AN IMPLEMENTATION TECHNIQUE OF MULTI-CYCLED ARITHMETIC FUNCTIONS FOR A DYNAMICALLY RECONFIGURABLE PROCESSOR

Miwa Miyata, Hideyuki Tsuchiya, Yuichiro Shibata, Kiyoshi Oguri

Department of Computer and Information Sciences, Nagasaki University, Japan
email: {miyata, tsucchy, shibata, oguri}@pca.cis.nagasaki-u.ac.jp

ABSTRACT

Dynamically Reconfigurable Processor (DRP) released by NEC Electronics is expected to have potential for high degree of parallel processing. Applications for DRP are described in C language, and parallelism in the source code is automatically extracted by a compiler. On the other hand, it is also important to optimize descriptions so that the potential performance of the device is effectively brought out. In this paper, arithmetic algorithms and an optimized coding technique to efficiently implement applications with multi-cycled arithmetic functions on DRP are discussed, focusing on the required number of the states. In this technique, the same kind of multi-cycled functions are aggregated into single functions, and arithmetic algorithms whose behavior is steady on operand values are utilized. The effects of the technique are evaluated with fixed-point arithmetic functions and polynomial arithmetic functions over a finite field, showing 2.68~3.09 times performance improvement without large increase in the number of states nor severe degradation of the frequency.

1. INTRODUCTION

Recently, technologies of coarse-grained dynamically reconfigurable processors such as Dynamically Reconfigurable Processor (DRP) by NEC Electronics have been rapidly developed [1, 2, 3, 4]. They are strong in executing applications that process stream data including image, voice, and encryption processing [5, 6, 7]. Applications for DRP are described in C language, and parallelism in the source code is extracted by a compiler. On the other hand, it is also important to optimize descriptions so that the potential performance of the device is effectively brought out. In addition, the number of states of a finite state machine (FSM) for application control, which is also derived from the source code by the compiler, must be considered, since the number of states is limited in DRP. Therefore, parallel processing of multi-cycled arithmetic functions should be carried out as simply and regularly as possible to save the states.

In this paper, arithmetic algorithms and an optimized coding technique to efficiently implement applications with multi-cycled arithmetic functions on DRP are discussed, focusing on the required number of the states. In this technique, the same kind of multi-cycled functions are aggregated into single functions, and arithmetic algorithms whose behavior is steady on operand values are utilized. The effects are evaluated with applications using fixed-point arithmetic functions and polynomial arithmetic functions over a finite field.

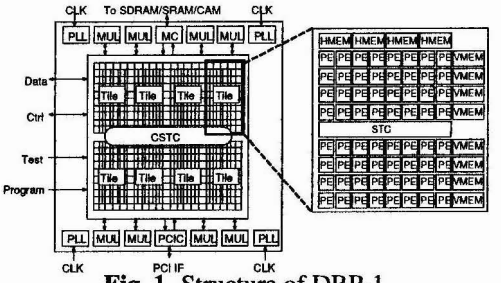


Fig. 1. Structure of DRP-1.

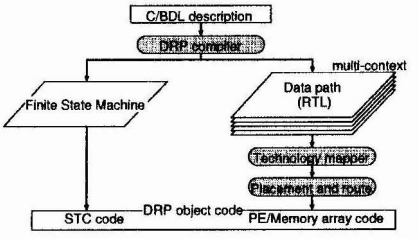


Fig. 2. Compile flow.

2. DRP OVERVIEW

NEC's DRP is a coarse-grained multi-context dynamically reconfigurable processor architecture [1]. The structure of prototype chip DRP-1 is shown in Fig. 1. It contains 4×2 reconfigurable units called Tiles and eight 16-bit multipliers. Within each Tile, there are 8×8 processing elements (PEs), a State Transition Controller (STC), eight 2-port memory blocks (VMEM), and four 1-port memory blocks (HMEM). The capacity of VMEM is 8×256 bits, and that of HMEM is 8×8K bits. DRP-1 has 512 PEs, 80 VMEM (VMEM blocks are placed between Tiles and at both ends of the chip) and 32 HMEM in all. A PE has an 8-bit computing unit, a register file, and an instruction memory. The instruction memory contains 16 instruction (context) sets, and hardware structure can be changed by fetching a new instruction according to a pointer issued by an STC. This context switch can be done in 1 clock cycle, and the next context can be chosen from up to 5 candidate contexts.

1-4244-0312-X/06/$20.00

A compile flow for DRP is shown in Fig. 2. The DRP compiler extracts an FSM and data paths from source code. Then the FSM is translated to be executed on an STC and the data paths are compiled into PE/Memory array code mapped on Tiles. While a context is basically build up by data paths of the corresponding state, up to 4 states can be allocated to the same context. That is, applications with up to 64 states can be implemented using all of 16 contexts.

3. AN AGGREGATION DESCRIPTION FOR ARITHMETIC FUNCTIONS

For efficient parallel processing on DRP, each context must be implemented within a limited number of PEs, as in the case of conventional FPGAs. In addition, we must consider a capacity of states, which is a peculiar property of DRP.

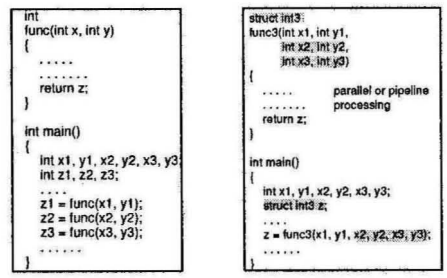
3.1. Synthesis Methods of Functions on DRP

Operators in source code such as addition are realized as combinational circuits, and operator level parallelism is automatically extracted by the DRP compiler. Meanwhile, it is natural that complicated multi-cycled operations are described as functions whose arguments and return value are operands and the result of the operation, respectively.

There are two ways of synthesizing functions for DRP; *inline expansion* which repetitively generates individual circuits for each function call, and *goto conversion* which generates a single circuit for each function and a function call is carried out by state transition. With inline expansion, the DRP compiler can exploit extensive parallelism beyond the function boundaries since the structure of functions is destructed. However, generation of multiple circuits for multiple function calls results in a large number of states. With goto conversion, the required number of states is constant regardless of the number of function calls. However, since function calls are carried out by state transition, inter-function parallelism cannot be extracted. For example, synthesizing three consecutive arithmetic functions in Fig. 3(a) with goto conversion produces the state transition like in Fig. 4(a). Whenever the function is called, the state moves to the function's states, and the three functions are executed sequentially. Thus, goto conversion produces less parallelism than inline expansion.

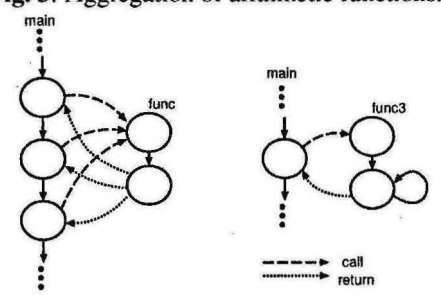
3.2. An Aggregation Technique

When frequently called functions are synthesized with inline expansion, expensive parallelism can be extracted but many states are required. To alleviate increase in the states while exploiting a certain degree of function level parallelism, a coding technique in which the same kind of consecutive functions are *aggregated* into a goto-converted function can be devised. For example, the three functions in Fig. 3(a) are joined up into a new single function whose arguments and a return value are extended to handle multiple operand sets as shown in Fig. 3(b). Inside the function, statements for multiple operations are described to be executed in parallel or in a pipelined manner. When the same arithmetic functions are aggregated, pipelined processing can be described with



(a)before aggregation (b)after aggregation

Fig. 3. Aggregation of arithmetic functions.



(a)before aggregation (b)after aggregation

Fig. 4. State transition diagram.

simple and regular loop sentences like for software pipelining, thus increase in the number of states can be restricted. Fig. 4(b) illustrates the state transition diagram corresponding to Fig. 3(b). Synthesizing the aggregated function with goto conversion, the same sequences of the same arithmetic functions appeared in other portions of the source code can also share the states.

By contraries, when different kind of arithmetic functions were aggregated, state transition of parallel processing would be complex and irregular. This would not contribute to saving of the states. Similarly, arithmetic functions whose behavior is widely varied depending on their operand values are difficult to be efficiently aggregated. This means choice of arithmetic algorithms influences the effect of the aggregation. In order to aggregate as many functions as possible, restructure of source code is also important such as a swap of sentences keeping data and control dependency.

4. FIXED-POINT ARITHMETIC FUNCTIONS

In this section, we evaluate the effect of aggregation of fixed-point arithmetic functions taking ray tracing as an example. Through implementation of various kinds of arithmetic, we discuss how arithmetic influences effects of aggregation.

4.1. Fixed-Point Arithmetic

Our preliminary evaluation has revealed at least 30-bit fixed-point arithmetic is required for a ray tracing algorithm to keep 40-dB PSNR, which is a standard threshold used in irreversible image compression processing. Based on this result, we chose 32-bit fixed-point arithmetic with 12-bit integer part and 20-bit fraction part. We have designed a multiplication function, a square root function, and three types

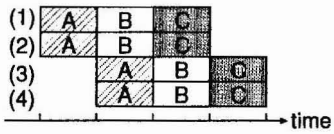


Fig. 5. Aggregation of multiplication.

Table 1. Design of aggregated functions.

arithmetic	aggregated in	states	cycles
Mul	pipeline	2	$n/2+2$
Div (Restore)	parallel	2	54
Div (Convergence)	pipeline	5	$n+15 (n \leq 3)$ $4n+7 (n > 3)$
Square root	parallel	2	28

of division functions using different algorithms; restoring divider, convergence divider, and Kantabutra divider [8].

For 32-bit fixed-point multiplication, four 16-bit embedded DRP-multipliers are required. Arithmetic results can be generated in 3 clock cycles. Here, the process for each clock cycle is represented as A, B and C; signs of operands are recognized and the operands are input to four DRP-multipliers (A), results from pipelined DRP-multipliers are waited for (B), and obtained data from DRP-multipliers are added and shifted (C). Since four of eight DRP-multipliers are used to multiply 32-bit operands, up to two multiplication can be executed in parallel. When more than two multiplication are aggregated, the corresponding operations are executed in a pipelined manner as shown in Fig. 5.

Table 1 shows aggregation type, the number of states, and clock cycles for each aggregated function, where n represents the number of functions aggregated. Multiplication and convergence division functions use DRP-multipliers, thus they are processed in a pipeline on aggregation. Restoring division and square root functions do not use severely limited resources, thus they are executed completely in parallel. The number of states is constant regardless of the degree of aggregation. The required clock cycles are constant for the functions whose operations are processed in parallel. For functions with pipelined processing, the clock cycles are influenced by the degree of aggregation n.

An aggregated version of the Kantabutra division function has not been implemented since aggregation is difficult due to its irregular state transition. This is because that the Kantabutra division algorithm produces 1 or 2 quotient bits per iteration, and the number of iterations to generate a result varies depending on operand values.

4.2. Effects of Aggregation

To evaluate effects of aggregation, the 7 patterns of ray tracing in Table 2 are implemented. We evaluate three degrees of aggregation; no aggregation (Ag0), aggregation of the same arithmetic functions that appeared consecutively (Ag1), and aggregation with loop unrolling (Ag2). There are three algorithms for division; Restoring (R), Convergence (C) and Kantabutra (K). Table 2 also illustrates the maximum degree of aggregation for each arithmetic function. While up to 8 multiplication functions are aggregated in Ag1, the maximum degree of aggregation for Ag2 is improved to 12. Similarly, for division and square root functions, the degree of aggregation is increased by loop unrolling.

Table 2. Evaluated patterns.

	aggregation	division algorithm	max degree of ag. Mul	Div	Sq
Ag0.R	not used	restoring	1	1	1
Ag0.C	not used	convergence	1	1	1
Ag0.K	used	kantabutra	1	1	1
Ag1.R	used	restoring	8	3	1
Ag1.C	used	convergence	8	3	1
Ag2.R	used(unrolling)	restoring	12	6	2
Ag2.C	used(unrolling)	convergence	12	6	2

Table 3. Implementation of ray tracing.

	critical path[ns]	execution time [ms]	states	PEs ave	PEs max
Ag0.R	43.1	15.81	36	80	224
Ag1.R	39.7	8.23	39	97	213
Ag2.R	46.6	5.11	43	160	257
Ag0.C	48.3	9.74	41	68	165
Ag1.C	50.3	6.56	42	101	250
Ag2.C	46.8	3.64	49	185	323
Ag0.K	43.9	10.28	39	101	246

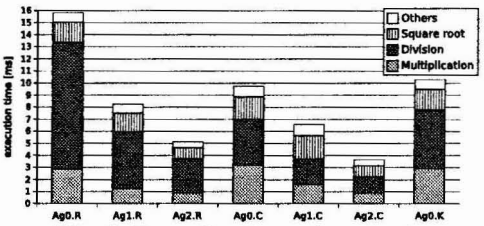


Fig. 6. Execution time of ray tracing.

Table 3 shows implementation results for the evaluated patterns. Critical paths for all patters are approximately 40[ns] to 50[ns], which means the aggregation does not seriously affect the critical path. Increase in the states is also relatively conservative. Table 3 shows the average and maximum number of required PEs as the PE utilization is different for each context. The highest usage ratio is achieved for Ag2.C, where 323 of 512 PEs are utilized.

Fig. 6 illustrates the overall execution time. For the implementation patterns using restoring division, Ag1.R outperforms Ag0.R by 1.92 times. Furthermore, Ag2.R achieves 3.09 times performance improvement by aggregating more functions using loop unrolling technique. Similarly, for the patters of convergence division, Ag1.C and Ag2.C improve the performance by 1.48 times and 2.68 times compared to Ag0.C, respectively.

Focusing on the division algorithms, Ag2.C outperforms Ag2.R by 1.40 times. The convergence division efficiently uses DRP-multipliers and the clock cycles are considerably reduced. Although the effect of aggregation for restoring division is higher than that of convergence division, aggregated convergence division shows the best performance of all. The performance of Kantabutra division algorithm could be further improved [8]. However, it is not suited to aggregation because of irregularity of the state transition.

Table 4. Implementation of multiplication over $GF(2^{4m})$.

	critical path[ns]	execution time[us]	states	PEs ave	PEs max	VMEM
NAg	17.5	39.6	15	51	115	32
Ag	23.4	18.9	14	144	376	32

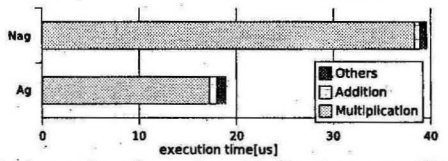


Fig. 7. Execution time of multiplication over $GF(2^{4m})$.

5. POLYNOMIAL ARITHMETIC FUNCTIONS

In this section, the aggregation method is applied to polynomial arithmetic functions over a finite field, which are used for the Tate pairing. The Tate pairing is a mapping on elliptic curves and provides good functionality for constructing cryptographic protocols such as Identity Based Encryption (IBE) [9]. In this work, we implement multiplication over $GF(2^{4m})$, which is the dominant calculation for the Tate pairing, using x^{241} x^7 1 for the irreducible polynomial (m 241).

5.1. Polynomial Arithmetic in a Finite Field

In the fixed-point arithmetic functions described in the previous section, operands and a result of an operation are passed by value through arguments and a return value of a function. However, in polynomial arithmetic, this style requires a lot of registers shared among contexts since long operand words are used, and tends to provide tight constraints on place and route processing. Therefore, we adopt a pass-by-reference style in which actual operand values are stored in VMEM blocks. By interleaving an operand into multiple VMEM blocks, a large bandwidth is provided and a 242-bit operand value can be read out with a single access. In addition, since VMEM is a 2-port memory, two operands for an arithmetic operator can be fetched at the same time. While a VMEM access contains 1-cycle latency, pipelined accesses can hide it to a certain degree.

Multiplication over $GF(2^{4m})$ contains addition and multiplication operations over $GF(2^m)$, and multiplication over $GF(2^m)$ is dominant in $GF(2^{4m})$ multiplication. Therefore only multiplication functions are aggregated in this implementation. We compare two types of implementation; "NAg" in which the aggregation method is not used and "Ag" in which up to three $GF(2^m)$ multiplication functions are aggregated. A non-aggregated $GF(2^m)$ multiplication function consumes 3 states and requires 244 clock cycles to generate a result. On the other hand, an aggregated function consumes 5 states and generates 3 sets of results in 246 clock cycles. Since the aggregated $GF(2^m)$ multiplication function does not use DRP-multipliers, processes for individual arithmetic can be executed in parallel. However, pipelined read accesses to VMEM increase the required clock cycles slightly compared to the non-aggregated function.

5.2. Effects of Aggregation

Implementation results of multiplication over $GF(2^{4m})$ with and without aggregation are summarized in Table 4. Although the impact of aggregation on the critical path is relatively larger than that for the ray tracing, the higher operation frequency is still achieved. The number of states for NAg and Ag are almost equal, but the maximum PE usage is improved by 3.3 times.

Fig. 7 illustrates the execution time and its breakdown. Almost 95% of the overall execution time for the $GF(2^{4m})$ multiplication is accounted by $GF(2^m)$ multiplications. Aggregating these arithmetic functions, 2.09 times performance improvement is achieved at the cost of the one extra state.

Compared with the ray tracing, critical paths are significantly reduced despite higher utilization of PEs. Multiplication over $GF(2^{4m})$ is executed with VMEM blocks, and interconnection between PEs seems not to be so complicated. These implementation results suggest that efficient use of rich embedded resources like distributed memories is another key to high speed design for DRP.

6. CONCLUSION

In this paper, an optimized coding technique to efficiently implement applications with multi-cycled arithmetic functions on DRP is discussed, focusing on the required number of the states. As a result, the aggregation technique achieves 2.68~3.09 times performance improvement without large increase in the number of states nor severe degradation of the frequency.

Acknowledgment

The authors would like to express their sincere gratitude to Dr. Awashima of NEC, Prof. Amano of Keio University, NEC Corp., and NEC Electronics Corp. for technical advises.

7. REFERENCES

[1] M.Motomura, "A dynamically reconfigurable processor architecture," *Microprocessor Forum*, Oct. 2002.

[2] V.Baumgarten, F.May, A.Nuckel, M.Vorbach, and M. Weinhardt., "Pact XPP — a self-reconfigurable data processing architecture," *J.Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[3] T. Sugawara, K. Ide, and T. Sato, "Dynamically reconfigurable processor implemented with IPFlex's DAPDNA technology," *IEICE Trans. on Inf.&Syst.*, vol. E87-D, no. 8, pp. 1997–2003, Aug. 2004.

[4] M. Saito, H. Fujisawa, N. Ujiie, and H. Yoshizawa, "Cluster architecture for reconfigurable signal processing engine for wireless communication," *Proc. FPL*, pp. 353–359, Aug. 2005.

[5] M.Suzuki et al., "Stream applications on the dynamically reconfigurable processor," *Proc. FPL*, pp. 137–144, Dec. 2004.

[6] S.Kurotaki, N.Suzuki, K.Nakadai, H.G.Okuno, and H.Amano, "Implementation of active direction-pass filter on dynamically reconfigurable processor," *Proc. IEEE/RSJ ICIRS*, pp. 515–520, Aug. 2005.

[7] Y.Hasegawa, S.Abe, H.Matsutani, K.Anjo, T.Awashima, and H.Amano, "An adaptive cryptographic accelerator for IPsec on dynamically reconfigurable processor," *Proc. FPL*, pp. 163–170, Dec. 2005.

[8] V. Kantabutra, "A new algorithm for division in hardware," *Proc. ICCD*, pp. 551–556, 1996.

[9] R.Dutta, R.Barua, and P.Sarkar, "Pairing-based cryptographic protocols: A survey," Cryptology ePrint Archive, Report 064/2004, 2004.