An Architecture for Exploiting Coarse-Grain Parallelism on FPGAs

Davor Capalija and Tarek S. Abdelrahman

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto
10 King's College Road, Toronto, ON, M5S 3G4, Canada
{davor,tsa}@eecg.utoronto.ca

***Abstract*—We propose the use of a novel architecture, called the Multi-Level Computing Architecture (MLCA) to efficiently exploit coarse-grain parallelism on FPGAs. The central component of the MLCA is its Control Processor (CP), which is analogous to an out-of-order scheduling unit of a superscalar processor. The CP schedules coarse-grain units of computation, or tasks, onto Processing Units (PUs). In this paper, we explore the FPGA implementation of the CP and demonstrate the scalability of the MLCA for multimedia applications. We design, test and evaluate an 8-PU MLCA system. Our evaluation using 4 realistic multimedia applications indicates that the applications exhibit good scalability up to 8 PUs. Furthermore, the evaluation indicates that our CP design poses no bottlenecks to performance and has little overhead in terms of resource usage.**

I. Introduction

The challenge of translating applications written in traditional software programming languages into efficient FPGA hardware has been met with considerable interest in recent past. Key to this challenge is the discovery, extraction and mapping of the parallelism in an application into efficient hardware. We believe that the key to addressing this challenge is the decoupling of computation on the one hand and synchronization and scheduling on the other. Such decoupling allows designers to reason about the performance of computation units and to better determine which computation units should map onto soft processors and which to dedicated hardware. To this end, we propose the use of a novel architecture, called the Multi-Level Computing Architecture (MLCA) [1], as a template for efficiently exploiting application parallelism on FPGAs. The MLCA consists of two levels. At the lower level is a set of processing units (PUs), which can be programmable processors or custom FPGA accelerators. The PUs execute coarse-grain computation units called *tasks*. At the upper level, a dedicated Control Processor (CP) automatically extracts parallelism among tasks, synchronizes and schedules the tasks on the PUs. It does so using techniques that are similar to ones used in superscalar processors to extract and execute parallelism among instructions, including register renaming and out-of-order execution.

The MLCA completely decouples computations from synchronization and scheduling, while being able to support applications with complex control flow and data sharing [1]. All synchronization and scheduling is performed by the upper level CP, which is logically and physically decoupled from the PUs. Tasks contain only computations; they have no synchronization code. Once a task is scheduled, it runs to completion. Thus, designers can focus on the performance of individual tasks and can easily determine execution-critical ones [2]. These critical tasks can be accelerated using standard synthesis tools, such as Altera's C2H or Xilinix's AccelDSP. Furthermore, the CP itself can be parameterized and its parameters can be customized to match the complexity of a single application or a set of applications. Thus, we believe that the MLCA presents an intuitive approach for designers to parallelize applications and to further accelerate them on FPGAs.

The CP is a central component of the MLCA and hence may become a performance bottleneck of the system. Therefore our immediate goal is to investigate the scalability of the CP using realistic multimedia applications. In this paper, we present the design of an MLCA system, focusing on the microarchitecture of the CP. We implement the entire CP in SystemVerilog, and we utilize Altera's Nios II processors for PUs and Altera's Avalon for interconnect. Our system consists of 8 PUs with caches and is organized as a shared memory multiprocessor. Our experimental evaluation shows that i) applications exhibit scalable performance for up to 8 PUs, ii) our CP design introduces little overhead in terms of resource usage, iii) the CP can provide enough throughput for applications with medium- and coarse-grained tasks. We thus conclude that the CP is not a bottleneck in our endeavor to use the MLCA as a template for exploiting application parallelism on FPGAs.

The remainder of this paper is organized as follows. Section II gives a brief overview of the MLCA and its programming model. Section III explains our system and CP design. Section IV presents our experimental evaluation. In Section V, we list related work and in Section VI we conclude and give directions for future work.

II. The MLCA

The overall organization of the MLCA is shown in Figure 1. The PUs execute coarse-grain units of computation that are referred to as *tasks*. The top-level consists of a *Control Processor* (CP) and a *Universal Register File* (URF) [1]. The CP fetches *task-instructions* (TIs) that each represents an invocation of a task. The CP decodes each task-instruction and then schedules it to a PU where the corresponding task gets executed. Each TI (and the corresponding task invocation) takes its inputs from registers in the URF and deposits its outputs also to registers in the URF. The novelty of the MLCA is that the CP uses techniques such as register renaming, out-of-order-execution (OoOE), speculation and multiple-issue to

extract parallelism among tasks. In this respect, it is similar to how a superscalar processor[1] extracts parallelism among instructions.

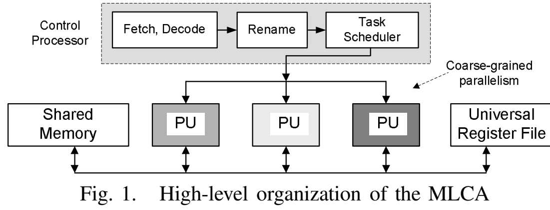

Fig. 1. High-level organization of the MLCA

The MLCA gives rise to a simple and intuitive coarse-grain data-flow programming model [1]. Similar to the architecture, it consists of a set of *task functions* that each represents a task and a top-level *control program* that consists of task-instructions (with URF inputs and outputs). The control program in itself is a sequential program and all parallelism among the tasks is extracted by the CP.

Figure 2 shows a typical MLCA program. It contains a loop whose body has five TIs. For instance, the task B reads the URF registers R_1 and R_6, and writes to R_2 and R_6. Task functions (not shown in the figure) read and write their input and output arguments using `Read_arg` and `Write_arg` calls. For instance, a call `Write_arg(2)` in the task function of the task B writes the task's second output argument to R_6.

The task graph shown in Figure 2 illustrates the main benefits of the MLCA execution model. It demonstrates the process of extracting parallelism across loop iterations in addition to the parallelism within the body of a loop. It shows the task graph for the first two iterations of the loop. The indices denote the iteration number, e.g., A_1 is a task from the first iteration. Solid edges represent true dependencies and dashed edges false dependencies. The only true dependence between iterations is among the instances of the task B. This is because the task B has a state that is carried over iterations (assigned to R_6 which is an input and also an output of the task B). However, there is false dependence caused by the dynamic recycling [3] of register R_1 across iterations. Without register renaming and OoOE (the task graph to the left) this false dependence precludes almost any overlapped execution between iterations; task A_2 has to wait until the task C_1 has finished its execution. With register renaming and OoOE (task graph to the right) there is considerably more overlap in execution.

In addition to TIs, the control program contains CP instructions that support control flow. These instructions access special *control registers* (CRs) in a Control Register File (CRF) and are executed directly by the CP. For example, the second internal CP instruction in the program of Figure 2 is a conditional branch instruction which tests CR_1. The control flow can be influenced by the output of tasks, since a task can write to one control register. In this example, the task E writes to CR_1.

The MLCA has a tool chain that allows the expression of control programs in a C-like language called *Sarek* [1]. In addition, the tool chain facilitates the "taskification" of

[1]We use the term *superscalar processor* to refer to a processor that employs multiple-issue, out-of-order and speculative execution.

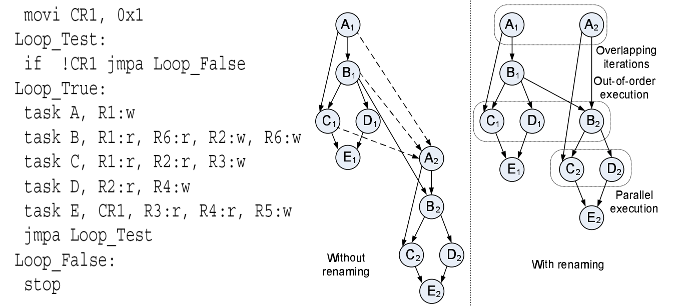

Fig. 2. An example MLCA control program and the corresponding task graphs and extraction of parallelism

sequential multimedia programs [4] and the optimization of the resulting tasks [5].

III. The MLCA System Design and Implementation

Figure 3 shows the overall organization of our implementation of the MLCA system comprising 8 PUs. We utilize Altera's Nios II processors for PUs and Altera's Avalon for system interconnect. All of the components are placed on the FPGA device except the shared memory (DDR2 SDRAM). The off-chip memory is accessed via Altera's High-Performance DDR2 controller. The URF and CRF register communication among tasks goes through the CP, which also manages the CRF and the URF. The control program is placed into a dedicated on-chip memory. The off-chip memory is logically divided among PUs, such that each PU has its own stack, heap and a copy of the task functions. PUs can share data through the heap. There is no operating system or virtual memory. Each PU runs a monitor program which receives task IDs form the CP and executes the corresponding task functions. Task input and output registers are communicated via designated input and output memories of each PU (small internal FPGA memory).

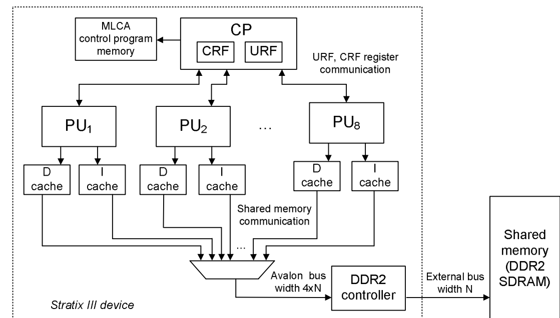

Fig. 3. Organization of the MLCA system on our platform

A. Control Processor Microarchitecture

We designed the CP's microarchitecture and implemented it in about 20,000 lines of SystemVerilog HDL code. The main functionalities provided by the CP microarchitecture are parallel and out-of-order execution of tasks. The two microarchitectural techniques that enable these functionalities are *register renaming* and *dynamic task scheduling* [6].

Figure 4 depicts the main units of the CP. At a high-level, the overall design resembles the pattern common to the pipeline of contemporary high-performance microprocessors.

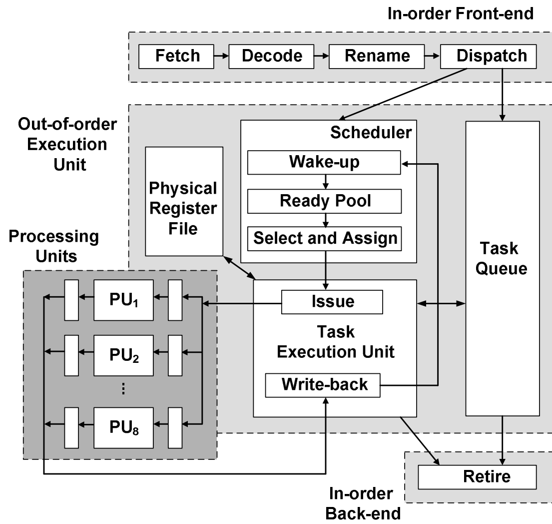

Fig. 4. Main physical units of the CP

It consists of an in-order front end, an Out-of-Order Execution (OoOE) unit and an in-order back end.

Although the CP is based on established principles commonly used for extracting parallelism at the instruction granularity, there are key differences. Task instructions have many inputs and outputs and tasks execute for many cycles as opposed to just few cycles, as is the case with instructions. This leads to different constraints and also to opportunities in the implementation of the MLCA. Table I summarizes the differences between a superscalar processor and the MLCA. Capalija [6] details the impact of these constraints and opportunities on the design of the microarchitecture and describes new mechanisms for efficient task instruction wake-up. A discussion of these mechanisms is beyond the scope of this paper.

Feature	Superscalar processor	MLCA
Computational element	Instruction	Task
Computation granularity (clock cycles)	Fine 1-100's	Medium/Coarse 1000's-100,000's
Number of processing units	4-10 FUs	4-32 PUs
Execution time variability	Lower variability	Higher variability
Number of inputs/outputs	Small (0-3)	Large (10-100)
Computation atomicity	Output is produced at the end of execution	Outputs are produced at arbitrary times and in any order

TABLE I
DIFFERENCES BETWEEN ARCHITECTURAL PARAMETERS OF A SUPERSCALAR PROCESSOR AND THE MLCA

We designed the CP microarchitecture as a pipeline of *macro stages*, each performing its operation in multiple cycles. This decision was driven by the considerably larger granularity of computation, which alleviates the need for units with complex logic that perform all the work in a single cycle. Thus, each unit shown in Figure 4 is a circuit that performs its operation in series of cycles. For instance, the renaming unit will take 3 cycles to rename an input register. Thus, if a task has 20 inputs, renaming its inputs will take 60 cycles. All the CP's units operate in parallel and communicate via FIFOs. Hence, stalls in one unit will not immediately stall neighboring units. Also, the units can operate at different rates. Such design allows for a relatively higher latency of individual units, but due to the concurrency between units, it results in satisfactory throughput.

A key challenge in the CP design is determining the required throughput of the CP. The two critical components are the issue unit and the write-back unit. Thus, the two questions we faced were: at which rate should the CP be able to issue tasks to PUs and at which rate should the CP be able to process outputs received from PUs. In both cases the rates are influenced by the variability of key architectural parameters, including task granularity, number of PUs, number of inputs/outputs, etc.

To answer these questions, we elect to use the following target parameters: average task granularity of 1000 clock cycles, 8 PUs, and tasks with 10 inputs and 10 outputs on average. Thus, the CP has to process a single output register in approximately 12 cycles ($\frac{1000}{8\times10}$). We performed similar calculation for each unit and used these rates to guide our design. We designed each CP unit striving for the simplest (and hence fastest) design that meets the latency/throughput requirements that we calculated. We use a set of realistic multimedia benchmarks to choose the above target parameters. The benchmarks exhibit variability in terms of the key architectural parameters, which gives us confidence that the target parameters are representative of a wide range of applications.

We often had a choice of a memory-based implementation or a logic-based implementation. We weighed the alternatives in terms of operating frequency and area. For instance, some circuit design choices that would be natural in standard cell ASIC design technology proved to be prohibitively expensive in FPGAs. One example is content addressable memories (CAMs). They do not exist as embedded blocks and must be implemented using generic FPGA logic resources, which can be very expensive and slow, especially with high-associativity. Similarly, FPGA memory blocks have only two ports, which dictates careful design and distribution of memory-based structures. In all of our design we avoided the choice of multi-ported memory structures. We carefully distributed the CP's memory structures across multiple memory blocks in order to obtain more throughput to those structures. It is possible that for larger MLCA systems with 16 or more PUs, several units of the CP might become a bottleneck. The memory structures in those CP's units will require more than two ports, hence they will have to be replicated or further distributed in order to provide more ports.

CP pipeline overview

The CP's front end is responsible for bringing TIs into the OoOE unit and it also employs a register renaming mechanism. A TI is fetched, decoded, and its inputs and outputs are renamed in an in-order fashion. The last stage of the front end is the dispatch unit which inserts TIs into the OoOE unit, and allocates required resources for each TI in the OoOE unit.

The OoOE unit is the central part of the CP. Its main components are the task queue, the dynamic scheduler and the task execution unit. Upon arriving into the OoOE unit, TIs are placed into the task queue and also into the wake-up unit of the dynamic scheduler. TIs wait in the wake-up unit until their inputs become ready. The task queue holds information about all TIs in flight. It provides the functionality similar to that of the reorder buffer in a superscalar processor. TIs are considered to be in-flight after they have been inserted into the task queue by the dispatch unit. A TI ceases to be in-flight

once it is retired by the retire unit.

Once all inputs of a TI are available, the corresponding task becomes ready for execution. The wake-up unit then updates the state of the TI and forwards it to the pool of TIs ready for execution. The select and assign unit selects one of the ready TIs from the pool and assigns it to one of the PUs that are free. The issue unit sends the TI to the assigned PU[2], where the corresponding task is executed. The writeback unit collects the registers written by tasks and forwards them to the wake-up unit which processes the registers and determines if any of the waiting TIs have become ready. The task execution and the writeback stages can be overlapped since tasks produce multiple outputs during their execution.

The back-end comprises only the retire unit which removes completed TIs from the task queue and other CP structures. A TI that has finished its execution is removed from the CP only if all tasks earlier to its corresponding task in program order have also finished executing. At this point the resources that the TI has occupied in the CP are released. The in-program-order completion is intended to support precise interrupts and recovery from exceptions and misspeculation, although the implementation of these features is left to future work.

B. Software-managed coherence

There is no hardware support for cache-coherence. We employ a software-managed mechanism that flushes shared data from a cache before it is used by dependent tasks. There is no flushing needed for simple scalar dependencies (e.g., integers), which are handled by the URF. However, tasks can also share complex data structures (such as buffers) and synchronize through URF registers that hold pointers to those data structures. This data must be flushed.

We have two methods of flushing shared data. The `FlushRegion(ptr,size)` routine flushes only the region of the cache where the target structure resides. The routine is implemented using the Nios II `flushda` instruction for flushing a single dirty cache line. In some cases it is overly complicated to flush individual data structures. Hence, we resort to a `FlushFull()` routine, which flushes the entire cache. We also prevent false sharing by ensuring that individual memory regions are aligned to cache lines.

Our flushing mechanism increases execution time in 3 ways. First, it adds execution time to flush dirty lines. Second, it introduces extra memory traffic that can lead to contention among PUs and further prolong task execution time. Third, when `FullFlush` is used, the cache lines corresponding to PU's stack region are also flushed and must be reloaded when a new task is started.

C. Memory subsystem design

We are limited in the design of the memory system by the options available for the Nios II processor. It has only L1 data (D) and instruction (I) caches as shown in Figure 3. The caches are direct-mapped and write-back. However, even in the face of these limitations, the design space for the memory system is large, and it is not feasible to explore it all. There are many design options: i) capacity of I/D caches, ii) line size for D cache, iii) burst modes for I/D caches, iv) arbitration priorities, v) off-chip memory bus width. We opted for the largest cache line possible (32B for Nios II) and bursting mode for both instruction and data caches (the details are listed in Table IV in Section IV).

[2]Only the task ID and task's inputs are sent to the PU, the code of the task function resides in the shared memory and is fetched by the PU itself.

We hypothesize that this configuration provides the best memory bandwidth for the following reasons. The Nios II data bus master is 4 bytes wide regardless of the cache line size. Thus, to fill a relatively large cache line (e.g., 16B), the processor will issue 4 individual transactions. However, these D-cache transactions of different PUs interleave and hence randomize memory access patterns. This in turn increases row changes in an active bank, and/or bank changes, which increases memory access latency. Therefore, we synthesize a system that includes a burst mode to fill both data and instruction caches, thus reducing non-sequential addressing. For a 32B cache line, a data master is granted 8 consecutive 4B-transactions from the controller. That is, it has exclusive access until the entire cache line is filled. Similar reasoning is outlined by Altera [7], but using arbitration priorities. However, neither exact line sizes for I/D caches nor actual arbitration priorities are given.

IV. Experimental Evaluation

We evaluate our system by synthesizing and testing it on an an Altera Stratix III FPGA platform. Our evaluation includes performance analysis in terms of application speedup and overhead analysis in terms of area. Table II contains the characteristics of our FPGA platform and the Altera tools that we used.

FPGA Board	Altera DE3 Prototyping system
FPGA device	Altera Stratix III EP3SL340 (Speed grade 3)
Logic Elements (LEs)	338,000
Embedded Memory (Kbits)	16,272
Off-chip SDRAM	DDR2 266.6 MHz f_{max}, 256 MB
Synthesis CAD/compilation tool	Altera Quartus II/Nios II EDS 9.0

TABLE II
FPGA board characteristics and design tools

An instance of our MLCA experimental system is described with three sets of parameters: the microarchitectural parameters of the CP, the PU parameters, and system-wide parameters. The process of obtaining the CP parameters is described in our earlier work [6]. It is possible to customize the parameters of the CP for a single application or a set of them. In this work we synthesize a single CP with the parameters that are customized to suit our set of four benchmarks. The parameters of our CP are shown in Table III. The PU and system parameters are shown in Table IV.

Maximum number of task inputs/outputs	64
Number of architectural registers	256 URF/32 CRF
Task queue size	512 tds
Wake-up unit size	2048 regs
Physical register file size	2048 regs
Ready task pool size	512 tds
Scheduling policy	first ready first served
Fetch unit size	8 128-bit words
Register rename queue	64 regs
Task header dispatch queue size	4 tds
Number of PU output (input) queues	8

TABLE III
CP parameters (td=task descriptor,reg=register)

In our evaluation, we use four multimedia benchmark applications: MAD, an open source MPEG audio decoder [8], FMR, a program that performs FM demodulation [9], GSM,

CP/PU clock frequency	100 MHz
DDR2 memory freq/width/capacity	200 MHz/8 bits/32 MB
DDR2 controller clock frequency	100 MHz, half rate
DDR2 controller arbitrator	All masters equal priority
DDR2 controller Avalon width	32 bits
Number of PUs	8
PU type	Nios II fast, HW mul and div
Instruction cache	16 KB, direct mapped, 32 B line, burst
Data cache	8 KB, direct mapped, 32 B line, burst

TABLE IV
PU AND SYSTEM PARAMETERS

an implementation of the GSM standard for speech transcoding [10], and JPEG, an image encoder [11]. We use manually ported versions of the applications on our platform [9]. Each ported benchmark comprises a control program and a set of task functions. The task functions are compiled using Nios II GCC compiler with `-O2` level optimization.

A. *Performance*

We measure performance in terms of *relative speedup* which is defined as the ratio of the application execution time on a 1-PU to its execution on an N-PU MLCA system[3]. Figure 5 shows relative speedup of the applications on our target system. These speedups are good and compare favorably to those obtained by functional MLCA simulator in earlier work [1], [5], [2]. FMR scales well to 8 PUs. JPEG and MAD scale well up to 6 PUs, whereas GSM scales well up to 4 PUs.

We investigate the possible sources of performance bottlenecks by measuring three different metrics: overall PU utilization, average task prolongation, and task prolongation due to data cache flushing overhead.

The utilization of a single PU is the ratio of the time the PU spends executing tasks (T_e) to the total PU time (T_t), which is the sum of T_e and the time it spends idling (T_i). The overall PU utilization is the average of T_e/T_t utilization ratios across all PUs, and is shown in Figure 6. For FMR the PU utilization is just below 90% for 8 PUs. This high PU utilization suggests high rates of ready tasks issued by the CP. Hence, we hypothesize that the CP is not a bottleneck. For JPEG, MAD and GSM utilization decreases as number of PU grows. However, this is expected since earlier simulation work [1], [5], [2] shows that there is less available parallelism for larger number of PUs.

The second metric is average task execution *prolongation*. We measure the execution time of each task during its parallel execution on the PUs. We then aggregate by summing all execution times (T_p). We compare this aggregate to the sum of the execution time of all tasks on a single PU system (T_s). The ratio T_p/T_s reflects the extent by which tasks are prolonged. The tasks are prolonged due to two reasons: memory contention between the PUs and flushing overhead. We measure total task prolongation and the time tasks spend executing our cache flushing calls, which is only the first component of the overhead of our flushing techniques (see Section III-B). Note that experiments with one PU require no flushing, whereas in experiments with more than one PU we must employ our flushing technique.

The task prolongation is shown in Figure 7. The curves labeled with a **D** reflect only the overhead of the flushing calls, whereas the curves labeled with a **T** show task prolongation due to all flushing overhead and memory contention.

[3]We do not flush the cache in 1-PU experiments. The total 1-PU execution time is the sum of task execution times only. Thus, all idle time is excluded.

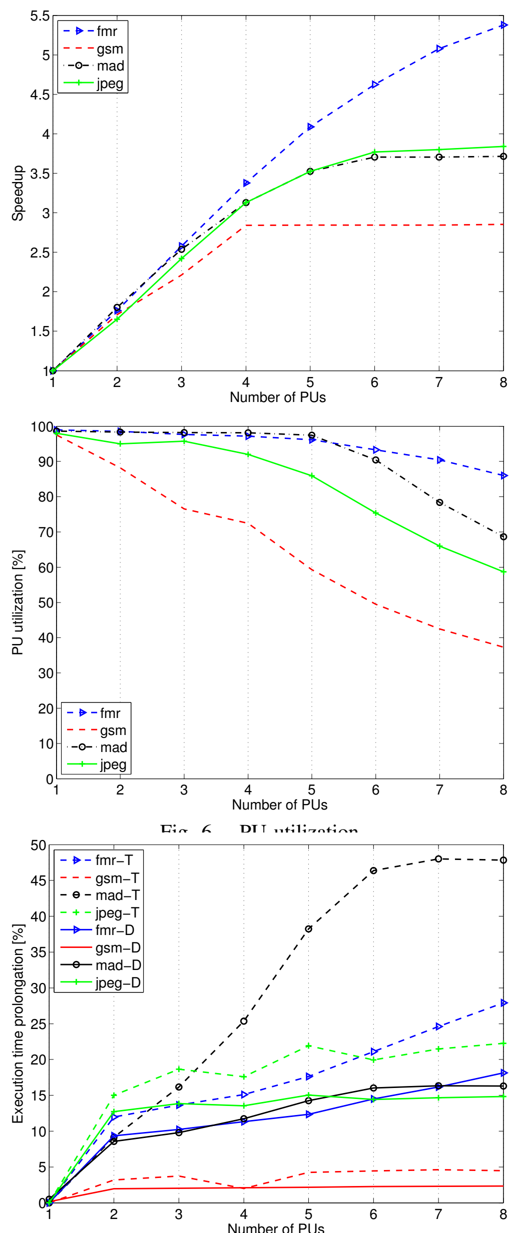

Fig. 7. Total prolongation of task execution time (**T**), and prolongation due to D-cache flushing calls (**D**)

The figure shows that total task prolongation is considerably increased with the number of PUs for MAD (up to 50%), while the flushing calls overhead remains below 15%. This suggests that this benchmark would benefit from additional memory throughput. For other benchmarks tasks are prolonged up to 28%, and most of this prolongation is due to flushing calls. This indicates sufficient memory bandwidth for these 3 applications.

B. *CP throughput*

The CP's throughput depends on the processing speed of its units and the sizes of the structures that units are composed of. We investigate the processing speed with respect to two factors: the number of PUs and the average execution time of tasks. We set the sizes of the structures to large values so that they do not pose a limitation. We use a synthetic benchmark whose main loop contains a series of tasks, which are made to have only one output and two inputs on average. The benchmark contains a mix of task, data and pipeline

parallelism. We perform functional simulation in ModelSim which allows us to employ idealized memory subsystem and interconnect which impose no contention. Hence, any speedup degradation is due to the CP. We specify the execution time of each task using artificial delays.

Figure 8 shows the speedup of the synthetic benchmark for different average tasks execution times. We run the experiment for up to 16 PUs. We can observe from the figure that there is a small difference in the speedup curves for average task execution times of 530 and 1061 clock cycles (cc). Thus, at these average task execution times, the speedup is limited by the amount of parallelism available in the application. We can also notice that as we reduce the average execution time, the speedup degrades. Also, we can see the effect of increasing the number of PUs. For 4 PUs the differences in speedup caused by varying task granularity are not significant. As we grow the system to 16 PUs, the speedup gaps between the curves become much larger. We targeted the CP throughput for the tasks with an average execution time of 1000 cycles, so the performance degradation for shorter task execution times is expected. We believe that the CP's bottleneck at these shorter execution times is its issue unit, which is currently sequential and can issue only one task at a time.

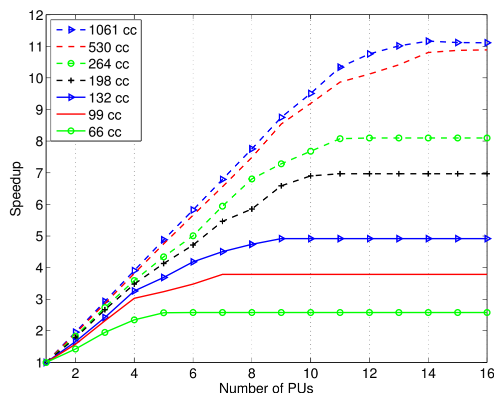

Fig. 8. Performance scalability for the synthetic benchmark

To further verify that the CP is not the bottleneck in experiments with our real benchmarks we synthesize an MLCA system in which the CP runs at 150 MHz, and the PUs remain at original 100 MHz. The performance curves are the same as before proving that the CP indeed has enough throughput for that set of benchmarks.

C. Maximum frequency and area

We further evaluate our CP and the overall system in terms of maximum operating frequency (F_{max}) and area. We determine these two metrics using Altera Quartus II Design Space Explorer (DSE). To obtain the upper-bound on F_{max}, the DSE was setup to search for the fit that is most optimized for speed. The attained F_{max} for the CP was 160 MHz. However, in our work we do not attempt to achieve the highest possible frequency of the system. The target frequency for the on-chip components system was 133.3 MHz and 266.6 MHz for the DDR2 memory. We were able to meet this requirement for our baseline system and also for systems with varying D-cache line sizes. However, we noticed that the inclusion of the burst adapters had negative impact on the maximum frequency. Hence, the system using bursting mode was synthesized at 100 MHz (DDR2 at 200 MHz). We preserved processor-to-memory clock ratio, hence allowing the comparison of the performance of both systems.

Our preliminary analysis shows that the CP's critical path is within the scheduler unit and its connection to the dispatch unit. The logic and routing delays equally contribute to the critical path delay. The path goes from the dispatch FIFO to the wake-up unit's memory block and consists of routing delays and several levels of logic. The CP microarchitecture comprises many memory blocks so their placement has great impact on the routing delays, and thus on the critical path. The analysis also show that the critical path of the entire MLCA system consist of a PU-to-memory interconnect logic which includes burst adapters and the large memory controller's arbitrator. Improving the F_{max} of both the CP and the entire system is a part of future work.

We measure the resources used by the CP in our final 8 PU 100 MHz system and compare to that of an average of all Nios II processors in the system. The results are shown in Table V.

Resource	CP	PU_{avg}	DDR2 ctrl	Total	CP % increase
ALMs	2852	1360	1458	23532	14%
Logic registers	2335	1432	1542	23478	11%
Mem (Kbits)	488	219	1	2241	28%
M9Ks	101	33	2	367	38%
M144Ks	0	0	0	0	0%
18-bit DSPs	0	4	0	32	0%
f_{max}	160	240			

TABLE V
SYSTEM RESOURCE USAGE

We can observe that the CP consumes roughly two times as many logic cells (called ALMs in Stratix III) as the PU. Similarly, the CP uses twice as many memory bits than the PU. The CP memory bits count includes all CP units, the CP instruction memory and 8 PU input buffers. The majority of the PU memory bits belong to the data and tag blocks of the PU caches. The high memory block usage in the CP is because the CP microarchitecture comprises many small FIFOs that are implemented using separate memory blocks.

Table V also shows the total resources consumed by the entire MLCA system, which includes the CP, the PUs, the interconnect and the DDR2 memory controller. The memory controller resources are also shown separately. Overall, in our 8-PU system the CP represents a 14% increase in the ALM usage, and 28% increase in memory bits usage.

The breakdown of the area across the CP units is given in Figure 9. The amount of resources consumed by each unit

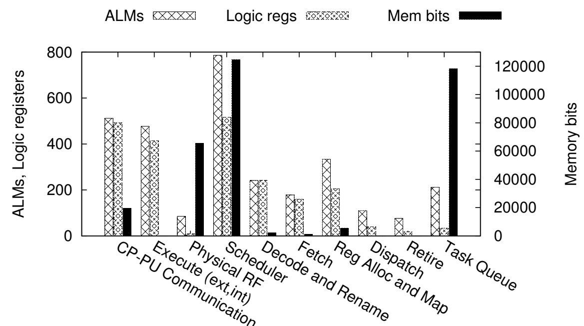

Fig. 9. Breakdown of the CP resource usage across units

is proportional to its parameters (shown in Table III). We can observe that the majority of resources are consumed by the units comprising the CP's back-end: the scheduler, task

queue, CP-PU communication, and the execute unit. Overall, the most resources are consumed by the scheduler unit. This unit consumes the most logic resources. It is important to emphasize that CP resources are proportional to its parameters which are in turn proportional to the number of PUs, in this case 8. Therefore, for 4-PU system, a smaller CP is needed, but for a 16-PU system, a larger CP is needed.

V. Related Work

Tripp et al. [12] propose a synthesizing compiler for FPGAs using Java programs as input. Their compiler translates inter-communicating threads into as a set of inter-communicating hardware circuits, each representing a single thread. Channels are used for communication and synchronization among threads. Singh and Greaves [13] similarly transform mainstream C# applications with explicit parallelism into parallel circuits on FPGAs. They generate hardware equivalents for monitors, events, channels and use them to synchronize coarse-grain units that correspond to software threads. Leow et al. [14] propose OpenMP as a medium-grain HDL that is synthesizable into FPGAs. They synthesize arbitrating coordinator for every shared data and for each critical section.

Bondhugula et al. [15] present a framework for synthesizing FPGA circuits from nested loops. They generate regular processing element array from loops with constant dependencies. Similarly, Guo et al. [16] develop ROCCC, an optimizing C to HDL compiler that targets loop nests. In these works loop nests with small bodies are targeted to exploit fine-grained parallelism. In contrast, our approach targets loop nests with coarser granularity and it tackles task-level parallelism.

Unnikrishnan et al. [17] and Plavec et al. [18] develop infrastructures that compile parallel stream-based applications into custom soft multiprocessors, possibly customizing soft processor or using dedicated hardware for computations. Although, streaming languages also decouple synchronization from computation, they target only static control-flow applications and cannot support more dynamic flow and scheduling supported by the MLCA.

The scalability of the memory system is one of the main challenges for soft processors systems. Labrecque et al. [19] explore multi-threaded/multi-PU systems in the presence of off-chip memory bandwidth limitations. Yiannacouras et al. [20] improve memory system performance for their vector processor. They use wide processor buses that match their cache line sizes. Kulamala et al. [21] explore different instruction memory architectures for soft multiprocessors. They use a single code image for all PUs and are thus able to place it in shared on-chip memory. This unloads some pressure from off-chip memory bandwidth and is a possible optimization to allow better scalability. In contrast, we use cache bursting to improve memory throughput.

VI. Conclusion and Future Work

In this paper, we advocate the use of a novel architecture, called the MLCA [1], as a template for efficiently exploiting application parallelism on FPGAs. The MLCA decouples computation and synchronization while being able to support applications with complex control flow and data sharing. We believe the MLCA is suitable for performance debugging, since the critical path can be discovered and critical tasks can be accelerated using a tool such as C2H.

We designed the microarchitecture of the MLCA, focusing on the CP. We implemented the entire CP in SystemVerilog and utilized Altera's Nios II processors for PUs and Altera Avalon for interconnect. Our system consists of 8 PUs with caches and is organized as a shared memory multiprocessor. We show that i) applications exhibit scalable performance up to 8 PUs, ii) our CP introduces little overhead in terms of resource usage, iii) CP can provide enough throughput for medium- and coarse-grain tasks.

Our immediate future direction is to investigate scalability of the system with more PUs at higher frequencies. We will experiment with DDR2/DDR3 memory interfaces that run at 400+ MHz in order to provide the system with enough memory throughput. We believe that we can synthesize the entire system at 200+ MHz by pipelining the system interconnect. We also believe that more in depth investigation of the memory subsystem is beneficial and we can leverage earlier work in this area [19], [21], [20]. Our second goal is to develop techniques for improving the performance critical tasks in an application by using hardware accelerators instead of soft processors. Finally, we plan to port more applications to our system.

Acknowledgments

We are grateful to Altera's University Program for providing us with a DE3 board.

References

[1] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman, "A Multi-Level Computing Architecture for Embedded Multimedia Applications," *IEEE Micro*, vol. 24, no. 3, 2004.

[2] I. Matosevic, T. Abdelrahman, F. Karim, and A. Mellan, "Power Optimization for the MLCA Using DVS," in *Proc. of SCOPES 2005*.

[3] J. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2004.

[4] K. Stewart and T. Abdelrahman, "Automatic Task Generation for the Multi-Level Computing Architecture," in *Proc. of PDCS 2007*.

[5] U. Aydonat and T. Abdelrahman, "Parallelization of Multimedia Applications on the MLCA," in *Proc. of PDCS 2006*.

[6] D. Capalija, "Microarchitecture and FPGA implementation of the multi-level computing architecture," Master's thesis, U. of Toronto, 2008.

[7] *Using High-Performance DDR, DDR2, and DDR3 SDRAM with SOPC Builder*, http://www.altera.com/literature/an/an517.pdf.

[8] *MAD: MPEG Audio Decoder*, www.underbit.com/products/mad/.

[9] U. Aydonat, "Compiler support for a system-on-chip multimedia architecture," Master's thesis, U. of Toronto, 2005.

[10] J. Degener and C. Bormann, *GSM 06.10 lossy speech compression*, 1994, http://cs.tu-berlin.de/˜jutta/toast.html.

[11] *Independent JPEG Group*, http://www.ijg.org.

[12] J. Tripp, P. Jackson, and B. Hutchings, "Sea cucumber: A synthesizing compiler for FPGAs," in *Proc. of FPL 2002*.

[13] S. Singh and D. Greaves, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Proc. of FCCM 2008*.

[14] Y. Leow, C. Ng, and W. Wong, "Generating hardware from OpenMP programs," in *Proc. of FPT 2006*.

[15] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "Automatic mapping of nested loops to FPGAs," in *Proc. of PPoPP 2007*.

[16] Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient hardware code generation for FPGAs," *ACM TACO*, vol. 5, no. 1, 2008.

[17] D. Unnikrishnan, J. Zhao, and R. Tessier, "Application-specific customization and scalability of soft multiprocessors," in *Proc. of FCCM 2009*.

[18] F. Plavec, Z. Vranesic, and S. Brown, "Towards compilation of streaming programs into FPGA hardware," in *Proc. of FDL 2008*.

[19] M. Labrecque, P. Yiannacouras, and G. Steffan, "Scaling soft processor systems," in *Proc. of FCCM 2008*.

[20] P. Yiannacouras, G. Steffan, and J. Rose, "Improving Memory System Performance for Soft Vector Processors," in *Proc. of WoSPS 2008*.

[21] A. Kulmala, E. Salminen, and T. Hamalainen, "Instruction Memory Architecture Evaluation on Multiprocessor FPGA MPEG-4 Encoder," in *Proc. of DDECS 2007*.