The DRACON Embedded Many-Core: Hardware-enhanced run-time Management using a Network of Dedicated Control Nodes

Daniel Gregorek, Alberto García-Ortiz
Institute of Electrodynamics & Microelectronics, University of Bremen, Bremen, Germany

Abstract
Many-core systems provide abundant computing power for parallel applications. The run-time manager of an embedded system has to efficiently exploit the available resources while guaranteeing a high responsiveness. We propose a dedicated hardware infrastructure to improve the scalability and responsiveness of a run-time task manager. The hardware enhancements constitute a hierarchy of global and local control nodes which communicate by means of message passing. The global nodes facilitate a distributed task manager which performs the task scheduling and a flexible task synchronization scheme at runtime. A low-latency interface between the run-time system and the processing cores is provided by the local nodes.
Based on simulations using a SystemC model, we demonstrate the advantages of our approach in terms of application performance. The design feasibility is substantiated by means of gatelevel analysis. We compare our results against state-of-the-art software and hardware-based run-time management systems.

Published in:
International Symposium on VLSI (ISVLSI) 2015, 2015
Date of Conference: 8-10 July, 2015
Page(s): 416-421
DOI: 10.1109/ISVLSI.2015.90

Conference Location : Montpellier, France
URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?&arnumber=7309603
Publisher: IEEE

This page was intentionally left blank

The DRACON Embedded Many-Core: Hardware-enhanced run-time Management using a Network of Dedicated Control Nodes

Daniel Gregorek, Alberto Garcia-Ortiz
Integrated Digital Systems Group
ITEM, University of Bremen, Germany
{gregorek,agarcia}@item.uni-bremen.de

***Abstract*—Many-core systems provide abundant computing power for parallel applications. The run-time manager of an embedded system has to efficiently exploit the available resources while guaranteeing a high responsiveness. We propose a dedicated hardware infrastructure to improve the scalability and responsiveness of a run-time task manager. The hardware enhancements constitute a hierarchy of global and local control nodes which communicate by means of message passing. The global nodes facilitate a distributed task manager which performs the task scheduling and a flexible task synchronization scheme at run-time. A low-latency interface between the run-time system and the processing cores is provided by the local nodes.**

Based on simulations using a SystemC model, we demonstrate the advantages of our approach in terms of application performance. The design feasibility is substantiated by means of gate-level analysis. We compare our results against state-of-the-art software and hardware-based run-time management systems.

***Keywords*-run-time task manager; embedded many core; hardware enhancements;**

I. Introduction

Power-density and reliability are becoming primary concerns in ultra-deep-submicron chip design. Many-core architectures are a promising candidate to address the upcoming challenges [2]. Dynamic hardware faults or changes of the user requirements make run-time task scheduling a necessary feature of the many-core system. But finding an optimal schedule for parallel applications usually requires an unaffordable amount of resources (NP complete) leading to the employment of heuristics for the run-time manager.

The demands for high performance, low power and deterministic response time advice for a hardware implemented solution of the run-time task manager [14][8]. Goal of the dedicated hardware is to reduce the management overhead and improve the overall system performance. It is therefore predicted [16], that hardware support for run-time system management will attain into mainstream for many-core platforms. Especially for embedded systems, where low-power is a crucial design constraint, the application specific hardware implementation of a task manager can become necessary.

Different architectural approaches for the dedicated system management have been reported in the literature. Centralized approaches have been shown to lack scalability for the on-line computation with an increasingly large number of cores [13]. Also, traffic hot-spots may become a bottleneck for such architectures. On the other hand, the overhead introduced by a fully-distributed approach can supersede the potential benefits. As a consequence, choosing the right granularity for an on-chip task manager remains an insisting optimization problem.

The idea of having a separate network for task synchronization has been described before by Herkersdorf [9]. State-of-the-Art many-core designs contain such dedicated hardware for synchronization (e.g. Tilera [1], Intel SCC [18], Kalray [6] and STHORM [23]). But to our best knowledge, we are the first to present and to analyze a *full-fledged* task management infrastructure using a dedicated hardware network. Our contribution, the DRACON many-core (**D**edicated **R**un-Time **A**rchitecture **Co**ntrol **N**etwork) focuses on low latency and predictability for run-time task management. The control nodes implement a message passing protocol for communication. The task management is charged with scheduling and synchronization of user tasks. Key feature of the dedicated network is the capability to perform parallel computation and to minimize the overall management latency.

The remainder of this paper is organized as follows: Sec. II discusses related work from the domain of run-time task management, Sec. III and Sec. IV present our system architecture and give details about the task manager. In Sec. V we show and analyze our experimental results and finally Sec. VI concludes the paper.

II. Related Work

The design space for run-time system management can be categorized into distributed versus centralized architectures, as well as into hardware versus software implementations [16]. Further, we localize along the hardware-software axis programmable approaches with optimized instruction sets.

A representative for a symmetric software-based run-time system is the Linux OS kernel. Linux has been shown to provide good scalability and high performance, even in the many-core domain [3]. Due to the shared-memory design, Linux depends on high-performance cache coherence and fine grain lock access.

A centralized OS-specific circuit resolving time critical task dependencies at run-time has been implemented by Nexus++ [5]. It has been designed to overcome a performance bottleneck involved by a software-based Master/Slave run-time system. An optimized processor architecture has been proposed by TMU [22] which employs an application specific instruction set optimized for task management. The TMU contains the capability to perform look-ahead computation. However, both solutions lack scalability due to the drawbacks of centralism in computation and communication.

A globally distributed and dedicated hardware approach has been implemented by Isonet [13]. Isonet applies a fully-distributed network of dedicated management nodes for hard-

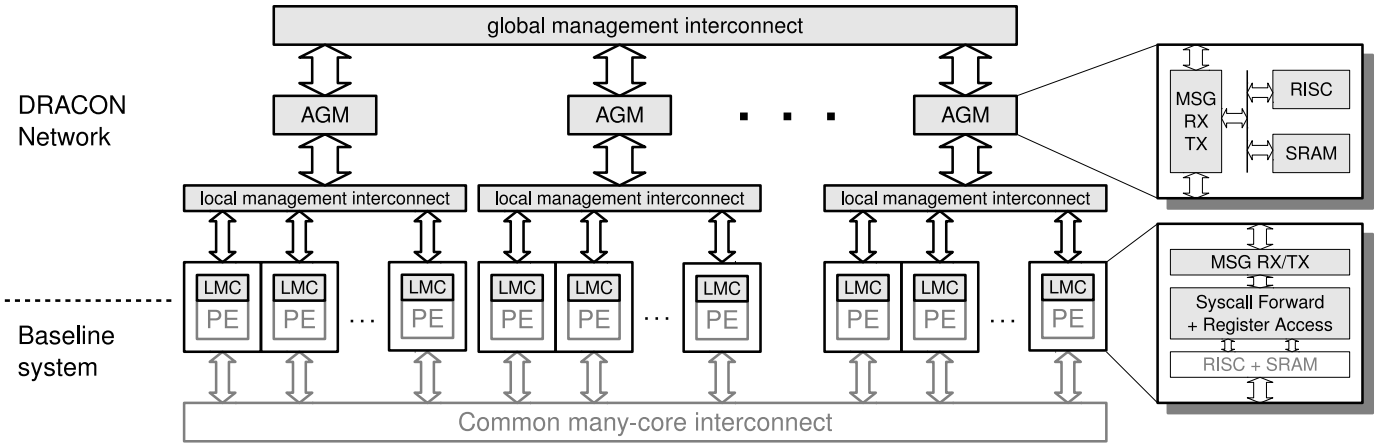


Fig. 1: Architecture of the dedicated control network for task management (drawn gray) on top of a baseline many-core system (drawn dimmed). The DRACON network consists of global nodes (AGM) and local nodes (LMC). The nodes communicate by means of message passing and perform the task scheduling and synchronization at run-time. The processing elements are connected by a common many-core interconnect.

ware supported load balancing. Using independent tasks, the approach claims to be scalable for more than 1024 cores. Yet, due to a limited synchronization scheme, the Isonet nodes may not find a globally optimal load balance. A lightweight HW/SW run-time framework for task management is presented by ARTM [17]. But, while ARTM uses the hardware semaphores included in the STHORM many-core architecture their evaluation is limited to intra-cluster task synchronization. As an advancement (see Tab. I) we propose the DRACON architecture as a full-fledged and global-view HW/SW run-time task manager.

TABLE I: DRACON vs. related work.

Arch.	Performance	Scalability	Area Overhead
Software	o	+	++
Nexus++	+	o	o
Isonet	+	+	-
DRACON	++	+	-

III. System Architecture

We propose a hierarchical and distributed network for run-time task management. The network enhances a common many-core system and is hierarchical by means of a clustering of the resources to be managed. It is also distributed, since each cluster can be managed autonomously. Fig. 1 gives an outline of our proposed system architecture. The hardware enhancements establish an infrastructure of dedicated management resources (drawn gray). The baseline many-core system (drawn dimmed) consists of numerous processing elements (PE), connected by a common multiprocessor-interconnect. As a low-latency interface, we closely couple a local management controller (LMC) to each PE. The LMCs are connected to a local management interconnect, which constitutes a cluster of PE + LMC pairs. Each cluster is controlled by an autonomous global manager (AGM). Communication between the AGMs is done via a global management interconnect. For the sake of readability we explain the particular hardware and software components side-by-side to our SystemC model.

A. Baseline System

As an initial baseline, we consider a *homogeneous* many-core system for our analysis. The many-core consists of RISC-like processing elements which are connected by a common many-core interconnect. For the SystemC model we consider the common interconnect to be a simplified Network-on-Chip without virtual channels having a mesh topology and XY-routing. The baseline system is not necessarily clustered by itself, the hierarchy is only constituted by the additional hardware enhancements. We employ a functional model for the processing elements, which interprets a task-based programming language (see also Sec. V). The employed network model considers effects like hop-distance, communication-volume and link utilization. The baseline NoC is only responsible for data transfer between the PEs (task communication). The NoC is not connected to any external memory or I/O, which is out of the scope of this paper.

B. Autonomous Global Manager

Each instance of the AGMs realizes one instance of the task manager on a dedicated processor and further contains an interface to the dedicated messaging protocol. The AGMs run a OS based on Micro-C/OS-II [11] which has been extended to a light-weight and distributed multicore OS. The AGMs implement the run-time task scheduling, the task synchronization and a cluster status communication mechanism (see Sec. IV). They monitor the activity (number of running tasks), and the number of mapped tasks inside their cluster. Each AGM has its private address-space, the communication between the AGMs and between one AGM and LMC is determined by the message protocol presented in Sec. III-D.

C. Local Management Controller

The local management controller (LMC) is constituted by a messaging interface, a system-call dispatcher and a tightly-coupled interface to the PE. The dedicated hardware LMC can be implemented with low area overhead (see Tab. V) and operates in parallel to the PE. Any system-call from a user task is fetched by the LMC and dispatched to its global node by means of a dedicated message. Due to the dedicated

TABLE II: System calls with inputs, outputs, and type of required context-wwitch.

Name	in	out	Context switch	Description
os-task-spawn	imem, dmem, cnt		SW only	Recursively spawn number of child tasks (cnt) with given instruction- (imem) and data-memory (dmem) address.
os-task-exit	addr		SW only	Send a signal to the synchronization barrier given by addr and terminate recursive child task. The parent task is restarted, if all child tasks have finished.
os-info-send	key, value		SW only	Send 32-Bit value to key.
os-info-recv	key, cnt	value(s)	SW and LMC	Receive 32-Bit value(s) by key. Blocks the task until all values are available.
os-get-pe		pe-id	none	Get numeric identifier of PE

infrastructure for the task management the PE is discharged from the execution of the OS service.

D. Messaging Protocol

We send messages via the dedicated interconnects to implement the communication between the hardware nodes. Each message has a header and one or more 32-Bit data fields. The header contains the message type, at least the source address, the message priority and a broadcast flag. The size of the message header depends on the actual hardware configuration (i.e. number of nodes/address-width) and the direction of the message. In the following we give a short overview about the implemented messages ordered by the direction of the conversation. A comprehensive reference can be found at the end of the paper (Tab. VI).

LMC to AGM: Most of the messages from the LMCs to the AGMs directly correspond to the system calls given in Tab. II and are send from a local controller to its global node. Messages from an LMC have a unique destination and therefore have a reduced header.

AGM to LMC: We send the message msg-task-start from an AGM to a LMC. The message transports a task identifier (tskd), and the task's stack-pointer as message data. It invokes the start of a task at the LMC. If the task has previously raised a system call, the message also transports the return value(s) of the call.

AGM to AGM: An AGM may also request to start a task at a remote cluster, therefore the message msg-task-give can be send from one AGM to another AGM. Since communicating task may be located inside different clusters, the task synchronization presented in Sec. IV-C also requires inter-AGM communication. To broadcast the current cluster workload status (see Sec. IV-D) the AGM uses the message msg-beacon.

IV. Task Management

Principal duty of the task manager is the scheduling and synchronization of the user tasks. Each task is considered to be sequential while the whole application is defined by a task graph [21] and consists of numerous depending tasks. The task graphs may contain task-level parallelism which must be exploited by the run-time task manager.

The run-time task management requires computation time to service the system-calls and to schedule (map) user tasks to processing elements [20]. Fig. 2 illustrate the inherent advantages of our hardware-enhanced run-time manager versus a software-based approach (for a more general HW-SW comparison see also [16]). A software approach will, at least initially, service a system call at the local processing element and requires more context switches (Fig. 2a). The hardware enhanced DRACON task manager forwards a system call (Fig. 2b) by means of a message to the AGM. The AGM services the system calls and may pre-compute the scheduling before going idle. If a PE becomes idle, the next tasks can be dispatched immediately. The responsiveness of the management system is therefore improved and the PEs are given more time for processing the user task. The response-time of the user applications is increased, while the standard deviation of the response-time is decreased.

Throughout this paper the user tasks can access the system calls given in Tab. II. Most of the system-calls require a context switch in a software-based implementation. Using the hardware enhancements only the call to os-info-recv triggers a context switch while the calling task is blocked and transported to its AGM.

A. Recursive Task Fork

Since our targeted task scheduling problems consist of task sets having a large number of tasks, we use a recursive task spawning/fork strategy [12]. Every recursive task spawns two additional system tasks and then blocks until its child's have terminated. The recursive start-up follows a dynamic cluster mapping procedure which tries to equally distribute the recursive system tasks onto the clusters. After the binary fork-tree has stopped to expand, the actual child tasks of the application are spawned. The final number of working child tasks is fixed and determined by the application profile.

B. Task Scheduling

The spatial and temporal task scheduling is performed on a cluster-basis by the AGMs. Each AGM contains one ready queue for waiting tasks. Our scheduling is quasi-preemptive: a running user task may be preempted by a higher priority system task but not by another user task. We use different scheduling algorithms depending on the application type. The algorithms have different time complexities depending on the number of ready tasks n or the number of PEs m. Usually, there is a trade-off between the time complexity and the quality of the scheduling algorithm [10].

(1) Round-Robin: We apply round-robin scheduling for independent tasks. The tasks are mapped and dispatched to the first idle PE. Scheduling time complexity is $\mathcal{O}(1)$.

(2) Max-Bottom-Level: For applications containing task dependencies we use a Max-Bottom-Level-First algorithm [21]. The ready tasks are stored inside a priority queue which is implemented as a Red-Black Tree and has a logarithmic time-complexity for inserting and removing [4]. The tasks are mapped and dispatched to the first idle PE. Time complexity is $\mathcal{O}(\log n)$.

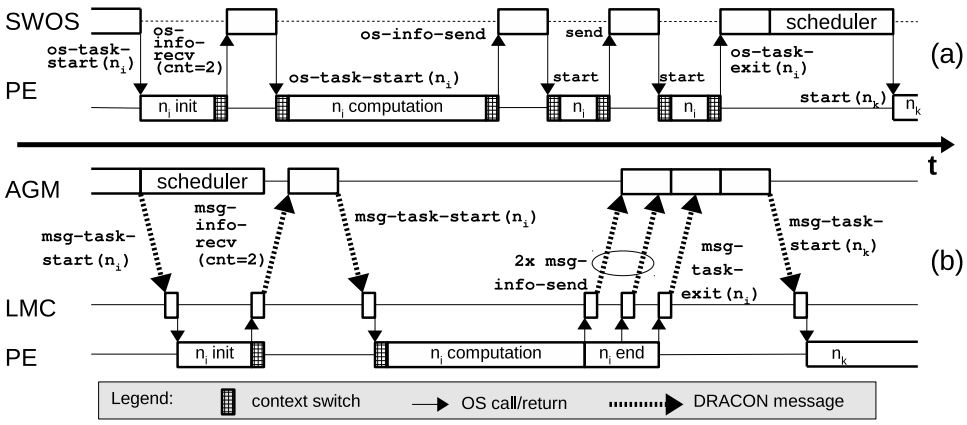


Fig. 2: Comparison of task management for one task n_i having two input and two output edges. The input data has arrived before the start of the given sequence. The software approach **(a)** runs at the same PE and interrupts the user task for OS service. The software approach schedules as late as possible.
The hardware-enhanced solution **(b)** implements the same OS interface but sends messages between the LMC and the AGM. The hardware approach requires less interuption of the user task and schedules as soon as possible in parallel to the PE. The outcome is an earlier dispatching of the next waiting task n_k.

(3) Nearest-Idle: To address the Manhattan distance between communicating tasks we apply the Max-Bottom-Level task selection. The selected task is mapped and dispatched to an idle PE nearest to the location of the application's parent task. Scheduling time complexity is $\mathcal{O}(\log n + \log m)$.

C. Task Synchronization

To have a flexible task synchronization sub-system which is able to work in a distributed environment, we developed a communication infrastructure based on key-value hash tables. Each global management node maintains one hash table autonomously at run-time. A task synchronization event is assigned to a unique key. User tasks can request to send a (non-empty) value to an arbitrary key. To allow tasks to wait until all of their input values have arrived, the system-call `os-info-recv` has been implemented. A task may therefore wait for a specific amount of values and then becomes ready for scheduling.

The synchronization sub-system does not contain critical races and utilizes a dynamic *localize-at-receive* mechanism: Every time a read request occurs, and the key position is not yet known or has changed, the position of the new key is broadcasted by the AGM via the global interconnect. The other AGMs read that broadcast and update their key positions accordingly. The broadcast is fully transparent to the user tasks, and allows their synchronization, even when tasks change their position from one cluster to another. Any value which is send to a key, who's position is not yet discovered, is pending until the required broadcast occurs. As soon as the position of a given key is known, a transport of the waiting values to the discovered position is triggered. In our implementation, we use the task synchronization sub-system to transport memory addresses between communicating tasks.

D. Cluster Status Communication

To perform the run-time task scheduling in an efficient manner it depends on reliable information about the global and local workload status. We use a broadcast message to inform all global managers about the local workload and apply a threshold-based mechanism to decide, whether to send that broadcasting message. The mechanism triggers the transmission, when the change in the number of locally active tasks reaches a certain threshold Δn_{th}.

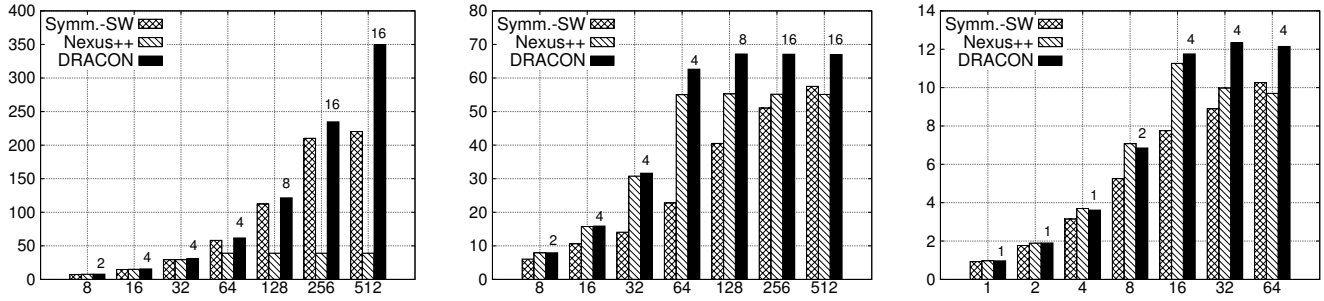
TABLE III: Benchmark characteristics: (a) number of workers, (b) total amount of work, (c) maximum task workload, (d) number of communication edges, (e) total sum of communication volume, (f) maximum edge communication volume.

Name	(a)	(b) Ticks	(c) Ticks	(d)	(e) 32b	(f) 32b
Indep.	1e4	2e7	2e3	0	0	0
Horiz.	8160	1.632e8	2e4	8092	0	0
Sparse	96	309760	5440	67	13668	204

V. Experimental Results

In our analysis we consider three different types of parallel applications. Their characteristic values are given in Tab. III. The first one assumes totally independent child tasks, the second has horizontal dependencies between child tasks, and the third one (Sparse) is a real-world task graph and is taken from the MCSL benchmark suite [15]. We use a transaction-level framework [7] implemented in SystemC to evaluate our hardware/software architecture. It uses a blocking transaction level methodology and TLM 2.0 sockets. The framework employs a task-based programming model to describe the behaviour of the user applications. The tasks are defined by trace instructions which are interpreted by the PE model. The traces consider the execution time, input edges and output edges of the tasks as well as their communication volume. Task communication is modeled by means of reading via the common NoC from the remote memory of a processing element.

For comparison with current state-of-the-art, we employ a symmetric software-based operating system and a centralized hardware implementation [5]. Since Isonet is only targeting on applications having independent tasks we do not compare to them. Both the reference measurements and DRACON provide the same syscall interface as given in Tab. II and make use of the same scheduling algorithms. In contrast, the software OS is only running at the baseline hardware of the many-core system. Due to the symmetry, one instance of the software OS is running at each PE. The software OS uses shared resources for scheduling, which are protected by fine-grained locks. The OS must acquire a lock before reading from the resource but we allow an asynchronous write to the shared resource. The locking ensures sequential read but allows concurrent computation and write to the shared resources. We do not consider any further memory contention or communication overhead for the software OS but assume fully coherent caches



(a) Independent Tasks (b) Horizontal Dependencies (c) Sparse-Matrix Solver

Fig. 3: Speedup versus number of PEs. The number of AGMs is given on the top of each DRACON column

TABLE IV: Timing parameters of the transaction-level model

Description	Value	Stakeholders
Scheduling delay coefficient	24 Ticks	SW-OS, AGM
System call	8 Ticks	SW-OS, LMC
Context switch	16 Ticks	PE
Task-Info Synchronization	16 Ticks	SW-OS, AGM
Lock access	8 Ticks	SW-OS
Software FIFO access	4 Ticks	SW-OS, AGM
TX/RX of message	4 Ticks	AGM, LMC
Local man. interconnect	64 Bit	AGM, LMC
Global man. interconnect	64 Bit	AGM
Common interconnect	128 Bit	Baseline NoC

in the background. For the second reference measurement we compare to the centralized hardware implementation Nexus++. We use a single AGM and calibrate our model according to the speedup values available for Nexus++ [5] and the synthetic benchmark containing horizontal task dependencies.

Tab. IV shows the parameters, which have been set for our transaction-level model. Both the DRACON architecture as well as the software reference have to cope with the same timing parameters for the task scheduling and task synchronization and the same baseline hardware. In our current work the management interconnects are implemented as shared buses, but our architecture allows to use other topologies as well. As a metric for the performance we measure the application speedup $S = t_{seq}/t_{par}$. In a first instance, each benchmark is run individually while the number of processing elements is increased. The resulting values for the speedup S are given in the Fig. 3. DRACON provides good scalability for fine-grain **independent tasks (Fig. 3a)** and the analyzed number of up to 512 PEs. The software approach reveals a scalability bottleneck due to a locking in the task mapping part of the scheduler. The speedup for the centralized approach Nexus++ does not scale beyond 64 PEs. The **horizontal benchmark (Fig. 3b)** contains more coarse-grained tasks and has a maximum parallelism of 68. The parallelism is almost fully exploited by DRACON. Nexus++ benefits from the larger task size and exhibits good performance. The software approach must compete for computing power at the PEs and requires a much higher number of PEs to achieve the same speedup. The **sparse-matrix solver (Fig. 3c)** contains moderate inter-task communication and has a theoretical speedup limit of $S = 15.78$ due to it's critical path in the task graph. For 16 PEs both DRACON as well as Nexus++ reveal their strengths due to its parallel scheduling capacities. For a larger number of PEs the distance between communicating tasks increases and the speedup slightly decreases.

In Tab. V we provide values for the gate-level area overhead of one AGM, LMC and PE. The values have been obtained using an industrial 65nm low-power process. Each AGM and each PE contain one dedicated RISC processor which is implemented as mLite/PLASMA CPU [19]. We consider 32-Bit RX buffers for message communication inside each AGM. Assuming a system of 256 PEs and 16 AGMs the RX buffers have a size of 16 entries which gives one RX entry per connected node at the local and the global dedicated interconnect. To address the impact of on-chip memory we also consider 4kB of SRAM per PE and AGM.

TABLE V: Area [μm^2] for 65nm technology

Unit		Total	Comb.	Non-comb.	f_{max}
AGM		81003.8	20072.8	60931.0	502.5 MHz
	core	30397.3	17313.1	13084.2	
	mem	50606.5	2759.7	47846.8	
PE		63367.9	14246.2	49121.7	529.1 MHz
	core	27627.8	14241.9	13385.9	
	mem	35740.1	4.3	35735.8	
LMC		858.6	375.5	483.1	543.4 MHz

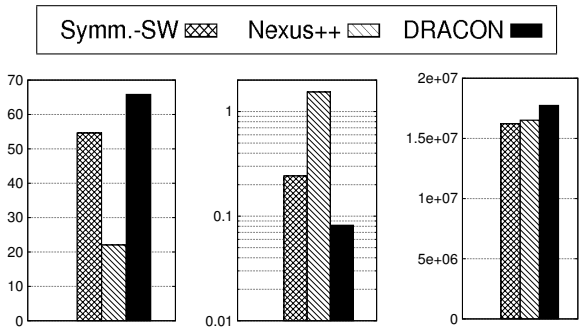
As a final evaluation we run all of the three benchmarks simultaneously for 10 iterations on a many-core system containing 256 processing elements. The DRACON network is configured to use 16 AGMs and the threshold for the cluster status communication has been set to $\Delta n_{th} = 4$. We measure the linear average for the application speedup $\overline{S}$ and the standard deviation σ of the normalized application response-time t_{par}/t_{seq}. The results of the simulations and the approximated total area are given in Fig. 4. The DRACON architecture has an average speedup $\overline{S}$ which is 20.56% higher compared to the software approach while at the same time the standard deviation is reduced by a factor of 2.96. The centralized approach Nexus++ is simply overburdened due to the large number of PEs. The area overhead of DRACON is approximated to be 9.4% compared to a common software approach.

VI. Conclusion

This paper presents DRACON, a scalable network of control nodes for run-time task management on embedded many-cores. The network of control nodes performs the communication and computation for task management in parallel to the processing cores. This is exploited to prepare task management decisions without disturbing the user application and to improve the responsiveness of the system.

TABLE VI: DRACON messages. Size is given in 32-bit words

Name	Size	Parameters	Dir.	Description
`msg-task-start`	2+	`tskd`, `stack`, `size`	AGM to LMC	Start task at destination LMC by means of the task descriptor (`tskd`), stack pointer (`stack`). The message may contain additional data words
`msg-task-spawn`	4	`parent`, `imem`, `dmem`, `cnt`	LMC to AGM	Spawn new tasks of count `cnt` with given parent, instruction- and data memory address
`msg-task-give`	4	`parent`, `imem`, `dmem`, `rcsv`	AGM to AGM	Request start of a tasks at another AGM. The parameter `rcsv` is required to implement the recursive task fork mechanism
`msg-task-exit`	2	`tskd`, `join`	LMC to AGM, AGM to AGM	Terminate task and/or signalize to join barrier `join`
`msg-beacon`	1	`status`	AGM to AGM	Broadcast load status
`msg-info-recv`	2	`tskd` , `key`, `cnt`	LMC to AGM	Let task receive `cnt` synchronization values for `key`.
`msg-info-send`	2	`key`, `value`	LMC to AGM, AGM to AGM	Send synchronization value to key
`msg-info-cast`	1	`key`	AGM to AGM	Broadcast key position



(a) Speedup (b) Deviation σ (c) Area [μm^2]

Fig. 4: Comparison of speedup, deviation of response-time and total area for a symmetric software-approach, a centralized hardware task manager (Nexus++) and DRACON on 256 PEs.

Our performance evaluation employs a transaction-level SystemC model and a task-based programming language. We implemented fundamental task scheduling and task synchronization algorithms to realize the distributed and hardware-enhanced task management. The results fortify the DRACON perspective to efficiently handle the complexity of run-time management for ultra-deep-submicron chip designs.

For 64 processing cores and horizontal task dependencies we achieve a 12% higher application speedup compared to a centralized hardware implementation and 70% higher speedup compared to a software approach. The 10% area investment for DRACON, although not negligible, is justified by the much higher performance outcome and better predictability. For 256 processing cores, DRACON outperforms a software approach by 20% in terms of speedup and has around three times smaller standard deviation of application response time when running a mixture of benchmarks.

REFERENCES

[1] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, ISSCC*, pages 88–598. IEEE, 2008.

[2] S. Borkar. Thousand core chips - a technology perspective. In *DAC*, 2007.

[3] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. 2010.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[5] T. Dallou and B. Juurlink. Hardware-based task dependency resolution for the starss programming model. In *Parallel Processing Workshops (ICPPW), 41st International Conference on*. IEEE, 2012.

[6] B. D. d. Dinechin, P. G. d. Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[7] D. Gregorek and A. Garcia-Ortiz. A transaction-level framework for design-space exploration of hardware-enhanced operating systems. In *International Symposium on System-on-Chip (SOC 2014)*. IEEE, 2014.

[8] N. Gupta, S. Mandal, J. Malave, A. Mandal, and R. Mahapatra. A hardware scheduler for real time multiprocessor system on chip. In *VLSI Design, 2010. VLSID'10. 23rd International Conference on*, pages 264–269. IEEE, 2010.

[9] A. Herkersdorf, A. Lankes, M. Meitinger, R. Ohlendorf, S. Wallentowitz, T. Wild, and J. Zeppenfeld. Hardware support for efficient resource utilization in manycore processor systems. In *Multiprocessor System-on-Chip*, pages 57–87. Springer, 2011.

[10] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[11] J. J. Labrosse. *Microc/OS-II*. R & D Books, 1998.

[12] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000.

[13] J. Lee, C. Nicopoulos, H. G. Lee, S. Panth, S. K. Lim, and J. Kim. Isonet: Hardware-based job queue management for many-core architectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 21(6):1080–1093, 2013.

[14] L. Lindh. Fastchart-a fast time deterministic cpu and hardware based real-time-kernel. In *Real Time Systems, 1991. Proceedings., Euromicro'91 Workshop on*, pages 36–40. IEEE, 1991.

[15] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang. A noc traffic suite based on real applications. In *VLSI (ISVLSI), IEEE Computer Society Annual Symposium on*, pages 66–71. IEEE, 2011.

[16] V. Nollet, D. Verkest, and H. Corporaal. A safari through the mpsoc run-time management jungle. *Journal of Signal Processing Systems*, 60(2):251–268, 2010.

[17] M. Ojail, R. David, Y. Lhuillier, and A. Guerre. Artm: a lightweight fork-join framework for many-core embedded systems. In *Conference on Design, Automation and Test in Europe, DATE*, pages 1510–1515. EDA Consortium, 2013.

[18] P. Reble, S. Lankes, F. Zeitz, and T. Bemmerl. Evaluation of hardware synchronization support of the scc many-core processor. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12), Berkeley, CA, USA*, 2012.

[19] S. Rhoads. Plasma-most mips i (tm) opcodes: overview. *Internet: http://opencores. org/project, plasma [May 2, 2012]*, 2006.

[20] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *50th Annual Design Automation Conference (DAC)*. ACM, 2013.

[21] O. Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, 2007.

[22] M. Sjalander, A. Terechko, and M. Duranton. A look-ahead task management unit for embedded multi-core architectures. In *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*, pages 149–157. IEEE, 2008.

[23] F. Thabet, Y. Lhuillier, C. Andriamisaina, J.-M. Philippe, and R. David. An efficient and flexible hardware support for accelerating synchronization operations on the sthorm many-core architecture. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 531–534. IEEE, 2013.