Noname manuscript No.
(will be inserted by the editor)

Towards efficient execution of RVC-CAL dataflow programs on multicore platforms

Ilkka Hautala · Jani Boutellier · Teemu Nyländen · Olli Silvén ·

Received: date / Accepted: date

Abstract The increasing number of cores in System on Chips (SoC) has introduced challenges in software parallelization. As an answer to this, the dataflow programming model offers a concurrent and reusability promoting approach for describing applications. In this work, a runtime for executing Dataflow Process Networks (DPN) on multicore platforms is proposed. The main difference between this work and existing methods is letting the operating system perform Central processing unit (CPU) load-balancing freely, instead of limiting thread migration between processing cores through CPU affinity. The proposed runtime is benchmarked on desktop and server multicore platforms using five different applications from video coding and telecommunication domains. The results show that the proposed method offers significant improvements over the state-of-art, in terms of performance and reliability.

Keywords Dataflow Process Networks · RVC-CAL · Orcc · Multicore

1 Introduction

In recent years multicore computing has spread rapidly to all mainstream computing. The reason behind the recent popularity of multicore originates from the fact that performance improvements with constant wattage are not achieved anymore simply by increasing clock frequencies [13]. Even mid-range smartphones can have quad-core processors and high-end mobile chips can have over 16 processor cores.

From the perspective of software development this evolution has set new challenges. Widely used software programming languages and development tools originate from the time when single core platforms and sequential programming models were mainstream. For this reason programmers have to invest a lot of effort in efficient mapping of their software on these multicore architectures. In addition, the huge amount of existing software, originally intended for single core platforms, is slowing down the adoption of novel programming models and computer architectures that would be better suitable for parallel programming.

Dataflow is a well-known and widely used programming paradigm for expressing the functionality of signal processing applications. Dataflow applications are modeled as directed graphs, which consist of actors as vertices and unidirectional first-in-first-out (FIFO) channels as edges. Dataflow description of an application ensures application modularity, re-usability and also reveals its concurrency, which simplifies distributing the application execution to multiple processing elements. These characteristics enable efficient application development for multicore platforms without in-depth expertise of low level parallel programming techniques.

RVC-CAL is a standardized dataflow language based on the Dataflow Process Network (DPN) model of computation [16] and the CAL actor language [11]. The Open RVC-CAL compiler (Orcc) toolset is used to translate RVC-CAL dataflow descriptions to platform independent source code (C, LLVM) or hardware description language (Verilog) which can be compiled or synthesized to an executable or logic ports by the target platform's own compiler [25].

Ilkka Hautala, Teemu Nyländen and Olli Silvén
Department of Computer Science and Engineering
University of Oulu, Oulu, Finland
E-mail: {ilkka.hautala, teemu.nylanden, olli.silven}@oulu.fi

Jani Boutellier
Department of Pervasive Computing
Tampere University of Technology, Tampere, Finland
E-mail: jani.boutellier@tut.fi

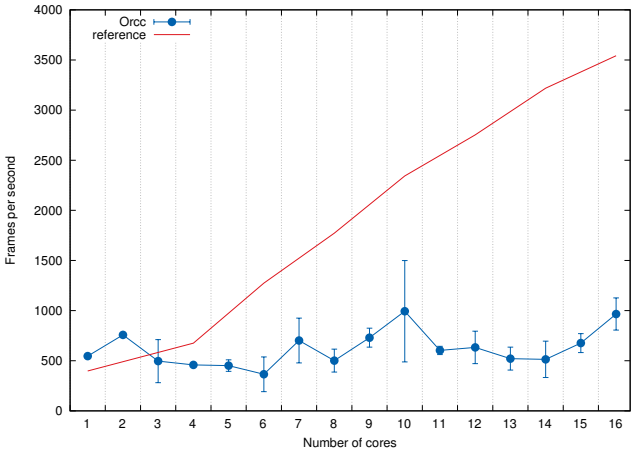

Fig. 1 Orcc multicore average performance in video motion detection application. Errorbars show standard deviation of four consecutive runs. The reference is the Distributed Application Layer (DAL) framework [6].

The Orcc C-backend can generate multithreaded C source code for multicore platforms [24] [23]. (From now on, when Orcc is mentioned in this paper it refers to Orcc C-backend) However, poor multicore performance of Orcc generated code has been reported for parallel applications [6]. Fig. 1. illustrates the problem demonstrating Orcc multicore performance in video motion detection. Firstly, distributing the Orcc-generated multithreaded application across multiple cores does not speed up execution as expected, and in some cases multithreading can even worsen the performance. Secondly, application execution times can have huge standard deviation.

Fig. 1. shows that the execution of RVC-CAL applications using the present Linux multicore runtime can yield very low speed-ups. The hypothesis behind this work is that such a low speedup is caused by one or both of the following reasons: a) the application has been divided to too many actors, which wastes computation cycles in unnecessary communication, b) the multicore runtime of RVC-CAL locks code to cores, which is a suboptimal choice. Whereas a) is a matter that has to be resolved either by revising the application descriptions, or by high-level compiler transformations [4], this paper addresses reason b). By proposing a new runtime that lets the OS freely assign code to cores, we show that an RVC-CAL application consisting of coarse grained actors can produce higher speedups than the state-of-the-art DAL framework.

This paper presents an efficient and compact solution to the aforementioned problems by introducing a multicore runtime which enables efficient execution of highly parallel and scalable DPN applications (implemented in the RVC-CAL language) on multicore systems. The essential contributions of this work are:

- A multicore runtime with up to 4 × performance improvement compared to current state-of-the-art for DPN execution, Orcc
- Extensive performance benchmarks using three multicore platforms and five RVC-CAL applications.

This paper is organized as follows: Section 2 gives a brief introduction to RVC-CAL and reviews related work. Next, the proposed multicore runtime is presented in Section 3. Section 4 presents the results of our experiments. Finally, the results and future work are discussed in Section 5.

2 Background

In RVC-CAL, a dataflow application consists of actors which are communicating with each other using FIFOs. An actor can *consume* (read) data tokens from its input FIFOs and *produce* (write) processed data tokens to its output FIFOs. The computations that an RVC-CAL actor can perform are implemented to actor-specific *actions*. The order of action executions is determined by 1) action-specific firing rules, 2) the current state of the actor, and 3) availability of input tokens. Actors communicate with each other solely over FIFO buffers, which effectively decouples actors from each other and enables concurrent execution. This kind of actor gives freedom to make choices how actors are scheduled by separating scheduling of actors from algorithm described by actor network and enables concurrent execution, as well as a multitude of scheduling and mapping possibilities..

On multicore platforms, actors can be executed in parallel on different processing elements to improve performance. However, when the number of actors exceeds the number of cores on the platform, multiple actors have to be mapped to the same core. Finding an efficient execution model in case of a complex dataflow application can be a challenging task, including problems like finding an efficient mapping of actors to cores of a homogeneous or heterogeneous platform and determining efficient execution schedule for actors on a processing core. [7]

There are a lot of papers in literature, which are describing different types of mapping algorithms [22] [19]. Mapping algorithms can be divided into dynamic and static ones. The dynamic algorithms are observing processor load at runtime and based on that execution of an actor can be transfered to another core on the fly. Static mapping algorithms make the actor-to-core

mapping decision at design time (offline) and mapping is constant during execution.

Scheduling of dataflow graphs has been a popular research topic and different scheduling methods for different dataflow models of computation have been proposed extensively, for example [17], [18], [5] and [3] to name a few. Recently, there has been some research on executing RVC-CAL applications on multicore platforms. In [24] Yviquel et al. presented several scheduling techniques to execute dynamic dataflow programs on multicore systems. They used a distributed scheduler scheme, where each processor core has its own local scheduler. Actors of the dataflow network are statically mapped to different cores and to their local schedulers. To arbitrate actor executions on an individual core, the local scheduler uses round-robin (RR) or RR combined with data-driven / demand-driven (DD) scheduling strategy. In DD scheduling, an actor, waiting data to arrive to its input FIFO or free space on its output FIFO, can request the local scheduler to schedule the predecessor of an actor (FIFO is empty) or successor (FIFO is full). If the requested actor is not mapped onto the same local scheduler (same core), communication between cores is required to schedule the actor.

Yviquel et al. extended their work by presenting a runtime actor mapping methodology [23], where they use metric based actor mapping. The metrics used are the available computational resources of the target platform, communication between actors and actors' workload. Communication between processor cores can be reduced by mapping actors, which have a lot of connections between each other, to the same processor core. Actor workload can be used to balance the workload of processor cores and thus reduce waiting times due to data dependency. To measure actor workloads they use low-cost profiling and therefore the target platform has to support these profiling mechanisms. The authors have evaluated the metric based actor mapping using RVC-CAL dataflow descriptions of three different video decoders: MPEG-4 Part 2 Simple Profile, MPEG-4 Part 10 Progressive High Profile (AVC) and High Efficiency Video Coding (HEVC). The current implementation of Orcc is based on these two aforementioned works [24] [23].

Chaverrias et al. [10] presents an automatic tool to assist in actor mapping to cores. Their tool simulates and profiles all possible mappings and recommends the best in terms of speed. First they reduce the number of possible mappings by forming groups of actors. When forming the groups, they are exploiting hierarchy of the RVC-CAL actor network descriptions. The formed groups are assigned to cores and profiling is performed using each possible group-core combination ($\#cores^{\#groups}$). The combinations that provide a performance improvement over the single-core baseline (and above a user-specified threshold), are stored for later use.

Chavarries et al. [9] proposed a Open Multi Processing (OpenMP) based runtime for shared memory multicore platforms, which do not have operating system and multithreading support. Based on performance profiling, static actor network partitioning to processing elements is made by hand so that actors are divided to *scheduler routines* and each processing element acquires one routine. Each routine has a *#pragma omp section* inside a *#pragma omp parallel* block. When the Open MP program reach a parallel region, it forks as many independent OpenMP threads as there are *omp sections* defined in the *omp parallel*-block. *Scheduler routines* use a Round Robin-algorithm to test and execute actors. Threads are pinned to cores using thread affinity.

Boutellier and Ghazi [6] proposed the Distributed Application Layer [21] (DAL) based design flow to execute RVC-CAL programs on multicore platforms to address Orcc performance scalability problems. They translate RVC-CAL actors to DAL processes by using the *DAL Backend* of Orcc. One problem in this approach is that the DAL model of computation assumes that dataflow networks are Kahn process networks (KPN) [14] instead of DPN. Therefore, a restricted set of RVC-CAL descriptions can be translated into DAL descriptions. In addition, DAL has poor performance on low data rate applications [6]. These low data rate applications are, however, the most important RVC-CAL application domain.

Several other multicore dataflow programming frameworks are also available, for example the XKaapi runtime system [12] and the CnC programming model [20]. Lately, TensorFlow [1], a dataflow based programmable system for machine learning has been getting attention due to increasing popularity of neural networks. TensorFlow networks, as in many other dataflow frameworks as well, are static [2], whereas the RVC-CAL is the only notable framework where the DPN model of computation is realized.

3 Proposed multicore runtime system

Processor affinity, also used by Orcc, is a technique that forces a process or a thread to be executed only on designated processor cores. One approach for implementing processor affinity is the use of a *processor affinity mask*, which the process scheduler of an operating system (OS), responsible for assigning threads to processor

cores, takes into account in process and thread scheduling decisions. Potential benefits of processor affinity are based on data locality: a recently executed thread can find useful data in the processor caches also during its next executions.

Processor affinity is commonly used in preventing thread migration from one core to another in non-uniform memory access (NUMA) systems. In NUMA systems, processors have their own fast *local memory* access and slower *remote memory* resources. The remote memories are, in turn, other processors' local memories, connected to a shared memory bus. Therefore a thread migrating from its initial core to another core increases memory access times.

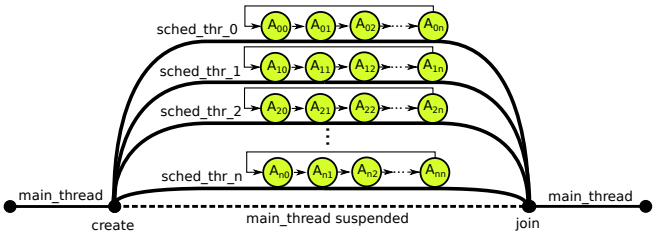

Fig. 2 Proposed thread creation and distributed round-robin actor schedulers.

On the other hand, processor affinity may interfere with load-balancing performed by the OS's process scheduler. The support for NUMA has been available for schedulers for a while and it is providing fair performance in typical applications [15]. Thus, processor affinity appears to be beneficial in a dataflow context only if the actor workloads are known and there are mechanisms available for balancing the actor workloads over different cores.

The main hypothesis behind this work is that processor affinity and runtime actor mapping [23], which have been adopted in Orcc, might be limiting multicore performance of Orcc by preventing OS managed CPU load-balancing. To test this hypothesis in practice, a compact multicore runtime system based on distributed RR actor schedulers was designed from scratch to manage execution of actors without processor affinity.

The proposed multicore runtime grants the user the possibility to decide how many threads are created. As in the work of [24], each created thread acts as a distributed scheduler and has a predefined set of actors to schedule. Static actor partitioning for schedulers can be done automatically, or alternatively allowing the user to decide actor partitioning over distributed schedulers. Each distributed scheduler fires its actors in a RR-fashion. After the threads of distributed schedulers have been created, the main thread is suspended until the created threads have finished. The POSIX Threads API is used for thread management; mutex and condition variables for synchronization. The OS's scheduler has the freedom to perform CPU load-balancing by thread migration. Fig. 2 shows thread creation and illustrates the RR actor scheduling. The scheduler is designed from scratch to be clean and simple and depending only from the POSIX thread library. In listing 1 the proposed runtime is described by using pseudo-code.

Listing 1 The runtime scheduler pseudo-code

```
scheduler(actors, #cores, premapping){
  actorsInThreads[#cores]

  IF(premapping):
      FOR(i : 0 to #actors)
          assign actors to
              actorsInThreads[i]
              respecting premapping
  ELSE assign actors to actorsInThreads[i
      ] automatically

  FOR(i : 0 to #cores)
      create_thread(thread(
          actorsInThreads[i]))
  wait_threads()
}

thread(actorsInThread){
  FOR(i: 0 to #actors):
      intialize actors[i]

  WHILE(!KILL_NETWORK):
      FOR(i: 0 to #actors):
              execute actors[i]
  exit_thread()
}
```

The focus of this work was not to find optimal actor to core mappings so our runtime only offers a naive static automatic actor partition algorithm. The partitioning algorithm of our runtime divides actors evenly for the distributed schedulers based on the total number of actors in the application. The resulting actor partitioning is static and the actor set of the distributed scheduler is not changing over time. However the proposed runtime works with the mapping tool presented in [10] which can produce more optimal mappings.

4 Experimental results

The performance of the proposed multicore runtime is measured using three different platforms presented in Table 1. The first test platform is a consumer grade desktop processor Intel Haswell i7 4770K @ 3.5GHz (Ubuntu Linux 14.04, gcc 4.8.4). The i7 consists of four physical and eight logical cores which each have 32KB

Table 1 Test platforms

id	platform name	type	log. cores	phy. cores	memory	operating system
i7	Intel i7 4770K @ 3.50GHZ	desktop	8	4	16 GB	Ubuntu Linux 14.04, gcc 4.8.4
Xeon	Intel Xeon E5-2650 v2 @ 2.60GHz	server	32	16	192 GB	CentOS Linux 7.2, gcc 4.8.5
Odroid	Samsung Exynos5422	mobile	8	8	2 GB	Ubuntu Linux 14.04, gcc 4.8.2

L1 cache, 256KB L2 cache, shared 8MB L3 cache over all cores and 16GB external RAM, with no NUMA. The second test platform is Intel Xeon E5-2650 @ 2.60GHz (CentOS Linux 7.2, gcc 4.8.5) which has two NUMA nodes. Both NUMA nodes have eight physical (16 logical) cores which each have 32KB L1, 256KB L2, 20MB shared L3 cache and 192GB external RAM. The third test platform is Odroid-XU4 powered by the mobile Samsung Exynos 5422 processor, which includes four ARM Cortex-A15 and four Cortex-A7 CPUs and utilizes the ARM big.LITTLE heterogeneous multi-processing solution.

Five different RVC-CAL dataflow programs have been selected as test applications: video decoders including HEVC, MPEG4 Simple Profile, MPEG4 AVC, video motion detection and a dynamic predistortion filter for telecommunications. Detailed descriptions of video motion detection and a dynamic predistortion filter can be found in [6]. The video decoders have been used in many studies and detailed descriptions of them can be found e.g. in [8] and [26]. Table 2 summarizes characteristics of the test case applications.

During the performance profiling there were no other processes running on the platform except critical OS related processes whose CPU-time is nearly zero according to the Linux **ps**-command.

Benchmarks for the proposed scheduler were performed using both predefined and automatic actor mapping. For comparison, Orcc benchmarks were performed using four different configurations: RR-scheduling using predefined and automatic actor mapping and DD-scheduling using predefined and automatic mapping. Predefined actor mappings were manually tuned and *identical mappings were used for benchmarking Orcc and the proposed scheduler*. The RR algorithm (Orcc default) was used for dynamic runtime actor mapping. Fig. 3 presents differences of Orcc performance between different configurations. In most of the cases the predefined mapping appeared to be better than automatic mapping on Orcc. To make figures clearer and easy to compare, Orcc automap results are omitted. Video motion detection and dynamic predistortion filter performance results on on the DAL based flow are extracted from [6] and therefore provide results only for the Xeon platform.

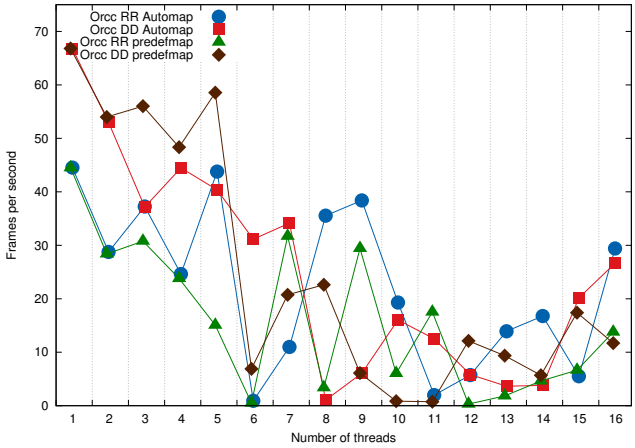

Fig. 3 Performance of four different configurations of Orcc C-backend in AVC application on the Xeon platform.

In Fig. 5 performance results on Intel i7 desktop and Intel Xeon server platforms are presented. Error bars show the standard deviation of the execution time, which is based on four consecutive runs of the application with the proposed configuration.

The i7 platform. On Intel i7 desktop processor Orcc with DD scheduler works slightly better than the proposed approach, provided that no more than four threads are used. When the thread count is increased over four, performance of Orcc starts to decrease significantly and execution time variance increases and it seems that performance decreases more in applications which have many actors. The best performance is always achieved using three or four threads. Meanwhile the proposed method can keep the performance level steady also with a higher number of threads. However, in AVC performance is varying a lot when thread count between six and eight is used. Only in video motion detection the best performance is achieved by using eight threads.

Performance differences between the proposed runtime and Orcc while using more than four threads can be explained by exploring the i7 architecture. The i7 processor exploits hyper-threading technology, in which one physical processor core appears as two logical cores that can be used for simultaneous execution of two threads. However, if two threads running on the same

Table 2 Properties of benchmark applications

Application	#Actors	#FIFOs	FIFO Size	Sequence
Video Motion Detection	4-16	4-28	256KB	-
Dynamic Predistortion Filter	15	48	64 KB	-
MPEG-4 SP	41	82	1024 KB	PussInBoots
AVC / H.264 Part 10 PHP	114	240	64 KB	Football 480p
HEVC / H.265 MP	38	64	64 KB	KristenAndSara 720p

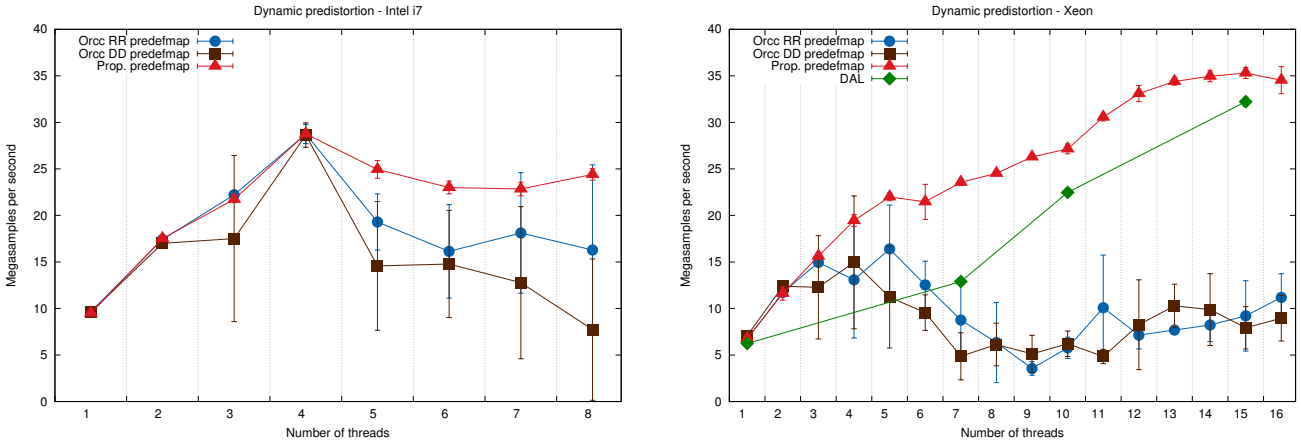

Fig. 4 Dynamic predistortion filter performance on i7 and Xeon platforms

physical core have similar workloads, execution can stall if the core runs out of resources. If execution of a thread is limited to a certain core, the OS is not able to balance the CPU-load by switching between different threads, which causes performance degradation.

To determine potential effects on power consumption when using to the proposed runtime, the Intel Vtune Amplifier 2018 is used to capture CPU loads during the execution of the dynamic predistortion application. In contrast to the ODROID platform, the measurements on i7 show that there is no clearly observable difference in the cpu loads. On both runtimes cpu load is near to maximum from beginning to end of the execution.

The Xeon platform. On Intel Xeon server processor the proposed runtime outperforms Orcc in all test cases. The Orcc runtime is not able to utilize more than three threads on the Xeon. The proposed runtime can exploit higher thread counts significantly better, always achieving peak performance with a thread count of five or higher.

The most significant difference is in video motion detection where the proposed method is almost four times better than Orcc. Additionally, standard deviations of consecutive executions using the proposed runtime are clearly smaller when compared to Orcc, excluding the AVC application. Video motion detection (Fig. 5) and dynamic predistortion filter (Fig. 4) benchmark figures for Xeon include Orcc DAL Backend results [6]. The proposed runtime is slightly better than the DAL based design flow in both of applications.

The Odroid platform. Fig. 6 presents measurements for the Odroid platform. The ARM big.LITTLE heterogeneous multiprocessing configuration includes four high performance cores and four lightweight cores, which emphasizes the significance of actor mapping over cores. The results show that by using identical mappings the proposed implementation clearly outperforms Orcc in all test applications. The Difference can be explained mainly by the fact that the Orcc runtime maps actors to threads and then pins threads to cores by in an ascending order starting from *coreid 0*. On the Odroid *core ids* 0-4 refer to Cortex-A7 cores and *core ids 5-7* to the Cortex-A15, which means that Orcc uses only Cortex-A7 cores when less than five threads are used.

When running the Dynamic Predistortion and Motion Detection application on the Odroid platform it was noticed that file I/O between the microSD card that contained the input data files and the processor caused a performance bottleneck. The problem was solved by reading input directly from the DRAM and writing output only to the DRAM and after that expected performance speed up was observed by increasing number of threads.

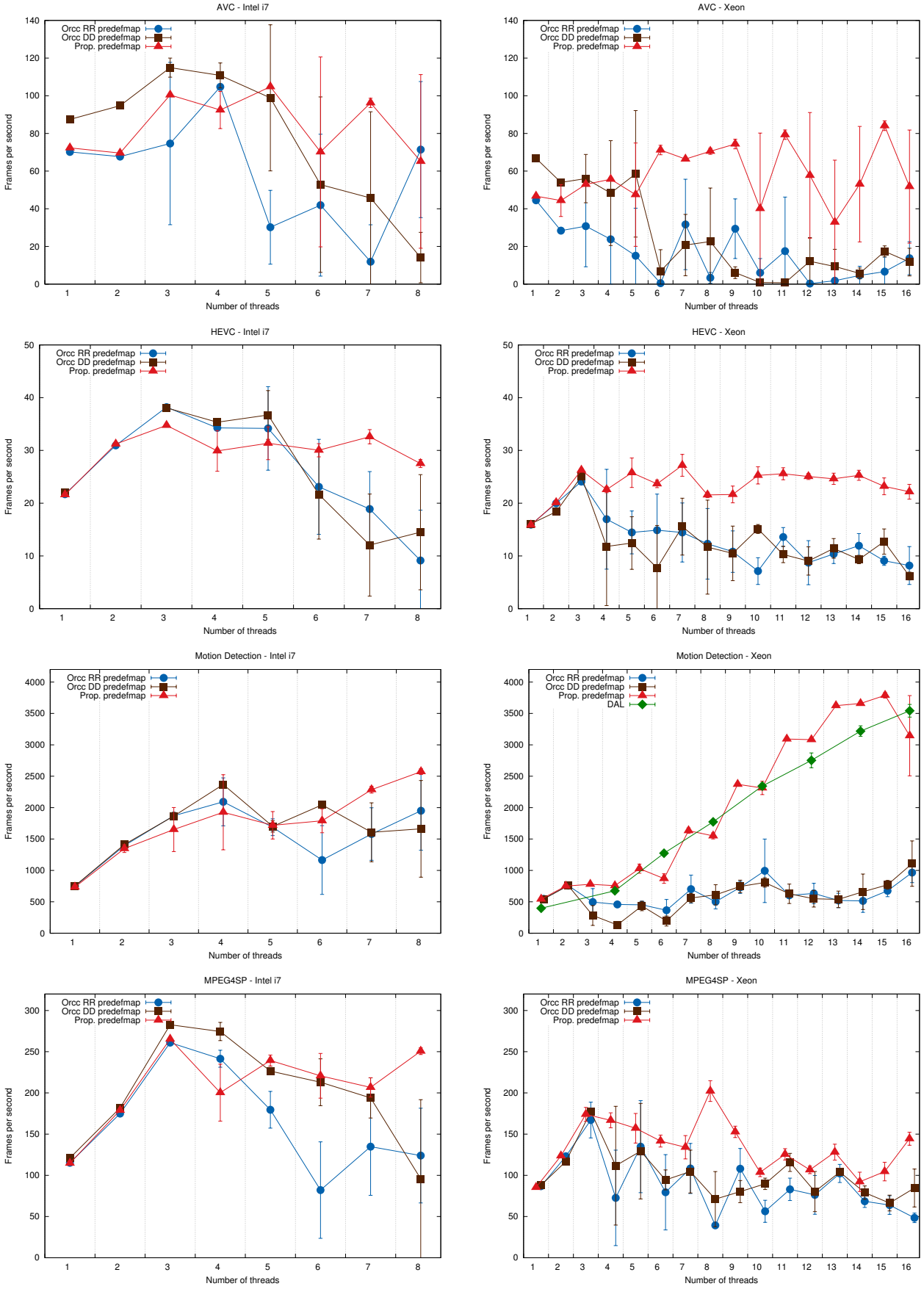

Fig. 5 Intel i7 and Xeon benchmark results

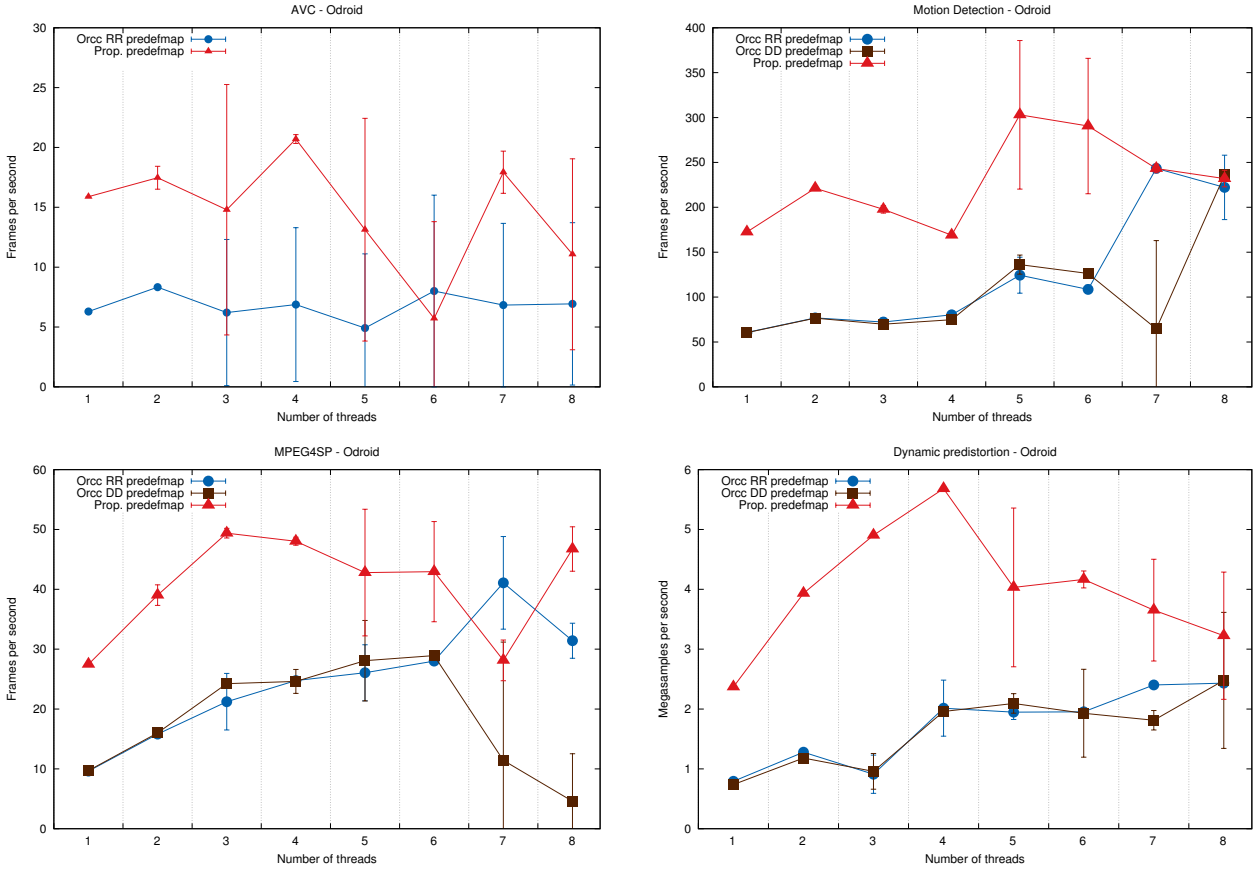

Fig. 6 Odroid benchmark results

Table 3 shows power and total energy consumptions of the proposed runtime and Orcc reference when running the dynamic predistortion application. The power figures include total power of the whole Odroid platform. Idle power of the platform is measured to be 4.3 watts (W). Energy is total energy that is consumed during application execution. The results show that the proposed runtime with four threads is the best setup alternative in terms of energy efficiency. Power figures also reveal that the Orcc runtime uses low-power Cortex-A7 cores in case of 1-4 threads as explained above. Increasing the number of Cortex cores in use increases the power consumption about 0.4 W per A7 core and about 1-2 W per A15 core.

The presented benchmarks show that disabling processor affinity and allowing the OS to make decisions on thread migration has to be seriously considered for efficient multicore execution of RVC-CAL programs. On the i7 processor the advantages are clear when more than four threads are used. Also on the Xeon with NUMA memory hierarchy, the proposed runtime clearly outperforms the current Orcc runtime. On the Odroid, which uses ARM heterogeneous multiprocessing configuration, disabling processor affinity also makes sense due to prominent performance gain.

5 Conclusions

Based on the results, enabling OS-managed CPU load balancing (as the proposed runtime does) provides superior results on the Xeon platform and the Odroid platform over fixed CPU affinity. In video motion detection and dynamic predistortion filter the proposed implementation illustrates how performance can be boosted by increasing the number of threads provided that the program can be efficiently divided for parallel execution. This suggests that for HEVC and AVC applications the RVC-CAL application descriptions form the parallelization bottlenecks rather than the runtime system itself.

In this work an efficient multicore runtime for RVC-CAL dataflow multicore execution is proposed. The proposed multicore runtime outperforms the reference approach with a clear margin in terms of performance and reliability (execution time variance). The proposed

Table 3 Odroid power consumption in Dynamic Predistortion application

	ORCC		PROPOSED	
#threads	**Power (W)**	**Energy(J)**	**Power(W)**	**Energy(J)**
1	4.8	201.12	7.6	108.68
2	5.2	138.84	10.4	88.4
3	5.6	131.6	12.2	90.28
4	6.2	99.2	13.8	84.18
5	8.8	105.6	13.6	179.52
6	11.6	205.32	14.4	118.08
7	13.0	185.9	14.4	109.44
8	13.8	197.34	13.8	193.2

runtime also outperforms the DAL based design flow [6].

Based on the results, it is fair to say that by using the proposed runtime without CPU affinity it is possible to achieve the same or even significantly better as with complex runtime actor mapping techniques like the one presented in [23] on the benchmarked platforms.

To further improve performance of the proposed runtime, the DD-algorithm could be adopted to the proposed scheduler. This is justified based on the results of Orcc which show that in most of cases DD performs slightly better than RR.

References

1. Abadi, M et al. (2016) Tensorflow: Large-scale machine learning on heterogeneous distributed systems In *arXiv preprint arXiv:1603.04467*.
2. Abadi, M., Isard, M. and Murray, D. Actor merging for dataflow process networks In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2017, pages 1–7.
3. Bonfietti, A., Benini, L., Lombardi, M., and Milano, M. An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 897–902.
4. Boutellier, J. Ersfolk, J., Lilius, J., Mattavelli, M., Roquier, G., and Silven, O. (2015) Actor merging for dataflow process networks In *IEEE Transactions on Signal Processing*, volume 63, number 10, pages 2496–2508.
5. Buck J. T. and Lee E. A. Scheduling dynamic dataflow graphs with bounded memory using the token flow model In *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing* pages 429–432.
6. Boutellier, J. and Ghazi, A. (2015). Multicore execution of dynamic dataflow programs on the distributed application layer. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 893–897.
7. Carlsson, A., Eker, J., Olsson, T. and Von Platen, C. (2010) Scalable parallelism using dataflow programming. In Ericsson Review, volume 2, number 1, pages 16–21
8. Chavarrias, M., Pescador, F., Garrido, M., Juarez, E., and Raulet, M. (2013). A DSP-Based HEVC decoder implementation using an actor language dataflow model. *IEEE Transactions on Consumer Electronics*, 59(4):839–847.
9. Chavarrias, M., Pescador, F., Garrido, M., Juarez, E., and Sanz, C. (2015). A multicore DSP HEVC decoder using an actor based dataflow model and OpenMP. *IEEE Transactions on Consumer Electronics*, 61(2):236–244.
10. Chavarrias, M., Pescador, F., Garrido, M. and Juarez, E. (2014). An automatic tool for the static distribution of actors in RVC-CAL based multicore designs. *IEEE 2014 Conference on Design of Circuits and Integrated Circuits (DCIS)*, pages 1–6.
11. Eker, J. and Janneck, J. (2003). Cal language report. Technical report.
12. Gautier, T., Lima, J. V., Maillard, N., and Raffin, B. (2013). Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel &Distributed Processing (IPDPS)*, pages 1299–1308.
13. Horowitz, M. (2014). Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14.
14. Kahn, G. (1974). The semantics of a simple language for parallel programming. *In Information Processing*, 74:471–475.
15. Lameter, C. (2013). Numa (non-uniform memory access): An overview. *Queue*, 11(7):40.
16. Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801.

17. Lee, E.A. and Messerschmitt, D. G. Static scheduling of synchronous data flow programs for digital signal processing *IEEE Transactions on computers*, 1987, 100(1):24–35.
18. Chao L-F. and Hsing-Mean S. Scheduling dataflow graphs via retiming and unfolding *IEEE Transactions on Parallel and Distributed Systems*, 1997, 8(12):1259-1267.
19. Sahu P. K. and Chattopadhy S. A survey on application mapping strategies for Network-on-Chip design *Journal of Systems Architecture*, 2013, 59(1):60–76.
20. Sbîrlea, A., Zou, Y., Budimlíc, Z., Cong, J., and Sarkar, V. (2012). Mapping a data-flow programming model onto heterogeneous platforms. In *ACM SIGPLAN Notices*, volume 47, pages 61–70.
21. Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S.-H., and Thiele, L. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proc. International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES)* , 2012, pages 71–80.
22. Singh, A. K., Shafique, M., Kumar, A. and Henkel, J. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *Proceedings of the 50th Annual Design Automation Conference*
23. Yviquel, H., Casseau, E., Raulet, M., Jääskeläinen, P., and Takala, J. (2013a). Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms. In *2013 8th International Symposium on Image and Signal Processing and Analysis*, pages 732–737.
24. Yviquel, H., Casseau, E., Wipliez, M., and Raulet, M. (2011). Efficient multicore scheduling of dataflow process networks. In *IEEE Workshop on Signal Processing Systems (SiPS)*, pages 198 – 203, Beyrouth, Lebanon.
25. Yviquel, H., Lorence, A., Jerbi, K., Cocherel, G., Sanchez, A., and Raulet, M. (2013b). Orcc: Multimedia development made easy. In *Proceedings of the 21st ACM International Conference on Multimedia*, pages 863–866.
26. Yviquel, H., Sanchez, A., Jääskeläinen, P., Takala, J., Raulet, M., and Casseau, E. (2015). Embedded multi-core systems dedicated to dynamic dataflow programs. *Journal of Signal Processing Systems*, 80(1):121–136.