Resource Recycling: Putting Idle Resources to Work on a Composable Accelerator

Yongjun Park[1], Hyunchul Park[2], Scott Mahlke[1], and Sukjin Kim[3]

[1]Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI
{yjunpark, mahlke}@umich.edu

[2]Texas Instruments, Inc.
Houston, TX
{parkhc}@ti.com

[3]Samsung Advanced
Institute of Technology
Giheung, Republic of Korea
{sukj.kim}@samsung.com

ABSTRACT

Mobile computing platforms in the form of smart phones, netbooks, and personal digital assistants have become an integral part of our everyday lives. Moving ahead to the future, mobile multimedia support will become a key differentiating factor for customers. Features such as high-definition audio and video, video conferencing, 3D graphics, and image projection will lead to the adoption of one phone over another. However, in contrast to wireless signal processing which is dominated by vectorizable computation, mobile multimedia applications often contain complex control flow and variable computational requirements. Moreover, data access is more complex where media applications typically operate on multi-dimensional vectors of data rather than single-dimensional vectors with simple strides. To handle these complexities, composable accelerators such as the *Polymorphic Pipeline Array*, or PPA, present an appealing hardware platform by adding a degree of hardware configurability over existing accelerators. Hardware resources can be both statically as well as dynamically partitioned among executing tasks to maximize execution efficiency. However, an effective compilation framework is essential to partition and assign resources to make intelligent use of the available hardware. In this paper, a compilation framework is introduced that maximizes application throughput with hybrid resource partitioning of a PPA system. Static partitioning handles part of the resource assignment, but this is followed up by dynamic partitioning to identify idle resources and put them to use – *resource recycling*. Experimental results show that real-time media applications can take advantage of the static and dynamic configurability of the PPA for increased throughput.

Categories and Subject Descriptors

D.3.4 [**Processors**]: [Code Generators]; C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and Embedded Systems]

General Terms

Algorithms, Experimentation, Performance

CASES'10, October 24–29, 2010, Scottsdale, Arizona, USA.

Keywords

Coarse-grained reconfigurable architecture, Composable accelerator, Dynamic partitioning, Modulo Scheduling, Workload balancing

1. INTRODUCTION

The mobile devices market, including cell phones, netbooks, and personal digital assistants, is one of the most highly competitive businesses. The computing platforms that go into these devices must support ever increasing performance capabilities while maintaining low energy consumption. Advanced multimedia and signal processing applications are key drivers. Traditionally, application-specific integrated circuits (ASICs) were used for the heavy lifting to perform the most compute intensive kernels in a high performance but energy-efficient manner. However, several features push designers to a more flexible and programmable solution: supporting multiple applications or variations of applications, providing faster time-to-market, and enabling algorithmic changes after the hardware is constructed.

For wireless signal processing, programmable designs that exploit high degrees of single-instruction multiple-data (SIMD) parallelism have emerged to challenge ASICs [2, 1, 5, 15, 24]. While these solutions suffice for wireless signal processing, multimedia applications contain more complex data dependence patterns and frequent control flow for which wide-SIMD is inefficient. Thus, a different approach is necessary.

Polymorphic pipeline arrays (PPAs) are attractive alternatives for accelerating multimedia applications because the hardware is more flexible and can accelerate the code in multiple ways [19]. Coarse-grain pipeline parallelism is exploited by concurrently executing filters in streaming applications [7, 8, 12], as well as fine-grain instruction level parallelism is also found by modulo scheduling innermost loops [21]. A PPA is a generalization of a coarse-grain reconfigurable architecture (CGRA) shown in Figure 1 [17]. It 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.

While PPAs provide the opportunity for hardware customization, an effective compiler is necessary to configure the hardware to maximize application performance. In this work, we adopt the stream programming paradigm. Stream programming is generally

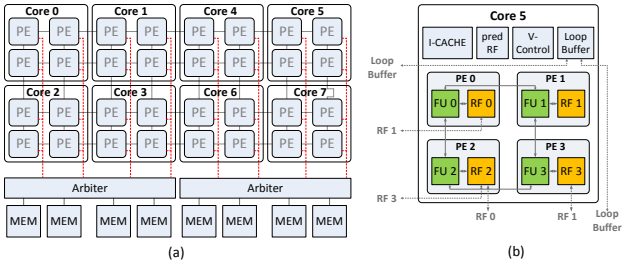

Figure 1: PPA Overview: (a) PPA with 8 cores, (b) Inside a single PPA core

based on synchronous dataflow wherein the application is represented as a directed graph (stream graph) where each node represents an actor and each arc represents the flow of data [13]. The number of data samples produced and consumed by each node are specified a priori. For this work, we focus on stream-style C code where a program is represented as a set of autonomous actors (also called filters) that operate on data and communicate through first-in first-out data channels [23]. During program execution, actors fire repeatedly in a periodic schedule [8]. Each actor has a separate instruction stream and an independent address space, thus all dependences between actors are made explicit through the communication channels. Compilers can leverage these characteristics to plan and orchestrate parallel execution.

Given a streaming application, the primary challenge is to perform resource allocation and assignment so as to achieve maximum throughput. More specifically, a PPA compilation framework must not only partition filters across the available cores, but also aggregate cores together into core-groups to jointly execute the assigned filters. Larger core-groups are effective for long-running filters because higher levels of fine-grain parallelism can be exploited. By modulo scheduling across more resources, higher performance is achieved. However, selecting large core-groups reduces the overall number of groups and hence the amount of coarse-grain pipeline parallelism that can be exploited. Greedily speeding up a small portion of the application often results in poor overall performance. Thus, an intelligent compiler must achieve a balance.

In this paper, the goal is to solve the joint filter assignment and core aggregation problem for mapping streaming applications onto a PPA. We start by defining the main scheduling constraints on PPA architectures, and propose a new compilation process to solve the difficulties. In this framework, we adapt the key concept from the stream graph modulo scheduling algorithm for coarse-grain parallelism [12]. The main difference is that parallel composition of the each filter is not performed with split-joins, but by modulo scheduling across larger core-groups. With this change, the PPA compiler can be used for more generic code by removing the restrictions of static data rates on stream programming languages like StreamIt [23]. Edge-centric modulo scheduling (EMS) [18], which focuses on routing of values between functional units, is used as the modulo scheduling technique for exploiting fine-grain parallelism.

The compilation process consists of three steps. First, filters are assigned to virtual cores using static partitioning and an approximate load balancing algorithm. Next, core allocation is performed to map the virtual cores to the physical cores considering core locations and the inter-filter communication patterns. Finally, fine-grain dynamic partitioning is performed to identify and recycle underutilized resources.

This paper offers the following three contributions:

- An analysis of the scheduling difficulties for composable accelerators such as the PPA.
- A compilation framework for jointly partitioning streaming applications across hardware resources and selecting resource aggregations that jointly exploit coarse-grain parallelism between filters and fine-grain parallelism within filters.
- An efficient resource borrowing technique is proposed to reduce the execution time of the largest coarse-grain pipeline stage by borrowing resources from underutilized stages.

2. BACKGROUND AND MOTIVATION

2.1 Composable Accelerators

As chip multiprocessors (CMPs) have become commonplace in today's desktop environment, their importance is growing rapidly in the mobile environment. The disparity between the granularity of parallelism in workloads and the granularity of processing cores inspired a flexible execution model that allows the aggregation of small cores to create larger logical cores [11],[10].

Composable accelerators are multi-core accelerator designs that incorporate this flexible execution model in embedded systems. Multiple small cores enable the parallel execution of individual tasks, exploiting task level parallelism. Additionally, when there is a high degree of parallelism within a task, such as loop level parallelism or instruction level parallelism, a larger core can be created by merging small cores. With this flexible execution model, different levels of parallelism can be exploited with a single piece of hardware.

Our specific compilation target is the *Polymorphic Pipeline Array* (PPA) shown in Figure 1. A PPA is a composable accelerator for embedded systems that can exploit both the fine-grain parallelism found in innermost loops and the pipeline parallelism found in streaming applications. A PPA consists of multiple simple cores that are tightly coupled to neighboring cores in a mesh-style interconnect. A PPA with 8 cores is shown in Figure 1(a). There are a total of 32 processing elements (PEs) in this PPA, each containing one function unit (FU) and a register file (RF). 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 4 PEs to a memory access arbiter that provides sharing of scratch pad memories among the cores.

The detailed diagram of a single PPA core is shown in Figure 1(b). Each PE contains a 32-bit FU and a 16 entry register file. PEs are connected to a mesh-style interconnect. The distributed nature of PPA provides low power consumption and hardware cost making it an attractive solution for embedded systems. The mesh interconnect also connects the neighboring PEs in different PPA cores. This allows fast inter-core communication for mapping compute intensive loop nests across multiple cores. A detailed description of PPA cores can be found in [19].

2.1.1 *Supporting Different Levels of Parallelism*

The major feature of the PPA is its ability to exploit both fine-grain and coarse-grain pipeline parallelism. Since each PPA core can process its own instruction stream, coarse-grain parallelism can be exploited for streaming applications. The communication between pipeline stages can be efficiently supported with DMA connections between cores. Abundant fine-grain parallelism within a pipeline stage can also be exploited by aggregating multiple cores to form a larger logical core allowing for maximized resource utilization. This is efficient since the PPA provides fast inter-core communication using a mesh-style interconnect.

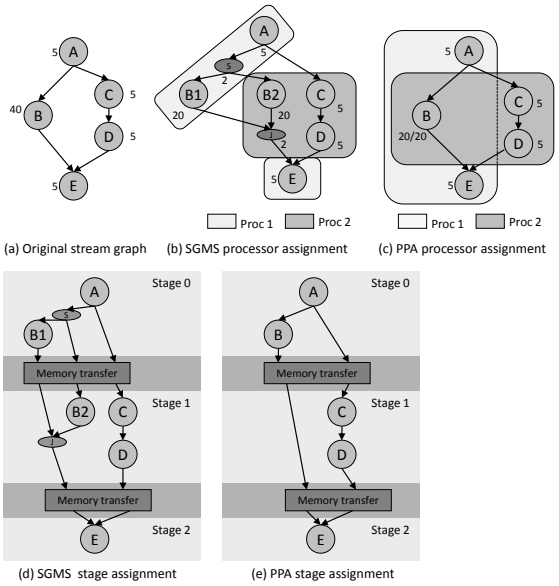

Figure 2: Example of processor and stage assignment for SGMS and PPA scheduling.

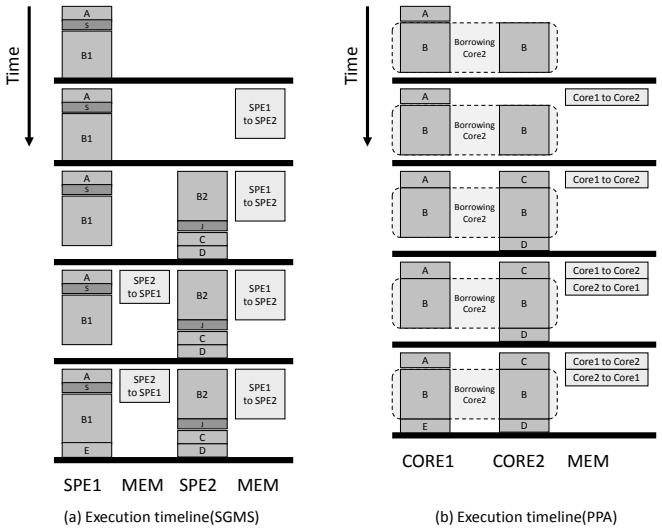

Figure 3: Example of running a SGMS on multi-core and a modulo scheduling on PPA.

2.1.2 Virtualization

One of the major characteristics of a PPA is virtualized execution of software pipelined loops [19]. Virtualized modulo scheduling generates a unified schedule that can be mapped onto different target sub-arrays of the PPA. At runtime, the PPA cores are dynamically merged to create larger logical cores based on the resource availability. With virtualization support, tasks can execute on different sized cores without rescheduling, improving the overall performance when the resource requirement in the workloads varies dynamically during execution [19]. However, there are some limitations of virtualization on a PPA, such as sub-optimality of the unified schedules and runtime overhead for virtualization.

2.1.3 Partitioning Schemes

Static Partitioning. The PPA array can be partitioned statically based on the resource requirement of each coarse-grain pipeline stage. Static partitioning has its benefit in achieving high quality schedules, but it cannot adapt to dynamically changing resource availability. When an application has a large variation in execution pattern, static partitioning can either result in low utilization of resources, or may not be able to fully accelerate the application when there are not enough resources available.

Dynamic Partitioning. Coarse-grain pipeline stages in multimedia applications have different execution patterns, resulting in fluctuating resource requirements. Dynamic partitioning can come in handy with the presence of dynamic variation of resource requirements. The partitioning of the PPA array can change during runtime on demand. For a single pipeline stage, a single core can be assigned to an acyclic region of code, but more resources can be assigned to the compute intensive loop kernels to exploit fine-grain parallelism. Dynamic partitioning assumes the sharing of resources between neighboring pipeline stages. The resources sitting idle in one stage can be utilized by neighboring stages through resource borrowing. So, it is not guaranteed that the required resource is available at all times in dynamic partitioning. When the required resource is not available, the stage stalls and waits for the resource. Virtualization can avoid stalls due to resource contention by generating a schedule that can be modified easily at runtime to run on different number of resources.

2.2 Stream Graph Modulo Scheduling

This paper presents a compiler technique specifically for composable accelerators based on *stream graph modulo scheduling*, or SGMS [12]. SGMS is a modulo scheduling algorithm for mapping streaming applications onto multicore systems. Modulo scheduling is traditionally a form of software pipelining applied at the instruction level to find a valid schedule for a loop such that the interval between successive iterations (initiation interval, or II) is minimized [21]. SGMS is the same technique on a coarse-grain stream graph to pipeline the actors across multiple cores. The objective is to maximize concurrent execution of actors while hiding communication overhead to minimize stalls.

SGMS consists of two steps: 1) integrated fission and processor assignment and 2) stage assignment. The first step is to assign actors to each processor with maximum load balance using an integer linear program formulation. Stateless data actors are replicated and fissed to achieve even work distribution. In stage assignment, the compiler decides a pipeline stage for each actor at runtime. The optimization process in this stage is to maximally hide inter-processor communication latency and not to violate data dependences.

Even though this paper adapts the basic concept of the SGMS, task scheduling in PPAs is different in several aspects. First, the PPA scheduler is proposed using legacy C code, hence it has less restrictions than SGMS using streaming languages such as StreamIt. For example, SGMS can exploit parallelism for only stateless actors, but modulo scheduling also can be applied to stateful actors. In addition, PPAs do not incur fission overhead (split, join) to assign multiple cores due to the tightly coupled inter-core scalar network for aggregation.

Figure 2 shows the differences between SGMS and PPA scheduling. Given an example stream graph (Figure 2(a)), all actors are assumed data parallel. When SGMS schedules the graph on 2 processors (Figure 2(b)), the resultant II is 32 because the slowest node B is fissed once and corresponding split-join overhead is incurred. Figure 2(c) is the resultant schedule for the PPA, enabling the processor assignment to achieve an II of 30 as node B is accelerated

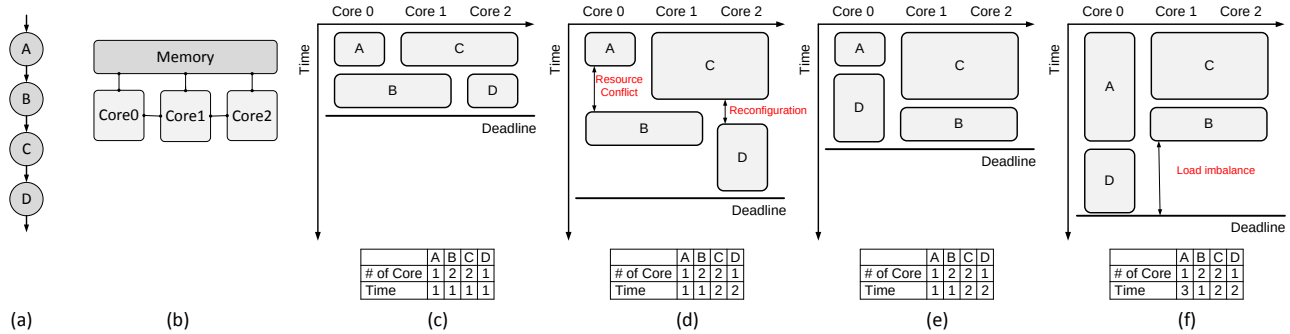

Figure 4: Examples of the runtime overhead: (a) original task graph, (b) simple 1x3 PPA, (c) expected ideal schedule with high resource utilization, (d) runtime overhead: stall, reconfiguration time, (e) static partitioning with low runtime overhead, (f) a possible problem of the static partitioning: workload imbalance.

by core aggregation without overhead. Finally, Figure 2(e) shows the stage assignments for PPA schedule in which the entire node B is executed in stage 0 within 20 time units by using both cores.

Figure 3 shows the execution timeline of both SGMS and PPA schedules. The main difference between the two schedules is the locations of node B: it is split into two independent pieces using SGMS on a multicore and with the PPA it is executed as a whole by aggregrating the resources of both core 1 and 2. Note that with the PPA, node B must be scheduled at the same time on both cores in order to exploit resource aggregration. Another interesting point is that since tightly-coupled memory system in the PPA provides lightweight memory synchronization mechanism, scheduling is more tolerable to high memory transfer.

2.3 Compilation Challenges

Efficient scheduling for composable accelerators is now emerging as an interesting, and challenging problem due to the high degree of freedom in both the hardware and software. Some factors that make scheduling difficult are:

Resource Requirement Variance: The optimal resource requirement for efficient parallelism depends on the task-specific characteristics. For example, cyclic code regions can be accelerated efficiently by appropriating more resources, but the performance of acyclic code with sequential dependences cannot be improved by supplying additional resources [20]. Assuming worst-case requirements for all code segments leads to either over-provisioned designs to achieve a desired performance or under-performance for a fixed design.

Execution Time Variance: Composable accelerators typically have multiple tasks running in parallel, and they usually have complex dependences. Thus, it is hard to predict the resource usage pattern and accommodate the optimal execution of multiple instruction streams.

Geometry: In CMPs, full connectivity between processors is often provided. However, in a low-cost accelerator, the interconnect is much more sparse and merging cores should be performed in a connectivity-aware manner.

To illustrate these difficulties, Figure 4 shows some simple, but frequently occurring examples that result in resources being wasted. The simple dataflow graph (DFG) in Figure 4(a) is being scheduled on a simple composable architecture (Figure 4(b)). Assuming the optimal resource requirements of each node(A, B, C, D) is 1, 2, 2 and 1 cores with the same execution time, the expected schedule is similar to Figure 4(c). However, even though the optimal number of cores is assigned, the different amounts of work in each node results in different execution times. On top of that, if C and D have long execution time, node B cannot start execution at the completion of task A, but must wait until the execution of node C is finished because of resource conflicts (Figure 4(d)). Another potential source of resource waste occurs when changing the core assignment. In Figure 4(d), task D is delayed by the reconfiguration time even though enough resources are available.

Static partitioning of the cores can potentially eliminate these problems, such as stalls and reconfiguration overhead (Figure 4(e)). Static partitioning means the core aggregation is not changed at runtime and each task is assigned to a suitable merged core. In this scheme, task A is not preferred to be executed in core group (1, 2) because the best resource requirement for A is one core. If A is assigned to 2 cores, resources cannot be utilized sufficiently. However, the workload of each core may not be balanced well because we categorized all the tasks based on optimum resource requirements (Figure 4(f)). To minimize this side effect, a final performance tuning phase is performed using dynamic partitioning of cores. For example, task D can be changed to run using 3 cores after final tuning because all the other resources remain idle. Additionally, we also propose a core reallocation mechanism to avoid geometry-based runtime overhead.

In this paper, our work is focused on finding the optimal partitioning of cores for a given task graph rather than changing the task graph itself. Although modifying the task graph is also a common load balancing strategy, it usually cannot be applied well to the graph itself without changing the source code due to the memory and control dependences.

3. COMPILER FRAMEWORK

In this section, we describe our new compilation framework based on the insights discussed in the previous section. The purpose of this framework is to achieve the highest throughput by minimizing stalls due to resource contention and reconfiguration processes. The compilation process consists of three different stages: prepass static partitioning, core allocation, and postpass dynamic partitioning. Prepass heuristically fuses virtual (no geometry information) PPA cores to accommodate larger pipeline stages based on the profile workload information with static partitioning. Core allocation maps the virtual cores onto physical cores, avoiding failures that occur when cores in same group are not connected together. Postpass performs final performance tuning to reduce the completion time of bottleneck pipeline stages by exploiting resource borrowing.

All compilation steps are performed at compile time. Virtualiza-

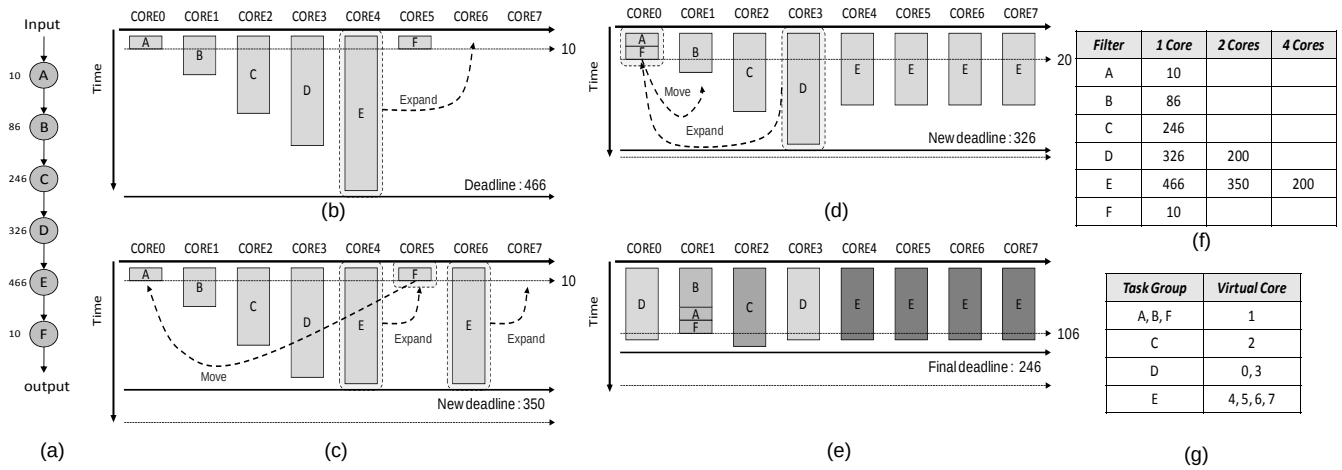

Figure 5: Static Partitioning example: (a) example data flow graph, (b) phase 0: each task is assigned to one core, (c) phase 1: the slowest task E gets one more core to accelerate, (d) phase 2: task E is still the slowest and gets two more cores(5, 7), thus task F loses own core(5), (e) phase 3: new slowest task D is accelerated as getting more core(0) and finally task C with one core(2) takes the maximum execution time, (f) execution time estimate table , (g) final core assignment: D has 2, E has 4 cores.

tion is not considered in this framework because of performance overheads, both on the hardware and compiler sides. For the hardware, a virtualization controller has execution time overhead for checking the resource availability of the neighbor cores. In addition to this, virtualized modulo schedule also has some performance degradation as it generates only one schedule to support various core configurations [19]. Despite these performance overheads, virtualization can improve the overall performance in specific situations, such as when running an application on a small number of resources or running an application with huge dynamic variance [19]. However, we just generate one schedule per stage and disable virtualization even when using dynamic partitioning to evaluate the real effectiveness of this strategy.

3.1 Prepass: Static Partitioning

As we discussed in Section 2.3, the goal of this compilation stage is to minimize idle and reconfiguration time between tasks and to create high quality schedules that maximize resource utilization in order to minimize execution time of assigned work. To achieve this goal, we propose resource grouping using static partitioning. This section describes our method for effectively grouping tasks requiring similar number of cores. The performance improvement achieved by this stage mainly comes from recognizing the huge variance between the optimal resource requirements and execution times of each task. The key idea is to categorize all tasks into some number of available resource combinations, enabling high utilization and assigning the different portions of composable cores based on this information. This method basically enables all the tasks to use the resources efficiently, achieving high throughput. This stage also performs coarse load balancing because the throughput of the program depends on the slowest pipeline stage. Therefore, imbalance between core groups leads to performance degradation even if all the tasks execute efficiently. Load balancing is also performed in the postpass step after identifying the optimal static partition with maximum resource utilization.

Algorithm 1 shows how the optimal core groupings (to support the assigned tasks) are identified to exploit fine-grain parallelism effectively. The general idea is to heuristically assign more cores to larger tasks based on the execution time estimate. However,

Algorithm 1 Prepass: Static Partitioning Algorithm

Input: *G:(V, E)*, *#virtualCores*, *balance, quality*
1: *groups* ← `PartitionGraph`(*G, #virtualCores*);
2: **while** *true* **do**
3: `SortGroupsByExecTime`(*groups*);

{ Find task groups with max and min execution time estimate. }
4: *maxTaskGroup* ← `MaxExecTimeTaskGroup`(*groups*);
5: *numCores* ← `NumRequiredCoresToExpand`(*maxTaskGroup*);
6: *minTaskGroups* ← `FindContractTaskGroups`(*groups, numCores*);

{ Generate candidate for new task groups. }
7: *maxTaskGroupCand* ← `ExpandGroup`(*maxTaskGroup*);
8: *minTaskGroupCand* ← `ContractGroup`(*minTaskGroups*);

{ Test the availability of the new task groups. }
9: **if** (`ExecTime`(*maxTaskGroupCand*) > `ExecTime`(*maxTaskGroup*) ∗ *quality* || `ExecTime`(*maxTaskGroupCand*) < `ExecTime`(*minTaskGroupCand*)) **then**
10: Finish;
11: **end if**

{ Update task groups.}
12: `Remove`(*maxTaskGroup*, *minTaskGroups*);
13: `Add`(*maxTaskGroupCand*, *minTaskGroupCand*);

14: **if** (`ExecTime`(*maxTaskGroupCand*) < `ExecTime`(*minTaskGroupCand*) ∗ *balance* || *timeOut*) **then**
15: Finish;
16: **end if**
17: **end while**

assigning too many resources to larger cores may not be the best solution because performance enhancement depends on the task-specific characteristics and may result in missed opportunities to accelerate other tasks, given a limited number of cores. Therefore, a *quality factor* is introduced to define the minimum performance gain that must be achieved to justify the assignment of additional cores.

Algorithm 1 starts from assigning one core to each task (Line 1). If the number of tasks is larger than the number of cores, tasks are grouped by the total execution time estimate(`ExecTime`). Based on this initial assignment of one core to each task group, the while loop in Algorithm 1 identifies the optimal number of cores per task group. Line 3-6 finds the task groups with the maximum Exec-

Time(*maxTaskGroup*), and minimum ExecTime(*minTaskGroups*). *maxTaskGroup* is the candidate for receiving more cores to enable faster execution while *minTaskGroups* will potentially lose cores. The number of task groups in *minTaskGroups* varies because number of additional cores, for *maxTaskGroup* to be the larger fused core, are set by the current assigned core topology of *maxTaskGroup* (Line 5) and the mininum ExecTime task group may not have enough number of cores to give. In this case, an additional second minimum ExecTime task group is required. If current *maxTaskGroup* has 1 core with 1x1 configuration, just 1 more core is required to be 1x2 or 2x1 array-style fused core. However, if current configuration of *maxTaskGroup* is 1x2 with 2 cores, 2 more cores are required to expand because 1x3 or 3x1 array-style core group is not allowed and next available core configuration is 2x2, 1x4, or 4x1 with 4 cores on current PPA. Moreover, an additional task group may be required to subsume the tasks from the minimum task group if the minimum workload group loses all its assigned cores. Then, line 7-8 creates the candidates of new maximum and minimum task groups given the new core assignments. `ExpandGroup` is the function for *maxTaskGroup* to get more cores to accelerate execution and `ContractGroup` is to take cores from *minTaskGroups*. Line 9-11 checks the benefit of these new resource assignments and determines whether new combinations are updated. First, the new ExecTime estimate of *maxTaskGroupCand* should be less than some relative ratio of the original ExecTime(*example quality factor = 0.9*), meaning that the performance gain should be at least 10%. Also, the ExecTime estimates of the *minTaskGroupsCand* should not become a new bottleneck. Line 12-13 updates the changes to the core assignment and this process is repeated until the load imbalance is less than the balance factor or the task group combination does not change within the defined timeout period.

Figure 5 shows an example of the prepass static partitioning algorithm. An original task graph (Figure 5(a)) with 6 nodes is mapped onto a PPA with 8 cores. The original graph only has 6 nodes and each node is initially scheduled using 1 core. The annotated numbers show the ExecTime estimate for each node. The prepass algorithm performs ExecTime estimation of the partitions then tries to appropriate more cores to the heavier workloads to balance the task groups. More specifically, node E is *maxTaskGroup* and gets 1 additional core because 2 cores are idle (Figure 5 (c) Phase 1). Then, node E is selected again as *maxTaskGroup* because the reduced ExecTime is still the highest at 350. In this case, an idle core and another core is selected to accelerate node E. As a result, node E is scheduled with 4 cores. Since node F lost all its assigned cores, it is merged into another task group with minimum ExecTime estimate, node A(Figure 5 (d) Phase 2). *maxTaskGroup* then becomes the task group with node D and is accelerated by taking one more core from nodes A and F. Again, nodes A and F lost all the cores and are merged into node B(Phase 3). At Figure 5 (c) Phase 3, the process is finished since it meets the balance condition (example balance factor 2.5) and 8 cores are divided as 4 task groups with different core numbers (Figure 5 (e)).

3.2 Core Allocation

After static partitioning, the number of PPA cores assigned to each task group is known, but their relative positions on the PPA array is not determined yet. Core allocation maps virtual PPA cores assigned to task groups onto the physical structure of the PPA. As discussed in Section 2.3, most composable accelerators, including PPA, provide limited interconnects. The fast scalar network connecting adjacent cores in PPA can be utilized to exploit fine-grain parallelism. So, cores assigned to the same task group are placed next to each other. Core allocation also attempts to place cores assigned to task groups with maximum ExecTime next to the cores with minimum ExecTime. This is to increase the opportunities for dynamic partitioning in postpass. With dynamic partitioning, idling resources can also be loaned to the neighboring task groups, further increasing the resource utilization. Algorithm 2 shows the process for core allocation. First, all the task groups are sorted by ExecTime estimates. In each attempt, the *maxTaskGroup* and the *minTaskGroups* are identified(lines 3 - 5) with Prepass-similar process, and they are placed closely on the PPA array to enable sharing cores at runtime(lines 6 and 7). Continuing the example from the prior section, Figure 6 shows the core allocation results and the slowest task group (C) is assigned the core next to the core reserved for the fastest task group (A, B, F).

Algorithm 2 Core Allocation: Physical Core Mapping

Input: *groups*, *#physicalCores*
Output: *phyTaskGroups*
1: `SortGroupsByExecTime`(*groups*);
2: **while** `HasGroup`(*groups*) **do**
3: *maxTaskGroup* ← `MaxExecTimeTaskGroup`(*groups*);
4: *numCores* ← `NumRequiredCoresToExpand`(*maxTaskGroup*);
5: *minTaskGroups* ← `MinExecTimeTaskGroups`(*groups, numCores*);

{ Assign physical cores.}
6: `SetPhysicalCores`(*maxTaskGroup*);
7: `SetPhysicalCores`(*minTaskGroups*);

{ Update task groups.}
8: `Remove`(*maxTaskGroup*, *minTaskGroups*, *groups*);
9: `AddTo`(*maxTaskGroup*, *minTaskGroups*, *phyTaskGroups*);
10: **end while**

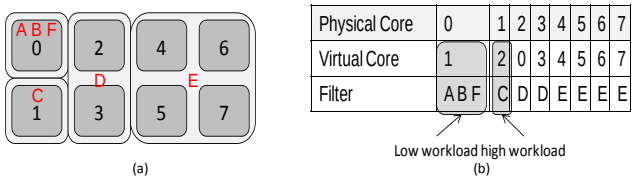

Figure 6: Core Allocation example: (a) physical placement of cores, (b) the slowest group is placed next to the fastest group.

3.3 Postpass: Dynamic Partitioning

In this section, we propose the final performance acceleration process: dynamically adjusting the resource assignment of the bottleneck task groups. The basic concept is to accelerate the slowest stage by dynamically acquiring the idle resources of neighboring cores at runtime. While the static partitioning achieves a good load balancing of PPA cores, workload variation still exists leaving some time slack for cores assigned to lightly loaded task groups. The idle time of cores can be exploited by neighboring cores using dynamic partitioning proposed in this section.

Algorithm 3 begins the optimization process by constructing the group adjacency information map (Line 1). The compiler automatically identifies which group is physically adjacent based on the PPA core connection information. Then, it identifies the slowest task groups and tries to find physically connected task groups. Among these task groups, the task group with the lowest ExecTime estimates is selected (Line 4-7). Line 8 calculates the performance estimate when dynamic partitioning is enabled between these groups. In this process, only tasks from the maximum ExecTime task group are allowed to execute with dynamically varying resources. The other task groups are restricted to their initial static resource assignments. This is to limit dynamic resource assignment

Algorithm 3 Postpass: Dynamic Partition Algorithm

Input: *phyTaskGroups*, *sharing*

```
1:  adjMap ← ConstructAdjacentMap(phyTaskGroups);
2:  while true do
3:     SortGroupsByExecTime(phyTaskGroups);
       { Find task groups with max and min execution time estimate. }
4:     maxTaskGroup ← MaxExecTimeTaskGroup(phyTaskGroups);
5:     nextMaxTaskGroup ← NextMaxExecTimeTaskGroup(nextMaxTaskGroup);
6:     numCores ← NumRequiredCoresToExpand(maxTaskGroup);
7:     minTaskGroups ← MinExecTimeAdjacentGroups(phyTaskGroups,
       numCores, adjMap);

       { Test the availability of dynamic partitioning of shared execution.}
8:     newMaxTaskGroupExecTime, newMinTaskGroupsExecTime
       ← EstimateExecTimeSharing(maxTaskGroup, minTaskGroups,
       sharing);
9:     if (newMaxTaskGroupExecTime < ExecTime(maxTaskGroup)
       && newMinTaskGroupsExecTIme < ExecTime(maxTaskGroup)) then
10:       UpdateSharing(maxTaskGroup, minTaskGroups, groups);
11:    end if

12:    if (newMaxTaskGroupExecTime > ExecTime(nextMaxTaskGroup)) then
13:       Finish;
14:    end if
15: end while
```

only to the performance limiting groups to minimize the reconfiguration overhead. The compiler identifies resource-constrained loop nests in the *maxTaskGroup* that can further exploit fine-grain parallelism with the extra resources. Then, the compiler gradually changes the resource assignment for the loop nests, until the ExecTime estimate of the *minTaskGroups* reaches a performance threshold. This threshold is set to the relative ExecTime of the second limiting group (*nextMaxTaskGroup*). The *sharing* coefficient is introduced to determine the threshold and it depends on the application characteristics (dynamic variance) for each task at runtime. For example, a stage execution time of AAC fluctuates between 150k and 200k cycles [19], and the coefficient will be smaller than 0.75 considering dynamic overhead. Line 9-11 updates the new assignment if there is any performance gain with the resource sharing. This process will finish if the new ExecTime is still larger than the ExecTime of the *nextMaxTaskGroup*. Another key point of this process is that the *quality factor* is not considered in this phase because the objective of this process is to accelerate the pipeline limiting stage using marginal resources.

An example of the postpass optimization is shown in Figure 7. In this example, the slowest task group(C) and the fastest task groups (A, B, F) are placed next to each other after the core allocation step in Figure 7 (a). The compiler identifies five candidate loop nests in task group C, and two of them are rescheduled using the additional resources(cores 0 and 1). The final result in Figure 7(b) shows that the pipeline deadline decreases from 246 to 200 cycles, achieving 20% performance gain for this stage. The overall resource utilization is improved by recycling the wasted resources of core 0 between cycle 106 to 197.

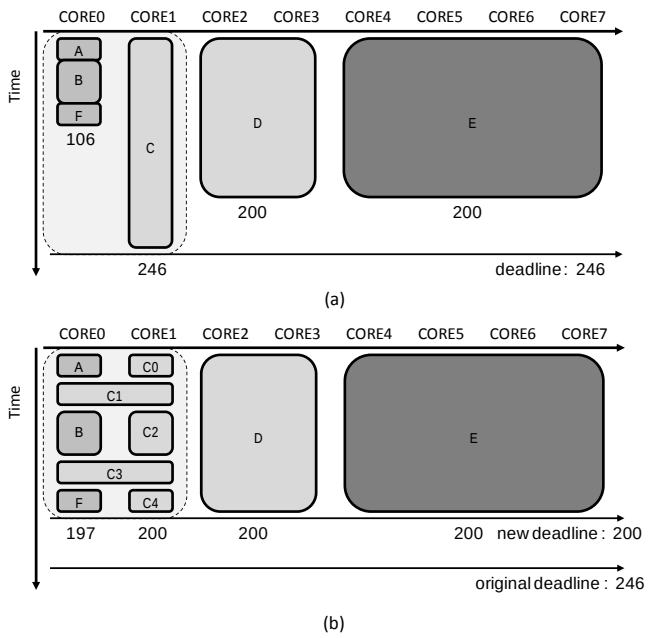

Figure 7: Dynamic Partitioning example: (a) coarse-grain pipeline using static partitioning, (b) coarse-grain pipeline with final performance tuning process

4. EXPERIMENTAL RESULTS

This section presents the results of the experimental evaluation of proposed high-level compilation techniques. We first present a brief explanation of the target architecture and benchmark applications. Performance measurement for prepass and postpass processes is explained based on the experimental environment described below.

4.1 Experimental Setup

Target Architecture PPAs are used to evaluate the performance of the compilation techniques. The PPA has 8 cores in the form of a 2×4 array as shown in Figure 1. Virtualization controller is disabled to evaluate the real performance of the compilation strategy. For the experiments using less than 8 cores, PPA is partitioned into two parts and the unused partition is disabled.

Target Applications and fine-grain parallelism To evaluate the performance, we used three application domains: audio decoding (aac), video decoding (h.264) and 3D graphics (3d). All software-pipelineable loops from these applications are taken and scheduled using edge-centric modulo scheduling with all available partitions. Topology of the core groups are also considered. For example, 2x1 and 1x2 core groups with 2 cores are individually scheduled. Performance is evaluated using the overall execution time.

For coarse-grain pipelining, three applications are split into multiple tasks that communicate in a feed-forward fashion and without any inter-iteration dependencies contained within a single task. Each task is able to have both loops and acyclic blocks of code. Based on the control and data dependency restrictions, aac, 3D, h.264 have 10, 5, and 3 tasks on experiments.

4.2 Performance Evaluation

Figure 8 shows the relative speedup obtained by various partitioning algorithms on 4 to 8 cores. Symmetric partition means that each task is scheduled using the same number of cores. If the number of tasks is smaller than the number of cores, the cores are divided by the number of tasks and each task has its own partition. If the tasks are more than the cores, the overall application is split by the number of cores and each task group is executed using one core. Smart partitioning means manually divided static partition based on the application characteristics. For example, tasks containing substantial portion of loops are executed on a large core group to exploit fine-grain parallelism and the others are run on only one core. Static partitioning represents the execution result when the program runs on an automatically divided partition with prepass. In dynamic partitioning, the program executes on the same parti-

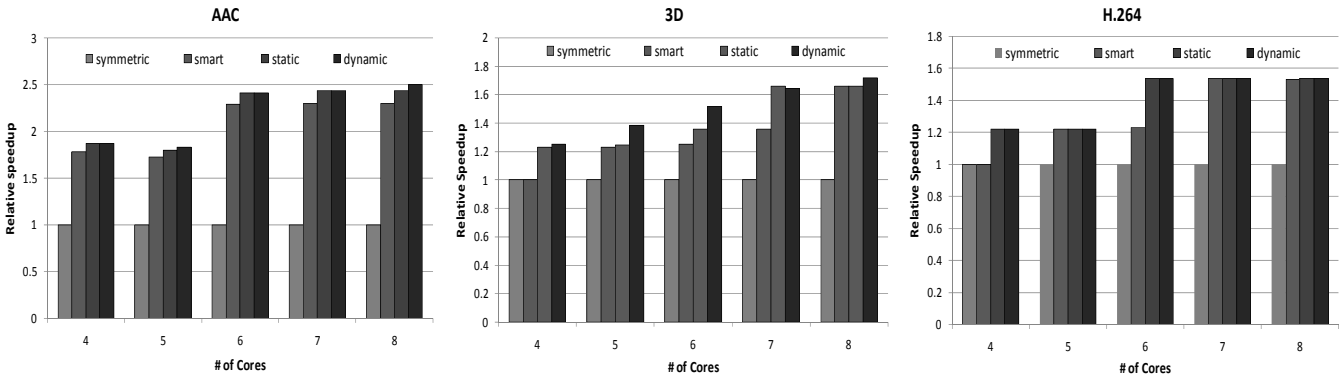

Figure 8: Relative speedup normalized to simple symmetric partitioning

tion with static partitioning and dynamic reconfiguration is applied as well.

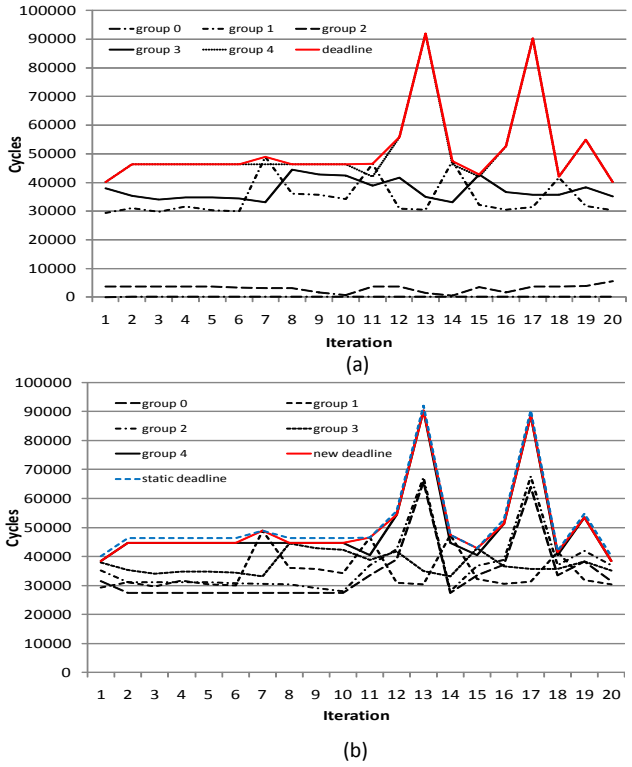

Figure 9: Stage execution time for AAC benchmark: (a) dynamic computation variance on static partitioning, (b) pipeline deadline reduction with dynamic partitioning

4.3 Static Partition

As shown in Figure 8, smart partitioning always outperforms symmetric partitioning by a significant amount because most of the loop-intensive task groups are accelerated using fine-grained pipelining. The promising point is that manual partitioning cannot achieve better throughput than our static partitioning algorithm, and the speedup of static partitioning on aac benchmark is always better than smart partitioning. As other benchmarks have small number of tasks, 3 and 5, manually partitioning with traditional load balancing algorithm can achieve the same speedup as with using the same partitioning with the result of prepass. However, if the application can be split into multiple subsets of tasks, our prepass optimization is able to minimize the performance degradation induced by low quality schedule, stall, and reconfiguration overhead. Note that tasks cannot be divided for perfect load balancing because memory and control dependences on the program prevent tasks from being partitioned from the middle. Despite these inherent difficulties, our algorithm successfully finds the throughput limiting tasks and accelerates them. On an 8-core PPA, static partitioning allows 2.44x, 1.66x and 1.66x speedup over symmetric partitioning.

4.4 Dynamic Partition

AAC Figure 8 shows that postpass with dynamic partitioning is effective when the number of cores are 5 and 8 but the gain is small, 1.7% and 2.8%, respectively. This is because the task group with the largest execution time on AAC application consists of a large amount of sequential code and a small portion of the software-pipelineable code. In prepass, this huge sequential task cannot reserve enough cores because of the low quality schedule and remains the performance bottleneck. This task is then accelerated by sharing its neighbors' resources during postpass since it doesn't need to meet the quality factor any more, hence the final performance is slightly enhanced by using the neighbor's idle resource.

Runtime observations of the real execution on both static partitioning and dynamic partitioning are shown in Figure 9. Figure 9 (a) shows that task group 4 is the performance bottleneck over time and execution times of task group 0 and 2 are small. Core allocation process places the cores, assigned to these three task groups, next to each other and group 4 gets some performance gain as shown in Figure 9 (b). Despite the small performance gain of group 4, 0 and 2 have substantial runtime overhead because these groups should share the low quality schedule.

Cores	Perf (smart)	Perf (static)	Perf (dyn)	Overall
4	1.79	1.05	1	1.87
5	1.73	1.05	1.02	1.83
6	2.29	1.05	1	2.41
7	2.30	1.06	1	2.44
8	2.30	1.06	1.03	2.50

Table 1: Relative speedup for AAC benchmark (normalized to the preceding column).

3D Rendering 3D rendering application has 5 tasks, two with small acyclic code and three with big software-pipelineable code.

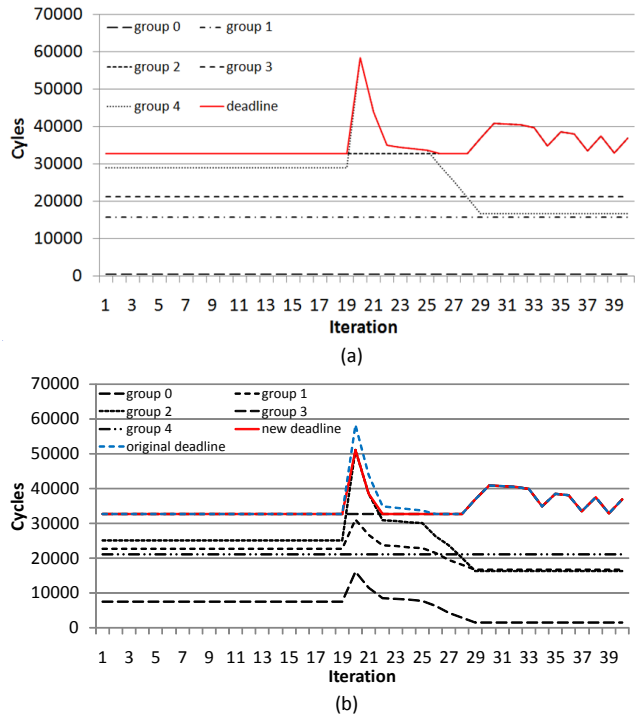

Figure 10: Stage execution time for 3D benchmark: (a) dynamic computation variance on static partitioning, (b) pipeline deadline reduction with dynamic partitioning

Dynamic partitioning increases the throughput by a large amount for all the cases because three huge tasks, which are easy to accelerate by fine-grain parallelism, have similar workload and quality of the schedule is still high when sharing the resources at runtime. The performance gain is up to 11.5% compared to static partitioning, just with reusing idle resources. Figure 10 shows how dynamic reconfiguration efficiently decreases the execution time of the slowest task group. On iteration 19-23, task 4 takes up to 60000 cycles to render 3D images and this work is finished in 50000 cycles by resource borrowing from task 0 and 1. After dynamic performance tuning, execution time on task 0 and 1 increases a large amount to help task 4 finish early on iteration 19-23.

Cores	Perf (smart)	Perf (static)	Perf (dyn)	Overall
4	1	1.23	1.02	1.25
5	1.23	1.01	1.11	1.38
6	1.25	1.09	1.11	1.52
7	1.35	1.22	0.99	1.65
8	1.66	1	1.03	1.72

Table 2: Relative speedup for 3D benchmark (normalized to the preceding column).

H.264 For H.264 benchmark, dynamic reconfiguration is not enabled because execution time of the performance limiting task group fluctuates too widely and is sometimes even smaller than the fastest task group. Therefore, the compiler decides not to adapt dynamic partitioning because runtime overheads of the fastest stage are much bigger than the gains of the limiting task and the overheads may adversely affect the final performance as the fastest task becomes the slowest. Figure 11 shows that execution time changes by a huge amount and sometimes is even lower than the fastest task. In this case, the compiler does not allow dynamic reuse of the neighbor resources because adopting dynamic partitioning is optional.

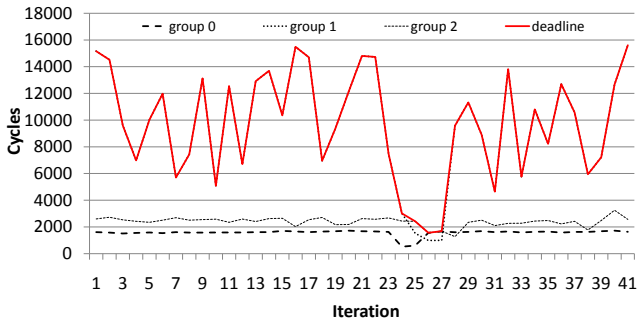

Figure 11: Stage execution time for H.264 benchmark: dynamic partitioning is not applied due to huge dynamic variance.

Cores	Perf (smart)	Perf (static)	Perf (dyn)	Overall
4	1	1.22	1	1.22
5	1.22	1	1	1.22
6	1.23	1.25	1	1.54
7	1.54	1	1	1.54
8	1.53	1.08	1	1.66

Table 3: Relative speedup for H.264 benchmark (normalized to the preceding column).

5. RELATED WORK

Architectures: Combining cores to create a bigger logical core is relatively a new technique, recently proposed by Core fusion [10] and Composable Lightweight Processors [11]. 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 [10] and [11] 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 are simple in-order cores similar to clustered VLIW processors [25]. 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 [16] is a reconfigurable architecture where PEs are connected to a mesh-style interconnect. Modulo scheduling using 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 [6] 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 pipelining while PPA can partition the array differently as to the charac-

teristics of the target application. RaPiD [4] is another CGRA that consists of heterogeneous elements (ALUs and registers) in a 1-D layout, connected by a reconfigurable interconnection network.

Exploiting Parallelism: Exploiting coarse-grained pipeline parallelism is one of the most attractive approaches to accelerate single thread performance as multicore architectures enter the mainstream. Even this type of parallelism has many advantages compared to other types of parallelism, adapting in real situation is difficult because of program-inherent data dependences [22]. To overcome this difficulty, [22] has proposed a dynamic analysis tool to extract a stream graph from legacy C code in order to give a programmer hints for manual parallelization. [22] also tries load balancing by changing a program but this paper's focus is more on compile time optimization for given program. [8] and [12] are similar to this paper to exploit coarse-grained pipeline parallelism but the parallelization mechanism is limited only to stateless components as using StreamIt language. Our work also considers composable architecture specific features such as resource conflict and reconfiguration overhead whereas these works targeted fixed multi-core solutions(RAW architectures [14] and Cell processors [9]). Resource borrowing on dynamic partition is a similar concept to Work stealing [3] but our approach is performed in more fine-grained level, not thread level.

6. CONCLUSION

The popularity of mobile computing platforms has led to the development of feature packed devices that support a wide range of software applications, ranging from high-definition audio and video to high-end 3D graphics. However, the variable resource requirements and complex data/control flow of these workloads limit the applicability of traditional acceleration techniques. In response, this paper proposes a novel, efficient compilation framework to enhance the throughput by maximizing resource utilization of a composable accelerator called a polymorphic pipeline array. The compilation consists of three phases: static partitioning into task groups, physical core allocation, and dynamic partitioning to reclaim idle resources to accelerate performance bottlenecks. The experimental results show that static partitioning achieves up to 2.44x speedup, with dynamic partitioning achieving even greater success in certain benchmarks.

7. ACKNOWLEDGMENTS

Thanks to Mark Woh, Shuguang Feng and Gaurav Chadha 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 is supported by Samsung Advanced Institute of Technology and the National Science Foundation grant CNS-0964478.

8. REFERENCES

[1] 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.

[2] 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.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 370–380, Dec. 2009.

[20] Y. Park, H. Park, and S. Mahlke. Cgra express: Accelerating execution using dynamic operation fusion. In *Proc. of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–280, Oct. 2009.

[21] 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.

[22] 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.

[23] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[24] 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.

[25] 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.