Exploring Multi-banked Shared-L1 Program Cache on Ultra-Low Power, Tightly Coupled Processor Clusters

Igor Loi
DEI, University of Bologna
Bologna - Italy
igor.loi@unibo.it

Davide Rossi
DEI, University of Bologna
Bologna - Italy
davide.rossi@unibo.it

Germain Haugou
ST Microelectronics,
Grenoble - France
germain.haugou@st.com

Michael Gautschi
Integrated Systems
Laboratory ETZ
Zurich - Switzerland
gautschi@iis.ee.ethz.ch

Luca Benini
DEI, University of Bologna
(Italy) and
Integrated Systems
Laboratory ETZ (Switzerland)
luca.benini@unibo.it

ABSTRACT

L1 instruction caches in many-core systems represent a sizable fraction of the total power consumption. Although large instruction caches can significantly improve performance, they have the potential to increase power consumption. Private caches are usually able to achieve higher speed, due to their simpler design, but the smaller L1 memory space seen by each core induces a high miss ratio. Shared instruction cache can be seen as an attractive solution to improve performance and energy efficiency while reducing area. In this paper we propose a multi-banked, shared instruction cache architecture suitable for ultra-low power multicore systems, where parallelism and near threshold operation is used to achieve minimum energy. We implemented the cluster architecture with different configurations of cache sharing, utilizing the 28nm UTBB FD-SOI from STMicroelectronics as reference technology. Experimental results, based on several real-life applications, demonstrate that sharing mechanisms have no impact on the system operating frequency, and allow to reduce the energy consumption of the cache subsystem by up to 10%, while keeping the same area footprint, or reducing by 2x the overall shared cache area, while keeping the same performance and energy efficiency with respect to a cluster of processing elements with private program caches.

Keywords

Near threshold computing, Shared instruction cache, FDSOI

1. INTRODUCTION AND RELATED WORK

The computational requirements of embedded applications and their growing portable nature caused by the increasing dependence on batteries, force embedded digital systems to satisfy extremely tight cost, performance and power constraints. Tightly-coupled clusters of multiple simple processors operating in near threshold offer a promising solution to satisfy performance and energy requirements of embedded applications [6]. By exploiting the intrinsic thread-level parallelism of applications in an efficient way, it is possible to lower the supply voltage to improve energy efficiency, while maintaining the required level of performance.

Instruction fetch is primary concern for all low-power processor based architectures, including multi-cores, as it highly affects the three main driving requirements of embedded applications: performance, power and area (cost). Keeping the miss-rate of instruction cache as low as possible is mandatory to preserve performance. Unfortunately, the instruction cache can be responsible for a significant portion of the energy consumption [8]. Finally, even though the caches usually consist of densely packed transistors, the area assigned to on-chip caches can be a significant fraction of the entire IC. Although traditional caches are often found on processor-based embedded systems, many specialized cache structures have been studied in the past to reduce energy requirements. Such structures include filter caches [9], loop caches [11], L-caches [1], and zero-overhead loop buffers (ZOLBs) [7]. These structures reduce the pressure on the instruction cache exploiting instruction locality, and they are mainly effective for architectures involving big instruction caches.

Within the field of ultra-low power multicore systems composed of tightly-coupled clusters, shared instruction cache per cluster can be seen as an attractive solution to improve performance and energy efficiency while reducing area. These systems are usually programmed with parallel programming models such as OpenMP [12] and OpenCL [10] that create workloads where all processors execute small critical kernels on different data. In this scenario, private caches are usually able to achieve higher speed, due to their simpler design, but the smaller L1 memory space seen by each core induces a higher miss ratio. Moreover, the coexistence of multiple program copies in the system requires more bandwidth to main memory in case of multiple concurrent misses. In contrast, the shared cache can offer a lower miss ratio and better memory utilization (no copies) at the cost of slightly increased hardware complexity.

A well-known class of many-cores that shares the instruction cache among several compute units is that of General

ACM International Conference on Computing Frontiers 2015, May 18-21, 2015, Ischia, Italy

http://dx.doi.org/10.1145/2742854.2747288 .

Purpose Graphic Processing Units (GPGPU). GPGPUs exploit massively multi-threading in order to hide the latency of accesses to external memory [14]. All the compute units in each multiprocessor execute their threads in lock-step according to the order of instructions issued by the instruction dispatcher, which is shared among all of them [17]. The same approach, which leverages the intrinsic SIMD nature of applications, has been proposed by Dogan et.al. [6] in the field of deeply embedded multicores for health monitoring. The architecture couples a shared instruction memory with broadcast mechanism. This allows forwarding the same instruction to all the processors within the cluster whenever they all request the same address on the same cycle thus reducing instruction fetch energy. Moreover, a hardware synchronization mechanism allows to dynamically manage the lockstep execution of cores during data-dependent program flows [5]. Although this approach achieves 60% energy reduction, its applicability is restricted to a strictly data-parallel code sequences, and it is intrusive from the software viewpoint, as it requires explicitly activating and deactivating lock-step execution. Moreover, although typical workloads executed by these systems are parallel, in many cases (e.g. code with conditionals) they do not execute the same instructions and forcing SIMD execution leads to efficiency drop due to loss of parallelism. In a previous work, an in-depth study of the private and shared cache architectures based on a SystemC platform running both micro-benchmarks and real OpenMP applications has been carried out [3].

In [4], a multicore processor for server and desktop markets, it is presented a sharing mechanism applied to L1 instruction cache within the CMT (Clustered Multi-Thread or module block). A single cache bank is used to feed instructions to a central instruction decoder that dispatches them to the two integer units and FPU. These cores are quite large and clocked at high speed to achieve high performance at the expense of very high power consumption (from 10W to 125W). Moreover the sharing infrastructure is not scalable with the number of cores, since this approach becomes inefficient (increase contentions in the fetch interface) when number of cores is greater than 2 (because of the instruction cache contention issues on the fetch side). Our target in this paper is a sharing mechanism for program caches, tailored for ultra low power systems, with small processing elements, and where energy efficiency is the first goal to achieve.

In this work, we describe for the first time a fully functional design and we present an accurate assessment of performance, area, energy. Results are based on the physical implementation of both the private and shared instruction cache. Summarizing, the main contributions of this work are:

- A shared instruction cache architecture and complete design for clustered ultra-low power multi-core clusters systems.
- A detailed analysis of the implementation of the proposed shared cache in several configurations including post place & route analysis of performance, area and power metrics.
- Several architectural variants and a comparison in terms of performance with respect to the private configuration with several signal processing applications.

The rest of the paper is organized as follows. Section 2 gives a description of private and shared program cache based, multi-cluster system on chip. Section 3 describes the internal components of the shared program cache system. Section 4 illustrates the experimental setup and results, and finally section 5 gives the conclusions.

2. SYSTEM OVERVIEW

This section presents the overall system architecture where the proposed shared program cache architecture is deployed, providing also a description of its integration in the multi-core cluster.

2.1 Target cluster architecture

An ultra-low power tightly coupled cluster is composed by several Processing Elements (PEs) that usually embed private data and instruction caches, and usually share a L2 cache. These cores are often grouped in clusters of processing elements as depicted in Figure1. A global interconnect provides the communication infrastructure between the clusters, the L2, the SOC peripherals and the IOs. The baseline cluster architecture considered in this paper is presented in Figure 2. It features a parametric number PEs composed of OR10n cores (OpenRISC OR1200 core evolution [15]), each one with a 1KB private, direct mapped instruction cache. No data caches are present, therefore avoiding any memory coherency overhead and increasing area efficiency [2]. PEs share a L1 multi-banked Tightly Coupled Data Memory (TCDM) acting as a shared data scratchpad memory. The refill ports of the instruction caches converge on a common cluster instruction initiator port through a AXI4 crossbar (refill node).

Private caches generally improve the overall performance of individual cores, at the expense of stringent constraints on their capacity, with higher power consumption even when the cores are in idle (due to leakage effects). The program (instruction) cache is slightly different from a data cache, because of its read-only nature. Program cache sharing is an attractive option that lead to increase the effective capacity of the I-cache, and at the same time can reduce the power consumption and improve performance. This is especially true for those applications that do not fit in the private cache, because of limited capacity. In such scenario, for every miss, the cores are blocked, waiting for the incoming refills from a L2 memory subsystem that generally is far in terms of latency cycles.

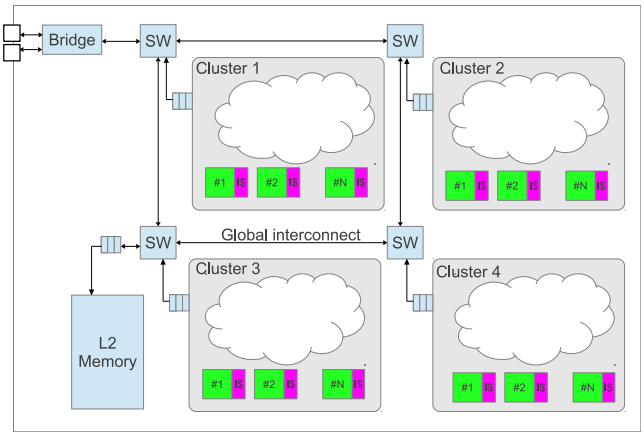

Figure 1: Overview of the multi-cluster, multi-core system on chip

A shared program cache must minimize the logic complexity in the processor fetch interface, and last but not least, must not introduce any additional latency cycle with respect to a private cache [1].
In the following subsections we refer to a single cluster composed by four OR10n cores [2], with 1KB of instruction cache for private configuration and a total capacity of 1KB and 4KB and fo the shared version. For both configurations we assumed a fetch data-width of 32 bit.

2.2 Private program caches

The instruction cache is deeply integrated in the processor pipeline, therefore its complexity is highly reduced, and then its interface is highly customized. It generally fetches one instruction per cycle and internally, only minimal information is tracked. After a miss, the PE is stalled and its cache blocked, until the refill comes back. Thanks to this behavior, the internal controller is very simple.

Data replication (multiple copies of program segments may exist in different caches) is the major drawback for multicore systems, which leads to a degraded energy and area efficiency. In such systems (e.g., in [2]) processors work in parallel and execute same kernel on different datasets, thus several cores fetch the same code segment, that is multiple stored in each private cache (copies). Therefore multiple refill requests are asserted at the same address, creating intense traffic to a slow and far L2 memory subsystem. As depicted in Figure 1 and 2, the cost for a single refill is given as sum of the internal latency of the cluster (the AXI4 refill node and cluster bus crossing in both directions), plus the amount of time to cross the global interconnect and reach the L2 and coming back to the cluster, and is usually on the order of magnitude of 10 - 40 cycles.

2.3 Shared program caches

Figure 3 illustrates the instruction cache sharing architecture, where private caches are removed from cores and shared through a very thin and fast crossbar interconnect based on logarithmic trees [16]. The processor fetch interface has been modified to match the flow control implemented in the low-latency interconnect, therefore caches have been designed to support multiple outstanding transactions and non-blocking behavior. Cache banks (I$) are linked to the off-cluster interconnect using an AXI4 crossbar to route and back-route refills from/to the cluster bus and then to the global interconnect (L2 memory subsystem).
To reduce the contention on cache banks during fetch requests, cache banks are interleaved at cache line granularity. Since cache banks are shared among PEs, the cache capacity per core is increased compared to its private configuration when we budget for area (e.g., 4 private 1KB vs. 4 shared 1KB cache banks), or if we budget for overall cache capacity, we can maintain similar performance while reducing silicon cost and energy consumption (e.g., 4x private 1KB vs. 4x shared 256B cache banks).

[1]Any additional latency cycle will stall the processor pipeline, leading to lower the Instruction Per Cycle (IPC) performance

[2]The number of cores is a degree of freedom of the cluster, and for sake simplicity we use 4 for our experiments

3. SHARED PROGRAM CACHES ARCHITECTURE

This section gives detailed description of the architectural modifications needed to support the shared program cache in a typical multi-cluster SoC with respect to the baseline architecture described in section 2.

3.1 Fetch core interface and L0 buffer

The core interface has been customized to work properly with a shared cache (handshaking and flow control), and the pipeline has been optimized to avoid any critical path from processors to the shared program cache. We added a core interface controller to monitor the flow control signals and to stall properly the pipeline in case of a miss. To reduce the pressure (and contention) on the shared program cache, we added an instruction buffer (L0) which is capable to hold 64 or 128bit of data (half or full cache line), therefore avoiding any access to the cache when the instruction to be fetched is available in this local L0 buffer. To avoid any IPC penalty, this buffer checks the local availability of that instruction, and in case of miss, the request is forwarded in the same cycle to the shared L1 program cache. Since this check is performed on the fly, in the same clock cycle this block has to check for local hit/miss and asserts a fetch to the program cache in case of local miss. Thus, the L0 buffer complexity

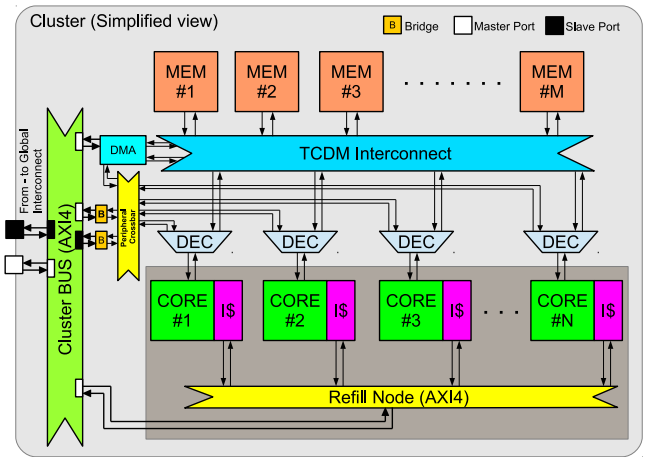

Figure 2: Overview of the cluster of processing elements with private program caches

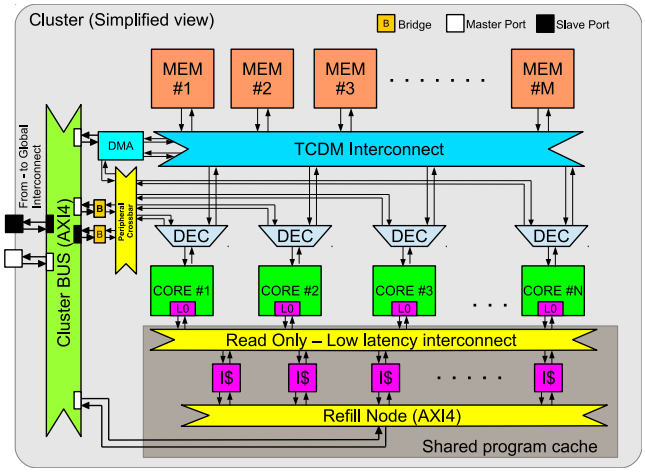

Figure 3: Overview of the cluster of processing elements with shared program cache

is reduced by hosting only one entry (one cache line).

3.2 Shared cache bank

The cache bank, as introduced at the beginning of this section, is designed to be non-blocking in case of miss, and to feed back the instruction in one cycle in case of hit. The first feature implies a slightly modified cache controller to handle multiple outstanding transactions and the need to track the outstanding refills (refill tracker). The second feature entails the adoption of a cache pipeline to process one fetch per cycle in case of hit, without blocking processors unnecessarily.
Figure 4 depicts the simplified view of the single cache banks. The cache controller is in charge of managing the pipeline, to proper route data from to DATA/TAG RAM and refill/response FIFOs.

TAG RAM is implemented with flip flops, while DATA RAM is a Standard Cell Memories (SCMs) [13], implemented with latches as base storage cell. Contrarily to traditional SRAMs, SCMs provide the key benefit of being functional in the same voltage range as the rest of logic, suitable for low power systems working with very low voltage supply. For instance, by lowering the cluster voltage (VDD) up to 0.3V, we can still guarantee the functianality of TAG and DATA arrays that compose the shared I-cache. This would not be possible with SRAM based I-cache, because they start failing below 0.6V.

In addition, with respect to standard SRAMs, SCMs consumes less energy per access [13]. For these reasons SCMs appear as a perfect candidate for the implementation of instruction caches for near-threshold multi-core architectures.

The refill tracker is a FIFO with a CAM (Content Addressable Memory) to search if the current transaction has already an outstanding refill associated. Finally the AXI4 FIFOs are inserted to decouple the critical path to a complex crossbar (AXI4), and to have some storage to alleviate the congestion issues during multiple refill assertions.

3.3 Low latency interconnect

The low latency interconnect [16] is a read only crossbar featured by N target ports and M Initiator ports. It routes in a combinational fashion the requests from PEs to the various cache banks. Initiator ports are mapped with cache line interleaving. The arbitration between different PEs that concurs to the same cache bank is made though distributed round robin arbiters, and completely combinational. All the

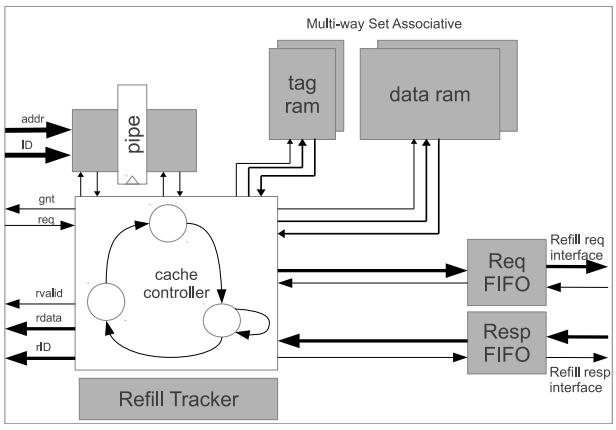

Figure 4: Single cache bank block diagram for the shared program cache

requests are arbitrated in the same clock cycles that are asserted.
Figure 5 presents a schematic view of the interconnect architecture, for a configuration of 4 initiator ports (e.g., PEs), 2 target ports (cache banks) 64 bit data-width and multi-cast support. This Intellectual Property (IP) block has been derived from [16], and customized to support caches. The arbitration is performed through distributed round robin arbiters (request blocks), arranged in a binary tree in order to reduce logic complexity to $log2(N)$ stages, where N stands for the number of target ports. The source ID is used to back-route the read data, using the Response Block.

3.3.1 Multi-cast feature

Programs tend to reuse instructions near those they have used recently, or that were recently referenced themselves. We extended the read-only low latency interconnect to exploit spatial locality, through a multi-cast mechanism. This approach allows to deliver (multi-cast), at the same time (in one clock cycle), the same instruction or block of instructions to multiple processors. The multi-cast mechanism aims to reduce the number of cache accesses to a shared cache, with a potential gain in power saving and performance. To support multi-cast, we extended the low latency interconnect with two blocks (refer to figure 5): the Mcast Check and the Mcast flow control handling block. Cycle by cycle, the Mcast check block compares its relative address to the others (N-1), and creates a match vector which carries the multi-cast information (valid multi-cast and matching sources). This vector is used in the same cycle to handle properly the flow control signals, and to deliver the data to the right PEs.

Let's consider figure 5, and let's assume a multi-cast example where target port 0 and 3 are fetching the same address (e.g., 0x0000). Both requests are sent to the Request Block 0, and only one wins the arbitration. Assuming that port 0 loses and port 3 wins, the request block 0, gives the grant to port 3 and stalls the port 0. The MCast flow control handling, then is listening all the grants that matches (valid multi-cast) with the current source port: for port target 0, it is considered as valid grant, both grant_0[0], and grant_0[3], and since port 3 gets the grant, this is forwarded to processor 0 as well.
Responses in case of hit come the cycle after the grant, while in case of miss, latencies are not deterministic because they depend on several factors that are not under control of caches (access to L2). The multi-cast flow control block, manages the responses in the same way it handles the grants. To perform this task, it samples the multi-cast match vector (during the arbitration stage), and then listens the response valid signals (rval) to route back the right read data. Following the previous example, when the read data is available at the initiator port 0, then the MCast flow control block listens both rval_0[0]and rval_0[3]: Since the rval_0[3]is asserted, this block informs the processor 0 (rval_out[0]) that the read data is available at its fetch interface. Once the processor samples the read data, the MCast flow control block clears its registers.

We also extended the multi-cast granularity, in order to better improve the program spatial locality. The multi-cast granularity is defined as the fetch-size that we want to broadcast to different processors. The lower-bound granularity is given by the interconnect data-width, while the upper-bound is dictated by the cache line size. In this last case, any simultaneous fetches that belong to the same cache line, can be multi-casted, in the same clock cycle, to different processors while keeping the same interconnect data-width (eg

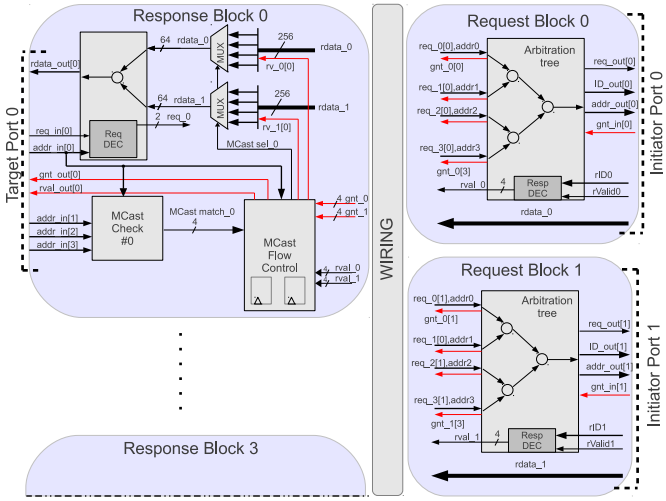

Figure 5: Details of the read only - low latency interconnect with multi-cast capability, for a configuration with 4 target ports, 2 initiator ports, 64bit wide, and multi-cast feature

32bit). Increasing the granularity implies that, the cache must export more than the nominal 32 bit, therefore on the cache side the interface is simplified (no cache line multiplexing), but on the other hand, the low latency interconnect interface becomes more complex because the multi-cast multiplexer are needed to select which data chunk (group of 32 bit) must be sent back to the processor. For an NxM interconnect, when the granularity is greater than the data-width (e.g., 128 for a 32bit interconnect) there are NxM multi-cast 4 ways multiplexers.

4. EXPERIMENTAL SETUP

We carried out the exploration for different cache configurations, for a baseline system composed by a single cluster of four processors (OR10n cores derived from the OperRISC OR1200 architecture). First we analyzed the private program cache system (state of the art), and then we compared it to several shared cache based systems. We divided the exploration in two scenarios:

- 4KB shared configuration: moving from a private to a shared configuration while keeping the same area footprint (basically the 4 private caches become single and shared , with 4X more capacity), leading to performance improvements (reducing the miss ratio).
- 1KB shared configuration: moving from a private to a shared configuration while keeping the same overall cache capacity, and leading to save some silicon area.

In both scenarios, we explored the effects of several parameters that influence the conflict ratio on the cache interconnect, and the number of cache access (L0 buffer that acts as cache filter). Finally we compared the main metrics (area, power and runtime) between private and shared program caches.

4.1 Performance evaluations

To evaluate performance, we performed simulations on the cycle-accurate RTL model of the cluster, with several bare-metal multi-core benchmarks running on the four PEs. To carry out a fair comparison between the private and shared cache, we have chosen as benchmark, kernels that fit the capacity of both configurations. Obviously, considering benchmarks that fit the shared cache configuration, but not the private one (e.g., 2KB of benchmark code, 1KB per core of private cache, 4 KB shared) would polarize the results in favor of the shared cache configuration.

For all these benchmarks, the code is located in the L2 memory (with a latency of 10 cycles, refer to figure 1), while cache banks are direct mapped, with FIFOs at their minimum depth. Private caches are 1KB each, while in the shared configuration we swept the total capacity from 1KB to 4KB. For each capacity, we swept both the program cache interconnect data-width from 32bit to 128bit (and L0 buffer) and we turned on/off the multi-cast feature. The performance metric is calculated as the number of cycles needed to complete the main kernel of the running application, normalized on the private cache results. To measure the application runtime, we used an embedded timer.

Figure 6 shows for all the benchmarks, the ratio between the runtime for a specific shared configuration and the private program cache. A value of 100% indicates equivalent performance while a value lesser than 100% indicates a speedup. Fig 6a) is related to the 4KB shared cache scenario, while b) to the 1KB version. All the results are measured with hot caches. In Fig 6a, when multi-cast is not enabled, and the interconnect is 32bit wide (refer to figure 6a and 7), we note longer runtime (up to 41% in average). When the multi-cast is enabled and the interface is 32bit, the Dijkstra application gets a huge boost on the execution runtime. This happens thanks to the fact that this application consists of a very simple parallel loop, and all the cores always process data stored in different TCDM banks eliminating all forms of contention. Thus, once the cores are synchronized at the begin of the application, they always execute the same instruction in the same clock cycle. More in general runtime decreases but it is still not attractive with respect to private caches.

In the fourth group, the interconnect is 64bit wide and the L0 buffer holds 2 instructions (filter cache). In this configuration, we measured a speedup of 4% in average above the baseline configuration. In the fifth group, by enabling the multi-cast, the runtime decreases in average 9% with respect to private. With 128bit interconnect and L0 buffer (128 bit, 4 instructions), we measured a boost of 11%, while, when implementing the multi-cast for this configuration we boosted the speedup up to 12%.

Table 1 reports the total miss ratio for the whole execution of these applications. In Dijkstra, the miss ratio improvement is around 11x, and in average we measured an enhancement of 4x. This is due the fact that, in the shared cache configuration, there are no multiple copies (data replication), and the cache refills are requested only once time, thus reducing the pressure over the L2 memory subsystem.

Fig 6b) illustrates the results for the second scenario. Since the total shared capacity is equivalent to a single private cache bank, this configuration, in principle cannot achieve better results than private cache, because the capacity is the same, and the conflicts on the cache interconnect reduce the available bandwidth. Nevertheless, this configuration aims to reduce the pressure on the L2 subsystem (as discussed before). The shared 32bit configuration shows a penalty of 60% when compared to private cache, while implementing the multi-cast, this penalty drops to 11%. Increasing the interconnect width to 64bit and introducing a L0 buffer, helps to reduce the pressure on the shared cache interconnect. In this case the slow-down drops to 7% while im-

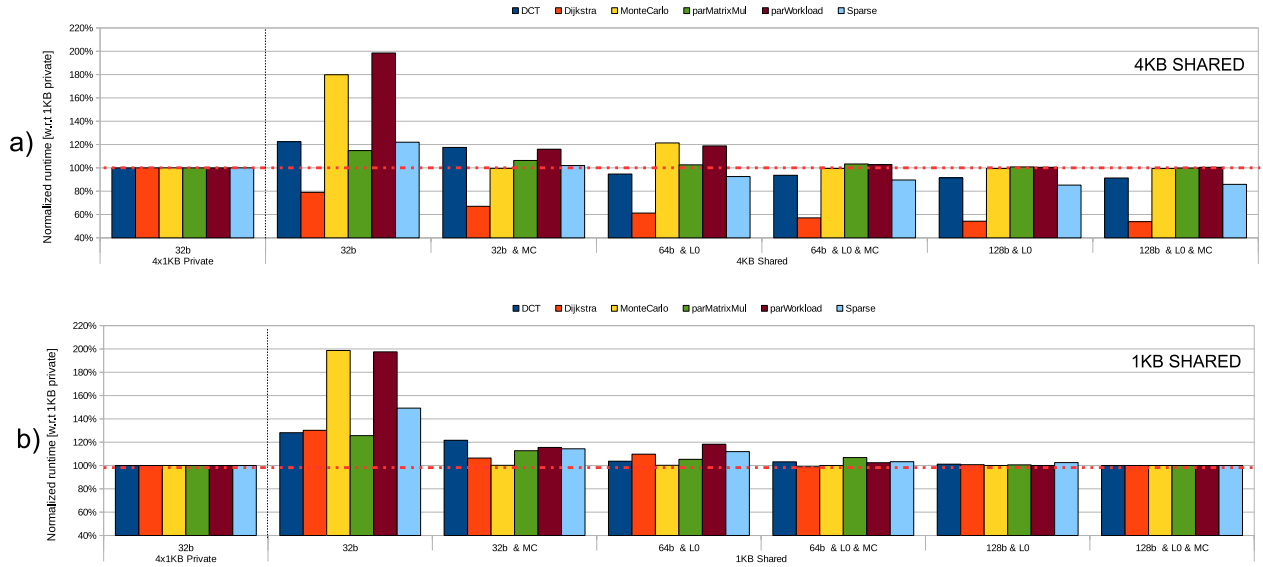

Figure 6: Normalized runtime (with respect to private cache configuration) for several benchmarks, and for different shared program cache configurations. 32b , 64b and 128b stand for the interconnect data-width, which is respectively 32,64 and 128 bit wide. L0 indicates the presence or not of the instruction buffer. MC stands for multi-cast feature implemented. a) shows the normalized runtime for the 4KB shared program caches, while in b) shows the 1KB shared results

	Private 4x1K			Shared 4K		
	HIT	MISS	**MR**	HIT	MISS	**MR**
parWorkload	140534	527	**0.37**%	140131	294	**0.21**%
parMatrixMul	26499	814	**3.07**%	26019	403	**1.55**%
Dijkstra	81136	5322	**6.56**%	80787	480	**0.59**%
MonteCarlo	16963	855	**5.04**%	16575	349	**2.11**%
Sparse	22133	891	**4.03**%	22133	388	**1.75**%

Table 1: Miss ratios (MR) for several applications with private and shared program caches

plementing the multi-cast feature, the runtime is increased by 1.5%. Finally, tuning the interconnect to 128bit, and 4 entries in the L0 buffer, the performance is equivalent to the private cache configuration (with multi-cast we measured 1% in speedup).

It is also observable in Fig 6 that, parWorkload, parMatrixMul and Sparse benchmark runtime are always greater than or equal to the private configuration results. For these applications the kernel code is smaller than the private cache capacity and there are no tangible benefits when moving to a shared configuration.

4.2 Area, power and energy results

This section presents the area, power and energy results and the comparison between the proposed shared program caches vs private. All the results are related to a fully functional cluster with 4 processors. Interconnect size is variable, while caches are direct mapped and FIFO depths are reduced to their minimum size allowed. For all these experiments, we assume a cache line of 128 bit (16 byte), and a total shared cache capacity of 1KB and 4KB, and 1KB private as reference configuration.

Our design flow is based on the STMicroelectronics FD-SOI CMOS-28nm Ultra Wide Voltage Range (UWVR) technology library, with hierarchical synthesis methodology with Synopsys Design Compiler Graphical and place and route with Synopsys IC compiler. UTBB FD-SOI technology, allows to dynamically modulating the threshold voltage of transistors through forward body biasing (FBB), trading the leakage power with the operating frequency at each voltage point. This technique allows boosting the operating frequency to improve the energy efficiency, and is mainly effective at extremely low voltage.

To extensively characterize the behavior of the proposed cache sharing mechanism, we considered the architecture running at three different operating points leading to different ratios between leakages and dynamic power. They are 0.9V with no body bias (super-threshold, 1% leakage), 0.4V with no body bias (near threshold, 20%leakage), 0.3V 1.8V FBB (near threshold, leakage dominated).

In section 3.1 we introduced some optimizations in the processor pipeline and fetch interface, that make this core attractive when plugged to a shared program cache system. Usually fetch interface is critical from the implementation point of view, but this core and the program cache have been optimized in a close loop fashion in order to make the sharing approach attractive. Moreover, our cluster is equipped with shared scratchpads (TCDMs) that are directly plugged to the load store unit of the processors, through a low latency interconnect. Therefore, data and instruction interfaces may become critical for timing closure, because of logic complexity, placement issues and routing congestion for such complex systems. However, in our implemented configurations, we localized the critical path along the core load-store unit (from PEs to TCDM scratchpads), and despite the worst latency from core to program cache (and vice versa) is close to the one across the load store unit (to TCDM), the actual target frequency for our synchronous cluster does not depend on the type on the program cache plugged (based on the configurations analyzed). This is an important milestone in our exploration, because any improvement in the application runtime and in the power consumption will be converted in better system energy efficiency (with respect to the reference cluster).

Both for private and shared caches, the cycle time is the same, and achieved target frequencies are 833MHz, 50MHz and 150MHz respectively for 0.9V, 0.4V and 0.3V-1.8FBB corners.

Figure 7 shows the post P&R power estimation results. To extract the power consumption, we simulated the post P&R cluster by feeding the simulation traces obtained from the post P&R simulation. Then we back-annotated the switching activity and the SPEF (parasitic file) in Synopsys Primetime and we extracted these power metrics both for 0.9V no-FBB, 0.4V no-FBB and 0.3V 1.8V-FBB corners. The first series represents the normalized cache area, and for the 1KB shared configuration, the cluster has 4x less capacity than the private configuration. In this case, only the TAG and the DATA array are scaled, while FIFOs, controller and all the internal logic remains the same. For the 1KB configuration we measured a 50% reduction in silicon cost, while the shared 4KB configurations cost about 11% more in average.

The power estimation is performed at cluster level, and therefore the potential benefits and drawbacks are mitigated by the cores, peripherals and TCDM power consumption. This estimation does not include the L2 memory subsystem because is outside the cluster. At 0.9V with the 4KB shared configuration, the cluster burns from 3% to 8% more power than private, while with the 1KB shared, the cluster from 6% to 1% less power.

When moving to maximum energy efficiency point for the cluster, the 0.4V corner, the leakage accounts for 20% of the overall power consumption and the 1KB shared cache becomes much more attractive.

At 0.3V, the power is leakage dominated, and since this is proportional to the silicon area, the cluster with 1KB shared configurations consumes less power, around 8% than the private. Despite we got better results with respect 0.9 and 0.4V, this is not the optimal corner to achieve the maximum energy efficiency, because is leakage dominated.

Figure 8 illustrates the silicon costs for both private and shared configurations. When the shared program cache size is set to 1KB (4 shared banks, 256 Bytes each), the shared cache is 2.1x to 2.3x smaller than the private one, depending on the configuration. When we quadruplicate the cache capacity in the shared configuration (1KB private vs. 4KB shared) the cache overhead is only 14% (with respect to the private cache configuration)

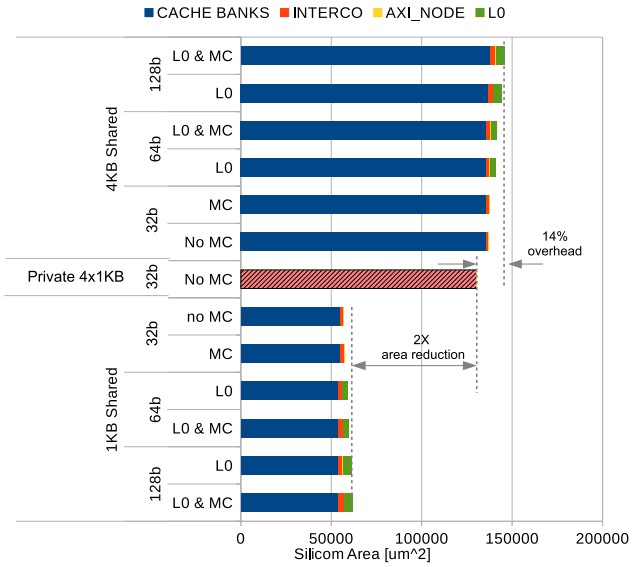

Figure 8: Area breakdown for the private and shared program cache configurations

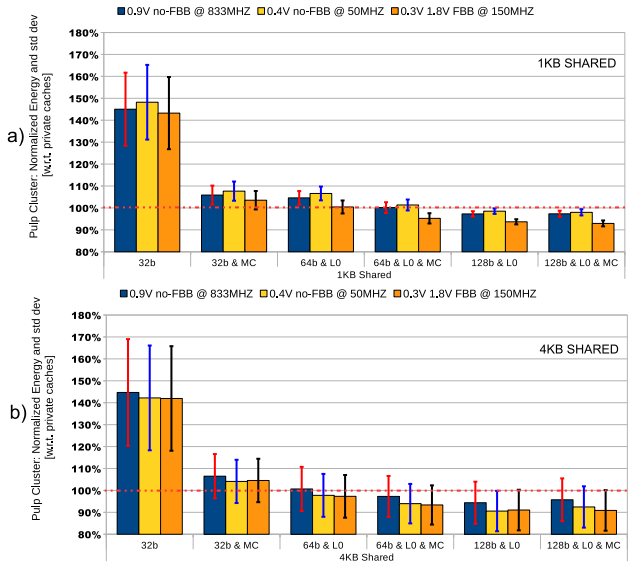

Figure 9: Normalized energy consumption (with respect to private configuration), for different shared cache configurations and operative corners.

Figure 9 shows the cluster energy cost associating the execution time of the benchmarked applications to the power related to the analyzed configurations. The results are normalized to the energy of the cluster equipped with the private cache configuration, used as reference. On average, and considering Fig 9b, the minimum energy point occurs for a configuration with 128-bit interconnect and L0, without multi-cast support. In this configuration the energy saving is 5% @ 0.9V, 10%@ 0.4V and around 8% @ 0.3V, 1.8V FBB [3]. This happens, because increasing the width of the interconnect is very effective in reducing the contention on the cache banks, and the runtime speedup of 11% affects directly the energy metric. On the other hand, multi-cast is effective only in the sub-optimal configurations, where interconnect width is 32 (or 64bit) and the contention on the TCDM is extreme (moderate). This leads to a significant performance variation depending on the access pattern of the different applications. Furthermore, Fig 9a, depicts the normalized energy trend for the shared 1KB configuration. This scenario aims to reduce the cache silicon cost by 2x, while giving a slightly better energy efficiency than the reference cluster. The amount of improvement achieved is 2.5% @ 0.9V, 2%@ 0.4V and around 7% @ 0.3V, 1.8V FBB for the 128bit version without multi-casting feature.

5. CONCLUSION

In this paper, we proposed a multi-banked, shared instruction cache architecture for clustered ultra-low power multi-core systems, where parallelism and near threshold operation is used to achieve minimum energy. A peculiar feature of this architecture consists in a combination of techniques like, L0 buffering and multi-cast mechanism, which make more efficient and attractive the shared program caches.

[3]0.3V 1.8FBB is not the optimal point to achieve the maximum energy efficiency

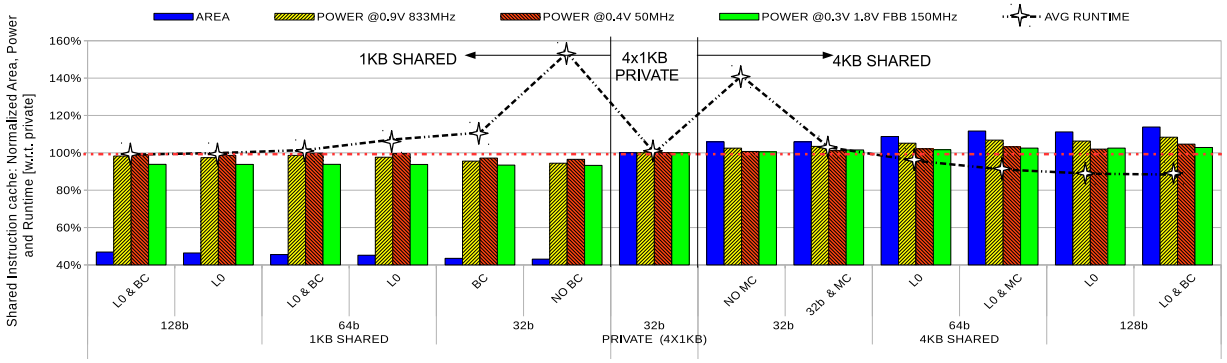

Figure 7: Shared cache metrics for different configurations and corners, normalized to the private cache configuration: cache area, cluster power (in three corners) and average runtime slow-down

We investigated about the benefits and costs with respect to the state of the art and we provided a detailed analysis of the implementation of the proposed shared cache in several configurations including post place & route analysis of performance, area and power metrics. We carried out an architectural exploration of the proposed instruction cache and a comparison in terms of performance, with respect the private configuration, with several signal processing applications. Experimental results, demonstrate that sharing mechanisms allow reducing the energy consumption of the whole cluster by up to 10% when keeping the same area footprint, while reducing the area by 2x when reducing the whole shared cache size capacity to be equivalent to the relative private configuration, and keeping same performance and no impact on the system operating frequency.

6. ACKNOWLEDGMENTS

This work was supported by VAMPA an Eureka EuroStars project (E! 7678 VAMPA).

7. REFERENCES

[1] N. E. Bellas, I. N. Hajj, and C. D. Polychronopoulos. Using dynamic cache management techniques to reduce energy in general purpose processors. *IEEE Transactions on Very Large Scale Integrated Systems*, 8(6):693–708, Dec 2000.

[2] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 983–987, March 2012.

[3] D. Bortolotti, F. Paterna, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini. Exploring instruction caching strategies for tightly-coupled shared-memory clusters. In *International Symposium on System on Chip (SoC)*, pages 34–41, Oct 2011.

[4] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15, Apr 2011.

[5] A. Dogan, R. Braojos, J. Constantin, G. Ansaloni, A. Burg, and D. Atienza. Synchronizing code execution on ultra-low-power embedded multi-channel signal analysis platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 396–399, March 2013.

[6] A. Y. Dogan, J. Constantiny, M. Ruggiero, A. Burg, and D. Atienza. Multi-core architecture design for ultra-low-power wearable health monitoring systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 988,993, March 2012.

[7] J. Eyre and J. Bier. Dsp processors hit the mainstream. *IEEE Computer*, 31(8):51–59, Aug 1998.

[8] C. H. Kim, S. Shim, J. W. Kwak, S. W. Chung, , and C. S. Jhon. First-level instruction cache design for reducing dynamic energy consumption. In *5th International Workshop, SAMOS*, pages 103–111, July 2005.

[9] J. Kin, M. Gupta, and W. H. Mangione-Smith. Filtering memory references to increase energy efficiency. *IEEE Transactions on Computers*, 49(1):1–15, Jan 2000.

[10] Krhonos. The open standard for parallel programming of heterogeneous systems. *https : //www.khronos.org/opencl*.

[11] L. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *International Symposium on Low Power Electronics and Design*, pages 267–269, Aug 1999.

[12] A. Marongiu, P. Burgio, and L. Benini. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 105–110, March 2012.

[13] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg, and J. N. Rodrigues. Benchmarking of standard-cell based memories in the sub-vt domain in 65-nm cmos technology. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(2):173–182, Jun 2011.

[14] NVIDIA. Next generation cuda compute architecture: Fermi. *www.nvidia.com*.

[15] OR1200. Openrisc processor. *http : //opencores.org/or1k*.

[16] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, March 2011.

[17] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246, March 2010.