A tightly-coupled Hardware Controller to improve scalability and programmability of shared-memory heterogeneous clusters

Paolo Burgio*†, Robin Danilo†, Andrea Marongiu*‡, Philippe Coussy† and Luca Benini*‡
*DEI – Università degli Studi di Bologna – Italy
†LabSTICC – Université de Bretagne-Sud, Lorient – France
‡Integrated Systems Laboratory – ETH Zurich – Switzerland

***Abstract*—Modern designs for embedded many-core systems increasingly include application-specific units to accelerate key computational kernels with orders-of-magnitude higher execution speed and energy efficiency compared to software counterparts. A promising architectural template is based on heterogeneous *clusters*, where simple RISC cores and specialized HW units (HWPU) communicate in a tightly-coupled manner *via* L1 shared memory. Efficiently integrating processors and a high number of HW Processing Units (HWPUs) in such an system poses two main challenges, namely, architectural scalability and programmability. In this paper we describe an optimized *Data Pump (DP)* which connects several accelerators to a restricted set of communication ports, and acts as a virtualization layer for programming, exposing FIFO queues to offload "HW tasks" to them through a set of lightweight APIs. In this work, we aim at optimizing both these mechanisms, for respectively reducing modules area and making programming sequence easier and lighter.**

I. Introduction

During the last decade, we witnessed the shift from single to multi- and then many-cores architectures for embedded systems. To improve energy efficiency and performance, designers are increasingly including application-specific units, to implement key computational *kernels* in ASIP or FPGA circuits. A recent trend is the one of isolating tightly-coupled *clusters* of cores and Hardware Processing Units (HWPUs), and in this template communication is implemented through shared memory banks, as shown in Figure 1. This is referred to as *zero-copy* data scheme [1] [4]: HWPUs have direct access to the on-cluster interconnection+shared memory system, rather than leveraging private memory buffers where data has to be copied before computation can start. Integrating a potentially high number of HWPUs in such a design raises two main issues, namely, architectural scalability and programmability. As the number of accelerators grows, so does the number of interconnection data ports, with both the negative effect of increasing its area, and in some the current technologies (such as the interconnections in STM STHORM [14] or Plurality HAL [18]) of creating longer critical paths, which reduce maximum achievable frequency target. Figure 2 models the area increase of a logarithmic interconnect modeled after Plurality [18], which grows quadratically with the number of connections. As an addition, application specific circuits often feature I/O parallelism, furtherly exacerbating this problem. To prevent the design becoming unmanageable, platforms reserve only a fixed number of interconnection ports for HWPUs, and these must be shared among them.

The second main challenge refers to programmability. Traditionally, embedded system programmers use unique IDs or "hooks" to reference HWPUs in their code, while the complexity of modern applications calls for new ways to efficiently expose them to the SW layer. A promising solution – following a hot trend in embedded system programming [16] [2] [13] – is to implement a queue system to support the offload of *HW tasks* to accelerators. To tackle these problems, in a previous work [3] we introduced the so-called **Data Pump (DP)**, which i) connects a high number of the HWPUs to the available interconnection data ports through a crossbar medium, and ii) provides a lightweight hardware queue system and its programming APIs for supporting application development. This work presents specific optimizations to increase performance of the module, at the same time reducing its area. Our first contribution is a methodology for efficiently connecting a high number of accelerators to the interconnection master ports. Doing this in a point-to-point manner introduces the previously described problems on area and critical path length. On-chip busses could be used, but they don't scale well with the number of connections, introducing an architectural bottleneck that hinders performance. We implemented a framework which profiles the traffic generated by each accelerator, and starting from a full crossbar, cuts away the unnecessary links. The resulting (ad-hoc) interconnection performs as a crossbar, but with minimal area. We call this technique *smart-sharing*. As a second contribution, we explored several task offloading strategies and optimizations aimed at reducing the area of DP and increase programmability. We validated our approach on a cycle-accurate Virtual Platform simulator, and we estimate the area gain for the DP module, for both the optimizations. They result in up to than 90% area saving, while keeping comparable performance to the baseline architecture and software stack. The paper is structured as follows. Section II gives a brief overview of works related to ours. Section III describes the target cluster. Section IV describes the Data Pump and our optimizations, whose effectiveness will be validated in Section V. Finally, Section VI concludes the paper.

978-3-9815370-2-4/DATE14/ © 2014 EDAA

II. Related Works

Tightly-coupled shared memory multi-core clusters have been adopted in several products, such as STM STHORM [14], Kalray MPPA [12], Plurality HAL [18]. It is therefore relevant to study heterogeneous evolutions of such clusters, where homogeneous processors leverage shared-memory communication with tightly-coupled HW blocks.

There are several notable architectures implementing shared-memory communication. AMD Fusion [1] is shared-memory GPU-based architecture, where the accelerator unit (APU) is not tightly coupled to cores, and hardware tasks are dispatched through a FIFO queue system. Carbon [13] provides a ring-based architecture where each node has its own task queue, and workload is balanced by work-stealing. However, in both of them, accelerators are loosely-coupled to cores. Fajardo et al [10] propose an system where cores and accelerators share a common L2 SRAM, called Buffer-Integrated-Cache (BIC). However, accelerators are not tightly-coupled at L1, and their

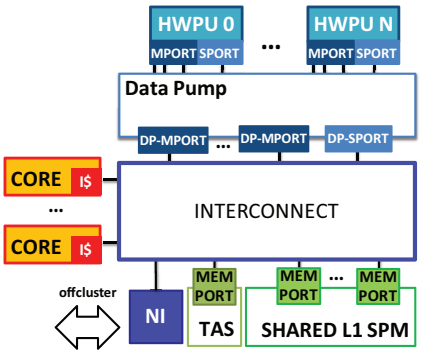

Fig. 1. On-chip shared memory cluster template

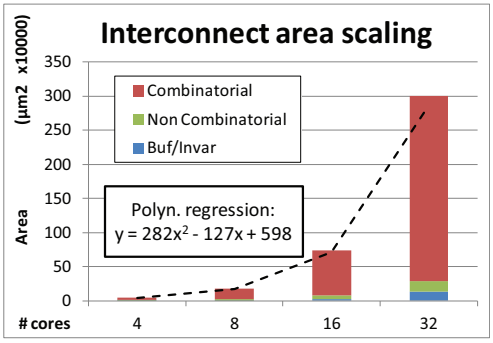

Fig. 2. Area increase of an Interconnection modeled after Plurality's

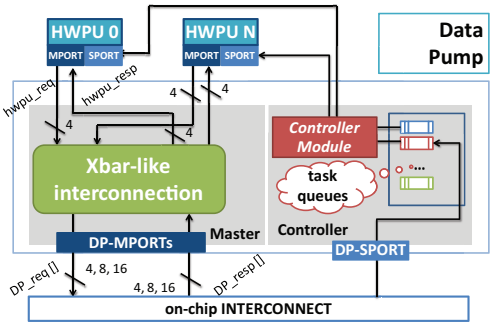

Fig. 3. Scheme of the Data Pump

design is not scalable to several accelerators, because their abstraction layer (*BIC substrate*) is implemented as a shared *"augmented conventional cache"*.
Some works attack the architectural scalability problem by providing smart controllers for the accelerators. Bin et al. [15] propose a Memory Access Engine (MAE) to hide the memory latency, as the number of HWPUs grows, with data prefetching. Hence, accelerators are not tightly-coupled neither to cores or to memory banks. Murali et al. [17] propose an optimization strategy for on-chip crossbars on MPSoCs, which is quite similar to our *smart-sharing*. Their scenario is however different: they connect general purpose cores to a memory space which is not completely shared. The work from Cong et al. [6] is probably the closest to ours from the point of view of programmability. They propose a hardware module (called GAM) for supporting the execution of accelerators, which allows *composing* the available HW blocks to realize complex *macro-tasks*. However, their accelerators are not tightly-coupled to cores.

III. Target Architecture

The heterogeneous shared memory cluster considered in this work is inspired by architectures such as STM STHORM [14] and Plurality HAL [18], and by works such as [4] or [9]. It is shown in Figure 1. It consists of an array of (up to 16) RISC-like processors with private instruction caches, plus a number of hardware accelerators called *HWPUs*. Cores and accelerators are connected through a low-latency, high bandwidth logarithmic interconnect similar to the one proposed by Plurality [18], and communicate through a multi-bank L1 scratchpad memory (SPM). Each memory bank of the SPM has a memory port, and overall the number of banks is a multiple K of the number of the master ports on the interconnect. The interconnect provides word-level address interleaving on the memory banks, aimed at reducing bank conflicts. If no bank conflicts arise, data routing is done in parallel for each core, and memory access happens in 2 cycles. Banking conflicts result in higher latency, depending on the number of conflicting requests. Synchronization among the processors is achieved through a segment of the local SPM address space featuring *test-and-set (TAS)* semantics. The L1 SPM has limited size, thus most of the program code and data are typically stored in larger L2 or L3 memory, while working data are moved back and forth e.g., via DMA transfers. In this work we consider a two-level memory system, with an off-cluster main memory.
HWPUs expose a set of memory-mapped registers for initiating an offload sequence, accessible through a slave port (*SPORT*). The semantics of accelerator execution is non-blocking: once a processor has successfully offloaded a task, it can asynchronously execute independent code. A generic HWPU embodies a *zero-copy* model, meaning that data is accessed directly from/to the shared SPM. Specific implementations might feature internal buffers to support data prefetching or post-storing, not considered in the current work. When the HWPU needs to access data from memory the corresponding transaction is appropriately packetized to match the interconnect protocol and brought to the accelerator master port (*MPORT*). Access requests to contiguous memory addresses are thus split into multiple parallel transactions on different ports. In the baseline template [4] [9], all HWPUs are connected to interconnection ports, thus their number can become very large, introducing all the practical problems discussed in the introduction (see Figure 2). However, as explained in Section I, the interconnection typically has a limited number of available connections: in [3] we introduced the Data Pump to deal with this. In this work, we consider 16 MPORTs for the interconnection (see [18], [7]), and we explore several configurations (i.e., number of DP-MPORTs and HWPUs) of the heterogeneous cluster. All HWPUs exploit I/O parallelism of 4, that is, they all have 4 MPORTs.

IV. Design of the Data Pump

The Data Pump (DP) module [3] interfaces between HWPUs and the on-cluster interconnection. It is split in two parts, called *Master* and *Controller* : the former handles the data paths between HWPU-MPORTs and DP-MPORTs, and the latter embeds a FIFO queues system for supporting offloading sequence. It is depicted in Figure 3. HWPU-MPORTs are connected to DP-MPORTs through a crossbar-like medium, whose area grows with the number of connections (similarly to what we show in Figure 2). We will now describe how we manage to keep this complexity low.

A. Optimizing the data connections: smart sharing

To optimize the connectivity between the HWPU-MPORTs and DP-MPORTs, we first profile the expected data traffic from different accelerators. This can be done either by statically analyzing how the HLS tool schedules operations and data accesses, or profiling data traffic with the help of a Virtual Platform. We follow the latter approach, that is, we simulated the target system and for each HWPU-MPORT we collected the number memory accesses, and divide it by the total HWPUs execution time (thus measuring its *throughput* in word/cycles). Then we model a constraint satisfaction problem as follows. Given:
N: number of HWPU-MPORTs
M: number of DP-MPORTs
H: number of HWPUs
FP_h: ID of the first MPORT of HWPU h
LP_h: ID of the last MPORT of HWPU h
$D_i, i \in \{1..N\}$: throughput on each HWPU-MPORT

We want to find the optimal assignment of HWPU-MPORTs to DP-MPORT, that is:
$X_{ij} \in \{0,1\}, i \in \{1..N\}, j \in \{1..M\}$

Where each HWPU-MPORT is connected to exactly one DP-MPORT, that is, the following Equation:

$$\forall_i \sum_{j=1}^{N} X_{ij} = 1 \tag{1}$$

And we aim at minimizing three partial cost functions:
1) to balance the throughput for each DP-MPORT:

$$\sum_{j=1}^{M} \left| \sum_{i=1}^{M} (D_i \times X_{ij}) - \frac{\sum_{i=n}^{N} D_i}{M} \right| \tag{2}$$

2) to reduce the combinatorial paths, we balance the number of HWPU-MPORTs connected to each DP-MPORT:

$$\sum_{j=1}^{M} \left| \sum_{i=1}^{M} X_{ij} - \frac{N}{M} \right| \tag{3}$$

3) different MPORTs for the same HWPU shouldn't be connected to the same DP-MPORTS, to avoid them interfering each other:

$$\forall_h \sum_{j=0}^{M} \left| \sum_{i=FP_h}^{LP_h} X_{ij} - (LP_j - FP_j + 1) \right| \tag{4}$$

Our global cost function is an equally weighted sum of the functions 2, 3 and 4. This optimization problem is quite generic, and applies to any scenario where a number of data requestors (being them either cores or accelerators) must be connected to a limited number of slave/memory ports. To estimate the area gain of the *smart-sharing* interconnection, we developed an area model of the DP. Figure 4 shows how DP area reduces – for different configurations of of DP-MPORTs and HWPU-MPORTs – compared to a "standard" full crossbar. The missing bars are for configurations with $1 \leftrightarrow 1$ connections, for which no optimization is needed. Our approach also applies to HWPUs whose traffic is not completely predictable (e.g., their behaviour is data dependant), by simply computing an average (or maximum) throughput requirement running several profiling simulations with different representative data sets.

B. Programming phases

The Data Pump acts as a virtualization layer for the accelerators, that is, it handles offloading requests by exposing a set of FIFO queues where tasks can be pushed using a set of lightweight APIs. To implement atomic updates, the requesting core acquires a lock (implemented using the TAS memory bank), enqueues the HW task descriptor in the FIFO, and then releases the lock [3]. This means that other cores requesting the DP are blocked until the full offload has completed, and is clearly extremely inefficient. Thus, we implemented an offloading strategy that we call **ASYNCH**, as opposite to the previously described **BASE** programming phase. A core which wants to offload requests a "slot" of the FIFO, and subsequently releases the DP. Then the core exclusively owns the FIFO element, and can fill the job descriptor without stalling other requesters. However, accessing the queue still happens through the DP-SLAVE port, and this causes a bottleneck. As a third variant, we propose to store the task descriptor queue in the on-cluster L1 SPM. The DP has a dedicated DP-MPORT to load it once the offloading sequence has completed. We call this the **SPM** programming phase. It has also the side effect of reducing the area of DP, because we don't store the queues in it anymore. Figure 5 shows the three programming phases, and figure 6 shows the area gain for the **SPM** programming phase over **BASE** and **ASYNCH**. As the number of DP-MPORTs and HWPU-MPORTs increases, so does the crossbar which connects them, reducing the area contribution of the *Controller* on the whole module. As previously explained, for some configurations no smart crossbar is needed (recall the missing bars in Figure 4). They are the white bars in Figure 6: in those configurations, the DP area is almost totally occupied by the *Controller*, resulting in almost 90% area saving on the whole module.

ARM v6 cores	Up to 16	DP-MPORTs	4, 8, 16
L1 SPM size	256 KB	# L1 SPM banks	16 (K=1)
L3 size	256 MB	L3 latency	$\geq$ 59 cycles
$I\$_i$ size	1 KB	$I\$_i$ line	4 words
t_{hit}	= 1 cycle	t_{miss}	$\geq$ 59 cycles

TABLE I. ARCHITECTURAL PARAMETERS

V. EXPERIMENTS

We prototyped the proposed heterogeneous cluster using a cycle-accurate Virtual Platform [8], with main architectural parameters as summarizes in Table I. With this setup, we validate both our optimization using two applications from image processing domains, namely a JPEG decoder and an application which performs the tracking of a color in an image. The first application performs an HW Inverse Discrete Cosine Transform, while in the second we accelerated the preprocessing Color Space Conversion (CSC) pass from RGB to YUV. We aime at validating our programming model and characterizing our architecture in terms of performance. We developed SystemC models for the DP and all the accelerators, as well as the RTL for DP using Calypto CatapultSL [5], to estimate its area. To support the *smart sharing* framework, we modified the simulator to statically configure the internal Data Pump MPORTs interconnections by loading them from a text file, and to dump throughput statistic for each of the HWPU-MPORT again on a text file. We solved the *smart sharing*

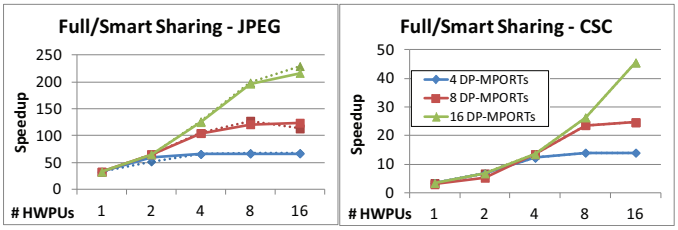

Fig. 7. Applications speedup for full and *smart sharing*

CSP with LocalSolver [11], a freely available programming solver based on local search. It accepts simple text files as input, and this simplifies the interaction with the simulator. Figure 7 Shows the speedup achieved over SW version for JPEG and CSC. Dotted lines refer to a system where HWPU-MPORTs and DP-MPORTs are connected with a "standard" full crossbar such as the one we presented in [3], while dashed lines show the results for *smart sharing*. We trade a negligible performance loss (or none at all for CSC) for up to 90% area saving of the overall DP module (Figure 4). Figure 8 shows the performance improvement of **ASYNCH** and **SPM** over **BASE**, for both the applications. For 4 DP-MPORTs there is no improvement due to the high memory boundedness of the JPEG application (i.e., more than 50%), that "hides" the benefits of **ASYNCH** and **SPM**. This is also the reason why we have a loss of performance for more than 4 HWPUs: 16 HWPU-MPORTs create a bottleneck towards the memory system. Figure 9 shows how for this experiment, **ASYNCH** and **SPM** programming phases are able to exploit the full bandwidth of the DP-MPORTs, while **BASE** reaches only $\approx$

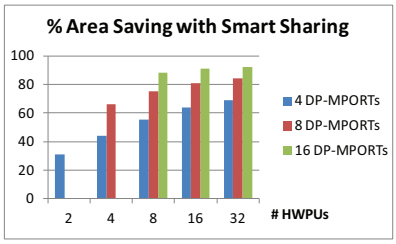

Fig. 4. Area saving with *smart sharing*

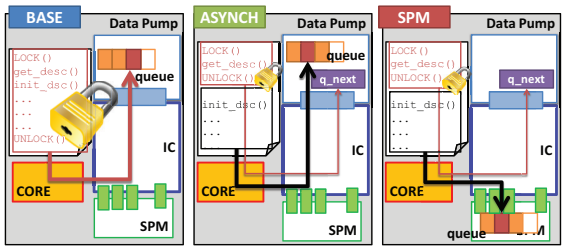

Fig. 5. The three programming phases

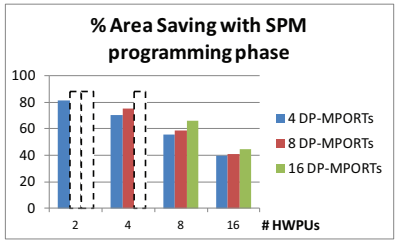

Fig. 6. Area saving with SPM prog. phase

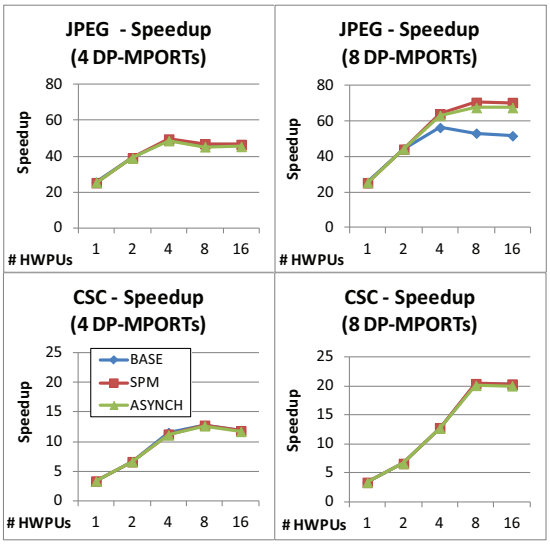

Fig. 8. Applications speedup for different programming modes

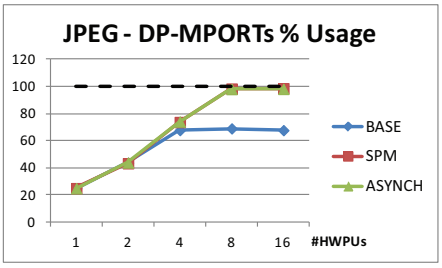

Fig. 9. DP-MPORTs utilization (8 DP-MPORTs)

70%. CSC kernel is less memory bound (i.e., $\approx$ %14) but it performs multiplcations which increase the granularity of the computation, hiding the benefits of **ASYNC** and **SPM**. However, the area gain with **SPM** is clear (see Figure 6).

VI. Conclusions

A promising trend in embedded many-core designs is to tightly-couple cores and hardware accelerators (HWPUs) in shared-memory clusters. To let cluster design scale the number of HWPUs, in [3] we introduced a *Data Pump (DP)* that multiplexes the data-paths from HWPUs and the on-cluster interconnection, and provides hardware support for queue-based programming. We propose two specific optimizations for the DP, estimating the area gain as compared to the baseline implementation, and measuring the performance with real image processing applications on a cycle-accurate virtual platform. Results prove the effectiveness of our optimization: the first reduces of up to 90% the DP cost with minimal performance loss, while the second relieves the performance bottleneck when multiple cores concurrently access the module, and at the same time reduces its area.

Acknowledgements

This work was supported by projects FP7 VIRTICAL (288574) and ERC-AdG MultiTherman (291125), funded by the European Community, and FUI-P, funded by French Fonds Unique Interministériel.

References

[1] AMD Inc. Fusion series.

[2] Apple, Inc. Grand Central Dispatch. http://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html.

[3] P. Burgio, A. Marongiu, R. Danilo, P. Coussy, and L. Benini. Architecture and Programming Model Support for Efficient Heterogeneous Computing on Tigthly-Coupled Shared-Memory Clusters. In *Design and Architectures for Signal and Image Processing (DASIP)*, 2013.

[4] P. Burgio, A. Marongiu, D. Heller, C. Chavet, P. Coussy, and L. Benini. OpenMP-based Synergistic Parallelization and HW Acceleration for On-Chip Shared-Memory Clusters. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 751–758, 2012.

[5] Calypto Design Systems Inc. Catapult family. [Online] http://calypto.com/en/products/catapult/overview, 2013.

[6] J. Cong, M. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. Architecture support for accelerator-rich CMPs. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 843–849, 2012.

[7] Daniele Bortolotti et al. Exploring instruction caching strategies for tightly-coupled shared-memory clusters. In *System on Chip (SoC), 2011 International Symposium on*, pages 34–41, 2011.

[8] Daniele Bortolotti et al. VirtualSoC: a Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip. In *013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*, pages 2182–2187. IEEE, May 2013.

[9] M. Dehyadegari, A. Marongiu, M. Kakoee, L. Benini, S. Mohammadi, and N. Yazdani. A tightly-coupled multi-core cluster with shared-memory HW accelerators. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 96–103, 2012.

[10] C. Fajardo, Z. Fang, R. Iyer, G. Garcia, S. E. Lee, and L. Zhao. Buffer-Integrated-Cache: A cost-effective SRAM architecture for handheld and embedded platforms. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 966–971, 2011.

[11] Innovation 24. LocalSolver. [Online] http://www.localsolver.com/.

[12] Kalray Corporation. Many-core Kalray MPPA, 2012.

[13] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News*, 35:162–173, June 2007.

[14] L. Benini et al. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*.

[15] B. Li, Z. Fang, and R. Iyer. Template-based memory access engine for accelerators in SoCs. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 147–153, 2011.

[16] Massachusets Institute of Technology. The Cilk Project. [Online] http://supertech.csail.mit.edu/cilk/, 1998.

[17] S. Murali, L. Benini, and G. De Micheli. An application-specific design methodology for on-chip crossbar generation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1283–1296, 2007.

[18] Plurality Ltd. The HyperCore Processor.