Reducing Control Power in CGRAs with Token Flow

Hyunchul Park, Yongjun Park, and Scott Mahlke

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

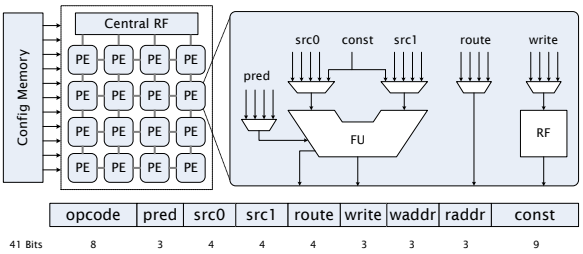

Figure 1: CGRA overview: 4x4 array of PEs (left), a detailed view of a PE (right), and a PE instruction (bottom)

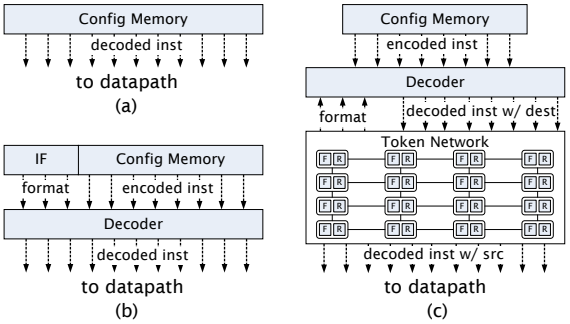

Figure 2: Different Control Path Designs: (a) No compression, (b) Fine-grain code compression with static instruction format, (c) Fine-grain code compression with a token network (F and R indicate FU token module and RF token module, respectively)

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. CGRAs generally consist of an array of a large number of function units (FUs) interconnected by a mesh style network(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 [2, 1]. Moreover, control bits are read from the on-chip memory every cycle regardless of the array's utilization. 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

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 [4]. However, instruction-level compression does not work well in CGRAs due to the highly distributed nature of the resources. We 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%.

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. Slide 6 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 the next Section.

3. TOKEN NETWORK

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. We can utilize a token network in dataflow machines [3] 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.

3.2 Overview

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.

Token modules are connected by a 1-bit token interconnect that has 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.

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.

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 Sender: For each output port of a datapath element (FU output ports, RF read ports), a token sender 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.

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 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).

FU token module:

The input MUXes of the FU is 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. 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.

			power (mW)				area (mm^2)			
design	# bits	perf	sram	dec	token	total	sram	dec	token	total
baseline	845	100.0	104.0	5.4	0.0	109.4	0.539	0.015	0.000	0.554
static	647	98.5	56.4	18.2	0.0	74.6	0.412	0.120	0.000	0.532
token	456	103.8	27.2	17.1	4.8	49.1	0.291	0.113	0.048	0.452

Figure 3: Performance, power and area comparison of control path designs

RF token module: Similar to FU token modules, a token receiver and token senders are created for the write port MUX and two read ports of RFs, 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.

4. EXPERIMENTAL RESULTS

Three control path designs were evaluated for performance, area, and power consumption and the results are shown in Figure 3. *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(Figure 2(b)). *token* also employs a fine-grain code compression and the instruction format is discovered dynamically with token network(Figure 2(c)).

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(only two dests are allowed).

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* to *static*. The only difference between two designs is how the instruction format is discovered. In *token* 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* 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.

5. REFERENCES

[1] 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.

[2] 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.

[3] 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.

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