CGRA Express: Accelerating Execution using Dynamic Operation Fusion

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

ABSTRACT

Coarse-grained reconfigurable architectures (CGRAs) present an appealing hardware platform by providing programmability with the potential for high computation throughput, scalability, low cost, and energy efficiency. CGRAs have been effectively used for innermost loops that contain an abundant of instruction-level parallelism. Conversely, non-loop and outer-loop code are latency constrained and do not offer significant amounts of instruction-level parallelism. In these situations, CGRAs are ineffective as the majority of the resources remain idle. In this paper, *dynamic operation fusion* is introduced to enable CGRAs to effectively accelerate latency-constrained code regions. Dynamic operation fusion is enabled through the combination of a small bypass network added between function units in a conventional CGRA and a sub-cycle modulo scheduler to automatically identify opportunities for fusion. Results show that dynamic operation fusion reduced total application run-time by up to 17% on a 4x4 CGRA.

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

Keywords

Coarse-grained reconfigurable architecture, latency-constrained, modulo scheduling, subgraph accelerator

1. INTRODUCTION

The embedded computing systems that power today's mobile devices demand both high performance and energy efficiency to support various high-end applications such as audio and video decoding, 3D graphics, and signal processing. Traditionally, application-specific hardware in the form of ASICs is used on the compute-intensive kernels to meet these demands. However, the increasing convergence of different functionalities combined with high non-recurring costs involved in designing ASICs have pushed designers towards more flexible solutions that are post-programmable. Coarse-grained reconfigurable architectures (CGRA) are becoming attractive alternatives because they offer large raw computation capabilities with low cost/energy implementations [13, 21, 15]. Example CGRA systems that target wireless signal processing and multimedia are ADRES [16], MorphoSys [13], and Silicon Hive [19].

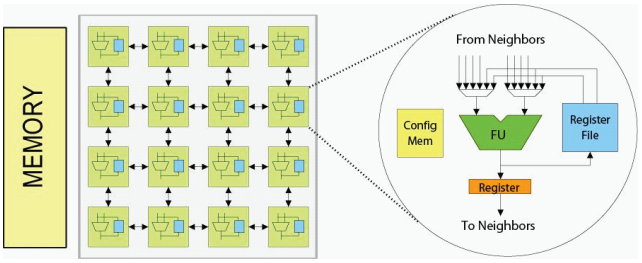

Figure 1: Overview of a 2x2 CGRA.

CGRAs generally consist of an array of a large number of function units (FUs) interconnected by a mesh style network, as shown in Figure 1. Register files are distributed throughout the CGRA to hold temporary values and are accessible only by a small subset of the FUs. The FUs can execute common integer operations, including addition, subtraction, and multiplication. In contrast to FPGAs, CGRAs sacrifice gate-level reconfigurability to achieve hardware efficiency. Thus, CGRAs have short reconfiguration time, low delay characteristics, and low power consumption.

While CGRAs are fully programmable, an effective compiler is essential for achieving efficient execution. The primary challenge is instruction scheduling wherein applications are mapped in time and space across the array. However, scheduling is challenging due to the sparse connectivity and distributed register files. On CGRAs, dedicated routing resources are not provided. Rather, FUs serve as either compute or routing resources at a given time. Therefore, the scheduler must manage the computation, flow, and storage of operands across the array to effectively map applications onto CGRAs. Compilers generally focus on mapping compute-intensive innermost loops onto the array. Early work focused on exploiting instruction-level parallelism [12, 2]. However, these approaches could not make efficient use of the available resources due to limited ILP, thus more recent research focuses on exploiting loop-level parallelism through modulo scheduling [15, 17, 18].

CGRA research has generally focused exclusively on efficiency for throughput-constrained innermost loops. However, real-world

CASES'09, October 11–16, 2009, Grenoble, France.

	sequential region			loop (resource)			loop(dependency)			total	
	#	execution	percentage	#	execution	percentage	#	execution	percentage	#	execution
aac	218	42.6M	71	34	17.1M	28	2	0.3M	0.53	254	60.0M
h.264	639	44.8M	65	78	23.2M	33	1	0.6M	0.84	718	68.6M
3d	752	77.8M	51	82	70.4M	46	13	4.3M	2.81	847	152.5M

Figure 2: Execution time breakdown for three multimedia applications (#: number of basic blocks, execution: number of cycles, percentage: percent of execution cycles). Execution time is broken down into three categories: sequential are all non-innermost loop regions, loop (resource) are inner-most loops whose performance is constrained by the availability of resources, and loop (dependency) are inner-most loops whose performance is constrained by cross-iteration dependences.

media applications consist of more than highly parallel inner loops. Specifically, substantial fractions of time are spent in non-loop or outer loop code, as well as recurrence dominated innermost loops. Traditional CGRAs do not handle such *latency-constrained* code segments in an effective manner as they have no mechanisms to accelerate dataflow graphs that are narrow and sequential. In fact, the majority of the resources sit idle in such situations.

This paper proposes a new technique referred to as *dynamic operation fusion* to accelerate latency-constrained code segments on CGRAs. The core idea is to dynamically configure the existing processing elements of a CGRA into small acyclic subgraph accelerators. Each cycle, any FU can be fused with multiple of its neighbors to create an accelerator capable of executing a small computation subgraph in a single cycle. In essence, small configurable compute accelerators are realized on the array to accelerate sequential code [4]. The necessary hardware extensions for a conventional CGRA are quite simple – an inter-FU bypass network is added between neighboring FUs in the array using a few multiplexors. The compiler scheduler automatically identifies opportunities to accelerate subgraphs by managing the scheduling process at the subcycle granularity. The net result is that the usefulness of CGRAs is extended beyond highly parallel loops to effectively operate in latency-constrained code regions.

The contributions of this paper are as follows:

- An analysis of common media applications to understand the limitations presented by latency constraints.
- CGRA design that supports dynamic operation fusing.
- A compiler scheduler that automatically identifies opportunities for dynamic fusion.
- An evaluation of dynamic operation fusion across a set of media applications.

2. MOTIVATION

2.1 Analysis of Multimedia Applications

To understand the effectiveness and limitations of traditional CGRAs, we examine the characteristics of commonly used multimedia applications. In mobile environments, three of the most widely used multimedia applications are: audio decoding, video decoding and 3D graphics acceleration. We first identify the characteristics of each application, and verify the importance of enhancing performance in latency-constrained code.

2.1.1 Baseline Architecture

In this paper, ADRES[16] is used for the baseline CGRA architecture. This architecture consists of 16 FUs interconnected by a mesh style nework. Register files are associated with each FU to store temporary values. The FUs can execute common integer operations. The architecture has two operation modes: one is CGRA array mode and the other is VLIW processor mode. In CGRA array mode, all 16 computing resources are available and loop-level parallelism is exploited by software pilelining compute-intensive innermost loops. The baseline architecture is also able to function as a VLIW processor to execute sequential and outer loop code. The four FUs in the first row and the central register file support VLIW functionality, while the other components are de-activated. This type of architecture provides high performance by eliminating huge communication overhead to transfer live values between host processor and the array as well as a multi-issue VLIW for non-loop code that is more powerful than a traditional general-purpose processor used as the host (e.g., an ARM-9).

2.1.2 Application analysis

Code of general applications can be categorized into sequential and loop regions. Sequential regions often perform control flow for decision making and handle setup for the compute-intensive loops by transferring live values between loops. Loop regions execute iterative work like calculating pixel data on graphic application. Multimedia applications typically have many compute intensive kernels that are in the form of nested loops. Software pipelining, which can increase the throughput of the innermost nest by overlapping the executions of different iterations, can decrease run time of this type of loops tremendously. In this section, we first decompose applications into various region types. The applications consist of :

- AAC decoder: MPEG4 audio decoding
- H.264 decoder: MPEG4 video decoding
- 3D: 3D graphics rendering accelerator

For our benchmarks, we analyzed the relative importance of sequential and loop regions by analyzing the execution time spent in each. Loops were also categorized loops as their performance was most constrained by resources or cross-iteration data dependences. This grouping provides more precise insights because the characteristics of dependence-constrained loops are more similar to sequential code rather than resource-constrained loops. Performance of the sequential regions was determined by scheduling those onto the VLIW subset of the ADRES CGRA (a 4-wide VLIW) [16]. Modulo scheduling, an efficient software pipelining technique that exploits loop level parallelism by overlapping the execution of different iterations [20], was used to compute the run time of loop regions executing on the 4x4 ADRES CGRA.

Figure 2 presents the execution time breakdown for each benchmark. Software pipelining can successfuly reduce the execution time of loop regions, making it less than 50% of the total execution time. To further improve the overall performance, it is clear that

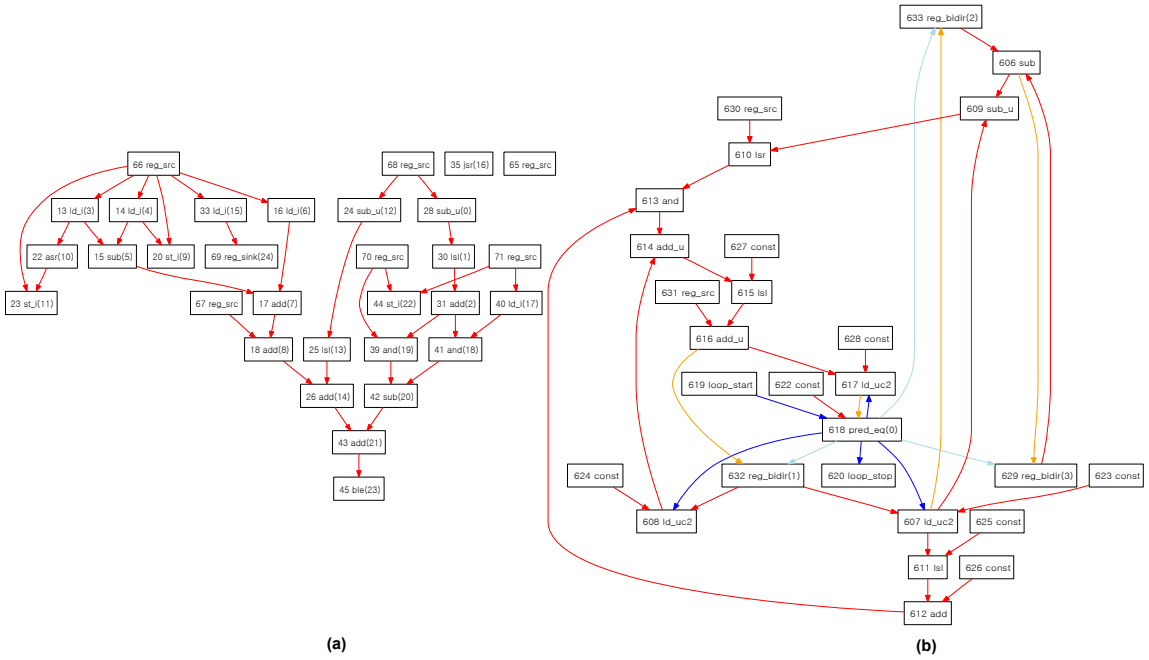

Figure 3: Example dataflow graphs in AAC: (a) Sequential code, (b) Loop code

improving the performance of sequential code regions is critical since they are taking more than 60% of the total execution time.

To get a better understanding of the structure of the code in both the acyclic and loop regions, consider the dataflow graphs in Figure 3 from the AAC benchmark. Figure 3(a) is a data flow graph of a sequential region that performs some control flow between compute-intensive loops and has many data dependences between instructions. Generally, this type of sequential code doesn't have a large number of instructions so providing abundant compute resource does not improve performance. Decreasing the dependence length through a chain of instructions is the only solution to accelerate such code. Figure 3(b) is an example of dependence-constrained loop. This loop also has a small number of instructions with long chains of sequential dependences. This type of code is also hard to overlap iterations by software pipelining because last instruction on each iteration has data dependence with the first instruction of the next iteration, and the next loop cannot start execution before finishing the execution of the prior loop.

2.2 Accelerating Sequential Code

Most prior research in CGRA has focused on improving the performance of innermost loops through intelligent parallelization or software pipelining techniques. However, none are effective at enhancing the performance of sequential code regions, which occupy a significant fraction of total execution time as demonstrated in Figure 2. In this work, we take a circuit-level approach to attack the problem of improving the performance of sequential and dependence-constrained loops on CGRAs.

One obvious approach to improve performance of all region types is to increase the clock frequency of the CGRA. However, this approach increases power consumption a large amount due to additional pipeline registers and higher voltage needed to operate the CGRA. Rather, our approach is to exploit the slack cycle-time to accomplish more work in a single clock cycle when the critical timing paths are not exercised through the CGRA. In this manner,

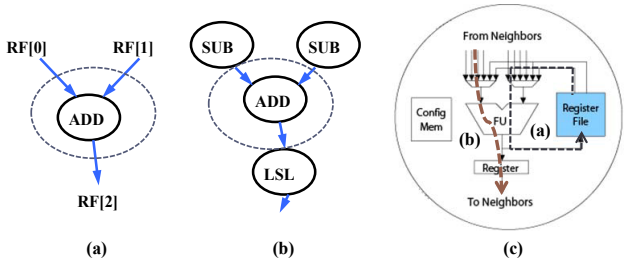

Figure 4: Comparison of flow of data through a processing element in a CGRA: (a) Operation with register file access, (b) Operation without register file access, (c) Flow of data for (a) and (b)

multiple arithmetic operations can be "chained" together when the critical timing paths are not exercised to accomplish more work in a single cycle.

Configurable compute accelerator (CCA) [4] is one related research based on this concept. CCA is also designed to execute a number of sequential instructions on fixed clock period in a general purpose processor. The clock period of a general purpose processor is larger than that of CGRA and the depth of maximum sequentialized instruction is quite large. However, this type of accelerator cannot cover all the subgraphs becasue of fixed numbers of input/output ports and limitations of subgraph depth. Expression-grained reconfigurable architectures [1] are proposed to solve these problems but they still cannot cover all the cases. In addition to coverage problem, low utilization of FUs is another critical drawback on this type of research. They put abundant resources to obtain high subgraph coverage on fixed hardware hence utilization of each individual FU becomes low. Thus, a more efficient strategy is required to enable the acceleration of sequential subgraphs without adding significant cost or power to a baseline CGRA.

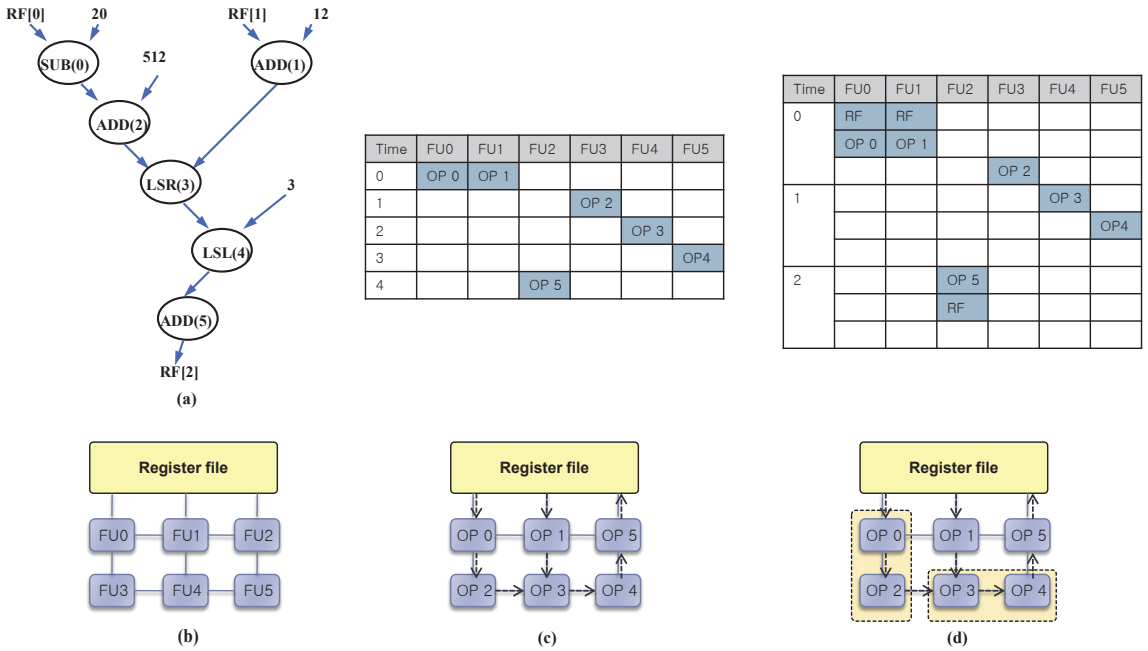

Figure 5: Dynamic operation fusion example: (a) dataflow graph under consideration, (b) target 2x3 CGRA, (c) conventional scheduling that requires 5 cycles, and (d) scheduling with dynamic operation fusion that requires 3 cycles.

3. DYNAMIC OPERATION FUSION

In this section, we propose dynamic operation fusion that can accelerate the execution of sequential code regions by executing multiple operations in a single cycle. The basic idea is explained first and the opportunities for dynamic operation fusion in multimedia applications is shown. Lastly, the hardware support is discussed.

The basic idea of operation fusion stems from the observation that the clock period of a CGRA is determined by the worst case delay (critical path delay) in the architecture. When the clock period is not fully utilized, the slack can be used to execute the successive operation if the delay fits into the slack.

The critical path of a CGRA usually consists of: register file read, longest execution in a FU, and write back to register file, as shown in Figure 4(a). While register file access is required for every operation in conventional architectures, CGRAs have distributed interconnect across the array that can directly transfer operands between FUs. When an operation is executed without a register file access through the interconnect, it does not fully utilize the clock period and there is significant slack left. For example, the ADD operation in Figure 4(b) reads the operands from its neighboring FUs and transfers its result directly to another FU. If the time slack is bigger than the delay of the successive operation LSL, both ADD and LSL can be executed in the same clock period. As previously mentioned, vertical collapsing of dependent operations is similar to the CCA [4]. In CCA, the subgraphs with simple operations (i.e., arithmetic, logical) are identified either at compile time [3] or at run-time [4]. The execution of the subgraphs are offloaded to a specially designed accelerator (Figure 6) that can collapse the execution of multiple operations into a single cycle.

Instead of using dedicated hardware as in CCA, we propose dynamic operation fusion that utilizes existing resources in a CGRA to collapse the dependent operations into a single cycle. Since there are a large number of FUs in a CGRA, a subset of them can be combined dynamically at run-time and execute dependent operations in a single cycle. A simple modification to the hardware can allow dynamic merging of FUs for operation fusion; providing an interconnect between FUs that bypasses the output registers. Figure 7 shows the additional interconnect from the combinational output of an FU to the input of its neighboring FUs. Here, three FUs on the right are serially merged together to execute the three dependent operations on the left (ADD - ADD - LSR) in a single cycle. So, the execution time of the sequential code region can potentially be reduced with dynamic operation fusion, while the hardware overhead is minimal.

Dynamic operation fusion has the following benefits over the CCA approach with a dedicated accelerator:

- Minimal hardware overhead utilizing the existing resources.
- Multiple subgraphs can be executed simultaneously when resources are available.
- Dynamic merging of FUs allow exploiting various shapes of the subgraphs.

We will compare the schedule results using dynamic operation fusion with traditional scheduling for a CGRA with the example shown in Figure 5. The dataflow graph on the left contains a series of dependent operations that read operands from register files and store the result back into them. It is mapped onto a hypothetical 2x3 CGRA in Figure 5(b). The conventional approach will generate a schedule shown in Figure 5(c), where the total execution time is 5 cycles. Because of the serial data dependences, the utilization of the FUs is quite low.

Figure 5(d) shows how the execution of the dataflow graph can be accelerated with dynamic operation fusion. Here, we assume that one register file access and two arithmetic operations can fit into the clock period. More detailed studies on the comparison between the clock period and operation latencies are provided in the following section. With the bypass network, two sets of back-to-back operations are collapsed into the same cycle as shown in the

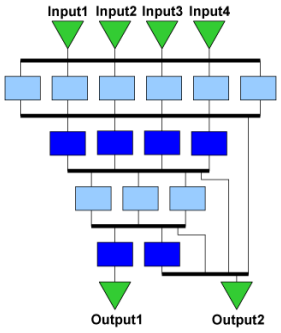

Figure 6: Configurable compute accelerator design.

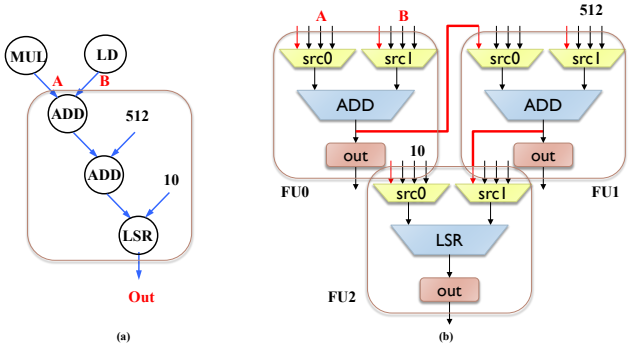

Figure 7: Combining of FUs for dynamic operation fusion: (a) Target subgraph, (b) 3 FUs combined.

schedule. At cycle 0, FU 0 and FU 3 are merged together to execute back-to-back operations 0 and 2 in an single cycle. In the same fashion, operations 3 and 4 are collapsed into cycle 1 on FU 4 and FU 5. Operation 5 cannot be scheduled at cycle 1 since it stores the result into the register file. By applying dynamic operation fusion, the total execution time is reduced by 2 cycles over the conventional approach.

3.1 Delay Statistics and Tick Time Unit

As shown in the previous section, dynamic operation fusion is an effective approach to accelerate the execution of sequential code region. However, the feasibility of dynamic operation fusion depends on the hardware characteristics of the underlying architecture. Dynamic operation fusion can be applied only if there is enough slack in a clock period to execute multiple operations. So, we investigated the delay characteristics of our CGRA design in a real implementation. Figure 8 shows the delay information when the clock period is 3.5 ns. The delays are computed with Synopsis Design Compiler and Physical Compiler using the IBM 90nm standard cell library in typical condition. The delay here includes the delay of input MUXes for each unit. In this table, single cycle operations are categorized based on their execution time. For multi-cycle operations, the delays of the last stage is shown in the table. The execution time of all instructions are smaller than half of a clock period. Logical operations show the minimal delay and four of them can be fused together into a single cycle. On average, two sequential operations can be collapsed. The opportunites for dynamic operation fusion maximizes when there are a large number of operations with a small delay. As in Figure 9, there are a large portion of comparison and logic operations, which suggests that dynamic operation

Group	Opcode	Delay(ns)	Tick (1=0.25ns)
Multi cycle op	MUL, LD, ST	1.65	7
Arith	ADD, SUB	1.74	7
Shift	LSL, LSR, ASR	1.36	6
Comp	EQ, NE, LT	0.93	4
Logic	AND, OR, XOR	0.73	3
RF Read		0.91	4
RF Write		0.70	3

Figure 8: Delay and tick breakdown for common opcodes.

Tick	aac (%)	3d (%)	h.264 (%)
Multi cycle	2419 (31)	17077 (34.5)	11579 (30.7)
Arith	2018 (26)	12339 (25)	11075 (29.3)
Shift	370 (4.7)	1165 (2.3)	2086 (5.5)
Comp	506 (6.5)	2788 (5.6)	1923 (5.1)
Logic	2492 (32)	15919 (32.2)	11024 (29.2)

Figure 9: Breakdown of opcodes for three target applications.

fusion can potentially improve the sequential code performance in multimedia applications.

Since multiple operations can be mapped into a single cycle, we need a smaller time unit than the traditional clock cycle used by compiler schedulers. We propose a new time unit called a *tick*, a small time unit based on the actual hardware delay information. The unit delay of one tick is set by the actual latency of the smallest logic component, normally a small MUX. With the tick unit, the clock period and the delays of other hardware components can be converted into tick numbers. Every logic component on CGRAs has their own tick information and the information is used for dynamic operation fusion scheduling. Tick information based on IBM 90nm library is shown in the last column of Figure 8.

3.2 Bypass Network

Figure 10 shows the real implementation of the bypass network with some practical considerations. Figure 10(a) is the original FU on the baseline architecture. Each FU has three source MUXes for predicate and data inputs. In addition to this, each FU has one additional MUX to increase the routing bandwidth of the array. Four predicate, compute, and routing outputs are generated from the FU and connected to other FUs through registers. Bypass connections between FUs are implemented by adding a small two-input MUX to two data outputs (Figure 10 (b)). The MUX has both an FU output and register output as inputs and one of these signals is chosen by the select signal of the MUX every cycle. This type of MUX is selected to minimize the additional area and delay cost to the baseline architecture. As FU and register outputs are shared, the bandwidth is restricted but the hardware overhead can be reduced by minimizing change of the baseline architecture. An additional 32 control bits and 32 MUXes with 33644 um^2 area are required and the costs are 3.8% and 2.3% overhead (Figure 11).

4. COMPILER SUPPORT

In this section, we describe the compiler support for dynamic operation fusion using the bypass network in CGRA Express. Taking the concept of edge-centric modulo scheduling (EMS) [18], we developed a scheduler that can support both sequential and loop code regions for CGRAs. We enhanced the original algorithm with the ability to place multiple operations in a single cycle without incurring the structural hazard of the resources. The concept of tick slot in Section 3.1 is introduced into the scheduler and scheduling

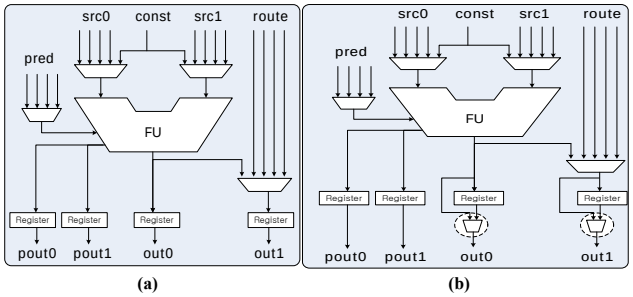

Figure 10: Comparison of bypass network implementation details: (a) baseline network and (b) network that supports dynamic operation fusion.

	baseline	modified	overhead(%)
control bit	845	877	3.8
area (mm^2)	1.447	1.48	2.3

Figure 11: Hardware overhead of the bypass network. Two forms of overhead are specified: control bits to control the bypass MUXes and area of the bypass network.

is performed on a tick basis rather than a conventional cycle-based manner.

First, we will briefly introduce the EMS framework and then describe the basic concepts of tick-based scheduling. Finally, we will provide the added features to attack the problems specific to tick-based scheduling.

4.1 Edge-centric Modulo Scheduling

The most distinctive feature of the EMS is that it takes routing of values as the first-class objective. The routing of operands is often ignored in traditional schedulers since it can be guaranteed by the centralized resources (i.e., central register file) of a traditional VLIW processor. Any value generated by a producer can be routed to its consumers by putting the operand into the central register file. However, the distributed interconnect and register files in CGRAs require the compiler to orchestrate the communications between producers and consumers explicitly. The modulo constraint that must be observed to create a correct modulo schedule allows only a limited available slots for each resource, making the routing of operands on the array even harder.

For this reason, EMS constructs the schedule by routing the edges in a dataflow graph, rather than placing the nodes. This approach allows both performance gain and compilation time reduction over the traditional node-centric approach. The following are the major features of the EMS that differentiate it from conventional schedulers.

- **No explicit backtracking.** With the distributed interconnect and abundant computation resources, the scheduling space for CGRAs can get quite large and the compilation time can be a critical issue. To reduce the compilation time, EMS does not have a backtracking mechanism. Especially for CGRAs, it is hard to make forward progress with backtracking since placing and unplacing of operations usually involves multiple resources for routing. Therefore, routing decisions are made just once.

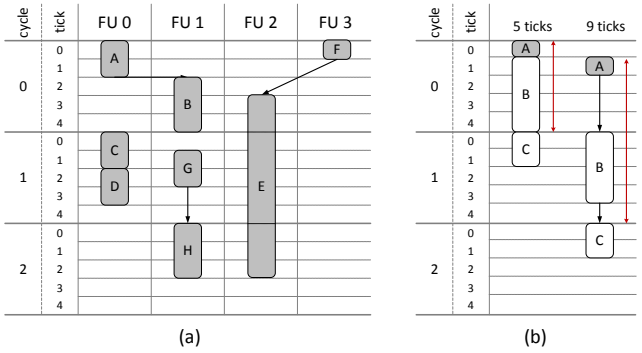

Figure 12: Tick-based scheduling example: (a) possible placements in the tick scheduling space and (b) different longest path delays per tick slots.

- **Proactive prevention of routing failures.** To compensate for the lack of backtracking, EMS proactively avoids routing failures using probabilistic cost metrics. Before routing an edge, the probabilities of the future usages of scheduling slots are calculated. By avoiding the slots with high probabilities, routing failures can be effectively prevented.

- **Recursive routing calls for critical components.** Some components in a dataflow graph require more cautious scheduling since they can easily make the scheduling fail. One good example is a recurrence cycle. To meet the timing constraints of the recurrence cycles, traditional schedulers usually treat them with highest priority. Additionally, EMS schedules the edges in a critical component altogether by routing them recursively. When an edge in a recurrence cycle is routed, it only finalizes the routing only if all other edges in the component are successfully routed in recursive calls. This recursive routing provides an implicit form of backtracking for scheduling critical components.

4.2 Tick-based Scheduling

To enable the scheduler to place back-to-back operations in the same cycle, it needs to keep track of where the operations are placed at the precision of ticks. Figure 12(a) shows the scheduling space for tick-based scheduling where each cycle is divided into multiple ticks. For illustration purposes, register file access time is ignored. The number of ticks in a cycle is determined by the frequency of the target architecture and is given as input to the scheduler. Here, operations are placed into tick slots, and the resource management is still done on a cycle basis; only one operation is allowed to be placed in a cycle for each resource.

To manage the cycle and tick times together, we defined *STime* which is a pair, *(cycle, tick)*. *STime* is used for two purposes: schedule time unit , and delay of resources and operations. For example, the input time of operation A in Figure 12(a) is scheduled at (0, 0) and its delay is (0, 2). For multi-cycle delays of pipelined operations, *STime* has an additional field of *init_tick* making it a tuple of *(cycle, tick, init_tick)*. *init_tick* indicates the number of ticks required to process the operation at the first pipeline stage. The load operation E shown in Figure 12(a) has a delay of (2, 3, 2). While the load operation will have a delay of 3 cycles in a traditional approach, it requires 2 ticks and 3 ticks for the first and last stages, respectively. Therefore, the pipelined operations can also participate in dynamic operation fusion.

Figure 12(a) shows some possible placements of operations in tick-based scheduling. Operations A and B are scheduled in the same cycle using the bypass network. However, since the resources are managed in cycles, only one operation can be mapped on a resource in a single cycle. So, it is illegal to place back-to-back operations C and D in the same resource/cycle. Also, an operation cannot be mapped across the clock boundary unless it has a multi-cycle delay. When there is not enough tick slots in a given cycle, the scheduler delays the operation to the next cycle as shown with operations G and H.

Operator Overloading We replaced all the time/delay units in the EMS with our *STime* unit, while keeping the basic structure of the scheduler. So, the changes applied to the original scheduler are minimized. The basic arithmetic operators such as +, -, *, / were overloaded in a way that the *cycle* field increases/decreases as the *tick* field crosses the cycle boundary. Often times, a delay is added or subtracted to a schedule time to create another schedule time. For example, the output time of operation B in Figure 12(a) can be calculated by adding the delay (0, 3) to the output time of operation A (0, 1).

However, there are two things to consider when a delay is applied to a schedule time. First, the clock boundary constraint should be checked so that the operation is not placed across the boundary. Also, when adding a multi-cycle delay to a schedule time, the resulting time should be adjusted along the clock boundary since multi-cycle operations should be aligned with the clock boundaries. Basically, the time gap between the output time of the producer and the consumer needs to be added to get the output time of the consumer. The equation below shows how the addition is performed between a schedule time and a delay. *num_ticks* denotes the number of tick slots in a single cycle. *T* is the schedule time and *D* is the delay. When adding a delay to a schedule time, the timing constraint is checked by looking at *init_tick* of the delay (Equation 1). When it passes the timing constraint, the delay is added using the overloaded operator '+'. For multi-cycle delays, the time is converted to its floor to align the resulting time along the clock boundary (Equation 2). After performing the addition, Equation 3 checks if the performed addition violates the clock boundary constraint.

$$if(D.cycle > 0)\ num\_ticks - T.tick >= D.init\_tick \quad (1)$$

$$add(T, D) = (D.cycle > 0)?(T.cycle, 0) + D : T + D \quad (2)$$

$$check(T, D) = (add(T, D).cycle - T.cycle == D.cycle) \quad (3)$$

4.3 Tick Specific Features.

By introducing the new *STime* unit, we could minimize the modifications applied to the original EMS. However, there are some features that need to be adapted to efficiently perform tick-based scheduling. Three major features are explained in this section.

ASAP/ALAP time calculations. In schedulers, ASAP and ALAP times are used to estimate how early/late an operation can be placed without destroying timing dependences between operations. The ASAP time of an operation C can be calculated by Equation 4. *p* denotes an placed predecessor of *C* and *d(x, y)* is the longest path delay between *x* and *y*.

$$ASAP(C) = MAX(time(p) + d(p, C)) \quad (4)$$

Basically, the scheduler looks at all the already-placed predecessors in the dataflow graph and adds the longest delay between the predecessor and the current operation, and picks the maximum

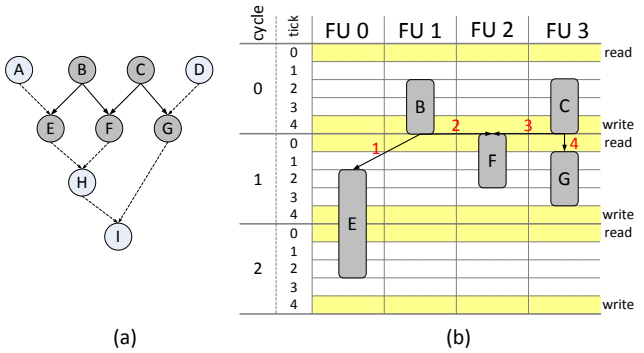

Figure 13: Register access regions in a tick schedule: (a) dataflow graph, (b) register read/write regions (shaded) within each cycle.

time. In cycle-based scheduling, the longest delay stays constant no matter which cycle the predecessor is placed. However, in tick-based scheduling, the longest delay changes depending on which tick slot the predecessor is placed. Figure 12(b) shows an example of the different delays between operation A and C. Here, we assume that A is already placed and B and C are not. Since the operations cannot be scheduled across the clock boundaries, the delays are different between the two cases. Therefore, the tick-based scheduler calculates the longest delay of two operations for each producer's tick slot in a cycle.

Identifying Subgraphs. To find the opportunities for dynamic operation fusion, the scheduler takes a greedy approach for finding the target subgraphs. When an operation is placed, the scheduler looks at its neighboring operations in the dataflow graph and checks the timing constraints to see if they can fit into the same cycle using the bypass network. If there is an opportunity for fusion, the scheduler recurses on the routing of an edge between the two back-to-back operations. The use of the bypass network is encouraged in routing by giving a penalty when the cycle is increased during the routing. The router will visit the available slots in the same cycle first using the bypass network. However, this can result in wasting FU slots just for routing since the bypass network connects neighboring FUs. For this reason, we only allow the use of the bypass network when back-to-back operations can be placed in neighboring FUs.

Register Access Region. Even though the register access time was ignored in Figure 12, the register read and write times need to be considered in reality. The shaded regions in the scheduling space in Figure 13 display the register access region. Here, we assume the register read and write time is 1 tick. For each cycle, the first tick slot is called the register read region and the last tick slot is called the register write region. When operations are placed in these regions, they cannot access register files due to timing constraints. For example, operation B's output is placed at (0, 4) slot and it can only route its value to neighboring FUs through the FU's output register. Therefore, routing flexibility is greatly limited for operation B. When all the neighboring FUs are occupied, the scheduling will fail since there is no backtracking mechanism. To avoid this situation, our scheduler performs recursive calls for routing edges when an operation is placed in the register access region. Figure 13(a) shows an example dataflow graph. When operation B is placed at cycle 0 as shown in the figure, its output is placed in the register write region. Therefore, the scheduler makes sure that all the edges coming out from operation B are success-

	seqential			loop(resource)			loop(dependency)			total		
	baseline	express	perf. ratio	baseline	express	perf. ratio	baseline	express	perf. ratio	baseline	express	perf. ratio
aac	42.64	36.47	85.53%	17.40	15.75	90.51%	0.32	0.24	75.34%	60.36	52.46	86.91%
h.264	44.77	39.29	87.75%	23.80	24.70	103.78%	0.58	0.29	50.01%	69.15	64.28	92.95%
3d	77.82	60.05	77.16%	74.70	65.94	88.28%	4.29	4.22	98.32%	156.81	130.22	83.04%

Figure 14: Performance evaluation of the baseline and CGRA Express architectures for three multimedia applications. Performance is broken down into non-innermost loop regions (sequential), inner-most loops whose performance is constrained by the availability of resources (loop (resource)) and inner-most loops whose performance is constrained by cross-iteration dependences (loop (dependency)).

fully routed before finalizing the placement. Therefore, it recurses on the routing of two edges (E and F). When operation F is placed in cycle 1, the scheduler also recurses on the edge to operation C since F is placed in the register read region. The numbers shown in the figure denote the order of routing call of each edge. Since the operations E, F, and G are not placed in the register write region, they can store values into the register files. So, the scheduler does not proceed with routing the outgoing edges from them. When all the edges with solid lines in Figure 13(a) are successfully routed, the scheduler finalizes the placement of operation B.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

Target Architecture Two CGRA architectures are used to evaluate the performance of dynamic operation fusion. The baseline architecture is the 4×4 heterogeneous CGRA shown in Figure 1. Four FUs are able to perform load/store instructions to access the data memory and 6 FUs support 2-cycle pipelined multiply. A 64-entry central register file with 6 read and 3 write ports and sixteen 8-entry local register files exist in the array. Only four FUs on the first row have direct access to the central register file and other FUs must use data buses to access the central register file. Local register files with one read and one write port are placed similar to the FUs and each register file can be written by FUs in diagonal directions. There is also one 64-entry predicate register file with four read and four write ports. The CGRA Express architecture has the same architectural shape except the addition of the bypass network.

Target Applications All the sequential and loop code are taken from three application domains: audio decoding (aac), video decoding (h.264) and 3D graphics (3d). The sequential code regions are mapped using VLIW mode of the array and loop code regions are mapped using CGRA mode of the array. Performance is evaluated by the overall execution time.

Power/Area Measurements Both the baseline and CGRA Express architectures are generated in RTL Verilog and synthesized with the Synopsys design compiler and Physical compiler using IBM 90nm standard cell library in typical operation conditions. Synopsys PrimeTime PX is used to measure power consumption. The SRAM memory power was calculated using SRAM model generated by the Artisan Memory Compiler. The target frequency of both baseline and the CGRA Express architectures are 200MHz.

5.2 Performance Measurement

In order to illustrate the effectiveness of dynamic operation fusion, performance of the three benchmarks is compared on the baseline CGRA and CGRA Express. In sequential code regions, run-time is measured by the schedule length multiplied by the frequency of execution. The run-time of the loop code regions is calculated by multiplying the Initiation Interval (II) achieved by EMS and the loop trip count. II means the interval between successive

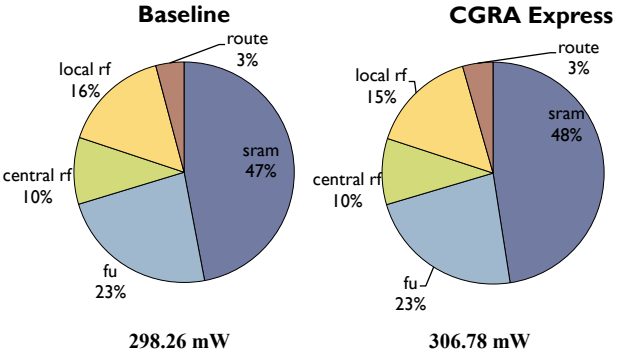

Figure 15: Power breakdown comparison for the baseline and CGRA Express architectures.

	baseline	express	ratio
power (mW)	298.26	306.78	102.86%
# of cycles (million)	156.81	130.22	83.04%
energy (mJ)	233.85	199.74	85.42%

Figure 16: Energy comparison for the baseline and CGRA Express architectures.

iterations thus II is the indicator of throughput in modulo scheduling. The results of this experiment are shown in Figure 14. The numbers in the table show the execution time in millions of cycles and perf.ratio is the ratio of execution time on CGRA express over the baseline.

Overall, dynamic operation fusion achieves 7-17% reduction in execution time over the baseline. This is a promising result because the hardware overhead is about 3% as discussed in Section 3. More specifically, most of the performance improvements are due to the schedule length reduction in sequential code regions, which was expected since dynamic operation fusion collapses the series of operations into a single cycle.

However, we could also observe a good amount of reduction in resource-constrained loops. This is primarily due to the additional bypass network. The additional connection doubles the number of reachable slots from an FU. With the bypass network, an FU can access its neighboring FUs results in the same cycle as well as in the next cycle. This gives the scheduler more flexibility and improves the throughput of the resource constrained loops. Also, when a loop has small trip count, schedule length will be more dominant than the II for run time, hence dynamic operation fusion can improve performance. The dependence-constrained loops show up to 50% reduction in execution time. This was expected since the throughput of these loops was mainly limited by the critical path

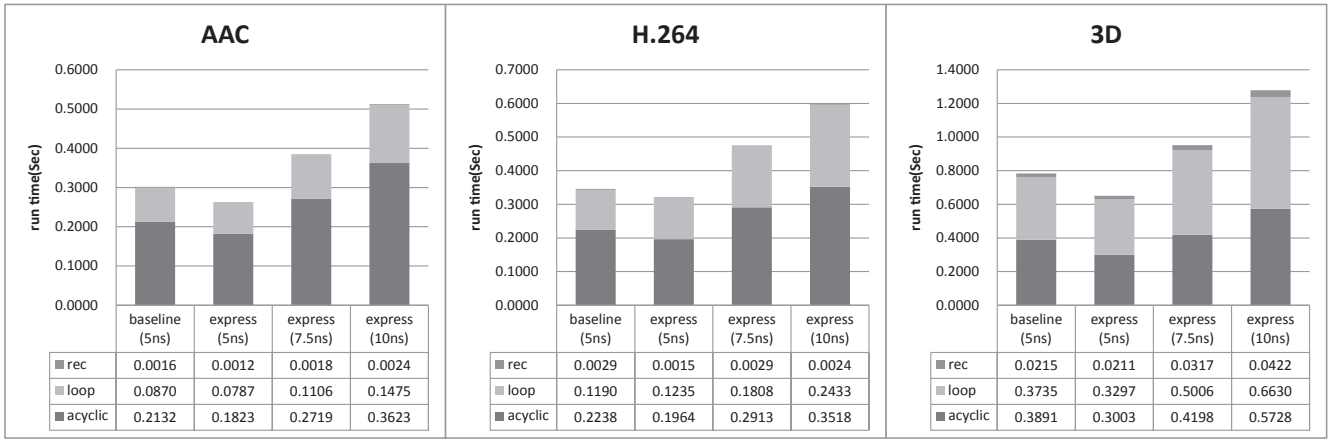

Figure 17: Performance comparison of the baseline and CGRA express architectures for different clock periods. Performance is broken down into dependence-constrained loops (rec), resource-constrained loops (loop) and non-innermost loops (acyclic) regions.

of a single iteration, which can be efficiently reduced by dynamic operation fusion.

5.3 Power and Energy Measurement

The instantaneous power consumption of CGRA Express architecture is seemingly higher than that of baseline architecture due to additional hardware overhead. However, the bypass network implementation can also decrease the total run time. Since there is such trade off between power and run time, we measured total energy consumption for running complete applications to determine the effectiveness of dynamic operation fusion.

Overall power consumption and the breakdowns of both architectures for 3D are shown in Figure 15. Overall, average power consumption on the CGRA Express architecture is 3.1% higher than the baseline architecture. Compared to the baseline architecture, the power increase observed for the datapath is smaller than the increase in the SRAM for control signals. The bypass network adds just a small amount of combinational logic (MUXes) on the baseline architecture, hence the overall effect is quite small. On other hand, adding control signals is more critical for power consumption on CGRAs because all the control signals must be read every cycle. Therefore, overall power overhead for adding bypass network is trivial but careful consideration is necessary due to the additional number of control signals.

An interesting result can be found on total energy consumption comparison between both architectures. Figure 16 shows that the CGRA Express architecture is 15% more energy efficient than the baseline architecture. Even though average power consumption of the new architecture is slightly higher, the decrease in application run time dominates the results.

5.4 Operating Frequency Optimization

As discussed in prior experiments, dynamic operation fusion can decrease total run time by decreasing number of cycles in fixed clock period. However, measuring total run time on various clock periods will be another interesting question with dynamic operation fusion. With different clock periods, total run time is calculated by multiplying the number of cycles and the clock period. If clock period is large, more operations can be chained into a single cycle. But, these gains must offset the losses in performance due to a reduced clock rate. We can expect some optimal smallest run time exists as the clock period is swept that represents the sweetspot of a fast clock rate while permitting some degree of chaining. Figure 17 shows the total run time of the three applications with various clock periods in nanoseconds.

Dynamic operation fusion works efficiently at 5ns compared to traditional scheduling but expanding the clock period to more than 5ns achieves no additional performance improvement. As the clock period becomes longer, sequential code regions require fewer cycles to execute and their characteristics start to resemble loop code regions. This behavior occurs because just 4 FUs are used for executing sequential code regions. With the most aggressive fusion, the dependences of 4 successive instructions are collapsed which basically eliminates all dependences that can constrain performance and converts the code region into a resource constrained one. Moreover, the number in sequentially dependent instructions before a memory instruction is encountered is typically smaller than 4, thus there are limited opportunities for fusion. As a result, using a clock period of 7.5ns results in 50% increase of total run time because there is no additional reduction of the number of clock cycles due to dynamic operation fusion (beyond those saw at 5ns), but the clock period is 50% larger.

6. RELATED WORK

6.1 Architecture

Many CGRA-based systems have been proposed in various papers and some of the models have been implemented. Each design has different scalability, performance, and compilability. ADRES [14] is the most well-known CGRA system with an 8x8 mesh of processing elements with central and local register file. As we mentioned prior sections, ADRES also supports CGRA array mode as well as VLIW mode using central register file and FUs on the top row. MorphoSys [13] is another famous example of 8x8 grid with a more sophisticated interconnect network. In MorphoSys, each node has an ALU and a small local register file. RAW architecture is more general system which node is small MIPS processor with memory, registers, and a processor pipeline. PipeRench [7] and RaPid [6] are also 1-D architectures have similar concept to CGRAs. In PipeRench, each processing elements are arranged in stripes to support pipelining. RaPid has a lot of heterogeneous elements (ALUs and registers), which can be connected by reconfigurable interconnection.

The results of recent research about general architecture exploration on CGRAs are also promising. Kim [10] focussed on the power consumption for configuration memory and proposed spatial and temporal mapping with pipelining. Moreover, Kim [9] proposed different approach based on data flow graph of applications.

Research on instruction set customization with configurable compute accelerator (CCA) is also closly related to this research. Clark [3] studied how to create efficient CCA based on sub graph modulation and improved the idea to virtualized execution accelerator [5]. Hormati [8] also studied CCA to be more faster and smaller. Lastly, Bonzini[1] adopt the CGRA idea to CCA and diminish disadvantages of CCA, such as logic depth limitation and low coverage.

6.2 Compilation Techniques

As dealing with sparse connectivity and distributed register file is huge challenge on compiler, many techniques have been proposed for compiling CGRAs. Lee [11] proposed a schedule approach for a generic CGRA, which generates pipeline schedules for innermost loop. Park [17] also worked on innermost loop, but they focussed on loop level parallelism while Lee worked on instruction level parallelism. Park's work is more similar to Mei at al [15]'s work on modulo scheduling.

Research on CGRA scheduling is partially similar to the research on VLIW machine scheduling. As clustered VLIW machines are also spatial architecture, many compilation techniques on VLIW can be adopted to CGRAs. However, VLIW machine does not have routing issues related to sparse interconnection network hence some modification is necessary to support CGRA.

On this paper, we introduce some cost function about actual delay of synthesized hardware (MUX, Adder, Shifter). This concept is similar to the research about module mapping and placement on FPGA area. Callahan [2] performed datapath module placement simultaneously with the mapping using area and delay cost. They used the area and delay cost to minize both area and delay on FPGA. We also adopt the delay cost to increase utilization of FUs on pre-defined clock period.

7. CONCLUSION

This paper proposes dynamic operation fusion, an effective approach to accelerate sequential code regions on CGRAs. As scheduling techniques for loops have been developed, the run-time for loops decreases by large factors as the compiler is able to make effective use of the abundance of resources available in a CGRA. However, the side effect is that sequential code region become more and more of the overall performance bottleneck as these regions have limited instruction-level parallelism. We introduce two key concepts to execute sequential code region faster. First, a bypass network is implemented to support dynamic operation fusing wherein existing function units on a CGRA are configured to execute back-to-back operations in a single cycle using any available slack in the cycle time. A simple hardware extension in the form of an additional connection between neighboring function units and a bypass MUX are required. Second, the compiler scheduler automatically identifies opportunities for dynamic fusion based on sub-units of clock cycles, called ticks. Overall, dynamic operation fusion reduces total application run-time by 7-17% and total energy by 15% on a 4x4 CGRA.

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] P. Bonzini, G. Ansaloni, and L. Pozzi. Compiling custom instructions onto expression-grained reconfigurable architectures. In *Proc. of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 51–59, Oct. 2008.

[2] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, Apr. 2000.

[3] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[4] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005.

[5] 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.

[6] 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.

[7] 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.

[8] A. Hormati et al. Exploiting narrow accelerators with data-centric subgraph mapping. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 341–353, Mar. 2007.

[9] Y. Kim and R. N. Mahapatra. A new array fabric for coarse-grained reconfigurable architecture. In *Proc. of the 34th Euromicro Conference*, pages 584–591, Sept. 2008.

[10] Y. Kim, I. Park, K. Choi, and Y. Paek. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proc. of the 2006 International Symposium on Low Power Electronics and Design*, Oct. 2006.

[11] J. Lee, K. Choi, and N. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Journal of Design & Test of Computers*, 20(1):26–33, Jan. 2003.

[12] 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.

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

[14] B. Mei et al. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Proc. of the 2003 International Conference on Field Programmable Logic and Applications*, pages 61–70, Aug. 2003.

[15] 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.

[16] B. Mei, F. Veredas, and B. Masschelein. Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. In *Proc. of the 2005 International Conference on Field Programmable Logic and Applications*, pages 622–625, Aug. 2005.

[17] H. Park, K. Fan, M. Kudlur, and S. Mahlke. Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 136–146, Oct. 2006.

[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] 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.

[20] 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.

[21] 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.