Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications

Hyunchul Park* Yongjun Park Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI 48109
{parkhc, yjunpark, mahlke}@umich.edu

ABSTRACT

Mobile computing in the form of smart phones, netbooks, and personal digital assistants has become an integral part of our everyday lives. Moving ahead to the next generation of mobile devices, we believe that multimedia will become a more critical and product-differentiating feature. High definition audio and video as well as 3D graphics provide richer interfaces and compelling capabilities. However, these algorithms also bring different computational challenges than wireless signal processing. Multimedia algorithms are more complex featuring more control flow and variable computational requirements where execution time is not dominated by innermost vector loops. Further, data access is more complex where media applications typically operate on multi-dimensional vectors of data rather than single-dimensional vectors with simple strides. Thus, the design of current mobile platforms requires re-examination to account for these new application domains. In this work, we focus on the design of a programmable, low-power accelerator for multimedia algorithms referred to as a *Polymorphic Pipeline Array*, or PPA. The PPA is designed with flexibility and programmability as first-order requirements to enable the hardware to be dynamically customizable to the application. PPAs exploit pipeline parallelism found in streaming applications to create a coarse-grain hardware pipeline to execute streaming media applications. PPA resources are allocated to each stage depending on its size and ability to exploit fine-grain parallelism. Experimental results show that real-time media applications can take advantage of the static and dynamic configurability for increased power efficiency.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems

General Terms

Design, Algorithms

*Currently with Texas Instruments, Inc.

MICRO'09, December 12–16, 2009, New York, NY, USA.

Keywords

Programmable Accelerator, Software Pipelining, Virtualization

1. INTRODUCTION

Mobile computing has become a ubiquitous part of society. More than half the world's population now owns on a cell phone, and in some countries, the number of active cell phone contracts out numbers the population. The embedded computer systems that power mobile devices demand high performance and energy efficiency to operate in an untethered environment. Traditionally, hardwired accelerators have done the heavy lifting in terms of computation. Mobile platforms are designed as heterogeneous systems-on-a-chip consisting of multiple processors (general-purpose and/or digital signal processors) and special purpose accelerators constructed for the most compute-intensive tasks. The performance/energy point achieved by these designs is impressive - performing tens of giga-operations per second at sub-Watt power levels.

Moving forward, there is a need to create more programmable mobile computing platforms. Programmable solutions offer several key advantages:

- *Multi-mode operation* is enabled by running multiple application standards (e.g., two video codecs) or even multiple applications on the same hardware. Accelerator-based solutions require a union of hardware blocks to accomplish all desired applications.
- *Time to market* of an implementation is lower because the hardware can be re-used across multiple platforms. More importantly, hardware integration and software development can progress in parallel.
- *Prototyping and software bug fixes* are enabled on existing silicon with a software change. On-going evolution of specifications are supported in a natural way by allowing software changes after the chipset and even the device have been manufactured.
- *Chip volumes* are higher as the same chip can support multiple standards without requiring hardware changes.

Traditionally, the design of programmable mobile computing platforms has focused on software defined radio [3, 2, 9, 18, 30]. These systems are geared towards wireless signal processing that contain vast amounts of vector parallelism. As a result, single-instruction multiple-data (SIMD) hardware is recognized as an effective strategy to achieve both high-performance and programmability. SIMD provides high efficiency because of its regular structure, ability to

scale lanes, and low control cost. However, mobile computing systems are not limited to wireless signal processing. High-definition video, audio, 3D graphics, and other forms of media processing are high value applications for mobile terminals. In fact, many believe the quality and types of media support will be the key differentiating factors of future mobile terminals.

Media applications in a mobile environment offer a number of different challenges than wireless signal processing. First, the complexity of media processing algorithms is typically higher than signal processing. Computation is no longer dominated by simple vectorizable innermost loops. Instead, loop bodies are larger with significant amounts of control flow to handle the different operating modes and inherent complexity of media coding. This results in differential dynamic computational requirements. Further, significant time is spent in outer loops and acyclic code regions. As a result, SIMD parallelism is less prevalent and less efficient to exploit in media algorithms [20]. Second, the data access complexity in media processing is higher. Signal processing algorithms typically operate on single dimension vectors, whereas video algorithms operate on two or three dimensional blocks of data where the block size is variable. Thus, video and other forms of media processing push designs to have higher bandwidth and more flexible memory systems. Finally, the power budget is generally more constrained for media processing than wireless signal processing because of higher usage times.

To address these challenges, this work focuses on the design of a flexible media accelerator for mobile computing referred to as a *polymorphic pipeline array* or PPA. Our design does not exploit SIMD parallelism, but rather relies on two forms of pipeline parallelism: coarse-grain pipeline parallelism found in streaming applications [11, 12, 16] and fine-grain parallelism exploited through modulo scheduling of innermost loops [26]. The PPA consists of an array of simple processing elements (PEs) that are tightly interconnected by a scalar operand network and a shared memory. Groups of four PEs form cores that are driven by a single instruction stream. These cores can execute tasks (filters in a streaming application) independently or neighboring cores can be coalesced to execute loops with high degrees of fine-grain parallelism. The use of a regular interconnection fabric allows the core boundaries to be blurred, thereby allowing the hardware to be customized differently for each application.

The PPA design is inspired by coarse-grain reconfigurable architectures (CGRAs) that consist of an array of function units interconnected by a mesh style interconnect [21, 22]. In CGRAs, small register files are distributed throughout the array to hold temporary values and are accessible only by a small subset of function units. Example commercial CGRA systems that target mobile devices are ADRES [22], MorphoSys [19], and Silicon Hive [25]. Tiled architectures, such as Raw, are closely related to CGRAs though are not intended for mobile computing [27]. CGRAs have two important weaknesses that limit their effectiveness on media applications. First, resources are statically organized as 4x4 or 8x8 grids. Within a CGRA, there is full connectivity between PEs and to memories, but across CGRAs only simple connectivity, such as a bus, exists. Mapping applications across more than one CGRA is thus inefficient. Second, CGRAs are geared towards executing innermost loops with high efficiency. Outer loop and acyclic code is often executed on a host processor, or in systems like ADRES, a small subset of the PEs function as a VLIW processor [22]. The net result is that a large fraction of the resources are not utilized unless an innermost loop is executing. The PPA extends the CGRA design to provide more inherent flexibility and increase efficiency in applications that are not dominated by innermost loops.

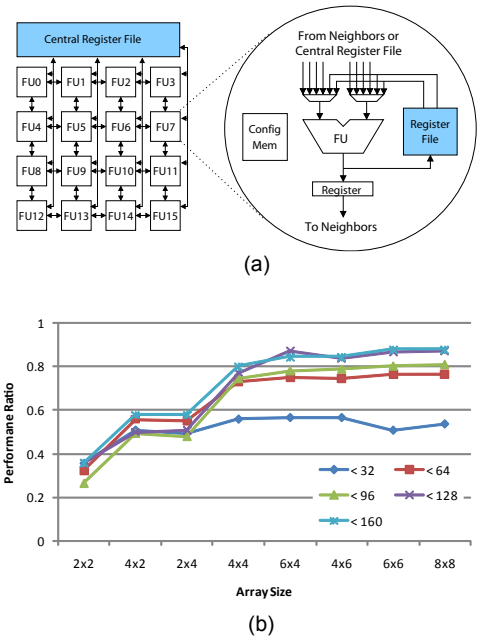


Figure 1: (a) CGRA loop accelerator, (b) Impact of the array size on the performance across loops with varying numbers of instructions.

This paper offers the following three contributions:

- An analysis of the available parallelism and its variability in three media applications (MPEG4 audio decoding, MPEG4 video decoding, and 3D graphics rendering).
- The design, operation, and evaluation of the PPA - a customizable media accelerator for mobile computing.
- A virtualized modulo schedule that can execute innermost loops with a run-time varying number of PPA resources assigned to it.

2. MOTIVATION

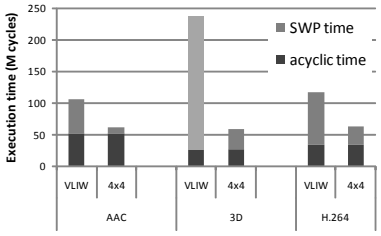
In this section, we identify the current limitations of CGRAs based on an analysis of three applications from different multimedia domains in mobile environments: audio decoding, video decoding, and 3D graphics. Then, we suggest some high-level architecture choices to overcome these bottlenecks and increase scalability for future embedded systems. The applications consist of:

- AAC decoder: MPEG4 audio decoding, low complexity profile
- H.264 decoder: MPEG4 video decoding, qcif profile
- 3D: 3D graphics rendering

A CGRA that is similar to ADRES [21] (Figure 1(a)) is used as the baseline accelerator. ADRES consists of 16 function units (FUs) interconnected by a mesh style network. Register files are associated with each FU to store temporary values. The FUs can execute common integer operations, including addition, subtraction, and multiplication. In contrast to FPGAs, CGRAs sacrifice gate-level reconfigurability to increase hardware efficiency. As a result, they have short reconfiguration times, low delay characteristics, and low power consumption. With a large number of computing resources available on CGRAs, loop level parallelism can

	total loops	SWP loops
AAC	102	36
3D	260	83
H.264	269	81

(a)



(b)

Figure 2: (a) Number of software pipelineable loops, (b) Breakdown of execution time for software pipelineable and acyclic regions.

be exploited by software pipelining compute intensive innermost loops.

ADRES can also function as a VLIW processor to execute acyclic and outer loop code. The first row of FUs and the central register file provide VLIW functionality, while the remaining three rows of FUs are de-activated for non-innermost loop code. Other CGRAs simply execute non-innermost loop code on the host processor and de-activate the entire array. ADRES provides a higher performance option by eliminating slow transfer of live-in values between the host and the array as well as dedicating more functional resources to the acyclic code than a typical host processor would have.

2.1 Fine-grain Parallelism

Multimedia applications typically have many compute intensive kernels that are in the form of nested loops. With a large number of computing elements available, CGRAs can effectively accelerate the nested loops with software pipelining that can increase the throughput of the innermost nest by overlapping the executions of different iterations.

Figure 2 shows how much fine-grain parallelism resides in the three target benchmarks. The number of total loops and the number of software pipelineable (SWP) loops are shown in Figure 2(a). The execution time breakdown between SWP loops and the remainder of the code is shown in Figure 2(b). Each bar in Figure 2(b) shows the breakdown of execution time spent in the software pipelineable regions (SWP time) and the rest of the application (acyclic time). The left bar of each benchmark in Figure 2(b) is the breakdown of execution time spent when only the VLIW processor (top row of the CGRA accelerator) is used for the whole application (no software pipelining), while the right bar shows the breakdown when SWP regions are executed on the entire CGRA. First, we can see that there are many opportunities for exploiting fine-grain parallelism in the benchmarks. On average, 35% of loops are software pipelineable and 71% of execution time is spent in SWP regions. When the CGRA accelerator is employed to map the software pipelineable loops, there are large performance gains of 1.76, 3.25, and 1.48 for AAC, 3D, and H.264, respectively. So, it is very important for multimedia applications to exploit fine-grain parallelism inherent in them, and CGRAs are effective platforms executing such loops.

An interesting question at this point is how we improve the performance even further when more resources are available in an embedded system. One possible solution is scaling the accelerator to a bigger array. By introducing more resources into the accelerator, we can possibly reduce the execution time spent in SWP regions. To assess the impact of scaling the accelerator, we took all the SWP regions in the three benchmarks and mapped them onto accelerators with various array sizes. First, we categorized the SWP loops into groups based on the number of instructions and measured how the

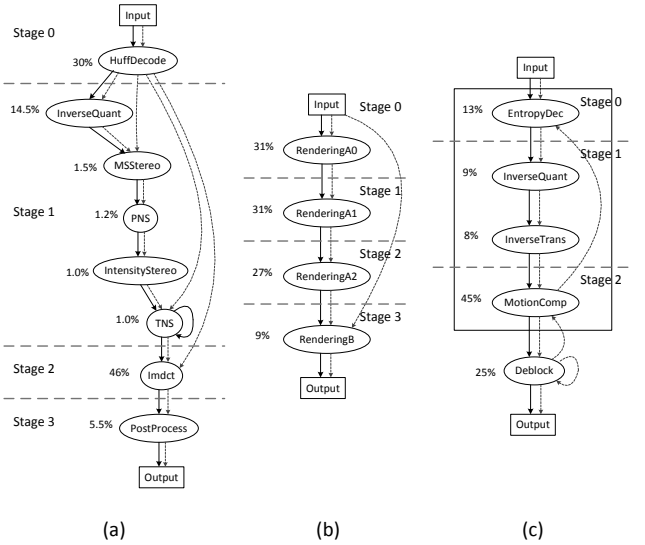


Figure 3: Task Graphs: (a) AAC, (b) 3D, and (c) H.264. Nodes represent tasks, solid edges show control flow, and dotted edges show data transfers.

average throughput of each group changes as the size of the CGRA increases. The array sizes of the CGRA are shown in the X-axis of Figure 1(b), and the scaled throughput on the Y-axis. Throughput is normalized to the theoretical upper bound of each loop when mapped onto the 4x4 array. Here, we can notice that the throughput saturates as we increase the size of the CGRA. Even for the biggest group, the throughput does not increase that much beyond the size of 4x4. Moreover, the execution time spent on SWP loops is relatively small when all the SWP loops are mapped onto the accelerator as shown in Figure 2(b). For H.264, the execution time of SWP region is only 20% of the total time after accelerating them.

The analysis here reveals the inherent weakness of CGRAs; acceleration is limited to the innermost loops. To further improve the overall throughput, it is necessary look beyond innermost loops to include outer loops and acyclic regions.

2.2 Coarse-Grain Pipeline Parallelism

An effective way to accelerate outer loops and acyclic regions is using coarse-grain pipeline parallelism that is rich in multimedia applications due to their streaming nature [12]. Figure 3 shows the task graphs of the target benchmarks. Solid lines indicate control flow edges, while data communications between tasks are shown as dotted lines. Each packet of data is processed in a streaming manner through various computing kernels represented as oval nodes. There is an implicit outer loop around these task graphs that loops over input data packets. Coarse-grain pipeline parallelism can be extracted when the task graph can be split into multiple stages that communicate in a feed-forward fashion and without any inter-iteration dependences contained within a single stage. By mapping stages onto different pieces of hardware, the execution of outer loop iterations can be pipelined and the overall throughput can be increased. Stages can consist of loops as well as acyclic blocks of code, hence parallelism of this form is not limited to innermost loops. The amount of execution time in each stage is annotated next to the nodes. Here, we assume no accelerator in the system and only the VLIW processor (first row of the CGRA) is used.

When pipelining the three benchmarks with the stages shown in Figure 3 (horizontal dashed lines denote stage boundaries), perfor-

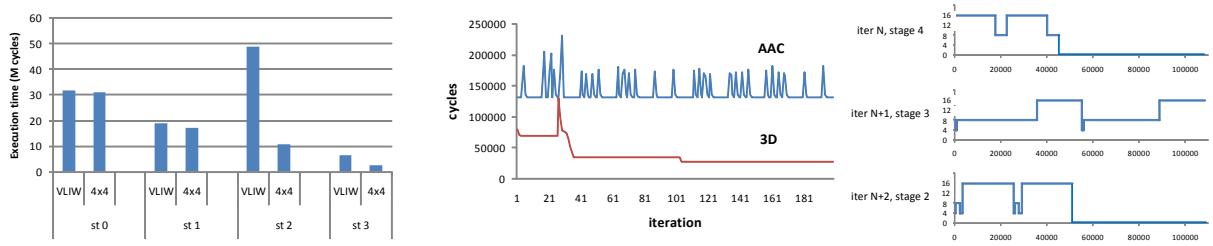


Figure 4: Static/dynamic computation variance: (a) Stage execution time reduction with 4x4 CGRA, (b) Stage execution time change over iterations, and (c) Resource requirement changes in a single iteration.

mance gains of 2.09x, 3.11x, and 1.93x are observed for AAC, 3D[1], and H.264[2], respectively. Indeed, the coarse-grain pipeline parallelism can expose a great deal of performance gain. The stages that are limiting the overall throughput of the pipeline can be accelerated if fine-grain parallelism is available.

2.3 Computation Variance

Exploiting coarse-grain pipeline parallelism along with fine-grain parallelism can overcome the limitation of CGRAs and further enhance the single-thread performance. The application can be partitioned into different stages and each stage can be mapped onto a different CGRA to exploit coarse-grain pipeline parallelism. The abundant computing resources in each CGRA can accelerate the innermost loops when fine-grain parallelism is available. The major challenge here is how to determine the size of CGRAs so that a right amount of resources are allocated to efficiently exploit fine-grain parallelism. In this analysis, two types of computation variance are identified.

Static Variance: The computational requirements vary across different pipeline stages. For AAC (Figure 3(a)), there are two nodes that have a high execution time ratio: HuffDecode (30%) and Imdct (46%). However, their computation patterns are quite different. HuffDecode performs huffman decoding that is a very sequential process, while Imdct contains a large number of filters with large amounts of fine-grain parallelism. Figure 4(a) shows how the execution time changes when more resources added for each stage. The first stage with HuffDecode node does not get much benefit from the extra resources due to its sequential nature. On the other hand, the third stage with Imdct can achieve 4.7x speedup when mapped on a 4x4 array. Similarly, RenderingA in 3D and MotionComp in H.264 can run 3.9x and 2.4x faster on a 4x4 array, respectively. So, not all pipeline stages can get benefit from the large number of computing resources and resource allocation should adapt to the computation variance across different stages.

Dynamic Variance: Another important behavior to characterize in these applications is the dynamic variance in computational requirements. This metric is important because it indicates whether a static apportioning of resources would yield predictable execution times and utilization of the hardware. Conventional wisdom is that processing time is relative constant for media applications, e.g., each iteration of a loop might operate on the row of an image.

Figure 4(b) shows the execution time for one stage in AAC and 3D over the first 200 frames (outer loop iterations). The x-axis is the iteration number and the y-axis is the execution time in cycles on a 4x4 CGRA. As shown, execution time is not constant. In fact, there is a large variation in execution time. AAC regularly oscillates between 150k and 200k cycles, while 3D starts off high and gradually becomes less. This behavior is due to several factors. First, there is an abundance of control flow in these applications that changes the amount of processing required. Second, there is some predictable regularity to the behavior. For example, frames of different types occur at regular intervals and require relatively constant processing time. Finally, in 3D, the processing time levels off after an initial startup. Again, such behavior is not constant, but is predictable.

To view the variability in a different manner, Figure 4(c) shows the resource requirements for three consecutive coarse grain stages from AAC over time. The x-axis is cycle number and the y-axis is the number of resources that achieve the best performance. As shown, resource requirements change during the execution of a single frame. For the top and bottom stages, the resource requirements are dramatic, going from 16 to near zero. In general, resources are allocated based on a worst-case scenario. In this case, each stage would require 16 resources. But, the utilization will be very poor with this approach. Rather, this behavior indicates that idle resources could possibly be loaned to neighboring stages or that shared resources could be designated.

2.4 Summary and Insights

The analysis of these media applications provides several insights. First, multimedia applications are rich in both fine-grain and coarse-grain pipeline parallelism. Further, these forms of parallelism are not mutually exclusive. Rather, they can cooperate to eliminate the opportunities that were left out when only one of them is exploited. Pipeline parallelism can accelerate the entire application including acyclic regions, while fine-grain parallelism can accelerate the pipe stages that limit the overall throughput of the pipeline. Second, resource requirements not only vary statically across different pipeline stages, but also dynamically both during the processing of a single frame of data and across different frames. Dynamic partitioning of resources is thus necessary to simultaneously achieve high performance and high utilization.

A central challenge is how to allocate finite resources across different pipeline stages. Pipeline stages have different levels of fine-grain parallelism. For example, the HuffDecode kernel in AAC and Deblock in H.264 are inherently sequential and putting more resources will not improve the performance. Conversely, Imdct can greatly benefit with more resources. Also, the high dynamic variance in computation continually changes the resource requirements. In real systems, worst-case execution times are often used. But, in these applications, worst-case will grossly exaggerate the number of needed resources. The conclusion is that a flexible execution substrate that facilitates changing the resource allocation over time is necessary.

[1]RenderingA was fully unrolled.

[2]Inter-iteration dependency prevented pipelining the outermost loop. Instead, an inner loop in the solid box was pipelined.

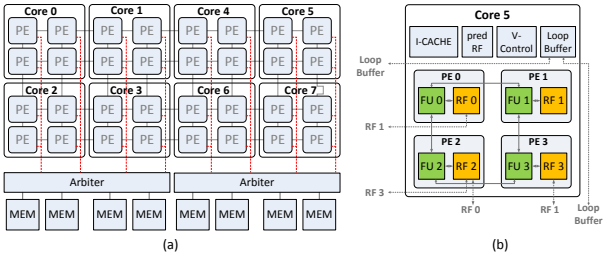


Figure 5: PPA Overview: (a) PPA with 8 cores, (b) Inside a single PPA core

3. POLYMORPHIC PIPELINE ARRAY

3.1 Overview

The Polymorphic Pipeline Array (PPA) is a flexible multicore accelerator for embedded systems that can exploit both fine-grain parallelism found in innermost loops and pipeline parallelism found in streaming applications. The PPA design is inspired by CGRAs but with extensions for both static and dynamic configurability. A PPA consists of multiple simple cores that are tightly coupled to neighboring cores in a mesh-style interconnect. A PPA with eight cores is shown in Figure 5(a). There are a total of 32 processing elements (PEs) in this PPA, each containing one FU and a register file. Four PEs are combined to create a core that can execute its own instruction stream. Each core has its own scratch pad memory and column buses connect four PEs to a memory access arbiter that provides sharing of scratch pad memories among cores.

The main characteristics of PPA can be summarized as follows:

- Simple and distributed hardware: The resources are fully distributed including register files and interconnect. Also, there is no dynamic routing logic. All the communications are statically orchestrated by compiler.
- Fast inter-core communications via direct connections between register files
- Cores can be combined to create a larger logical core to exploit the available fine-grain parallelism in large loop bodies.
- Virtualized execution: PPA can adapt to fluctuating resource availability and dynamically partition the array during the execution

3.2 Core Description

Inside a Core: Figure 5(b) shows a detailed diagram of a single PPA core. There is an instruction cache and a loop buffer. A loop buffer is a small SRAM that stores instructions for modulo scheduled loops. A loop buffer minimizes the instruction fetch power for high density code of loops. Each PE contains a 32 bit FU and a 16 entry register file with 3 read/3 write ports. Four PEs in a core share a 64 entry central predicate register file with 4 read/4 write ports, but there is no central register file for data. All FUs can perform integer arithmetic operations and one FU per core can do multiply operation. A simple mesh network connects the FUs in a core. Register files are accessed by the same topology of mesh interconnect(not shown in the figure). All the FUs can read/write from the central predicate file.

Inter-core Connectivity: Inter-core interconnect is shown in dotted lines in the figure. There are three types of inter-core interconnect in a PPA: data register file, virtualization controller, and loop buffer. Direct connections between neighboring RFs in different cores allow fast inter-core communications. These RF-to-RF

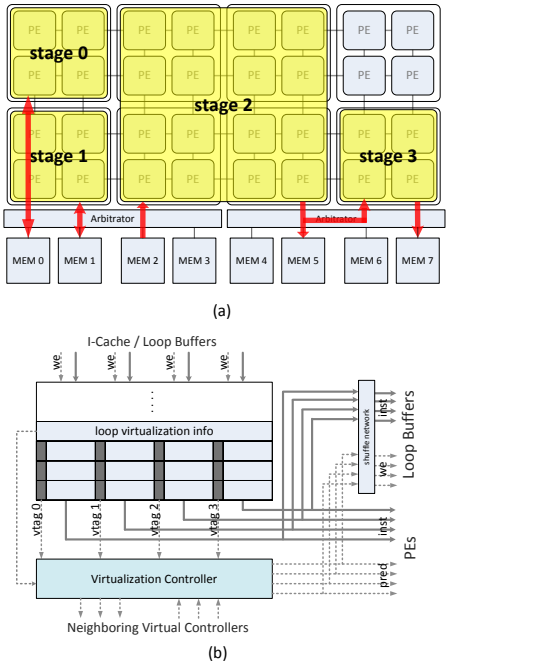


Figure 6: (a) An example of PPA running AAC, (b) Virtualization Controller

connections can be utilized when a loop is mapped onto multiple cores. We found the direct connections between register files more efficient than FU connections especially for virtualization (discussed later). Interconnect between neighboring virtualiztion controller is used for inter-core control communications such as hand-shaking for coarse-grain pipelining and resource availability check for virtualized execution. Finally, loop buffers can transfer instructions to the neighboring cores also for virtualized execution. The hardware components for virtualized execution is explained in details later in Section 4.

Memory System: For memory accesses, a memory bus connects FUs in the same column to the memory access arbiter, allowing only one memory access per cycle for FUs in the same column. A memory arbiter has three memory sharing configurations and provides different load latencies when multiple scratch pad memories are shared among FUs in different columns. The sharing modes for a memory arbiter is as follows.

- No sharing : FUs can access the memory in the same column only. Load latency is 2 cycles.
- Sharing of 2 : FUs in two columns can share memories in the same columns. Load latency is 4 cycles.
- Sharing of 4 : FUs in four columns can share memories in the same columns. Load latency is 6 cycles.

For example, when the arbiter operates in the sharing 4 configuration, all the FUs in four cores (i.e., cores 0, 1, 2, and 3) can access the four memories below them (mem 0, 1, 2, and 3) with a load latency of 6 and up to 4 memory accesses can be made per cycle. Memory sharing only occurs when cores are combined to create a bigger logical core for software pipelining of loops. The increased load latency is not really a big issue since software pipelining can often hide the long latency of operations. Memory sharing can also be used to form a bigger logical memory when memory requirement is high, behaving as a banked memory system.

Communication with Host Processor The PPA behaves as a media accelerator connected to a host processor such as ARM. Since the whole region of the application is off-loaded to the PPA, the communication between the host and PPA is limited to the processing data elements. The data transfer is performed through a standard AMBA bus along with a DMA. The arbiter in PPA only controls DMA transfer among PPA cores and is not shared by the host.

3.3 Supporting Coarse-Grain Parallelism

Figure 6(a) shows how the applications can be accelerated using different static resource partitions for a PPA with eight cores. Based on the analysis in Section 2, we provided the possible mapping of AAC on the PPA shown in Figure 5(a). For the stages with a high ratio of acyclic regions (not software pipelineable), a single core is allocated for execution. Stage 0 performs a Huffman decoding that is very sequential and one core is assigned to this stage. The memory requirement is not high in AAC, so all the memories operate in the no sharing mode. Bold solid lines in the figure show the stages that access memory. When a stage finishes processing data, the output of each stage is transferred to the next stage's memory by a DMA engine. DMA transfers can be omitted when memories are shared by the arbiter. For example, stage 2 can read the input from MEM 2 and write the processed data in MEM 5 that can be accessed by stage 3 directly. Even though memory sharing increases the load latency, both stage 2 and stage 3 contain high ratio of SWP region that can tolerate the latency overhead.

3.4 Supporting Fine-Grain Parallelism

The abundance of computation resources makes PPA an attractive solution for exploiting fine-grain parallelism. When there is large amounts of fine-grain parallelism in a inner-most loop, multiple cores in the PPA are merged together to create a bigger logical core. In the logical core, one core behaves as master and orchestrates the execution of all the participating cores in lock step.

Static Partitioning: The PPA array can be partitioned statically based on the resource requirements of each coarse-grain pipeline stage. An example of static partitioning was provided in the previous section with AAC(Figure 6(a)). The benefit of static partitioning lies in the highly optimized schedules since each compute intensive kernel is scheduled targeting only one sub-array. However, this approach does not adapt to dynamically changing resources availability discussed in Section 2.3. When an application has a large variation in execution pattern, static partitioning can either result in low utilization of resources, or not be able to fully accelerate the overall performance when there is not enough resources available.

Dynamic Partitioning with Virtualization: Coarse-grain pipeline stages in multimedia applications have different execution patterns. As a result, the resource availability in the PPA fluctuates at run-time and it is crucial to adapt to different availabilities and maximally utilize them for improving the overall throughput of the applications. One approach is to statically generate a set of different schedules each of which targets different numbers of resources. For example, a loop can be scheduled for 1x1, 1x2, 2x1 add 2x2 PPA cores beforehand, and an appropriate schedule can be selected at run-time depending on the availability of resources. Each schedule can be highly optimized since the target is fixed. However, the resulting code bloat prevents it from an attractive solution for embedded systems where minimizing code size is important.

The code bloating problem can be minimized through virtualized execution where a single schedule is converted into different schedules dynamically with regard to the changing resource availability. For virtualization, both compiler and hardware support are required. The compiler is responsible for generating schedules that can be easily converted at run-time (see Section 4). Then, the hardware can dynamically allocate resources and perform the conversion of schedules. However, the major down side is the sub-optimal scheduling result. Since the compiler has to target multiple sub-arrays, the scheduling result might not be as efficient as static partitioning approach. Also, there is run-time overhead for virtualization.

3.5 Hardware Support for Virtualization

The major challenges in hardware are how to migrate the schedule across different cores at run-time for virtualized execution and how to communicate with neighboring cores for checking the resource availability. For these purposes, a virtualization controller (VC) is implemented in each core (Figure 6(b)). Since each core has a loop buffer, the PPA can prepare for a virtualized execution by migrating particular sections of a schedule into neighboring cores from the owning core where the whole schedule is stored.

Each instruction in the loop buffer is tagged with two bits of information (virtualization tag). This information is used on two purposes. First, it tells to which core the instruction is migrated when resource allocation is changed at run-time. When more resources become available, the VC copies a subset of instructions to the neighboring cores through the connections between loop buffers. The loop buffer interconnect goes through a shuffle network that can change the orientation of the copied schedule. Depending on the location of the available core, the schedule needs to be flipped horizontally or vertically. Another use of vtag is predicating the execution of inter-core communications that only execute when the schedule is spread over multiple cores. The VC compares the current resource allocation status and the vtag, and generates a predicate input for the inter-core communication instructions.

4. COMPILER SUPPORT

4.1 Edge-centric Modulo Scheduling

In the PPA, all the communications including inter-core communications are orchestrated by the compiler. An effective compiler is essential to utilizing the abundance resources. The major challenge in scheduling with the presence of distributed hardware is in managing the communications among resources. Without any centralized resources, communication is often the bottleneck in finding optimal schedules, more so than the actual computation.

We extended the edge-centric modulo scheduler(EMS) [24] that was previously developed for CGRAs. EMS focuses primarily on the routing problem, with placement being a by-product of the routing process. Edges in the dataflow graph are categorized according to their characteristics, and EMS takes different strategies to route them. Since the schedule is constructed only by routing edges, a number of heuristics on routing cost metrics were developed to improve the quality of schedules. In our baseline accelerator(Figure 1(a)), only the innermost loops are mapped onto the array and the remaining (acyclic and outer loop) regions are executed on the VLIW processor. Since we are offloading acyclic regions onto PPA as well as loops, we modified the EMS algorithm so that it can support both cyclic and acyclic regions.

4.2 How to Virtualize

Virtualization requires the compiler to generate a single schedule that can be dynamically mapped onto different target arrays. There are two approaches for converting schedules: folding and expand-

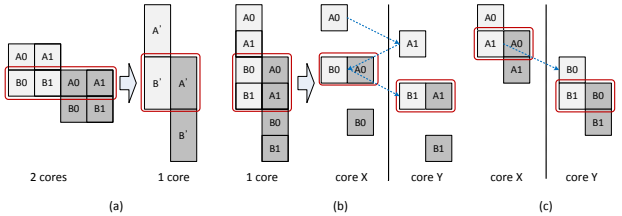


Figure 7: (a) Folding with interleaving, (b) Expanding with horizontal cut, (c) Expanding with vertical cut

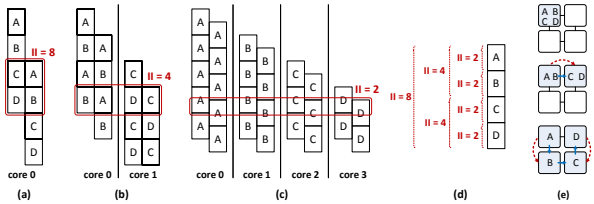


Figure 8: (a) Execution in a single core, (b) Execution in two cores, (c) Execution in four cores, (d) Multi-level modulo constraints, (e) Code expansion

ing. In both approaches, converting (transforming a schedule from one array to another) should be performed in a way that observes the following constraints:

- Modulo constraint: each resource in a converted schedule is used only once in every II cycles
- Register pressure: : virtualization should not drastically increase register pressure
- Dependency constraint: producer-consumer relation is observed in a converted schedule

4.2.1 Folding Approach

Figure 7(a) shows how the folding scheme works for 1x2 array(two cores). First, the compiler generates a schedule shown on the left. Here, the schedule is composed of two sections(A and B). Each section is divided into two sub-schedules for each core. A0 and B0 run on core 0, and A1 and B1 run on core 1. Two iterations of the target loop is shown in the figure; the light gray boxes show the first iteration, and the dark gray ones for the second iteration. In the kernel state(shown as an empty rectangle), all the sub-schedules(A0, A1, B0, and B1) run in parallel in two cores. When only a single core is available, the whole schedule needs to run on a single core and the original schedule is folded to create a narrower and longer schedule shown as A' and B'. The new schedule is created by interleaving the two sub-schedules cycle by cycle. For example, each operation at cycle N in A0 is placed at cycle 2N in A', and one at cycle N in A1 is placed at cycle (2N + 1) in A'. Cycle-wise interleaving is the only way to observe the dependency constraint in the new schedule without re-scheduling. The resource constraint is also kept naturally since (A0, B0) and (A1, B1) time-share the resources in a single core. Since A0 and B0 execute in parallel in the original schedule in the same core, the modulo constraint is observed in the new schedule(also observed for A1 and B1).

The major downside of folding is the increased register pressure. Since the two sub-schedules are interleaved cycle-wise, the communications bypassing the register file in the original schedule need to be buffered for one cycle. They need either passing through a register file(requires re-scheduling) or buffering latches inserted for each interconnect(hardware overhead). Also, the register live ranges in each section overlap with the other section, further increasing the register pressure.

4.2.2 Expanding Approach

The expanding scheme starts with a schedule targeting single core as shown on the left in Figure 7(b). Here, each section (A or B) is divided into two sub-sections(A0, A1 or B0, B1). The first expanding approach is pipelining each section across two cores(shown on the right in Figure 7(b)). Basically, the original kernel schedule is cut horizontally in half and each half runs on a different core. In each core, two sub-sections are running in parallel((A0, B0) or (A1, B1)). Since they were already running in parallel in the original schedule, the modulo constraint is naturally observed. The dependency constraint is also observed since operations are placed in the original order. However, this approach results in frequent inter-core communications shown as the dotted lines in the figure. The communications across the sub-sections(i.g. A0 and A1) incur the register copy operations via RF connections.

Another expanding approach is cutting the original schedule vertically as shown in Figure 7(c). This approach pipelines each section within single core, rather than across two cores. Again, the dependency constraint is naturally observed. Also, the inter-core communications are limited to the section boundary(between A1 and B0). The register pressure does not increase since the consecutive sub-sections are running back-to-back in a single core. The major challenge in this approach is the additional modulo constraints between sub-sections((A0, A1) or (B0, B1)). For example, A0 and A1 were running sequentially in the original schedule, but they run in parallel in the new schedule. Therefore, there is no guarantee that two sub-sections have exclusive resources usage in a single core.

Finally, there is an approach that converts the original schedule in Figure 7(b) into the left schedule of Figure 7(a). This cannot be done easily due to the dependency constraint of A0 and A1. Since there can be producer-consumer relations between A0 and A1, they cannot run in parallel without re-scheduling.

4.3 Virtualized Modulo Scheduling

Based on the observations in the previous section, we propose Virtualized Modulo Scheduling that takes the expanding approach with vertical cut in Figure 7(c). This approach has no hardware overhead of buffer latches in folding and less inter-communication over horizontal expanding. Also, the register pressure is minimal compared to the other two schemes. We proceed the discussion on VMS with a running example of a schedule that can be mapped onto three different target arrays: 1x1, 2x1, and 2x2 , shown in Figure 8.

4.3.1 Minimizing Inter-core Communication

In VMS, a schedule targeting smaller array is divided into the same number of sections as the number of cores in a bigger array. When the schedule is expanded at run-time, each section is individually pipelined in a single core. For example, a loop in Figure 8(a) shows a schedule for 1x1 array. This schedule can be dynamically converted into schedules for 1x2 and 2x2 arrays at run-time. Since it can be mapped onto up to 4 cores, the whole iteration is divided into four sections(A, B, C, and D). When it is mapped onto a 1x2 array, A and B run on core 0, and C and D run on core 1(Figure 8(b). Each section will be mapped to an individual core when 2x2 array is available(Figure 8(c)). Therefore, the inter-core communications can occur only at the section boundaries. Since they can only use the limited inter-core interconnect and the live register values need be transferred across the cores, it is important to minimize the communications across the section boundaries.

For this purpose, the dataflow graph of the target loop is partitioned into four clusters minimizing the number of edge cuts. This is a traditional min-cut problem where each edge-cut denotes an inter-core communication. For this, we implemented direct connections between neighboring register files that can transfer live values across the section boundaries. So, the inter-core communication can occur without wasting the existing computation resources with move operations.

When a loop is scheduled, special register transfer instructions are added for the live values across section boundaries by the compiler. These instructions are guarded with predicates that are only enabled when the schedule is expanded across multiple cores.

4.3.2 *Multi-level Modulo Constraints*

The biggest challenge in VMS is to enforce different levels of modulo constraint, so that no resource conflict occurs when the schedule is converted at run-time. Figure 8 shows a schedule that can be mapped onto different target arrays. In 1x1 array, the target loop runs with II=8 as shown in Figure 8(a). Here, the modulo constraint of II=8 is imposed for all four sections. When it is mapped onto 1x2 array(Figure 8(b)), two different schedules run on each core with II=4. A and B are running in parallel in core 0, so are C and D in core 1. Therefore, the additional modulo constraint of II=4 needs be applied for (A, B) and (C, D), separately. It is important to note that this second level modulo constraint is only applied to the sections running in the same core. Therefore, there is no modulo constraint of II=4 imposed between A and C, or between B and D. Finally, each individual section should observe the third level modulo constraint of II=2(Figure 8(c)). In conclusion, three levels of modulo constraints need be imposed when scheduling the target loop for 1x1 array as shown in Figure 8(d).

In general, modulo constraints limit the number of available scheduling slots to (II x # resources). For example, the number of available slots in Figure 8(a) is 32(= 8×4). One might think that the additional modulo constraints can further reduce the number of available slots. In reality, the number of available slot stays the same since each level of modulo constraints has different scopes. The scopes of three modulo constraints are shown in Figure 8(d). For example, when scheduling the first section A, the scheduler needs to observe both II=8 and II=2 modulo constrains(B is not scheduled yet, so II=4 is not imposed at this point). This reduces the number of slots to 8(2 x 4), but this is actually what is available in a single core when section A is individually pipelined in Figure 8(c). The scheduling slots that were limited by II=2 can be used when section B is scheduled, since the scope of II=2 is only valid for section A. When section B is scheduled, the modulo constraint of II=4 is imposed instead between section A and B.

Scheduling example: An example of scheduling with multi-level modulo constraints are shown in Figure 9. A dataflow graph is shown in Figure 9(a) and it is partitioned into section A and B. For illustration purposes, partitioning was performed arbitrarily(not min-cut partitioning). The target arrays are 1x1 array(1 core) and 1x2 array(2 cores) and target IIs are 4 and 2 for 1x1 and 1x2 array, respectively. First, section A is scheduled onto 1x1 array with II=4 and II=2. Figure 9(b) shows the scheduling result of section A on a single core with 4 FUs. Bold letters with underline show the actual placement of the operations. Gray letters with underline are occupancies due to the modulo constraint of II=4 and normal gray letters show occupancies due to the modulo constraint of II=2. After scheduling section A, only 12 slots are available(Figure 9(b)). However, when section B is scheduled, occupied slots due to modulo constraint of II=2 becomes available since this modulo constraint is limited to section A. Figure 9(c) shows the available slots

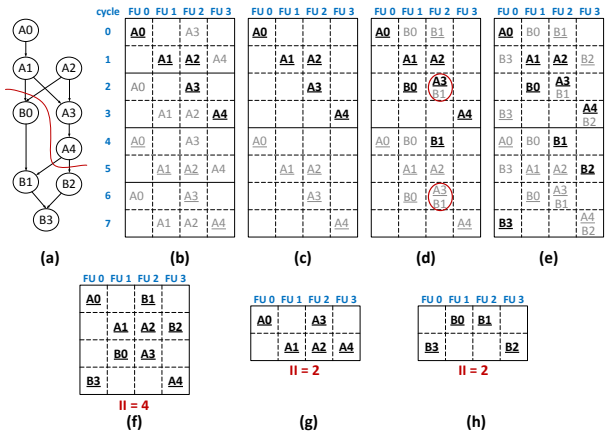


Figure 9: (a) Dataflow graph, (b) - (e) Mapping examples, (f) Modulo schedule for 1x1 array, (g) Modulo schedules for 1x2 array

for section B. First, operation B0 is placed at FU1 in cycle 2(slot (1, 2)) in Figure 9(d). The II=4 modulo constraint marks slot (1, 6) as occupied and the II=2 modulo constraint makes slot (1, 0) and (1, 4) occupied. So, there is no resource conflict so far. When operation B1 is placed at slot (2, 4), II=2 modulo constraint marks slot (2, 2) and (2, 6) as occupied, but they are already occupied by A3. However, this is not a real resource conflict since this II=2 modulo constraint is only valid for section B. II=2 modulo constraint becomes effective only when the schedule is expanded to 1x2 array where section A and section B run on different cores. The final schedule of section B is shown in Figure 9(e). Even though the II for a bigger array is multiple of the smaller II in this example, the IIs don't have to be multiple of smaller one in reality. The constraints among the multi-level IIs are explained in the following section.

Code migration at run-time is also shown in Figure 8(e). Here, a schedule in a single core is expanded over to 2x2 array. Migration in this case is performed in two steps. First, the instructions in section C and D are copied to the core on the right. In the next step, section B and C are copied to the cores below.

Determining Multi-level IIs: In conventional modulo scheduling, the minimum II is selected based on the number of available resources(ResMII) and the length of inter-iteration dependency cycles(RecMII). Starting from the minimum II, the scheduler increases the II until it finds a valid schedule. In VMS, scheduling is performed with multiple IIs as shown in Equation 1, where *N* refers to the number of cores in target array. Before scheduling with virtualization, VMS generates test schedules for each target array without virtualization. The achieved II of each level ($TestII_k$ in Equation 3) determines the benefit of virtualization.

For the first level II, which targets the smallest array, ResMII and RecMII are calculated in a conventional way.While the ResMII for level k changes depending on the number of cores(C_k in Equation 2), the RecMII stays the same since it is not related to the number of resources available. The calculated ResMII and RecMII for each level define the lower bound of II to try(Equation 3). The lower bound is also determined by the II in the next level. When the II for a bigger array becomes greater than the II for a smaller array, there is no point of running the loop on a bigger array. Finally, the II in each level(II_k) is limited to the achieved II of the previous level (II_{k+1}) in the test run. This one also tests the benefit of the virtualization. The scheduling order of II sets is determined by the weighted summation of all the IIs(Equation 4).

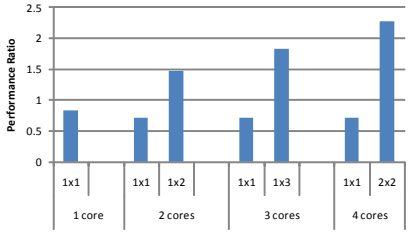


Figure 10: Performance evaluation of virtualized modulo scheduling across varying PPA sub-arrays. Performance is reported as a ratio of the theoretical upperbound performance for a single core.

$$IIs = (II_1, II_2, ..., II_N) \quad (1)$$

$$ResMII_k = ResMII_1 \,/\, C_k, \;\; RecMII_k = RecMII_1 \;\; (k > 1) \quad (2)$$

$$II_k \geq max(ResMII_k, RecMII_k),\; II_k > II_{k+1},\; II_k < TestII_{k-1} \quad (3)$$

$$cost(IIs) = \sum_{k=1}^{N} \Big(w_k \times II_k \Big) \quad (4)$$

Register Allocation with Multi-level IIs: Traditionally, register allocation is performed after scheduling, and spill code is inserted when the register requirement exceeds the register file capacity. Spilling in a highly distributed architecture like PPA is quite costly since it involves routing to/from the memory units and may require complete rescheduling of the loop. Moreover, spilling can easily happen due to the small size of the register files. For this reason, EMS performs register allocation during scheduling to avoid spilling and guarantee routability through the register files. We take the same approach of concurrent scheduling and register allocation in VMS. PPA supports rotating register files that implicitly copy the stored register values at II boundaries so that values can stay in the register file more than II cycles. Since VMS has multi-level of modulo constraints, register allocation needs be performed in a way that all the modulo constrains are observed inside the register files. To simply state, the same concept of different scopes in multi-level modulo constraints can be applied to the register allocation. The details are omitted due to the space limitation.

5. EXPERIMENTS

We evaluated the performance and power of a PPA that consists of eight cores as shown in Figure 5(a). First, the performance of VMS is presented for kernel loops in the three multimedia applications. The performance was measured by the execution time of the three multimedia applications with different configurations of core aggregation and the power was measured using a compute intensive loop in the H.264 application.

5.1 Virtualized Modulo Scheduling Evaluation

The performance of VMS is evaluated for 200 kernel loops that can be modulo scheduled in the three benchmarks. Figure 10 shows the performance ratio of the schedules targeting different sets of PPA sub-arrays. The performance ratio of the schedules is compared to the theoretical upper bound of each loop when mapped onto a single PPA core (MinII). The first bar shows the ratio of schedules targeting a single core. For all three benchmarks, VMS achieves 83% of the maximum throughput for a single core. Considering the distributed hardware in PPA, VMS can provide good quality schedules.

The rest of the bars show how the performance ratio changes when loops are scheduled targeting multiple sub-arrays for virtualization. The number of target sub-arrays is limited to two since we discovered that targeting more than two sub-arrays does not work well in VMS. Improving VMS on more than 2 target sub-arrays remains future work. The left bar in each '*N* core' group shows the performance of the schedule when running on a single core, and the right column shows the result for running in multiple cores. As we expected, the performance of the virtualized schedule in a single core decreases by 13% in average. due to the additional modulo constraint. However, mapping these virtualized loops onto multiple cores allows a big performance increase. On average, the speedup of virtualized schedules on 2, 3, and 4 cores are 1.78, 2.21, and 2.75, respectively. Even though there is some performance degradation for a single core, virtualization can accelerate the overall performance of the application in the presence of fluctuating resource availability.

5.2 Performance Evaluation of PPA

Figure 11 shows the performance from different configurations of the PPA across three applications. The graph shows the execution time for each application in million cycles. The first bar of each application(*acyclic*) represents the entire application executing on a single PPA core without modulo scheduling. The execution time for 3D goes off the chart and their numbers are shown in the graphs. The second bar(*modulo*) represents the execution time when the acyclic code runs on a single core, and the inner-most loops are modulo scheduled and execute on 2x2 PPA sub-array. The rest of the graphs shows the performance results when each application is mapped onto different number of PPA cores. Within each '*N* core' group, both static(*st*) and dynamic(*dy*) partitioning were applied.

First, we can notice that exploiting fine-grain pipeline parallelism with modulo scheduling allows 2.53x speed up in average on a 2x2 PPA sub-array. As shown in Section 2, increasing the number of cores beyond four does not allow much performance gain due to the limited amount of parallelism. When more resources are available, exploiting coarse-grain pipeline parallelism can further improve the overall performance. In this work, our focus is on the back-end scheduling of streaming applications. Here, we manually extracted the task graphs (Figure 3). Other streaming language such as StreamIt [12] can be employed to extract the coarse-grain pipeline parallelism and be fed to our VMS framework as input. We varied the number of PPA cores starting from the number of stages in each application, and increased up to 8 cores available in the PPA. For H.264, we merged the second and the third stages in Figure 3(c) since they have relatively small execution time. Since the outer-most loop in H.264 was not pipelined due to the memory dependency, it does not get as much benefit as AAC and H.264.

In general, increasing the number of cores provides the overall performance gain for all three applications and shows reduced execution time over both *acyclic* and *modulo* configurations, with an exception of 4 cores with static partitioning for AAC. This is because AAC spends most of the execution time in the third stage in Figure 3(c). Extracting fine-grain pipeline parallelism is mostly important to improve the overall performance. However, static partitioning in 4 cores only allows a single core to be utilized for the third stage. As a result, coarse-grain pipelining parallelism does not gives benefit over *modulo* configuration. Dynamic partitioning with virtualized execution becomes quite useful in this case. With virtualization, the compute intensive kernels in the third stages can utilize additional resources from other stages when they are sitting idle, allowing 1.66 speedup over *static* configuration.

The same trend appears on all three applications; the dynamic partitioning out-performs the static partitioning when there is not enough resources to fully exploit the available fine-grain pipeline parallelism. However, the benefit of virtualization diminishes as

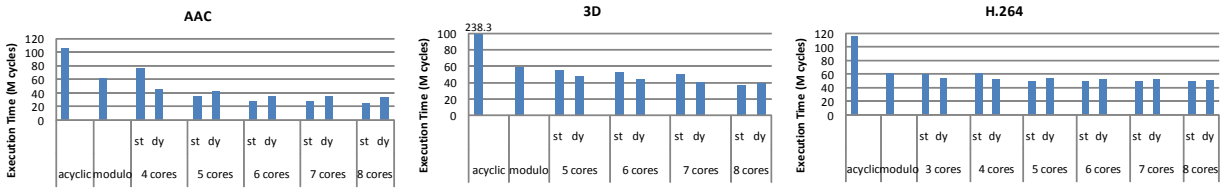


Figure 11: Performance evaluation of the PPA across three applications.

more resources become available in the target PPA configurations. For AAC, the static partitioning out-performs the dynamic partitioning when 5 cores are utilized. For 3D and H.264, the reversion of performance appears later at 8 cores and 5 cores, respectively. This is because the amount of fine-grain pipeline parallelism is richer in two applications than in AAC. To summarize, exploiting the coarse-grain pipelining parallelism does provide the overall performance improvement over *modulo* configuration with both static and dynamic partitioning. When there is a large number of resources available, static partitioning in 8 cores can achieve the speedup of 4.16 and 1.79 over *acyclic* and *modulo* respectively in average for three applications. The dynamic partitioning can maximally utilize the available resource in smaller arrays, out-performing the static partitioning.

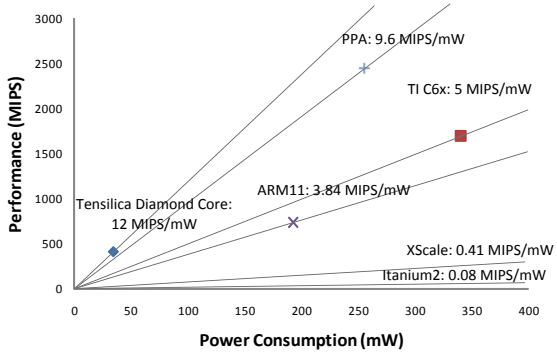
5.3 Power and Area Measurement

Area and power consumption was measured using the RTL Verilog model of the Polymorphic Pipeline Array (PPA) and synthesized using typical operation conditions in TSMC 90nm technology. The model contained both the datapath and control path and was targeted at 200MHz. Synthesizing higher frequencies was possible, but at 200MHz the target applications could be completed and more aggressive frequencies would generate a less energy efficiency design. The memories were generated using standard library models found in the Artisan SRAM memory compiler. Power consumption was calculated using Synopsys PrimTime PX. PrimTime calculates the total power consumption of the PPA using the synthesized netlist, parasitic data generated from Physical Compiler and activity files generated from behavioral simulations. The SRAM memory power was extracted from data generated by the Artisan Memory Compiler. The breakdown of average power when 8 PPA cores are executing the whole code region of H.264 is shown Figure 12(a). The major portion is in computation units like PEs and LRFs. The data memory power consumption is relatively low because H.264 has a high ratio of computation over memory operations. The average power for 8 cores running in pipelining mode is 255.06mW at 200Hz. The toal area of 8 core PPA is 3.37 mm^2.

Figure 12(b) plots the performance vs. power consumption of the PPA and other existing architectures. The numbers were obtained from a graph in [8]. On this plot, points on the same slope have roughly equivalent power efficiency in terms of MIPS/mW, with points towards the upper left having greater power efficiency. As can be seen from the plot, the PPA is able to achieve good power efficiency, only beaten by The Tensilica Diamond Core [28]. Embedded procoeessors like ARM11 and TI C6x show reasonable power efficiency, but their performance is significantly lower than PPA. Thus, they cannot meet the performace requirement of today's compute-intensive multimedia applications. The actual data points for XScale and Itanium2 are outside the range of the plot, but their efficiency lines are shown. As can be observed, the efficiency decreases significantly as the hardware becomes more general and less tailored for embedded applications.

	Total	PE	LRF	I-MEM	D-MEM	Interconnect
Power (mW)	255.06	79.83	108.66	28.57	7.65	30.35

(a)



(b)

Figure 12: (a) Power breakdown of PPA: running H.264, (b) Power/performance comparison.

6. RELATED WORKS

Architectures: Combining cores to create a bigger logical core is relatively a new technique, recently proposed by Core fusion [14] and Composable Lightweight Processors [15]. Core Fusion is a CMP architecture that can dynamically allocate independent cores together for a single thread execution maintaining ISA compatibility. CLPs also allows dynamic allocation of cores to form a larger and powerful single-threaded processors. It also keeps the binary compatibility for the special EDGE ISA. The major difference between [14] and [15] is the target environment. PPA is designed to exploit single thread performance in mobile environments where power consumption and hardware cost is a first-class constraint. The building blocks of PPA is simple in-order cores similar to clustered VLIW processors [31]. Also, the statically controlled point-to-point interconnect provides a fast inter-core communication, allowing PPA to exploit fine grain pipeline parallelism efficiently for multimedia applications.

The PE level view of PPA is similar to Coarse-Grained Reconfigurable Architectures. ADRES [21] is a reconfigurable architecture where PEs are connected to a mesh-style interconnect. Modulo scheduling using a simulated annealing is employed to exploit fine grain pipeline parallelism of nested loops. The top row in the array behaves as a VLIW processor with a multi-ported central register file. However, the non software pipelineable region of the application can only utilize the VLIW part of the array. So, it cannot pipeline the application in a coarser granularity as PPA. With identical resources, PPA outperforms our best approximation of ADRES by 1.43x. PipeRench [10] is a 1-D architecture in which processing elements are arranged in stripes to facilitate pipelining, but it has a fixed configuration of resource partitioning for pipelin-

ing while PPA can partition the array differently as to the characteristics of the target application. RaPiD [7] is another CGRA that consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network.

Exploiting Parallelism: Coarse-grained pipeline parallelism is becoming an attractive approach to accelerate single thread performance as multicore architectures enter the mainstream. [12] and [16] proposed to exploit coarse-grained pipeline parallelism for StreamIt language. Even their target architectures(RAW architectures [17] and Cell processors [13]) are not an embedded system, their technique of task level software pipelining can be applied to our execution model in PPA. [29] has proposed a practical approach to extract a pipeline parallelism from legacy C code. With a help of the programmer, their static analysis tool can extract the potential for streaming execution.

Virtualization: There is much related work on virtualization for binary compatibility for different architectures in literature. Transmeta Code Morphing Software [5] dynamically converts x86 applications into VLIW programs. DynamoRIO [1], Daisy [6], and DIF [23] are all examples that dynamically translate applications to target entirely different microarchitectures. Recent work [4] proposed dynamically binding to cyclic accelerators with modulo scheduling at run-time. The trace in a host processor is examined and run-time modulo scheduling is performed to map the kernel loops onto the cyclic accelerator. While this work focuses on acyclic-to-cyclic conversion, we propose dynamic conversion of modulo scheduled loops in homogeneous multi-core architectures.

7. CONCLUSION

This paper proposes the polymorphic pipeline array (PPA), a flexible multicore accelerator for mobile multimedia. Fine-grain and coarse-grain pipeline paralellism cooperatively improve a single thread performance of computation-rich multimedia applications. To efficiently extract both forms of parallelism on the same piece of hardware, the PPA supports a flexible execution model where cores can operate independently or be combined to create more powerful cores. The array resources can be either statically or dynamically partitioned depending on the computational requirements and behavior of the application. For dynamic partitioning, we propose virtualized modulo scheduling that can generate a single schedule of a target loop that can be easily converted to target different sub-arrays at run-time. With both static and dynamic partitioning, an 8-core PPA can achieve up to 4.16 speedup over a single core execution, and up to 8x over a 4-wide VLIW processor.

8. ACKNOWLEDGMENTS

Thanks to Hong-seok Kim, Sukjin Kim, Taewook Oh, and Heeseok Kim for all their help and feedback. We also thank the anonymous referees who provided good suggestions for improving the quality of this work. This research was supported by Samsung Advanced Institute of Technology, the National Science Foundation grants CNS-0615261 and CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

9. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[2] K. Berkel, F. Heinle, P. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal Applied Signal Processing*, 2005(1):2613–2625, 2005.

[3] H. Bluethgen, C. Grassmann, W. Raab, and U. Ramacher. A programmable platform for software-defined radio. In *Intl. Symposium on System-on-a-Chip*, pages 15–20, Nov. 2003.

[4] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, June 2008.

[5] J. Dehnert et al. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 15–24, Mar. 2003.

[6] K. Ebcioglu and E. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–38, June 1997.

[7] C. Ebeling et al. Mapping applications to the RaPiD configurable architecture. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–115, Apr. 1997.

[8] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 313–322, Feb. 2009.

[9] J. Glossner, E. Hokenek, and M. Moudgill. The sandbridge sandblaster communications processor. In *Proc. of the 2004 Workshop on Application Specific Processors*, pages 53–58, Sept. 2004.

[10] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, June 1999.

[11] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002.

[12] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–162, 2006.

[13] IBM. *Cell Broadband Engine Architecture*, Mar. 2006.

[14] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007.

[15] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 381–393, Dec. 2007.

[16] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of the SIGPLAN '08 Conference on Programming Language Design and Implementation*, pages 114–124, June 2008.

[17] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998.

[18] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006.

[19] G. Lu et al. The MorphoSys parallel reconfigurable system. In *Proc. of the 5th International Euro-Par Conference*, pages 727–734, 1999.

[20] A. Mahesri et al. Tradeoffs in designing accelerator architectures for visual computing. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 164–175, Nov. 2008.

[21] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the 2003 Design, Automation and Test in Europe*, pages 296–301, Mar. 2003.

[22] B. Mei, A. Lambrechts, J. Y. Mignolet, D. Verkest, and R. Lauwereins. Architecure exploration for a reconfigurable architecture template. In *Proc. of the 2005 Design, Automation and Test in Europe*, pages 90–101, Mar. 2005.

[23] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, June 1997.

[24] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, Oct. 2008.

[25] M. Quax, J. Huisken, and J. Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proc. of the 2004 Design, Automation and Test in Europe*, pages 230–235, Mar. 2004.

[26] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[27] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[28] Tensilica Inc. *Diamond Standard Processor Core Family Architecture*, July 2007. http://www.tensilica.com/pdf/Diamond WP.pdf.

[29] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, Dec. 2007.

[30] M. Woh et al. From soda to scotch: The evolution of a wireless baseband processor. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 152–163, Nov. 2008.

[31] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker. A distributed control path architecture for VLIW processors. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, Sept. 2005.