Implementation of Core Coalition on FPGAs

Kaushik Triyambaka Mysur, Mihai Pricopi, Thomas Marconi, Tulika Mitra
School of Computing
National University of Singapore
kaushik.mysur@gmail.com, {mihai,tulika,marconi}@comp.nus.edu.sg

***Abstract*—Embedded systems increasingly need to support dynamic and diverse application landscape. In response, performance asymmetric multi-cores, comprising of identical instruction-set architecture but micro-architecturally distinct set of simple and complex cores, have emerged as an attractive alternative to accommodate software diversity. Dynamic heterogeneous multi-core architectures take this concept forward by allowing on-demand formation of virtual asymmetric multi-cores through coalition of physically symmetric simple cores and thus adjust better to workload variation at runtime. In this paper, we present the first hardware implementation of a core coalition architecture and synthesize its functional prototype on FPGAs.**

I. Introduction

Traditionally embedded systems were designed to support only a few dedicated applications. Thus customized hardware-software co-designed solutions were ideal platforms. Current generation embedded systems — especially in the consumer electronics domain — are expected to accommodate a wide range of applications with very distinct characteristics. Consider smartphone as a canonical example of high-performance embedded system with strict power, area, and time-to-market constraint. The computing system of such a device needs to include some customized ASIC components for specific kernels such as audio, video, and image processing leading to a heterogenous multiprocessor system-on-chip (MPSoC) design [5]. However, the easy availability of App Store or Android Market implies that the user can download a diverse range of applications during the lifetime of a device, of which, MPSoC designers have no a-priori knowledge. Cores in the MPSoC are thus increasingly responsible for a wide variety of workloads starting from texting and email to compute-intensive tasks such as speech recognition and AI for gaming.

The first response to this issue has been the inclusion of symmetric multi-cores in the MPSoC following the same trend in general-purpose computing (e.g., dual-core and quad-core ARM Cortex-A9 processors [1]). Such multi-core solutions are perfect match for easily parallelizable applications that can exploit task-level or thread-level parallelism (TLP).

But most applications still comprise of a significant fraction of sequential workload. Amdahl's Law [3] states that such application will suffer from limited exploitation of instruction-level parallelism (ILP) in the simple cores. Single ISA (instruction-set architecture) but performance (micro-architecturally) asymmetric multi-cores [8] comprising of both simple and complex cores have recently been proposed as a promising alternative. Indeed, ARM has recently announced Big.LITTLE processing [11] for mobile platforms where high-performance, triple-issue, out-of-order dual-core Cortex A-15 processor is integrated with energy-efficient in-order dual-core Cortex A-7 processor in the same architecture on chip.

Even though asymmetric multi-cores are positioned to accommodate software diversity (mix of ILP and TLP workload) much better than symmetric multi-cores, they are still not the ideal solution as the mix of simple and complex cores has to be freezed at design time. The next logical step forward is to support diverse and dynamic workload with an adaptive multi-core that can, at runtime, tailor itself according to the applications. Such adaptive architectures are physically fabricated as a set of simple, identical cores. At runtime, two or more such simple cores can be coalesced together to create a more complex virtual core [4], [7], [13], [14]. Similarly, the simple cores participating in a complex virtual core, can be disjoined at any point of time. Thus we can create asymmetric multi-cores on-demand through simple reconfiguration.

Even though quite promising, the lack of acceptability of dynamic heterogeneous multi-cores arises from the complexity of the glue logic (either hardware or software) required to coalesce the simple cores together. The open question therefore remains whether dynamic heterogeneous multi-cores is a feasible alternative for high-performance, low-power embedded mobile platforms. In this paper, we attempt to answer this question through a concrete, functionally correct implementation of a core coalition architecture in hardware. To the best of our knowledge, ours is the very first hardware prototype of a dynamic heterogeneous multi-core architecture.

We have recently proposed a core coalition architecture called Bahurupi [12]. Our architecture offers a simple yet elegant approach towards coalition through a hardware-software cooperative solution. A high-level architecture of Bahurupi and an evaluation of its performance with cycle-accurate simulation appeared in [12]. However, a high-level simulation model cannot capture all the challenges involved in designing an architecture in hardware. In this paper, we design, implement, synthesize and place a micro-architecturally accurate prototype of Bahurupi on Xilinx Virtex 6 platform and report the area and latency overhead of core coalition.

Our implementation serves as a proof-of-concept prototype to confirm the feasibility of core coalition and adaptive multi-core architectures. Our FPGA prototype can also serve as a test-bed for future research in design and evaluation of dynamic heterogeneous multi-cores as it enables a faster emulation environment compared to slow software-only simulation.

II. Related work

A number of adaptive multi-core architectures have been proposed recently. The Core Fusion [4] architecture employs a reconfigurable, distributed front-end, and instruction cache organization that can leverage individual cores front-end structures to feed an aggressive fused back-end, with minimal overprovisioning of individual front-ends. TFlex processor [7] uses no physical shared resources among the cores. Instead, TFlex is dependent on a special distributed microarchitecture (EDGE) which is configured to implement the composable lightweight processors. Federation [13] is an alternative approach that makes a pair of scalar cores to act as a 2-way out-of-order core by adding additional complex stages to their internal pipelines. Voltron [14] uses multiple homogeneous cores that can be adapted for single and multi-threaded applications. Voltron relies on a complex compiler to perform code parallelization.

In MorphCore [6] architecture, an adaptive core is created by starting with a traditional high performance out-of-order core and making internal changes to allow it to be transformed into a highly-threaded in-order SMT core when necessary. Composite Cores [10] architecture reduces switching overheads by creating heterogeneity within a single core. The proposed architecture pairs simple and complex pipeline engines together inside a single chip.

To the best of our knowledge, there has been no attempt to implement hardware prototype of any of these architectures.

III. Core Coalition: High Level Architecture

Our core coalition design is based on Bahurupi [12] micro-architecture. Bahurupi, a dynamic heterogeneous architecture, uses a hardware-software co-operative design with minimal additions to the ISA and the underlying architecture. It is fabricated as a shared memory clustered architecture, where each cluster consists of four simple, 2-way, out-of-order (ooo) cores. These simple cores can run threads individually to exploit TLP as shown in the left-hand side of Figure 1. When required to execute a sequential application, 2–4 cores within a cluster form a coalition to create the illusion of a 4-way to 8-way complex ooo core. Thus the sequential application can now transparently exploit ILP through the virtual core. The right-hand side of Figure 1 shows an example configuration created on-demand comprising of one 8-way ooo core, one 4-way ooo core, and two 2-way ooo cores. The architecture can generate any multi-core configuration on-demand within the cluster. As ILP is fairly limited beyond 8-way superscalar processors, a cluster is restricted to four cores.

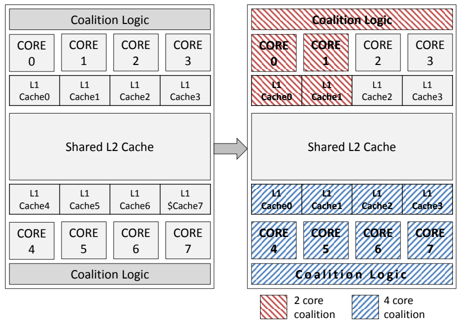


Fig. 1: *Bahurupi Dynamic Heterogeneous Multi-core.*

When running in coalition mode, participating cores co-operatively execute a single thread in a distributed fashion. They execute basic blocks of the sequential thread in parallel and fall back to a centralized unit for synchronization and dependency resolution. Dependency comes in the form of control flow and data dependence. Bahurupi handles these with compiler support and minimal additional hardware.

A new instruction called *sentinel instruction* is added to the ISA, which is the first instruction of each basic block. Basic blocks are constructed at compile time along with the information about *live-in* and *live-out* registers for each basic block. This information along with length of the basic block and whether it ends with a branch instruction, is encoded in the corresponding sentinel instruction to capture the dependency and control flow among the basic blocks. The same binary is used for both cases — in coalition mode or when the core is running alone. For the latter case, the sentinel instructions are dropped at the decode stage or directly skipped when predicted by the local branch predictor.

When not in coalition, each core has private direct mapped L1 instruction and data caches. When participating in a coalition, the private caches are reconfigured into a set associative cache shared among the cores in coalition. To add a new core into an already existing coalition, the system needs to migrate out all the tasks from the new core and write back the content of the L1 cache. Eventually the new core connects to the coalition bus and is ready to fetch a basic block. Note that the cores existing in the coalition do not need to be halted when a new core is added. The reconfiguration time, as described in [12], is 100 cycles.

A key aspect of Bahurupi is its execution model. It emphasizes on operating in a distributed way with only a few essential global structures. As a result, no core is aware of the existence of other cores in the coalition. This is important because any combination of cores can potentially become a coalition and the number of cores in the coalition does not change the way each core operates individually. The reader may refer to [12] for details on high-level architecture of Bahurupi.

IV. Hardware Implementation

We now present the implementation of core-coalition through minimal modification of a 2-way ooo pipeline emphasizing on architectural additions in order to fully realize Bahurupi on a hardware platform.

A. *Baseline 2-way out-of-order core*

We select synthesizable ooo core generated by Fabscalar [2], a parametric micro-architecture generation tool chain as the baseline simple cores. The cores use ARM-like ISA. We custom generate a 2-way ooo core with 5 in-order front end pipeline stages and an ooo backend. Figure 2 depicts the pipeline structure along with the coalition logic. The non-striped region is the baseline core.

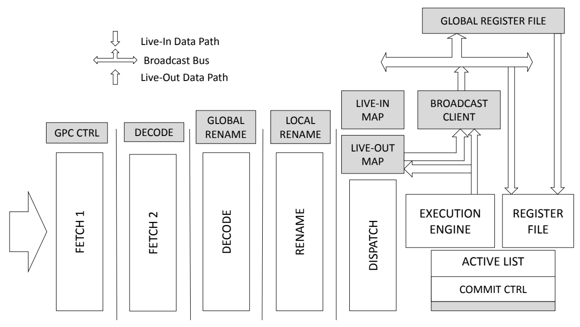


Fig. 2: *Pipeline in coalition mode.*

The critical component for coalition is the register file implementation and its renaming logic. The renaming structure consists of a *Register Map Table (RMT)* and an *Architectural Map Table (AMT)*. The live uncommitted mapping between the architectural registers and physical registers is maintained in the RMT. A dedicated FIFO buffer maintains the list of free registers. Once in-order renaming is complete, instructions are dispatched to the ooo execution engine. As instructions retire in program order from the active list, the logical-to-physical mapping of the destination registers are committed in the AMT. In case of mis-speculation, the renamed mappings in RMT is replaced with functionally correct mappings from AMT. As instructions retire, the physical register corresponding to the old mapping in AMT is disassociated from the logical register and is added back to the free list buffer. A physical register is busy (indicated by a busy bit in the physical register file) between the rename cycle, where it gets mapped to a destination register, and the write-back cycle for the corresponding instruction when the data is ready.

B. Core Coalition Logic

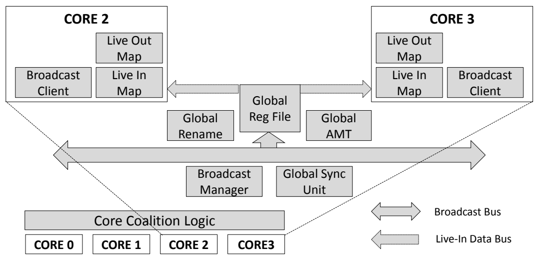


Fig. 3: *Coalition logic per cluster.*

Figure 3 shows the coalition logic required for a cluster of four 2-way cores. We add three new hardware units inside the core, apart from which there is no other significant modification to the existing core. One of key challenge is to make sure that the new hardware units are not in the critical path of existing core. The responsibility of these unit, *Live-In Map*, *Live-Out Map* and the *Broadcast Client*, are discussed next.

1) Sentinel Instruction Processing: We first describe the modifications required in the pipeline to support the sentinel instructions. Recall that the sentinel instruction precedes the normal instructions in each basic block. It encodes the live-in, live-out register information. Bahurupi architecture recommends using at most 3 live-in and 3 live-out registers. If the number of live-in or live-out registers in a basic block exceeds this number, it has to be split into two or more basic blocks at compile time.

The cores within a coalition uses a new register, *global PC*, to synchronize fetching of basic blocks. Each participating core requests for the global PC through the global synchronization unit. We use Least Recently Used (LRU) policy to grant the lock for the global PC. The core that gets the lock proceeds to fetch the sentinel instruction. Assuming that we have L1 cache hit, it takes 2 clock cycles to fetch and decode the sentinel instruction. We decode it in Fetch-2 stage, one cycle before the actual decode stage so that the lock can be released earlier. If the basic block ends with a branch (this information is encoded in the sentinel instruction), we calculate the branch address (as the length of the basic block is known from the sentinel instruction) and index into the branch target buffer and branch prediction unit in Fetch-2 stage.

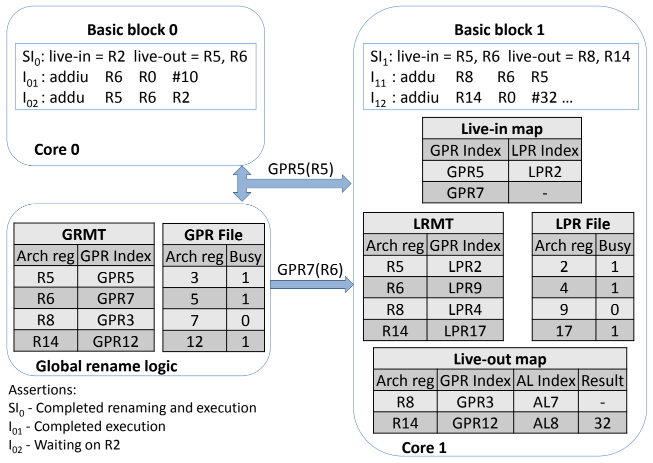
In the third clock cycle, we access global rename module to rename the live-out registers. We also get the existing mapping of the live-in registers. In parallel, the global PC is updated with the address of the next sentinel instruction, that is, a pointer to the next basic block. This will be a speculated address in case of a basic block ending with conditional branch. Otherwise, the next address is calculated by simply adding the length of the basic block to the current global PC. The lock is released at the end of the third clock cycle. The global rename module signals the global synchronization unit to free the lock and any other core is eligible to grab the lock.

After the lock is released, the core continues with sentinel instruction processing. The live-in register values reside in the *global physical register (GPR) file* — a new structure for each coalition cluster. To operate on these values, we need to create a local copy. Thus we rename each live-in register within the core to get a corresponding local physical register (LPR). If the data in the GPR is ready, it is directly copied into the corresponding LPR. However, if the data is not ready, the GPR-LPR mapping is maintained in the Live-In Map. With current design, the number of entries in the Live-In Map is equal to the number of entries in the global physical register file. An alternative approach is to carefully choose a smaller working set based on some heuristics and implement the Live-In Map as a Content Addressable Memory (CAM). We leave this optimization as part of future work.

Figure 4 provides an illustration. We have two basic blocks executing on two different cores. We only show the portion of the code that corresponds to the live-in, live-out registers. Each basic block is preceded by a sentinel instruction. We show the status of the various global structures (global RMT or GRMT, global physical registers or GPRs) and the local structures required for coalition in core 1 at the time instant when sentinel instruction (SI_1) from Basic block 1 (BB1) is to be dispatched. We assume that at this point, live-in register R6 is ready in the GPR; however, instruction I_{02} is yet to complete execution and hence live-in register R5 is not yet ready. As the data in the GPR corresponding to live-in register R6 (GRP7) is ready, it is directly copied into LPR9. However, as GPR5 corresponding to R5 is still busy, GPR-to-LPR mapping (GRP5 - LPR2) is maintained in the Live-In Map.

Similarly, while dispatching the sentinel instruction, the mapping of live-out registers to GPR are stored temporarily in Live-Out Map. In the example, SI_1 stores the live-out register R8 in Live-Out Map (R8-GPR3). Unlike Live-In Map, Live-Out map only consists of 3 entries corresponding to 3 live-out registers. Live-Out Maps are maintained until all the instructions in the basic block have been renamed. We only broadcast the result of the last instruction in the basic block that writes to a particular live-out. Significance of Live-Out Maps will be discussed during normal instruction processing.

2) Normal Instruction Processing: The coalition design is transparent to the processing of the normal instructions. It is visible in only two stages of the pipeline: Live-Out Map module in the dispatch stage and commit control logic in the active list module. Figure 2 shows the seamless integration of coalition logic into the base pipeline.



SI_0: live-in = R2 live-out = R5, R6
I_{01} : addiu R6 R0 #10
I_{02} : addu R5 R6 R2

SI_1: live-in = R5, R6 live-out = R8, R14
I_{11} : addu R8 R6 R5
I_{12} : addiu R14 R0 #32 ...

GRMT	
Arch reg	GPR Index
R5	GPR5
R6	GPR7
R8	GPR3
R14	GPR12

GPR File	
Arch reg	Busy
3	1
5	1
7	0
12	1

Live-in map	
GPR Index	LPR Index
GPR5	LPR2
GPR7	-

LRMT	
Arch reg	GPR Index
R5	LPR2
R6	LPR9
R8	LPR4
R14	LPR17

LPR File	
Arch reg	Busy
2	1
4	1
9	0
17	1

Live-out map			
Arch reg	GPR Index	AL Index	Result
R8	GPR3	AL7	-
R14	GPR12	AL8	32

Assertions:
SI_0 - Completed renaming and execution
I_{01} - Completed execution
I_{02} - Waiting on R2

Fig. 4: *Flow of register values across cores: An illustration.*

As instructions in the basic block are dispatched, the destination registers are searched for a match in the Live-Out Map (implemented as a CAM). If matched, the active list entry of the corresponding instruction is updated with the GPR index from the Live-Out Map. In our example, first instruction in BB1, I_{11}, finds register R8 in the Live-Out Map and adds GPR3 into its active list entry (AL7). This is to ensure that when the instruction I_{11} completes execution, it will broadcast the value. Also the Live-Out Map is updated with the active list ID (R8-GRP3-AL7) of the instruction. This happens in parallel to dispatching instructions to the ooo execution engine.

While the basic block is being renamed, if an instruction with live-out register destination completes execution, the result is stored in Live-Out Map. To illustrate, consider that the instruction I_{12} in BB1, completes execution while the following instructions in BB1 are still being renamed. As the system is not in a position to determine if I_{12} is indeed the last instruction in BB1 to write to the live-out register R14, it is illegal to broadcast the result on the *Global Broadcast Bus (GBB)*. Hence, the result of I_{12} is temporarily stored in Live-Out Map (value 32). When the last instruction of a basic block is renamed, we have the final list of instructions that need to broadcast their result onto the GBB. At this point, if some live-out register values are ready in the Live-Out Map, we broadcast them through the Broadcast Client. In our example, I_{12} has completed execution and assuming no other instruction after I_{12} writes to R8, we can now broadcast the result on the GBB. Live-Out register results produced after this point are broadcast as and when they are ready.

In our example, I_{02} would eventually complete and broadcast the result (GPR5) in its write-back stage through Broadcast client. Broadcast client module is the third significant addition inside the core. It is a simple FIFO buffer and has the logic to obtain broadcast bus access. The broadcast bus is monitored by a global module called *Broadcast Manager*. The access is given to one core at a time using LRU policy. Once the client gets the lock, a live-out value is broadcast on GBB.

Live-In Map module passively snoops all the data on the GBB. If an active Live-In Map for the broadcasted tag is found, the data is internally re-broadcasted to wake up the instructions waiting on that live-in operand. I_{11} of BB1 waits in the issue queue for the availability of live-in register (R5). When I_{02} of BB0 broadcasts the result (GPR5), Live-In Map module of Core 1 (running BB1) finds a match (GPR5-LPR2) and latches the result for internal broadcast. We moved the internal broadcast to the next cycle, which helps reduce the wire length for GBB. The cost of one additional cycle is amortized by the pipelined architecture.

A key thing to note here is that, though coalition logic is exposed at two stages of normal instruction processing, it does not alter the control path of the normal instructions. Coalition logic does not even appear in the critical path for normal instructions apart from searching for destination registers in the Live-Out Map. As there could be a maximum of only 3 live-out registers per basic block, this CAM search of 3 entries does not impact the clock cycle length. The one cycle access to GBB is not altered by the number of cores in the coalition, as we have split the broadcast into two cycles: global and internal. Increasing number of cores can create contention on the global broadcast bus. However, [12] shows that even for high ILP code, the number of broadcast requests per cycle is below 1.0 for 2-core coalition and below 1.6 for 4-core coalition.

3) Committing a Basic Block: A simple *ticketing system* is used for program order commit of the basic blocks. A core can commit instructions of a basic block if the serving ticket number is equal to the ticket number obtained for the basic block during sentinel instruction processing. Another implicit constraint is that all the instructions in the basic block must be renamed before the basic block can start commit. First the sentinel instruction needs to be committed. Though it is internally a NOP for the execution engine, it has the responsibility of committing the new mapping of live-out registers to GPR in the global AMT (architecture map table).

Another role of the sentinel instruction in commit stage is to release the mapping of the LPRs to the live-in registers. This simplifies our recovery mechanism a great deal as discussed next. All other instructions in the basic block commit normally. Finally, the serving ticket number is incremented when the last instruction in the basic block in committed.

4) Speculative Execution and Recovery Model: The baseline core employs a simple 2-bit predictor and a branch target buffer (BTB) to predict the control flow. In coalition mode, the prediction is distributed as each core individually predicts when it encounters a control flow. The prediction is performed at the time of decoding the sentinel instruction in the second cycle. As a basic block can have at most one branch instruction, we need only 1 read port for branch predictor and BTB. In comparison, a 2-way (4-way) ooo core needs 2 (4) read ports. We believe, the prediction algorithm could be tailor made considering the atomicity of basic blocks and by studying the distribution pattern of basic blocks with conditional branches in coalition mode. Additional control information could be encoded in the Sentinel instruction including type of control instruction and static predictions from compile time analysis for better accuracy.

When a branch is mispredicted, the recovery mechanism employed in coalition mode is similar to the model used in single core mode. When the prediction is incorrect, the core waits for all the instructions in that basic block to commit. When the last instruction is committed, all the cores in the coalition are flushed, global PC is updated, and a recovery of register maps is triggered from Global AMT to Global RMT.

The out-of-order address calculation of the load-store instructions in different cores may lead to a hazard. As an illustration, consider a load instruction following a store to the same address. If the address of the load is calculated before the store, the load-store disambiguation logic fails to perform store-to-load forwarding as it is unaware of the hazard. Such a load gets stale data from cache or memory. This is referred to as load-violation. We handle this problem in two steps.

When a store commits and detects a violating load, it marks a dirty bit corresponding to that load. When the load retires, the dirty bit is read and a recovery is triggered. The baseline core employs a pipeline flush plus register map recovery and restarts execution from the problematic load. However, in coalition mode, the recovery is complicated. As load instructions can be present anywhere in the basic block, part of the basic block including sentinel instruction might have been already committed. A step-by-step procedure for load violation recovery in coalition mode is shown below.

- Set the PC of the core where load violation is detected to the address of the violated load.
- For every basic block, maintain the next global PC that is updated at time of holding the global lock. When a load violation occurs, set the global PC to this next global PC.
- Flush all the other cores in the coalition.
- Trigger a recovery from local AMT to RMT in the core where load violation is detected.
- Trigger a recovery from Global AMT to Global RMT to recover the machine state to a clean state.

The decision to let sentinel instructions commit live-out, live-in mapping, is the key idea to achieve load violation recovery with minimal modifications to the base pipeline.

C. *Modifications to Bahurupi Architecture*

In order to fully realize core coalition on a hardware platform, we had to make some changes to the original proposal. Although Bahurupi architecture provides the design to communicate live-in and live-out registers across the baseline cores using Global Physical Register File and Global Broadcast Bus, it does not elaborate recovery handling due to load violation and branch mis-prediction. We provide two designs for recovery: (a) handling branch misprediction during the commit of the last instruction in the basic block and (b) handling load violation, which can happen anywhere in the basic block. The nature of such events are significantly different and need different approaches for recovery as presented earlier.

Another assumption made in Bahurupi is that decoding and renaming are performed in back-to-back cycles, so that the Global Lock can be released after renaming. However, with current state-of-the-art OOO cores, pipelines are deeper with many stages in between decode and rename stages. A classic example is an instruction buffer. Hence, we re-designed Sentinel instruction processing by splitting its register renaming stage into Global and Local rename stages.

In addition, we introduce a new structure to the baseline core called Live-In map. It maintains a list of maps (Global Physical Register of live-in to a correspondingly renamed Local Physical Register) for Live-Ins whose data are not available from Global Physical Register File at the time of global renaming of Sentinel instructions. The core snoops on the bus for broadcasts of these registers and then internally broadcasts it. This design was crucial to realize Live-in/ Live-out communication across cores and load-violation recovery mechanism with a deeply pipelined baseline core.

V. Prototype Synthesis and Evaluation

We implement our prototype on Xilinx Virtex 6 (XC6VLX240T-1FF1156) platform. In [12], we presented a preliminary evaluation of the impact of *only* the global register file and its renaming logic on the clock period. However, there was no attempt to integrate the coalition logic inside the base processor pipeline. This work presents a full-fledged implementation that brings out the challenges involved.

A. *Additional Resources for Core Coalition*

Table I shows additional read/write ports required in existing components and Table II shows block memories required to implement sentinel instruction processing within cores.

Component	AMT	RMT	Free List	LPRF	Issue Queue
Write Ports	1	1	1	2	0
Read Ports	1	1	1	0	2

TABLE I: *Additional ports within the baseline core.*

Module	RAM/CAM	Component	Bits	R/W Ports
Active List	RAMS	Ticket Number	128*9	2R/2W
		Live-Out Map	128*7	3R/3W
		Sentinel Commit Map	16*90	1R/1W
Broadcast Client		Broadcast FIFO Buffer	16*46	1R/4W
Live-In Map		GPR-to-LPR Mapping	96*7	1R/2W
Live-Out Map		Live-Out-Map	3*46	3R/3W
	CAMS	GPR Search	3*7	2R/3W
		Active List ID Search	3*7	3R/2W

TABLE II: *Additional RAMs/CAMs for coalition per core.*

The global structures added are independent of the number of cores participating in the coalition. The only exception is the global physical register file for which number of read ports increases linearly with the number of cores in the coalition. The placement of these global resources is a key to achieve good clock frequency. For two 2-way core and four 2-way core coalition, default placement strategies in Xilinx's Place and Route tool with no optimizations enabled could give us the desired clock frequency. Table III lists the global resources required for core coalition.

Module	Component	RAM in Bits	RAM Ports
Global AMT	Logical-to-GPR Mapping	34*7	3R/3W
Global RMT	Logical-to-GPR Mapping	34*7	3R/3W
Global Free List	Free GPR List Buffer	62*7	3R/3W
GPRF	Global PRF	96*32	NR/1W

TABLE III: *Global resources for coalition. N is # of cores.*

B. *Handling multiple ports in FPGAs*

A major challenge in synthesizing out-of-order cores in FPGAs is the multi-ported RAMs as most FPGA vendors only support 2-port memories. Many different solutions have been proposed for this problem [9]; but no single technique can fully provide our desired behavior from the RAMs. So we have used three different techniques: *Replication* to provide multiple read ports; *Live-Value-Table (LVT)* implementation for multiple write ports; and *Virtual Cycle* implementation to mitigate the one cycle delay due to Synchronous block RAMs.

As the block RAMs are synchronous, data from RAM reads are only available in the following clock cycle. However, our baseline core requires asynchronous read logic in some stages, i.e., data needs to be read in the same cycle in which the address is sent. We use Virtual Cycles to achieve this. A global virtual cycle generator unit connects to all the components and alternatively generates virtual stall signal for the entire core apart from the RAMs itself. This single cycle stall helps mitigate the asynchronous read problem. However, this effectively halves the frequency achieved; but as this applies to all the core configurations, we consider this a fair comparison.

C. Area Evaluation

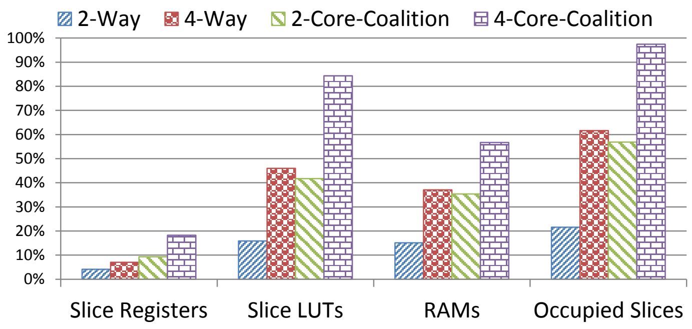


Fig. 5: *Area Utilization (8-way core equivalent in performance to 4-core coalition could not be synthesized).*

Figure 5 presents the area required for different core configurations including the baseline 2-way ooo core. Note that 2-core (4-core) coalition has similar performance to a native 4-way (8-way) ooo core. Our 2-core coalition requires less area than a native 4-way ooo core for most critical resources. We could successfully synthesize and perform place-and-route of a 4-core coalition on Virtex-6 board. However, we could not even synthesize an 8-way core on the same board. Thus Bahurupi has an area advantage over conventional ooo cores.

Figure 6 shows the break-up of area required for core coalition logic in comparison to a baseline 2-way core. The results show the utilization of the slice registers, LUTs and BRAMs. Core-coalition logic consumes additional 13% of Slice Registers, 26% of BRAMs and 27% of Slice LUTs compared to the the baseline core. Note that in our evaluation, the multi-core is directly connected to the rest of the system without any cache memory hierarchy. Still the results are encouraging with minimal additional resource requirement.

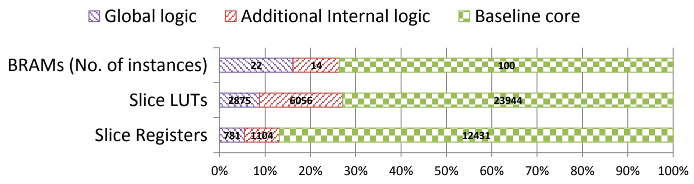


Fig. 6: *Area breakup of coalition logic w.r.t baseline core.*

D. Clock Frequency

The synthesis result concretely supports the simplicity and efficiency of Bahurupi architecture. We observe that core coalition logic has no impact on clock frequency of the baseline core. Figure 7 shows that the clock frequency remains almost same for 2-core (84.4 MHz) and 4-core (83.3 MHz) coalition as baseline 2-way core (84.8 MHz). In contrast, a 4-way core synthesizes to a much lower frequency (62.67 MHz) and an 8-way core could not even be synthesized. The high clock frequency of our prototype coalition architecture (83 MHz) makes it an ideal emulation tool for further research in dynamic heterogeneous multi-core architectures compared to extremely slow cycle-accurate software-only simulation.

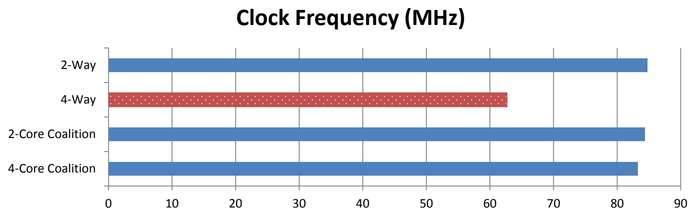


Fig. 7: *Clock frequency for various core configurations.*

VI. Conclusion

We have presented the first prototype implementation of a core coalition architecture in FPGAs. We can successfully create a virtual 4-way (8-way) out-of-order core from two (four) 2-way out-of-order cores. The area of the virtual core is slightly smaller while the clock frequency is significantly higher compared to the equivalent native core.

Acknowledgements: This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2009-T2-1-033.

References

[1] The ARM Cortex-A9 Processors. Technical report, ARM, 2009.

[2] N. K. Choudhary et al. Fabscalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. ISCA, pages 11–22, 2011.

[3] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 2008.

[4] E. Ipek et al. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. ISCA, pages 186–197, 2007.

[5] A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chip*. Elsevier Morgan Kaufmann, 2005.

[6] Khubaib et al. MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP. MICRO, pages 305–316, 2012.

[7] C. Kim et al. Composable Lightweight Processors. MICRO, pages 381–394, 2007.

[8] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. ISCA, pages 64–, 2004.

[9] C. E. LaForest and J. G. Steffan. Efficient Multi-Ported Memories for FPGAs. FPGA, pages 41–50, 2010.

[10] A. Lukefahr et al. Composite Cores: Pushing Heterogeneity Into a Core. MICRO, pages 317–328, 2012.

[11] G. Peter. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. Technical report, ARM, 2011.

[12] M. Pricopi and T. Mitra. Bahurupi: A Polymorphic Heterogeneous Multi-Core Architecture. *TACO*, 8(4), 2011.

[13] D. Tarjan et al. Federation: Repurposing Scalar Cores for Out-Of-Order Instruction Issue. DAC, pages 772–775, 2008.

[14] H. Zhong et al. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. HPCA, pages 25–36, 2007.