Enabling Massive Multi-Threading with Fast Hashing

Alberto Scionti [*], **Somnath Mazumdar** [†], **Stéphane Zuckerman** [‡]
[*] Istituto Superiore Mario Boella (ISMB), Turin, Italy
[†] Università degli Studi di Siena, Siena, Italy
[‡] Michigan Technological University, MI, USA

Abstract—The next generation of high-performance computers is expected to execute threads in orders of magnitude higher than today's systems. Improper management of such huge amount of threads can create resource contention, leading to overall degraded system performance. By leveraging more practical approaches to distributing threads on the available resources, execution models and manycore chips are expected to overcome limitations of current systems. Here, we present DELTA – a Data-Enabled muLti-Threaded Architecture, where a producer-consumer scheme is used to execute threads via complete distributed thread management mechanism. We consider a manycore tiled-chip architecture where Network-on-Chip (NoC) routers are extended to support our execution model. The proposed extension is analysed, while simulation results confirm that DELTA can manage a large number of simultaneous threads, relying on a simple hardware structure.

Index Terms—Dataflow, Hashing, Network-on-Chip, Thread-scheduling

1 Introduction

Exascale machines are expected to execute large, multiple applications at higher speed. To meet this goal, these machines have to manage a number of threads that is orders of magnitude greater than in current petascale machines, with more stringent power and resiliency constraints [4]. Recent CPU designs favour the integration of a vast number of simple, single-issue, in-order cores [1] to increase the number of threads that can be executed in parallel. However, traditional program execution models (PXMs) derived from the von Neumann model and used by such systems, exhibit a large thread synchronisation overhead. Their inherent sequential nature makes it tough to guarantee correctness and race condition freedom in multi-threaded program executions [8]. Furthermore, when fine-grain threads are exploited, their synchronisation activity quickly becomes the main performance limiting factor [5], also contributing to energy waste. Instead, PXMs which rely on explicit producer-consumer semantics and are self-scheduled [3] drive the design of efficient and less power hungry chip architectures (*e.g.*, [13]).

The eXplicit Multi-Threading (XMT) architecture [12] introduces an abstract execution model, where switching from serial to parallel execution is made through explicit spawn/join instructions. Specifically, such instructions create a group of concurrent threads executing the same code block, while micro-architectural support remains generic. Kyriacou et al. [7] proposed a scalable architecture which, requires a large amount of storage to maintain a local copy of the threads' dependency graph, while a flat thread distribution model is applied. Recently, TERAFLUX [6] proposed a chip architecture for the explicit exploitation of a dataflow PXM, where cores are organised into fixed-size nodes. Although it was proposed as a scalable solution, many drawbacks remain: locality of computations is not guaranteed (threads cannot explicitly restrict execution within a node), and an efficient selection of the target execution cores is not described. In recent years, GPUs emerged as the preferable platform to accelerate computations [9], thanks to their capability of running hundreds of fine-grain threads in parallel. However, their architecture is optimised for regular applications and does not adapt well to irregular data and control problems.

In this paper, we propose DELTA – a Data-Enabled muLti-Threaded Architecture – which attempts to: *i*) implement an effective mechanism to select target execution cores, as well as to guarantee locality of computations; *ii*) supporting the execution of a large number of concurrent threads with a lightweight synchronisation mechanism, and *iii*) provide a simple programming interface. Starting from a manycore tiled-chip, we augment the NoC router structure with a hardware unit responsible for the threads' creation and distribution over the application lifetime (we are agnostic in respect to the processing element architecture). Also, a fast hash-based mechanism allows the system to efficiently distribute the threads among the available processing resources, leading to more dynamic scaling-up capabilities and less power consumption.

2 Program Execution Model (PXM)

PXMs define how a computation must be carried on a target machine, on concurrency (how threads are created, scheduled, and destroyed), memory behaviour (how memory is addressed, and what ordering rules it obeys), and synchronisation (how threads can synchronise/wait for each other). Contrary to programming models, which describe *what* high-level action should be done (and when), PXMs describe *how* such an action is carried in the system. For example, OpenMP's programming model allows a programmer to define a region of code as *parallel* but makes no explicit mention of threads. However, OpenMP's PXM specifies that when a parallel region is encountered, a team of threads has to be created and must also be destroyed when the end of the region is reached. Here, we use an execution model directly derived from the Codelet model [11], where assisting hardware provides large performance improvements over a

Manuscript submitted: 26-Oct-2016. Manuscript accepted: 21-Apr-2017. Final manuscript received: 21-Apr-2017.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCA.2017.2697863, IEEE Computer Architecture Letters

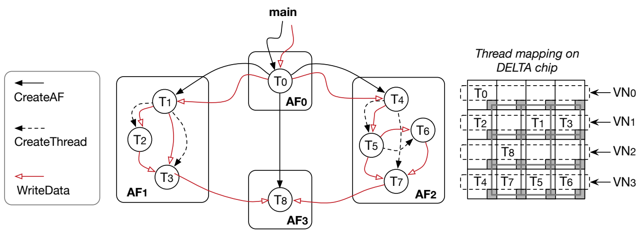

Fig. 1: A simple kernel application adhering with the proposed PXM and a possible mapping of threads on the PEs.

pure software implementation. The applications are divided into a set of fine-grain threads, each totalling no more than a few tens or hundreds of instructions. Fine-grain threads are exploited in a way that allows maximising the utilisation of the system.

Threads represent the quantum of execution: they exchange data with each other by resorting to an explicit producer-consumer scheme. It allows the construction of a data-flow graph (DFG) at compile time, which explicitly shows data dependencies among threads. Each thread holds a local storage space (*frame*) used to receive input data from producer threads, as well as to write intermediate results. It also contains a *scheduling slot* (SS) counting the number of inputs still required for the execution. A thread context contains the frame data and a unique thread identifier. To preserve locality and allow for better latency hiding, threads are grouped into *asynchronous functions* (AFs). Similarly, DELTA provides a mechanism for dynamically grouping processing elements to form virtual nodes (VNs) as part of the hashing mechanism, so that threads within an AF are forced to be executed on the same virtual node. With the aim of exposing these characteristics at the programming level, DELTA extends the processing element ISA with a reduced set of dedicated instructions (eventually wrapped by high-level programming language functions, *e.g.*, C/C++). In particular, `CreateThread` and `CreateAF` allow respectively to schedule a new thread and a new asynchronous function (*i.e.*, a new thread spawned outside the VN). `ReadData` reads data from a local frame, while `WriteData` writes a new data on a consumer thread frame; `DecreaseSS` decrements the scheduling slot of a consumer thread, while `DeleteThread` removes the context of a thread that completed the execution. Finally, `SetVN` and `ConfigRouter` allow respectively to set the size and to configure virtual nodes. With the aim of further optimising the execution, the compiler can aggregate multiple writes (*e.g.*, dealing with large loops) and use a single `DecreaseSS` signalling operation to update the corresponding SS field.

Every time a new thread is spawned, a processing element (PE) within the current virtual node is automatically selected and signalled. Similarly, every time a thread creates a new AF, the destination PE is selected within the whole chip. Hence, the creation of a new AF is led back to the scheduling of the root thread of the DFG contained in the AF. Figure 1 shows an example of a simple kernel application consisting of 4 asynchronous functions, each with its DFG. Both asynchronous functions and threads are directly managed by the compiler, which is responsible for mapping high-level programming constructs (*e.g.*, `#pragma omp for` when using OpenMP) with the correct sequence of `CreateAF` and `CreateThread` instructions. Figure 1 also shows that AF scheduling requests and write operations remain well confined on the local VN. By monitoring the scheduling slots, the hardware unit automatically fires threads became runnable, without the need of executing an explicit instruction. On the contrary, the execution completion is signalled by the `DeleteThread` instruction that allows freeing resources held by the thread.

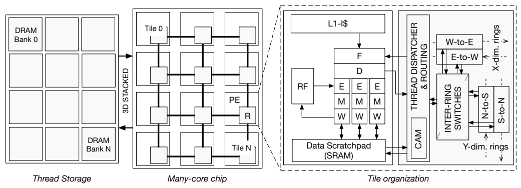

Fig. 2: DELTA chip organization: tiles contain a processing element (white box) and a ring-based router (gray box). The scratchpad substitutes the traditional L1-data cache.

3 DELTA Architecture

Figure 2 shows the whole DELTA chip organisation: a dedicated 2D-mesh Network-on-Chip, implemented with lightweight ring-based routers [10] (such kind of routers allows implementing a physical 2D-mesh topology on top of four unidirectional rings), connects a large group of tiles covering the entire chip area. Each tile contains a PE coupled with a lightweight router. Routers are augmented with our fine-grain thread hardware support: a local unit called *Thread Dispatcher* (TD) manages the threads during their lifetime. In particular, it allocates internal space for storing thread contexts every time new threads are created, removes previously allocated resources whenever threads complete, reads from (respectively writes to) associated frames. We assume that only one thread at a time can be executed in each PE, although our approach can benefit from the implementation of a form of local simultaneous multi-threading.

Chang et al. already showed how to apply a dynamic hashing mechanism to distribute the workload in a peer-to-peer system [2]. Here, we propose to apply hashing for thread allocation in a hardware system (in a much more constrained environment). On the one hand, the hashing mechanism must avoid the creation of hot-spots in the chip (*i.e.*, areas of the chip particularly stressed). In fact, an imbalance in the load distribution quickly and significantly increases the power and temperature of the more stressed portion of the chip, thus contributing to decrease the overall reliability (*e.g.*, device ageing is accelerated). Load imbalance can also create congestion in the network since some of the links drive more traffic than others. On the other hand, the hashing mechanism used to distribute the threads must guarantee locality of computations. To this end, our hashing scheme allows to place a group of dependent threads (asynchronous function) on the same group of PEs (virtual node), while still preserving a fair thread distribution within the VN by selecting PEs in a random fashion. Similarly, the hashing scheme allows to randomly schedule asynchronous functions on different VNs across the whole chip. It is worth to note that choosing PEs that minimise the communication distance between producer and consumer threads in a reasonable amount of clock cycles is not a trivial problem. In fact, our PXM implies more than one producer can generate input data for

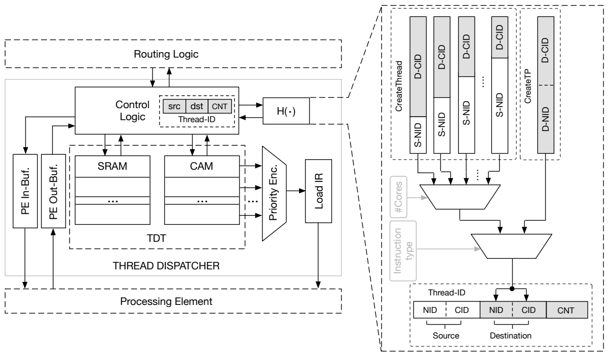

Fig. 3: Thread Dispatcher module organization (left) with the internal structure of the $H(\cdot)$ function (right).

a single consumer, as well as producers can be scheduled and executed at different points in time.

Each thread in the system is identified by a unique *thread identifier* (T_{id}) over the whole application execution. It is composed of three main fields: the *source field*, the *destination field*, and a local counter (*CNT*). The source and destination fields are in turn formed by two sub-fields representing the *core identifier* of the PE (C_{id}), and the *virtual node identifier* (N_{id}). While the content of the source field is fixed for each tile (once the number of cores in each VN has been selected), the content of the destination field is produced at run-time by the hashing function $H(\cdot)$. These fields allow the system to generate unique identifiers that are conveniently stored in a 64-bits register. The organization of the T_{id} is illustrated in figure 3. By passing to the $H(\cdot)$ module both the size of VNs (N_{pe}) and the indication of the executed instruction – I_{ex} (*i.e.*, the `CreateThread` or the `CreateAF` instruction), it uniquely identifies the PE responsible for the execution of the newly generated thread (*i.e.*, $\langle N_{id}, C_{id} \rangle_{dst} = H(N_{pe}, I_{ex})$). Once selected, the PE is signalled by sending a message over the network. Since the destination is encoded in the T_{id}, any subsequent operation on the thread can easily be forwarded to its corresponding PE, without any further calculation. This contributes to the speedup of the system.

Threads are managed through a data structure called *Thread Descriptor Table* (TDT) that is organised in two fixed-size local memory arrays (their total area is comparable with that of an L1 data cache). Input and intermediate data are stored in the scratchpad memory. Every time the context of a thread is updated, the T_{id} is used as a search key within the CAM. In case of a match, the returned base address of the frame F_b is added to an offset F_o to determine the location access (*i.e.*, $l = F_b + F_o$). Finally, a priority encoder selects the thread with the lowest T_{id} among those runnable ($SS = 0$ – see figure 3). Every time the selected PE is devoid of free resources, it can access to a larger but slower memory area called *Thread Storage*, implemented as a 3D-stacked DRAM layer. It is organised in banks (one for each PE) representing a larger TDT structure. When a PE receives a new thread, it first selects the entry in the local TDT and compares the SS value of the new thread and the one currently stored. The thread with the highest SS will be swapped on the DRAM memory bank.

4 Hash Scheduling Function

The purpose of the hash scheduling function $H(\cdot)$ is to map new threads to PEs for their efficient execution. In our case, the hardware module assigns to the newly created thread the tuple $\langle N_{id}, C_{id} \rangle_{dst}$ depending on the VN size and the executed instruction. To be effective, the scheduling function $H(\cdot)$ has to distribute `CreateAF` and `CreateThread` requests among the available resources fairly. The effectiveness of the hashing function derives from the ability to limit the number of times two distinct input values result in the same output value for the hashing. In our distributed scheduling scheme, this translates in avoiding different PEs selecting the same destination, given two different T_{id}. In that case, the PEs' load (*i.e.*, the number of threads to execute) is balanced, thus avoiding the formation of hot-spots and increasing the overall system reliability. A good hash functions must provide *determinism*, meaning that for the same set of input keys it has to provide the same set of hash values. More important, hash functions must exhibit *uniformity*: given a set of n input keys and m output buckets, each bucket shows a load $\lambda = \frac{n}{m}$. It directly translates in the same likelihood for each output bucket to be selected, thus limiting the number of collisions. Finally, since we are not considering cryptographic applications, it is not strictly required that the function be *non-invertible*, while it is more desirable to keep hardware implementation efficient in terms of area and power consumption. We found that the following scheme provides very good results while maintaining a low area overhead and preserving the capability of dynamically changing the size of VNs. Another important aspect of our scheme is that it works in a completely distributed fashion, meaning that a *single point of failure* is not present, as desired in a system equipped with thousands of PEs possibly.

4.1 Hardware implementation

The $H(\cdot)$ module contains a set of maximum-length linear-feedback shift registers (LFSRs), each providing a pseudo-random sequence with a different length L_{rnd}. Let n be the number of bits composing the LFSR, the length of the sequence is given by $L_{rnd} = 2^n - 1$, *i.e.*, the register cycles through all the 2^n configurations except for the configuration containing all zeros. The structure of the LFSR is thus modified, in such way all the 2^n configurations can be generated. The use of LFSRs allows the $H(\cdot)$ module to select every time a different PE in a round-robin fashion (although it is a random sequence). Compared to simple counters, LFSRs guarantee a homogeneous spatial distribution of the threads among the PEs in a VN over time. Indeed, since the LFSRs are initialised with a different seed the generated sequences are different. This preserves the chip from the emergence of local hot-spots and contributes to reducing pressure on the NoC links, hence reducing link contention and thus improving performance. Depending on the executed instruction (`CreateThread` or `CreateAF`), LFSRs are used to generate the destination C_{id} or both the C_{id} and N_{id}. In the case of the `CreateThread`, the destination VN is the same of the PE spawning the new thread. The $H(\cdot)$ module computes the destination PE for different VN sizes in parallel; while the selection of the actual one depends on the effective VN size, and it is performed by a multiplexer. For instance, a VN containing 64 PEs requires an LFSR 6-bits long. The executed instruction also controls which tuple (*i.e.*, the one formed by both the newly generated C_{id} and N_{id}, or the one formed by only newly generated

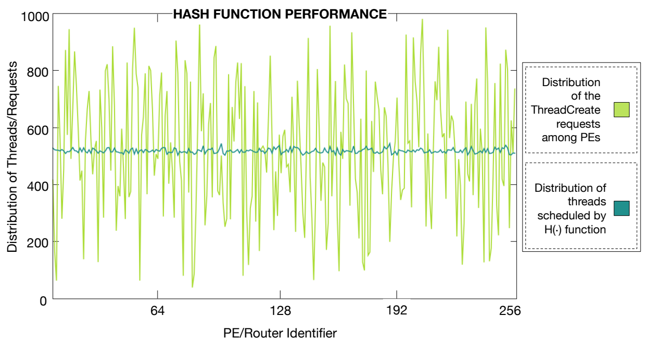

Fig. 4: Hash function performance: distribution of thread creation requests (green line), and distribution of threads on the PEs (blue line).

C_{id}) to copy in the destination field of the T_{id} through a second multiplexer. This mechanism is shown in figure 3 (right side). Note our scheme guarantees the locality of computations: threads belonging to the same asynchronous function are kept close to each other since they are executed on PEs placed closely in the chip, albeit randomly selected (see also figure 1). At the same time, asynchronous functions are homogeneously distributed among the VNs. Finally, to deal with space limitation for thread management, each tile can eventually reschedule a thread on a different PE for a limited number of time.

5 Evaluation

We evaluated the DELTA architecture regarding scalability and power consumption, as well as we evaluated the capability of the hashing function of well-distributing input requests. The simulated manycore design comprises up to 256 PEs implementing a 5-stage RISC-V compliant in-order execution pipeline (16 KiB I-cache + 16 KiB scratchpad memory), supporting our proposed instruction set extension, and integrating a 2-stage lightweight router [10]. We generated network traces using an in-house simulator and monitoring the set of requests send to the TD units. We performed the scalability and power consumption measurements implementing the NoC infrastructure on an Altera Stratix III-based device. Figure 4 shows the performance provided by the proposed hashing function implementation. The purpose of this experiment is to show how a huge number of input keys for the $H(\cdot)$ module (*e.g.*, thread scheduling requests) would be distributed among the PEs. To that end, we simulated a random traffic pattern in the NoC by allowing each tile to randomly injecting a schedule request (injection rate was 1.0) towards a randomly selected VN and PE. This kind of pattern is more effective in showing the capability of the hashing mechanism since the traffic cannot be predicted. As discussed in Sec. 4, an effective hashing scheme must provide a uniform distribution of the hash values. Looking at figure 4, the green line represents the initial distribution among the PEs of the `CreateThread` requests, while the blue line shows the effective distribution of threads as they have been scheduled by the $H(\cdot)$ modules. The high fairness in the assignment of the threads to different PEs greatly contribute to the high overall performance of the network, and it is confirmed by the χ^2- *test*. To this end, we considered a system with 100 degrees of freedom (*i.e.*, a system equipped with 101 nodes), a total number of more than $5.0 \cdot 10^4$ thread requests (*i.e.*, input keys), and an error between observed and expected distribution of 1.0% (*i.e.*, p-value equals to 0.010). Running the test, we obtained $\chi^2 = 10.838$ that is lower than expected threshold of 135.8 for a system with $n = 100$ degrees of freedom. Such value confirm that the proposed distribution mechanism provides a strong uniformity of the hash values, thus lowering the collision likelihood. Similar results (not showed for space limitations) have been obtained simulating the traffic pattern generated by a block matrix multiplication kernel. Compared to other (cryptographic) schemes, *e.g.*, the S-box, our solution provides good uniformity in an area efficient module capable of performing hash value computations in less clock cycles. The same traffic patterns have also been used to assess the NoC throughput and power consumption. Irrespective of the growing of the number of PEs in the system, the throughput grows almost linearly, while power consumption is relatively low if compared with an implementation based on traditional crossbar switch routers. In general, the area and power consumption for the scheduling logic remain very low, while that of the TDT is in line with that of an L1-data cache (it is worth noting the scratchpad substitutes the L1 D-cache, and represents the main data-exchange point between routers and PEs).

6 Conclusion and Future Work

The proposed DELTA architecture supports a higher degree of thread parallelism by extending the structure of NoC routers in such way that they can fairly distribute threads among all the computing resources, through an effective hardware hash scheduling mechanism. Preliminary simulations confirm the capability of our design to effectively scale with the increasing amount of PEs. Future investigations will focus on the detailed analysis of the DELTA architecture by porting complex applications to our PXM.

References

[1] N. Carter et al., *Runnemede: An architecture for ubiquitous high-performance computing*, in Proc. HPCA, pp. 198–209, IEEE, 2013.
[2] Ye-In Chang et al., *A Dynamic Hashing Approach to Supporting Load Balance in P2P Systems*, IEEE ICDCS, pp. 429–434, 2008.
[3] J. B. Dennis, *First Version of a Data-Flow Procedure Language*, In Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 362–376, Springer, 1974.
[4] J. Dongarra et al., *The international exascale software project roadmap*, Int. J. High Perform. Comput. Appl. vol 25, no. 1, pp. 3–60, SAGE Pub., 2011.
[5] T. Geng et al., *The Importance of Fine-Grain Synchronization for Many-Core Systems*, Int. Workshop on Languages and Compilers for Parallel Computing, 2016.
[6] R. Giorgi et al., *TERAFLUX: Harnessing dataflow in next generation teradevices*, Microprocessors and Microsystems, vol. 38, no. 8, pp. 976–990, Elsevier, 2014.
[7] C. Kyriacou et al., *Data-Driven Multithreading Using Conventional Microprocessors*, in IEEE TPDS, vol. 17, no. 10, pp. 1176–1188, 2006.
[8] E. A. Lee, *The problem with threads*, IEEE Computer, vol. 39, no. 5, pp. 33–42, 2006.
[9] M. J. Schulte et al., *Achieving Exascale Capabilities Through Heterogeneous Computing*, IEEE Micro, vol. 35, no. 4, 2015.
[10] A. Scionti et al., *Software defined Network-on-Chip for scalable CMPs*, IEEE HPCS, pp. 112–115, 2016.
[11] J. Suetterlein et al., *An Implementation of the Codelet Model*, Euro-Par, pp. 633–644, Springer, 2013.
[12] U. Vishkin et al., *Explicit Multi-threading (XMT) Bridging Models for Instruction Parallelism*, in Proc. SPAA, pp. 140–151, ACM, 1998.
[13] F. Yazdanpanah et al., *Hybrid Dataflow/von-Neumann Architectures*, IEEE TPDS, vol. 25, no. 6, pp. 1489–1509, 2014.