be seen when the time domain representations of the packet latencies given in Figure 10 and Figure 11 are compared directly. Although for different random start task topologies, it is seen that in the non task switching case the wormhole experiment has a high latency at the start. This is due to the first packet blocking subsequent packets until it is routed, after which the second packet does the same etc. This effectively enforces a time-slicing of network traffic as the nodes will not receive their packet, and therefore process it, until a blocking packet is finished. As all the rate parameters and processing times are the same for all nodes, resulting in that the next time a node is ready to send a packet it will have delayed enough from the first time to send it without waiting on the blocking packet (which will have already been sunk in the time the node was busy processing).

VI. Conclusions

Through hardware implementation we have shown that our social-insect-inspired task allocation is a viable approach to adaptive task allocation in many-core systems and we have also demonstrated the scalability of such an approach by increasing the number of nodes in the platform by nearly three times to 100 nodes, yet whilst maintaining a tiny hardware overhead of 3% that scales linearly with the number of nodes. However we have also discovered how a hardware implementation inevitably alters the underlying network dynamics and introduced patterns that the intelligence model cannot react to in the same manner as prior experiments. Despite this, decentralised task allocation which emerges from a far from optimal routing scheme is still observed and our intelligence model easily allows addition of extra local signal monitors that can exploit other dynamics that wormhole routing offers up to us. Indeed we have shown in our previous work that many biological models of task allocation exist and all models have some degree of sensory integration and evaluation, thus our

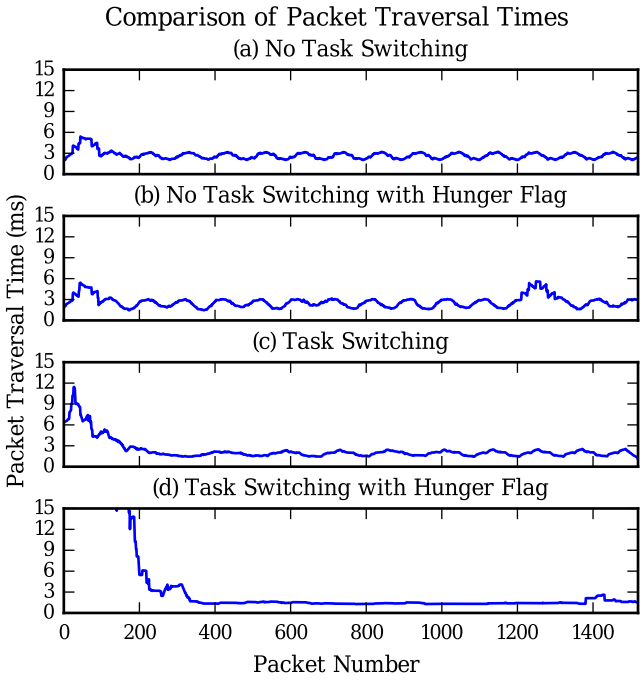

Fig. 10. Average packet traversal time represented in the temporal domain. A run from one of the 100 runs was chosen and a moving average was applied across the raw latencies of each packet. The same seed is used for each run of the different schemes and so the random starting topology and preloaded random routing tables are consistent across each scheme in this graph. Two effects are clear: the wormhole routing induced improvement in latency at the start as it enforces time-slicing of packets (discussed in Section V) and also the greater amount of time required for the hunger flag enabled schemes to stabilise and how they destabilise later on.

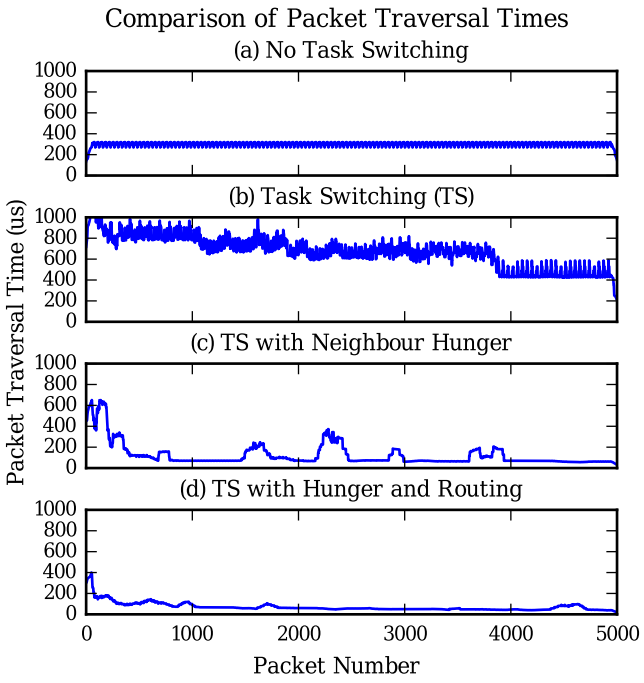

Fig. 11. The average packet traversal time represented in the temporal domain, from the simulation presented in our previous paper[10]. The difference in network dynamics between packet switched and circuit switched routing is clear to see when the two charts are compared directly.

current set of abilities of the distributed intelligence is only the baseline for fully adaptive many-core systems.

VII. Further Work

The Microblaze MCS provides an IO interface to the microprocessor that makes it very suited to the addition of embedded hardware accelerators. This would add heterogeneity to the many-core and so is of significant interest. Indeed the correct emergence of "generalist and specialists" members is observed as a fundamental part of the survival of some species of social insect and so our intelligence model can also take inspiration from this aspect as heterogeneity makes the problems of task allocation and routing problems even harder to design with a design-space exploration.

In addition to extra NoC sensory inputs to the intelligence model the hardware system also allows us to supplement the NoC information with information about the platform, such as thermal or power information, which will allow our many-core to be fully adaptive to Dark Silicon imposed constraints. Indeed this does not and should not be information at a global level; embedded sensors such as ring oscillators, results of CRC checking or even information on the processing load of neighbours to allow a node to detect when it is starting to push its local processing envelope and the network should react and adapt accordingly.

Acknowledgements

This work was supported by funding from the Department of Electronics and an EPSRC DTA award.

References

[1] R. Dennard and V. Rideout, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[2] H. Esmaeilzadeh and E. Blem, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 365–376.

[3] L. Benini and G. D. Micheli, "Networks on chips–a new SoC paradigm," *Computer*, 2002.

[4] A. Hemani, A. Jantsch, S. Kumar, and A. Postula, "Network on chip: An architecture for billion transistor era," in *IEEE NorChip Conference*, 2000.

[5] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote, "Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 3–21, jan 2009.

[6] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 23–es, aug 2007.

[7] K. Srinivasan, K. Chatha, and G. Konjevod, "Linear programming based techniques for synthesis of network-on-chip architectures," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* IEEE, 2004, pp. 422–429.

[8] U. Ogras and R. Marculescu, "Energy- and Performance-Driven NoC Communication Architecture Synthesis Using a Decomposition Approach," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 352–357.

[9] G. Tempesti, "Graceful Design," *International Innovation Issue 140*, pp. 76 – 78, 2014.

[10] M. Rowlings, A. Tyrrell, and M. Trefzer, "Social-Insect-Inspired Adaptive Task Allocation for Many-Core Systems," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 911–918.

[11] ——, "Social-Insect-Inspired Networking for Autonomous Load Optimisation," *Procedia CIRP*, vol. 38, pp. 259–264, 2015.

[12] ——, "Social-Insect-Inspired Networking for Autonomous Fault Tolerance," in *2015 IEEE Symposium Series on Computational Intelligence*. IEEE, dec 2015, pp. 1198–1205.
[13] C. Anderson and D. McShea, "Individual versus social complexity, with particular reference to ant colonies," *Biological Reviews (of the Cambridge Philosophical Society)*, pp. 211–237, 2001.
[14] D. Gordon, "The organization of work in social insect colonies," *Nature*, 1996.
[15] D. Gordon, B. Goodwin, and L. Trainor, "A parallel distributed model of the behaviour of ant colonies," *Journal of theoretical Biology*, 1992.
[16] P. Simmons and D. Young, *Nerve cells and animal behaviour*, 3rd ed. Cambridge University Press, 2010.
[17] Xilinx Inc, *MicroBlaze Micro Controller System v1.4 (PG048)*, 2013.
[18] P. Mohapatra, "Wormhole routing techniques for directly connected multicomputer systems," *ACM Computing Surveys (CSUR)*, vol. 30, no. 3, pp. 374–410, 1998.