A Dataflow-centric Approach to Design Low Power Control Paths in CGRAs

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

ABSTRACT

Coarse-grained reconfigurable architectures (CGRAs) present an appealing hardware platform by providing high computation throughput, scalability, low cost, and energy efficiency, but suffer from relatively high control path power consumption. We take the concept of a token network from dataflow machines and apply it to the control path of CGRAs to increase efficiency. As a result, instruction memory power is reduced by 74% , the overall control path power by 56%, and the total system power by 25%.

1. INTRODUCTION

Today's mobile applications are multimedia rich, involving significant amounts of audio and video coding, 3D graphics, signal processing, and communications. These multimedia applications usually have a large number of kernels in which most of the execution time is spent. Traditionally, these compute-intensive kernels were accelerated by application specific hardware in the form of ASICs to meet the competing demands of high performance and energy efficiency. However, increasing convergence of different functionalities combined with high non-recurring costs involved in designing ASICs have pushed designers towards programmable solutions.

Coarse-grained reconfigurable architectures (CGRAs) are becoming attractive alternatives because they offer large raw computation capabilities with low cost/energy implementations. Example CGRA systems that target wireless signal processing and multimedia are ADRES [9], MorphoSys [7], and Silicon Hive [12]. Tiled architectures, such as Raw, are closely related to CGRAs [16]. 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 FUs. The FUs can execute common word-level operations, including addition, subtraction, and multiplication.

A major bottleneck for deploying CGRAs into a wider domain of embedded devices lies in the control path. The appealing features in the datapath of CGRAs ironically come back as a major overhead in the control path. The distributed interconnect and register files require a large number of configuration bits to route values across the network. The abundance of computation resources simply adds up the list for configurations to the control path. As a result, the total number of control bits to configure the whole array can reach nearly 1000 bits each cycle, and the control path takes up to 43% of the total power consumption in existing CGRA designs [3, 2]. Moreover, control bits are read from the on-chip memory every cycle regardless of the array's utilization. Even when only a small portion of the resources are active in the array, the configurations for all the resources must be fetched, which makes CGRAs very inefficient for the codes with limited parallelism. This inefficiency prevents

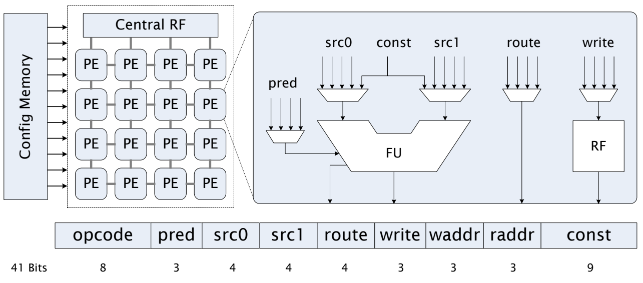

Figure 1: CGRA overview: 4x4 array of PEs (left), a detailed view of a PE (right), and a PE instruction (bottom)

CGRAs from wider uses including outer loop level pipelining [14] or simply running acyclic code to reduce the communication overhead with the host processors. Finding an efficient way to reduce the control power reduction will not only relieve the power overhead in the control path, but also opens the future application of CGRAs to more variety of workloads.

While there are many studies on architecture exploration, code mapping, and physical implementation [8, 1, 4], relatively little work has examined efficient control in CGRAs and other tiled accelerators. One exception is [3] wherein a hybrid configuration cache is proposed that utilizes the temporal mapping for control power reduction. Temporal mapping only utilizes a single column of PEs in the array to map the entire loop and the The execution of the loop is pipelined by running multiple iterations on different columns in the array. The control power can be substantially reduced by transferring the configurations in one column to its right each cycle, letting only the leftmost column read from the configuration memory. However, temporal mapping can be applied to only certain types of loops and it is not a general approach that can scale to different types of applications. [2] reduced the control path power of CGRAs as a by-product of an architecture exploration. A Pareto optimal design of a CGRA was discovered that required a lesser number of resources in the datapath thereby resulting in a power reduction in the control path. To our knowledge, no previous work has addressed a general solution for power-efficient control path design in tiled accelerators like CGRAs. In this paper, we propose a new control path design that improves the code efficiency of CGRAs by leveraging token networks originally proposed for dataflow machines.

2. MOTIVATION

Figure 1 shows our target CGRA, similar to [8]. There are 16 PEs connected in a mesh-style interconnect and a central register file for transferring values from/to the host processor. Each PE has one FU for computations and an 8-entry local register file that are shared by other neighboring PEs. An FU has three source multiplexors (MUXes) for predicate and data inputs. Here, we assume an additional MUX(route) in each PE to increase the routing bandwidth of the array. So, the PE can do both computation and routing in one cycle. There are several MUXes as a result of the distributed interconnect and each of them requires selection bits encoded in the instruction field. Also, each register read/write port requires an RF address field. Along with PE instructions, there are instructions for central register files, and other buses that also require config-

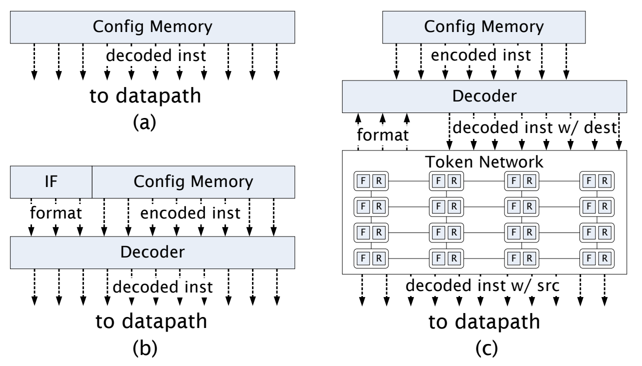

Figure 2: Different Control Path Designs: (a) No compression, (b) Fine-grain code compression with static instruction format, (c) Fine-grain code compression with a token network (F and R indicate FU token module and RF token module, respectively)

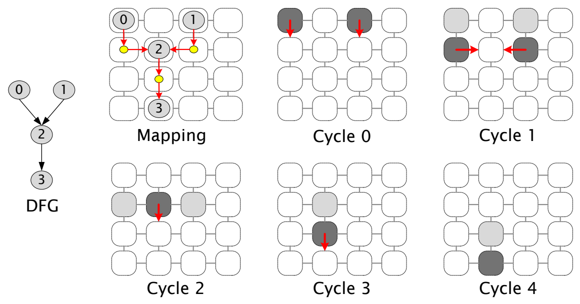

Figure 4: Dynamic configuration of PEs using tokens

uration. As a result, each PE instruction is 41 bits, and a total of 845 bits is required to configure the CGRA each cycle. Typically, control signals in CGRAs are stored as a raw data (fully decoded instructions) and directly fed to the datapath as shown in Figure 2(a). Fetching 845 bits every cycle is indeed a large overhead. Control path power can obviously be reduced by increasing code efficiency through some form of code compression technique.

Conventionally, code compression is performed at the instruction level with no-op compression or a variable length encoding. No-op compression is widely used in VLIW processors and many DSPs [17, 15, 10, 5, 6]. However, instruction-level compression does not work well in CGRAs due to the highly distributed nature of the resources. Even if an FU is sitting idle, the register file in the same PE can still be accessed by neighboring PEs. Also, the FU can be used for bypassing data from one PE to another. We examined the schedules of several hundred compute-intensive kernels taken from multimedia applications mapped onto our CGRA design and discovered that only 17% of PE instructions are pure no-ops (all the components in the same PE is not active), while the average utilization of FUs is 55%. Thus, no-op compression would have limited effectiveness.

However, there is a good opportunity for a fine-grain code compression: compressing instruction fields (e.g., opcode, MUX selection, register address) rather than the whole instruction. On average, only 35% of all instruction fields contain valid data, thus efficiency can potentially be increased by removing unused fields. Figure 2(b) shows a high-level organization that utilizes a static fine-grain compression approach. In the simplest variant, presence bits are added for each field to indicate whether the field exists or not. Instruction encoding consists of the presence bits(instruction format) followed by the subset of valid instruction fields concatenated together. With this approach, decoding can become complex due to the variable length nature of the encoding, but all unused fields can be removed in principle.

The biggest challenge for applying static fine-grain compression lies in the instruction formats. Using a simple fine-grain static compression scheme that we designed for a CGRA, the code efficiency increases by 24% with the average number of instruction bits decreasing from 845 to 647. However, 172 of the 647 bits are used for encoding the instruction formats. Since the instruction format of 172 bits needs be read from the configuration memory every cycle regardless of the number of fields present, the instruction format itself becomes a significant overhead in the control path. To address this limitation, we propose to dynamically discover the instruction formats by applying a dataflow token network explained in Section 3.

Another issue in employing the fine-grain code compression is decoder complexity. Since compression is performed in a finer granularity, the overhead of the decoder is more substantial than instruction-level code compression. In Section 4, we analyze the decoder features that affect the overall complexity and discuss an efficient partitioning of configuration memory to reduce decoder complexity.

3. DYNAMIC DISCOVERY OF INSTRUCTION FORMATS

In this section, we propose a dynamic discovery of instruction formats by adopting the concept of a token network from dataflow machines. The concept is explained first, and then we propose a token network that can assist the fine-grain code compression to reduce the overall power consumption in the control path of CGRAs. Lastly, we discuss how the token network is extended to support modulo scheduled loops [13] to exploit loop-level parallelism in kernel loops.

3.1 Concepts

The basic idea of dynamic instruction format discovery is that resources need configurations only when there is useful data that flows through them. By looking at the locations of data coming into a PE, we can infer the instruction format of the current instruction. For example, two data coming into src0 and src1 MUXes of the FU in Figure 1 indicate that this FU will perform an ALU operation. So, an opcode field and src0/src1 MUX selection fields are required in that cycle. If there is no data coming into the predicate input MUX, the ALU operation is not predicated and the selection bits for pred MUX is not needed. When there is only one data coming into either the src0 or src1 MUX, the FU is performing a move operation and the opcode field is not required. In the same way, a data coming out from the register file in Figure 1 indicates a read address field is required.

We can utilize a token network in dataflow machines [11] to provide information on where data flows in the distributed network. A token is sent from a producer to its consumers one cycle ahead of the actual data execution. Originally, the consumer fired when it accumulated sufficient tokens. However, this concept can be altered as all tokens for a single instruction are guaranteed to arrive at the same time. Hence, the set of tokens uniquely determine the instruction format so that the necessary fields can be fetched from the instruction memory. When the actual data arrives in the subsequent cycle, the required instruction fields are already decoded and the PE is ready to execute the scheduled operation.

Figure 4 shows the big picture of how PEs are configured dynamically in the token network. A simple dataflow graph (DFG) is shown on the far-left and its mapping onto the CGRA datapath is shown next to it. PEs with a small dot indicate they are used for routing. The PEs in the array are incrementally configured each cycle using tokens as in the figure. In each cycle, dark grey PEs are configured and send out tokens to their consumers. In the subsequent cycle, PEs executing the given instructions are shown in light grey. At cycle 0, PE[0,0] (row 0, column 0) and PE[0,2] are configured first to execute operations 0 and 1, respectively, and they send out the tokens to their consumers. At the next cycle, PE[1,0] and PE[1,2] receive the tokens from their producers and are configured to route the data to PE[1,1]. In a similar fashion, PEs are configured as tokens flow over the array and all the necessary PEs to execute the DFG are configured at cycle 4.

3.2 Token Network

To utilize tokens for instruction format discovery, a token network is inserted between the decoder and the datapath as shown in Figure 2(c). The token network consists of two components: token interconnect and token modules. Each datapath element, such as an FU, RF and MUX, has a corresponding token module in the token network. Example token modules are presented in Figure 3. Token modules are connected by a 1-bit token interconnect that has

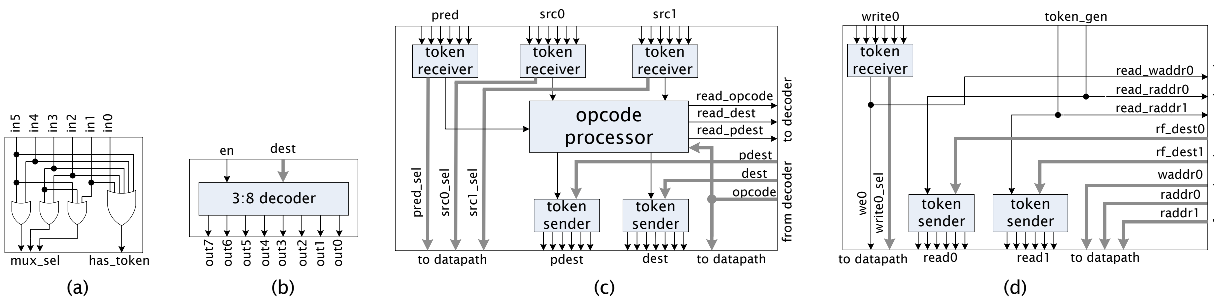

Figure 3: Token Modules: (a) token receiver, (b) token sender, (c) FU token module, (d) RF token module

the same topology as the datapath interconnect. The token network takes the decoded instructions from the decoder and sends tokens across the token interconnect. The token network has two responsibilities. First, the token network provides the instruction formats to the decoder. Second, it generates control signals for the datapath.

3.2.1 Token Generation and Routing

Tokens are first generated at the start of data streams in the dataflow graph: live-in values. A token generated at the top of the dataflow graph flows across the array visiting different resources and finally terminates when it either reaches a register file or merges into another token in an FU. A token terminated in a register file can be re-generated later, creating another token stream.

For tokens generated from live-in, the generation information (time and resource) needs be encoded in the configuration memory since there is no producer that sends token to those nodes. The tokens coming out from register files also require their generation information stored in the configuration memory since the tokens can be re-generated anytime once they are stored in the register file. Therefore, the configuration memory will hold the token generation information for all the tokens coming out from register file read ports. Each cycle, the token generation information stored in the configuration memory fires tokens into the token network and the configurations for the datapath are generated as tokens flow across the array. (token_gen signal in Figure 3(d)).

After tokens are generated, they are routed following the edges in the dataflow graph. To send tokens from producers to consumers, the destination information is stored in the configuration memory instead of the source information. The MUX selection bits in a PE instruction (Figure 1) are replaced by dest fields. As in dataflow machines, only two destinations are allowed for each data generating component (FU output ports, RF read ports). An analysis on the scheduling result of our benchmark loops shows that 86% of the communication patterns are unicast (requiring only one destination), and 98% of communications can be covered by two destinations. Therefore, the performance degradation with the limited number of destinations is minimal. The impact of this limitation is discussed in Section 5. For illustration purposes, only one dest field is shown in Figure 3(c) and (d).

3.2.2 Token Processing

Tokens flowing on the token network are utilized for two tasks. First, the instruction formats are discovered with tokens and they are sent back to the decoder. With these instruction formats, the decoder can decode the compressed instructions for the subsequent cycle. Also, the dest fields in the decoded instructions are converted into the source fields for MUX selection bits and sent to the datapath.

Token Receiver: Since only destination fields are encoded in the configuration memory, the source fields (MUX selection bits) for the datapath need be discovered when tokens are coming into the input ports of each resource. For each MUX in the datapath, a token receiver (Figure 3(a)) is created. A token receiver generates the MUX selection bits(MUX_sel) by looking at the position of an incoming token. Since only one input of a token receiver can have incoming token, the MUX selection bits can be generated with several OR gates as in the figure. Along with the MUX selection bits, it also notifies the attached module (FU/RF token module) whether there is a token coming into this input port or not (has_token).

Token Sender: For each output port of a datapath element (FU output ports, RF read ports), a token sender (Figure 3(b)) is created in the token network to send out tokens to the consuming resources. It simply decodes the dest field (dest) and sends out tokens to the connected modules.

FU token module: Figure 3(c) shows an example of FU token module that has both predicate and data parts. The input MUXes of the FU have been translated into token receivers and the FU itself is replaced with an opcode processor. For the output ports of the FU, token senders are created in the figure. The opcode processor first takes 'has_token' signals from the attached token receivers and discovers the instruction format. The opcode processor sends out a 'read_opcode' signal when both src0 and src1 have incoming tokens. Also, it sends out read signals for dest fields of both data (dest) and predicate (pdest) if there is any incoming token in the input ports. The opcode processor also determines the latency of computation by looking at the opcode field. The dest fields from the decoder are fed into the token senders directly. When the opcode processor signals the token senders with an enable signal, they send out tokens to the designated consumers specified in the dest fields.

RF token module: A token module for a RF with 2 read/1 write ports is shown in Figure 3(d). Similar to FU token modules, a token receiver and token senders are created for the write port MUX and two read ports, respectively. Any incoming token into the write port sends a read signal to the configuration memory for the write address field and it also sends a write enable signal. For the read ports of register files, there are no incoming tokens from the token network. Instead, the generation of tokens from the read ports are encoded statically in the configuration memory. When a token generation signal comes in, the RF module sends a read signal for the read address and the dest field.

3.3 Supporting Modulo Scheduled Loops

Loops are generally mapped onto CGRAs using modulo scheduling, thus its critical the token network efficiency support efficiencly support this paradigm. Modulo scheduling is a software pipelining technique that exposes loop level parallelism by overlapping the executions of different loop iterations. The basic concept of modulo scheduling is illustrated in Figure 5(a). In modulo scheduling, each iteration starts execution before its previous iteration finishes its execution. By overlapping the executions, modulo scheduling can exploit loop-level parallelism when there are enough resources. The time difference between beginnings of successive iterations is called initiation interval (II). In the steady state, modulo scheduling repeats the same pattern for II cycles and this is called kernel code. Only the kernel code is encoded into the instruction memory, while the pipeline fill/drain (prologue/epilogue) are controlled by staging predicates [13].

3.3.1 Initialization for Kernel Code Execution

In our encoding scheme, the configuration memory contains only the kernel code of target loops and this requires special support for executing modulo scheduled loops with the token network. Figure 5(b)-(c) illustrate the problem that arises. Here, we assume the loop kernel in Figure 5(a) is mapped onto an 1x4 CGRA. Figure 5(b) shows a possible mapping of operations X, Y, and Z on FU 2. The edges with an arrow head indicate tokens flowing on the token network. For operation X, a token arrives at cycle 3 and the operation is activated at stage 0. Similarly, operation Y and operation Z receive tokens at cycles 4 and 9, respectively, and they are activated at stages 1 and 2, respectively. The kernel code of the loop is presented in Figure 5(c). In the steady state, operations are executed in the order of Y, Z, and X and the opcodes for them are

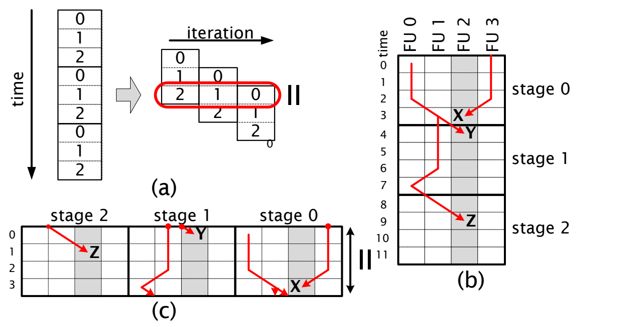

Figure 5: Modulo scheduling basics: (a) Concept, (b) An example mapping for FU 2, (c) Kernel mapping.

stored in the configuration memory in the same order. Therefore, opcodes in FU 2 should be consumed in the order of Y, Z, and X. The problem occurs in the prologue when a token arrives at FU 2 in cycle 3. Since Y and Z are activated in later stages (at cycles 4 and 9), the opcodes for Y and Z are not consumed yet from the configuration memory. As a result, FU 2 reads the opcode Y instead of X's opcode.

The solution for this problem is to maintain the kernel state from the beginning of the loop execution. We can achieve this by initializing the token network with the state of cycle II - 1. Once the state is initialized with the state of cycle II - 1, the tokens can flow through the network and generate the kernel code from the beginning. For initialization, the snapshot of the token network at cycle II-1 is stored separately in the configuration memory. At cycle -1, the initial state is loaded and the token network can maintain the kernel state during the prologue.

3.3.2 *Migrating Staging Predicates*

As previously mentioned, staging predicates are used to fill and drain the pipeline by selectively enabling operations to fill and drain the pipeline. A staging predicate is assigned to each stage of the schedule and it becomes true when current stage is activated in the pipeline. Staging predicates are routed through the predicate network in the datapath and separate configurations are required to manage the routing. Nearly 15% of the configuration bits are used for routing staging predicates in modulo scheduled loops. The kind of information carried with staging predicate is actually control data, hence its inefficient to manage it in the datapath.

For this reason, we propose to migrate the staging predicates from the datapath into the control path. We can simply increase the size of tokens by 1 bit and use the extra bit (valid bit) for the staging predicates. If a resource receives a token with the valid bit set, the incoming data is in the right stage and the operation mapped on the resource can execute. When a token terminates in a register file, it needs to store the valid bit in the register file so that the valid bit information can be retrieved when a token is re-generated later from the same register file. Therefore, RF token modules will include a 1 bit register file that has the same configuration as the original register file in the datapath.

There are several benefits to migrating the staging predicates. First, the configurations for routing the staging predicates in the datapath is not necessary anymore. The routing information of the valid bit in the control path is same as the token routing information, so no additional configuration is required. The second benefit is a performance gain for loops. Removing the staging predicates in the datapath also removes the staging predicate edges in the dataflow graphs. With less scheduling restrictions, the compiler can find better schedules for the same target loops. Also, the predicate network in the datapath is not used for routing staging predicates anymore and can be dedicated to support predicates for if-converted code. The overhead of this approach is mainly in the hardware side. The interconnect in the token network is increased by 1 bit and a 1 bit clone of each register file in the datapath is added to the RF decoders. Also, there is an encoding overhead for the activation stages for live-in values. The trade-off for migrating staging predicate will be discussed in Section 5.

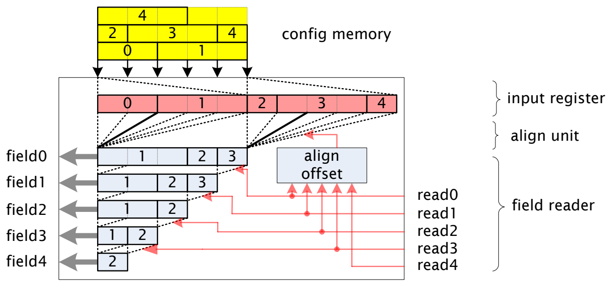

Figure 6: Decoder for fine-grained code compression

4. CONFIGURATION MEMORY PARTITIONING

The decoding logic for fine-grain code compression is shown in Figure 6. It is composed of three components: input register, align unit, and field reader. Encoded instructions are stored in the configuration memory as shown in the figure. Input register buffers each word line of the configuration memory and align unit makes sure that the instruction to be decoded is placed at the leftmost position in the field reader. Based the instruction format given(read signals in Figure 6), each instruction field is fetched in the field reader.

Obviously, having a giant 845-bit wide configuration memory is not a feasible design and also increases the complexity of the decoder drastically. Therefore, the configuration memory needs be partitioned and it needs be done in an efficient way that reduces the complexity of the decoder. Configuration memories are generally built with SRAMs and their power consumption is determined by the width of the memories. Partitioning the configuration memory into smaller SRAMs increases the total power consumption of all the SRAMs. This is because each individual SRAM has its own peripherals and they add up to the total power consumption. When a single 128 bit-wide SRAM is partitioned into eight 16 bit-wide SRAMs, the total power consumption for reading 128 bit data increases 46% for the partitioned case than the original 128bit-wide SRAM. On the other hand, a small configuration memory has the benefit of decreased complexity for the decoder attached to it. For example, a decoder with 4 fields can be built with 22 MUXes, but doubling the number of fields require 71 MUXes. Therefore, having two decoders with 4 fields is 40% more efficient than one decoder with 8 fields. Therefore, partitioning the configuration memory needs be done efficiently considering the trade-off between the SRAM power consumption and the complexity of decoders.

Field Uniformity: When partitioning the configuration memory, it is also important to determine which fields are bundled together and stored in the same memory. Different widths in the same configuration memory increase the complexity of the align unit and introduce an encoding overhead with padding bits. Therefore, we allow only same type of instruction fields to be bundled together.

Sharing of Field Entries: The width of each partitioned configuration memory determines the maximum number of instruction fields that can be fetched in each cycle. Since the width of the memory is also related to the complexity of the attached decoder, we can optimize the decoder complexity by limiting the maximum instruction fields for a single cycle. For example, let's assume that 4 constant fields from four FUs are bundled together and stored in the same memory. The worst case scenario is that all 4 constant fields are used in the same cycle, and the decoder has to have 4 field entries. If the worst case rarely happens, we can limit the number of active constant fields in each cycle. For example, the memory can have only 2 entries and 4 constant instruction fields can share them. While only two constant fields can be active in the same cycle, the complexity of the decoder decreases. The trade-off here lies between the performance of the schedule and the decoder complexity. We learned that the average utilization of instruction fields varies from 10% to 80% depending on the type of instruction fields. For under-utilized fields, it is definitely beneficial to allow instruction fields to share field entries in the decoder.

Design Space Exploration: The design decisions in each component have trade-offs with other components. Thus, we performed a design space exploration to find a good partitioning of the configuration memory. The configuration memory was partitioned differing the bundling of instruction fields, the number of partitioned memomries, and the sharing of field entries in a memory. Due to

field type	opcode	dest	const	crf read	crf write	ldrf	control
memory configurations	2x(8, 8)	4x(8, 5), 4x(8, 6)	2x(6, 10)	1x(11, 9)	1x(6, 7)	4x(6, 3)	1x(1,68)

(a)

					power (mW)				area (mm^2)			
design	m	v	# bits	perf	sram	dec	token	total	sram	dec	token	total
baseline	1	0	845	100.0	104.0	5.4	0.0	109.4	0.539	0.015	0.000	0.554
static	0	0	647	98.5	56.4	18.2	0.0	74.6	0.412	0.120	0.000	0.532
token 0	0	0	485	98.5	31.9	16.5	3.5	51.9	0.309	0.109	0.030	0.448
token 1	1	0	606	99.6	37.2	22.2	3.5	62.9	0.386	0.139	0.029	0.555
token 2	0	1	456	103.8	27.2	17.1	4.8	49.1	0.291	0.113	0.048	0.452
token 3	1	1	567	105.4	30.6	23.1	4.7	58.4	0.361	0.145	0.046	0.553

(b)

Figure 7: (a) Configuration memory partitioning, (b) Performance, power and area comparison of control path designs

the space limitations, only the final result is shown in Section 5.

5. EXPERIMENTS

In this section, we evaluate our control path design with the token network. we created four instances of the token network differing in multicasting capabilities and staging predicate and compared them with design (a) and (b) in Figure 2.

5.1 Experimental Setup

Target Architecture: The target CGRA architecture is a 4×4 heterogeneous CGRA shown in Figure 1. 4 PEs have load/store units to access the data memory and 6 PEs have multiply units. There is a 64-entry central register file with 6 read and 3 write ports wherein only FUs in the first row can directly read/write. All other FUs can only read from the central RF via column buses. The central register file is primarily used for storing live-in values from the host processor. There is also a predicate register file that has 64 entries and 4 read/4 write ports. Each FU has its own 8-entry local register file with one read and one write port. Local register files can be also written by FUs in diagonal directions (upper right/upper left/lower right/lower left). For example, local RF in PE 5 can be written by FUs 0, 2, 5, 8 and 10 and only FU 5 can read from it.

Target Applications: For performance evaluation, we took 214 kernel loops from four media applications in embedded domains (H.264 decoder, 3D graphics, AAC decoder, and MP3 decoder). The loops, varying in size from 4 to 142 operations, were mapped onto the CGRAs and configurations were generated by the compiler. The performance is measured by the average throughput of all 214 loops for each control path design.

Compiler Support: We developed a modulo scheduler that can supports our control path restrictions. First, the compiler makes sure that a value generated in an FU or a register file read port can be consumed up to two neighboring resources to meet the two destinations limit. Also, the compiler actively limits the number of active fields in each cycle as to the sharing degree of the configuration memories.

Power/Area Measurements: Area and power consumption were measured using the RTL Verilog model and synthesized with Synopsys design compiler using typical operation conditions in IBM 65nm technology. Power consumption was calculated using Synopsys PrimeTime PX. The SRAM memory power was extracted from data generated by the Artisan Memory Compiler. The model contained both the datapath and control path and was targeted at 200MHz. Our control path design with the fine-grain code compression decoder and the token network fits in a single pipeline stage between the configuration memory and the datapath, and it does not affect the critical path of the datapath.

5.2 Configuration Memory Partitioning

We performed a design space exploration for partitioning the configuration memory as explained in Section 4. The final result of the optimal partitioning is shown in Figure 7(a). The first row shows different types of instruction fields in our target CGRA and the partitioning result is shown in the second row for each field type. Three numbers in each entry of the table indicate the followings: the number of configuration memories, the number of field entries in a memory, and the bitwidth of each field. For example, there are two memories for opcode fields and each memory has eight 8-bit field entries. Since the opcode fields are frequently utilized, the total number of field entries in the opcode memories is equal to the number of FUs. This means that all 16 FUs can be

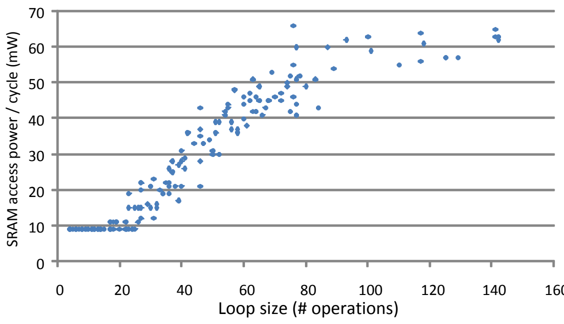

Figure 8: Cache effect on SRAM power consumption

activated at the same cycle. On the other hand, there are only 12 field entries for const fields(2 memories with 6 field entries). So, only 12 FUs can utilize the const fields at the same cycle. In addition to the configuration memories for instruction fields, the control memory in the last column is created to manage the behavior of the token network. The control memory has the token generation information as explained in Section 3.2.1 and read signals for other configuration memories.

In the original control path design shown in Figure 2(a), 845 bits of configurations are distributed in 7 configuration memories(six 128-bit memories and one 77-bit memory) with 128 word lines. In our partitioning scheme, there are 19 configuration memories and the total width of them is 881 bits. Even though the total bits of all the configuration memories has slightly increased, these memories are less frequently accessed since the code size is decreased. Therefore, we can achieve the power reduction in the control path. Also, the increased code efficiency decreases the memory requirements and the number of word lines in each memory can be reduced, resulting in area reduction of the SRAMs. When compared to a naive partitioning scheme where configuration memories are partitioned for each PE, our optimal partitioning achieves 22% power reduction and 33% area reduction for the decoder, while the performance degradation due to sharing of field entries is less than 1%.

5.3 Token Network Evaluation

Six control path designs were evaluated for performance, area, and power consumption and the results are shown in Figure 7(b). *baseline* design is the conventional control path of CGRAs that has no code compression(Figure 2(a)). *static* design employs a fine-grain code compression, but the instruction format is statically encoded in the configuration memory as shown in Figure 2(b). For control path designs with the token network, we created four instances that differ in multicasting capability and staging predicate support. The second column in Figure 7(b) indicates whether the design allows only two destination fields for each datapath component or allows multicasting as *baseline* design. The third column shows if the control path design contains valid bit network to support staging predicates(Section 3.3.2). For each control path design, the average number of configuration bits per cycle for all the target loops are shown in the fourth column. The performance of each design is normalized to the performance of *baseline* and shown in the fifth column. The rest of the table contains power consumption and area of the designs. The control path is broken down into three categories(SRAM, decoder, and token network) and each category's power and area are shown separately in the table. *baseline* and *static* don't have a token network and the decoder in *baseline* is composed of only a pipeline register between the configuration memory and the datapath. For other designs, the pipeline register is included in the decoder(*static*) or in the token network(*token* designs).

For the performance and the number of configuration bits, all 214 loops were evaluated and the average is shown in the table. The SRAM power in the table is the average power per cycle for all 214 loops. For power consumption of the decoder and the token network, an average activity equivalent to the average utilization of FUs(55%) was assumed. The area of SRAMs for each design was calculated based on the amount of configurations required for the loops in MP3 decoder which require 128 word lines in *baseline* design.

Fine-grain Code Compression: Comparison between *baseline* and *static* designs reveals that the fine-grain code compression can improve both power consumption and area of the control path with increased code efficiency. Overall, the power consumption was reduced by 32% and the area decreased by 4%. There is a small performance degradation of 1.5% due to the sharing of field entries and lack of multicasting capability.

We can notice that the SRAM read power reduction ratio(46%) is greater than the reduction ratio in the number of configuration bits(24%). This is due to the cache effect of the input register in the decoder(Figure 6). If all the configurations of a single loop can fit in the input register(two word lines in the configuration memory), the SRAM access happens only at the beginning and the content in the input register does not change throughout the execution of the loop. This occurs quite often when fine-grain compression is applied especially for less frequently used fields such as const fields or predicate fields. In *baseline* design, this cache effect is only achieved when loops are scheduled at II=1 (only 5% in our target loops).

Figure 8 shows the overall cache effect for the 214 target loops. X-axis shows the number of operations in each loop and Y-axis shows the average SRAM read power per cycle for each loop. In this figure, SRAM access power for instruction formats is not included. For small loops, the SRAM power is greatly reduced since most of the configurations can fit in the input register. As the size of a loop increase, the cache effect is minimized and the SRAM access power increases.

Among the average configuration bits of 647 in *static* design, the instruction format takes 172 bits and it needs be read from the memory every cycle. The power consumption of reading instruction format alone is 24.6 mW, which is almost one-third of the total power consumption in the control path. So, there is potential for further enhancing the control path design in the instruction format.

Token Network: We can evaluate the token network by comparing *token 0* to *static*. The only difference between two designs is how the instruction format is discovered. In *token 0* design, the token network is added for dynamic discovery of the instruction format. The overhead of the token network is relatively small, introducing only 3% and 5% of *baseline* design's power consumption and area, respectively. However, introducing the token network improves all three features of the control path: code efficiency, power consumption, and area. *token 0* design further reduces the power consumption by 31% over *static* design and by 53% over *baseline* design. The area of the control path also decreases even with the overhead of the token network since the instruction format is no longer stored in the configuration memory.

To evaluate the limitation of two destinations, we created *token 1* design by adding multicasting capability in *token 0* design. To enable multicasting, each destination field is extended to a bit vector whose width equals to the number of destinations. While there is a small performance gain of 1.1%, the lengthened destination fields lead to poor code efficiency and the power consumption increases by 21%.

An interesting result can be found with migrating the staging predicates into the control path. A valid bit was added to the token networks of *token 0* and *token 1* designs to create *token 2* and *token 3* designs, respectively. Although there is some overhead for having valid bits in the token network, this overhead is mitigated by the improvements in the SRAM, and the overall power consumption and area decrease. This is because the configuration bits for routing staging predicates are not necessary anymore and the code efficiency improves. Moreover, there is a performance improvement of 6% in both cases of *token 2* and *token 3*. By removing staging predicate edges in the dataflow graph, scheduling restrictions are lessened and the chance of the compiler's finding a better schedule increases.

System Power Consumption: From the results in Figure 7(b), we concluded that *token 2* design is the most efficient control path design for our target CGRA. When compared to *baseline* design, the power consumption was improved by 56% and the area was decreased by 19%. Even with the limitation of two destinations, migrating staging predicates into the control path provides the overall performance improvement of 3.8% over the *baseline* design. Figure 9 shows the comparison of the power consumption of the system including the control path and the datapath, with two control path designs of *baseline* and *token 2*. Power was measured by running a kernel loop in H.264 that was scheduled at II=5. The overall utilization of the FUs for this loop is 61%. The numbers at the bottom indicate the overall power consumptions of two designs. When the token network is introduced, the portion of the control path power decreases from 48% to 35%, and the overall system

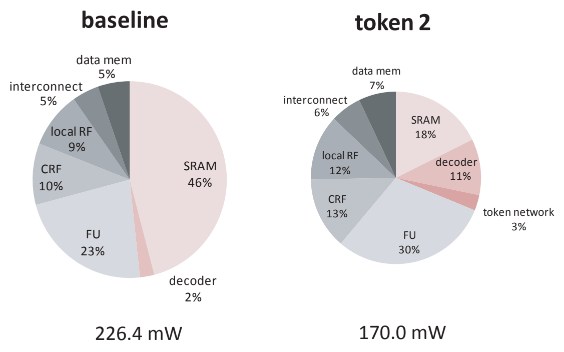

Figure 9: Power breakdown of *baseline* and *token 2* designs for a kernel loop in H.264

power is decreased by 25%.

6. CONCLUSION

This paper proposes a new control path design for CGRAs that utilizes the concept of a token network in dataflow machines for fine-grain code compression. The datapath is cloned to create a token network where tokens are flowing to discover the instruction formats. A design methodology for the control path with a token network is provided and an optimized solution was found through design space exploration. The resulting control path reduces the control power consumption by 56% while enabling a performance gain of 4%. Also, the area of the control path decreases by 19% since the configuration memory requirement is lowered with better code efficiency. Overall, our new control path design achieves a 25% saving in the system power consumption.

7. REFERENCES

[1] M. Ahn, J. W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proc. of the 2006 Design, Automation and Test in Europe*, pages 363–368, Mar. 2006.

[2] F. Bouwens, M. Berekovic, B. D. Sutter, and G. Gaydadjiev. Architecture enhancements for the adres coarse-grained reconfigurable array. In *Proc. of the 2008 International Conference on High Performance Embedded Architectures and Compilers*, pages 66–81, Jan. 2008.

[3] 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.

[4] A. Lambrechts, P. Raghavan, M. Jayapala, F. Catthoor, and D. Verkest. Energy-aware interconnect optimization for a coarse grained reconfigurable processor. In *Proc. of the 2008 International Conference on VLSI Design*, pages 201–207, Jan. 2008.

[5] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 194–203, Dec. 1997.

[6] S. Liao et al. Code optimization techniques for embedded DSP microprocessors. In *Proc. of the 32nd Design Automation Conference*, pages 599–604, 1995.

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

[8] 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.

[9] 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.

[10] H. Pan and K. Asanovic. Heads and tails: a variable-length instruction format supporting parallel fetch and decode. In *Proc. of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 168–175, Nov. 2001.

[11] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.

[12] 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.

[13] 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.

[14] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multidimensional loops. *ACM Transactions on Architecture and Code Optimization*, 4(1):7, 2007.

[15] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, thumb and the armt7tdmi. *IEEE Micro*, 15(2):22–30, 1995.

[16] 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.

[17] Texas Instruments. *TMS320C55x DSP CPU Programmer's Guide*, Aug. 2001. http://focus.ti.com/lit/ug/spru376a/spru376a.pdf.