Stream Applications on the Dynamically Reconfigurable Processor

Masayasu Suzuki, Yohei Hasegawa, Yutaka Yamada,
Naoto Kaneko, Katsuaki Deguchi, and Hideharu Amano
Graduate School of Science and Technology, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa 223-8522, Japan
drp@am.ics.keio.ac.jp

Kenichiro Anjo and Masato Motomura
NEC Electronics
1753 Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa 211-8668, Japan

Kazutoshi Wakabayashi, Takao Toi, and Toru Awashima
NEC System Devices Research Laboratories
1753 Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa 211-8668, Japan

Abstract

Dynamically Reconfigurable Processor (DRP) developed by NEC Electronics is a coarse grain reconfigurable processor that selects a data path from the on-chip repository of sixteen circuit configurations, or contexts, to implement different logic on one single DRP chip. Several stream applications have been implemented on the DRP-1, the first prototype chip, and evaluation results are presented. By pipelining the executions, DRP-1 outperformed Pentium III/4, embedded CPU MIPS64, and Texas Instruments DSP TMS320C6713 in some stream application examples. We also present programming techniques applicable on dynamically reconfigurable processors and discuss their feasibility in boosting system performance.

1 Introduction

SoC (System-on-a-Chip) which integrates an embedded CPU, standard I/O, and application specific hardware has been widely used for many electronic products and dedicated hardware. Devices implemented with SoC are well suited for intense applications and the custom design enables the reduction in die size and power consumption. Efforts to develop SoCs much faster than its current pace has led to the introduction of new design methodologies such as the C based description language and hardware/software co-synthesis models.

However, recent advances and introduction of new technologies in the areas such as signal processing, data communications, and network protocol handling have made the SoC a lessor attractive option. The higher development costs, diversification of the product line, necessity for swift and comprehensive response toward new standards, and low quantity of the devices shipped are some of the factors that discourage implementation onto an SoC. Moreover, floor-planning and chip layouts are becoming to be the new bottlenecks in design especially in advanced CMOS processes where wiring delay is critical. Unfortunately, high level design technologies for SoC design cannot contribute much in solving these problems.

A chip combining a CPU and a coarse grain reconfigurable fabric has received attention as a solution to this problem. Since the configuration of a coarse grain reconfigurable device is flexible, the same chip can be used for various applications. It can also be "refitted" after shipment by rewriting the configuration data. Because most applications do not need special types of computing units, fine grain reconfigurable architecture using LUTs is not always efficient in performance and cost. Although large scale FPGAs with embedded CPUs (i.e. Xilinx's Virtex-II Pro and Altera's Excalibur) are commercially available, their main target remains to be in prototyping due to high costs.

Recent coarse grain dynamically reconfigurable devices[1, 3, 5, 6, 7, 8] have been developed to achieve high performance and flexibility for a fraction of the cost of an FPGA. It incorporates the following properties: (1) A coarse grain cell consisting of ALU, data manipulator, and register files is adopted as the primitive processing element; (2) In reducing the cost and die size, time-multiplexed execution is introduced with multicontext functionality; (3) Multiple contexts are interchanged rapidly (of-

0-7803-8652-3/04/$20.00 © 2004 IEEE

ten in one clock) to implement different tasks; and (4) High level design entry and functional synthesis technique developed for SoC design can be adopted for designing these devices.

We have implemented several stream applications on the Dynamically Reconfigurable Processor(DRP)[5] prototype chip DRP-1 and have found five techniques that are applicable for designing hardware on any dynamically reconfigurable device. They are as follow:

1. The division of target application into each context should be carefully designed so as to not degrade parallelism.
2. For efficient parallel processing, data stream must be divided into small fragments and stored in the distributed memory modules.
3. When there are dependencies between processes within an application, pipeline execution of independent data streams are efficient.
4. Limited resources (i.e. multipliers) can be efficiently used by effectively scheduling the pipeline executions.
5. Because it is faster to reconfigure the processor than to move the data within the processor, keeping the data distributed around the PEs and "modifying" the logic around the data is substantially better than continuously moving the data to the individual processing elements.

Their feasibility in raising system performance is evaluated in Section 5.

In this paper, design and implementation of stream applications on the Dynamically Reconfigurable Processor(DRP)[5] prototype chip DRP-1 are presented and evaluated. In Section 2, a review of the DRP is done and several applications are discussed in Section 3. Performance comparison against other architectures are done in Section 4. Our proposal concerning the programming of reconfigurable processors are discussed in Section 5. The paper concludes in Section 6.

2 DRP Overview

DRP is a coarse-grain dynamically reconfigurable processor core which can be integrated into ASICs and SoCs. The primitive unit of DRP Core is called a 'Tile,' and a DRP Core consists of arbitrary number of Tiles. The number of Tiles can be expandable, horizontally and vertically.

The primitive modules of a Tile are processing elements (PEs), a State Transition Controller (STC), 2-ported memories (Vertical Memories or VMEMs), its controller (VMCtrl), and 1-ported

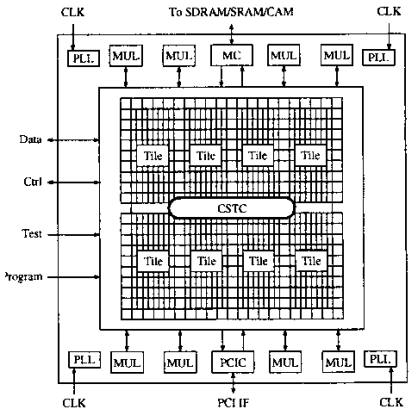

Figure 1. The DRP-1 Architecture and its Core

memories (Horizontal Memories or HMEMs). The structure of a Tile is shown in Figure 2.

There are 64 PEs located in one Tile. The architecture of a PE is shown in Figure 3. It has an 8-bit ALU, an 8-bit data management unit (DMU; for shifts/masks), sixteen 8-bit register file units (RFU), and an 8-bit flip-flop (FFU). These units are connected by programmable wires specified by instruction data, and their bit-widths range from 8Bytes to 18Bytes depending on the location. PE has 16-depth instruction memories and supports multiple context operation. Its instruction pointer is delivered from the STC.

The STC is a programmable sequencer in which finite state machine (FSM) can be stored. STC has 64 states, and each state is associated with an instruction pointer. FSM of STC operates synchronized with the internal clock, and generates the instruction pointer for each clock cycle according to the state. Also, STC can receive event signals from PEs to branch conditionally. The maximum number of branch is four.

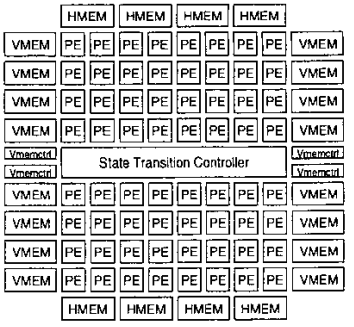

Figure 2. Structure of a Tile

As for the memory units, a Tile has sixteen 2-ported VMEMs on its right and left sides, and eight 1-ported HMEMs on its upper and lower boundary.

Each VMEM is 8-bit in width and 256 of them are provided for each Tile. Also, four VMEMs can be handled as a FIFO, using VMCtrl. HMEM is a single-ported 8-bit memory with 8K entries. Contents of these memories, flip-flops, and register files of PEs are all connected and shared by the contexts.

The DRP Core, consisting of several Tiles, can change its contexts every cycle with the instruction pointer distributed from the central STC (CSTC). The individual STCs within the Tiles can also be ran independently by programming different FSMs.

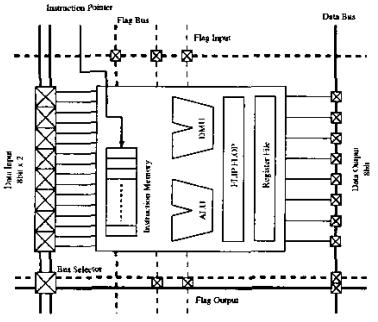

Figure 3. Architecture of a PE

As shown in Figure 1, the prototype chip DRP-1 consists of a DRP Core with eight Tiles. It is fabricated with 0.15-μm 8-metal layer CMOS process. It consists of 8-Tile DRP Core, eight 32-bit multipliers, an external SRAM controller, a PCI interface, and 256-bit I/Os. The maximum operation frequency is 100-MHz. Although DRP-1 can be used as a standalone reconfigurable device, Tiles of DRP can be used as an intellectual property (IP) on ASICs with an embedded processor. In this case, the number of Tiles can be chosen so as to achieve the required performance with minimum area.

An integrated design environment for DRP-1 which includes a high level synthesis tool, a design mapper for DRP, simulators, and a layout/viewer tool is provided. Applications can be written in a C-based high level hardware description language, synthesized, and mapped directly onto the DRP-1.

3 Application Examples

The following applications are implemented on the DRP-1: an Alpha Blender with Anti-aliasing, Block Cipher RC6, Discrete Cosine Transform (DCT), Inverse Modified Discrete Cosine Transform (IMDCT) for MP3, Discrete Waveform Transform (DWT), and Viterbi Decoder. Because of the page limitation, two applications (DWT and Viterbi Decoder) are omitted and others are briefly discussed in this paper.

3.1 Alpha Blender with Anti-aliasing

Alpha blending, an application from image processing, combines two images into one in relation to the parameter α. The formula for alpha blending is given as follows.

$$Image_{new} = \alpha Image_{zero} + (1-\alpha) Image_{one}$$

Anti-aliasing removes the jaggy (the border regions of an image that is not smooth) from the image generated by the alpha blender. By complimenting the pixel data in relation to its surrounding pixels, the jaggy edges of an image can be removed. The formula for anti-aliasing for an arbitrary pixel $K_{i,j}$ is given as follows.

$$K_{i,j(new)} = \frac{\omega}{10} \begin{pmatrix} K_{i-1,j+1} + K_{i,j+1} + K_{i+1,j+1} \\ +K_{i-1,j} + 2K_{i,j} + K_{i+1,j} \\ +K_{i-1,j-1} + K_{i,j-1} + K_{i+1,j-1} \end{pmatrix}$$

The weight ω is an arbitrary number. To derive an anti-aliased pixel data, one would need the data of the pixel in question and the data of its eight surrounding pixels.

For implementation onto the DRP-1, the total task was divided into five contexts that passed the processed data to the succeeding context. Context 0 is used for initialization, and executed only once. In Context 1, red and green pixels of RGB data are taken from outside the chip, and alpha blending is performed. In Context 2, the rest of the data (blue pixels) are processed, and with results from Context 1, they are written into the distributed memory VMEM with an address computed with an address generator. Context 1 and 2 are repeatedly executed, and when enough data for anti-aliasing is stored in the VMEM, the STC triggers Context 3 and 4 for anti-aliasing.

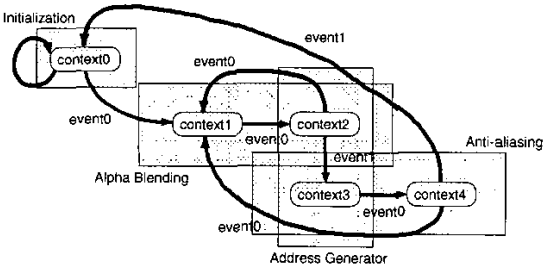

Figure 4. State transition diagram for alpha blender with anti-aliasing

Figure 4 outlines the state transition of this implementation. The initial setup time before anti-aliasing is $2(width_x + 2)$ clocks and it takes approximately four clocks to generate one anti-aliased image therein after.

Table 1. Required resources for alpha blender with anti-aliasing

CONTEXT	ALU	DMU	FFU	RFU	PE
0	2	2	0	31	31
1	9	7	0	10	10
2	24	7	0	19	24
3	34	21	0	11	34
4	16	17	0	8	17
MAX	**34**	**21**	**0**	**31**	**34**

Table 1 shows the required resources for this implementation; only a Tile is used for this implementation. The maximum operational frequency is 38MHz.

3.2 Block Cipher RC6

The RC6 block cipher [9] is a symmetric encryption algorithm developed by Ronald Rivest and his team in 1998. It is a parameterized algorithm where the block size, the key size, and the number of rounds are variables. After Rijndael[2] had been selected for the Advanced Encryption Standard (AES), efforts to promote RC6 further still remains as NP 18033 project (via the ISO/IEC JTC 1/SC27 [10]), and the Information-technology Promotion Agency in Japan is also considering RC6 for use within the Japanese Government.

In this implementation, the non-feedback Electronic Codebook (ECB) mode was selected. ECB mode divides the data into 128-bit blocks and encrypts/decrypts by these units. ECB is weak in that any combination of the same plain text and key returns an unique encrypted message but it can be processed in parallel. The data size was fixed to 32-bits while key size and round number were variables between $0 < b \leq 32$[bytes] and $0 < r \leq 255$, respectively. A block size of 128-bits was processed in this setting.

Figure 5 explains the basic data flow in the RC6 implementation. This RC6 processor is built from a key scheduler, four encryption modules, four decryption modules, and input and output interfaces.

The data stream is divided into 128-bit blocks and stored into input FIFO realized by the FFUs. Each encryption module gets a block from the FIFO and performs rounding operations r times. Because of the data dependence between different rounds, and because the eight multipliers can process 32-bits at maximum, there could only be four parallel processes at any given time. To enhance parallelism, rounding is divided into two and the rounding for two blocks are executed in the pipelined manner as shown in Figure 6.

In this Figure, Pipeline 0 processes multiply-add operations and fixed length rotation of the rounding,

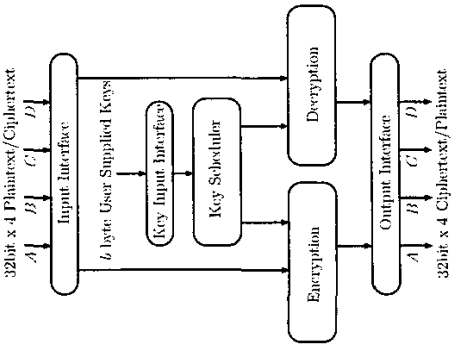

Figure 5. Diagram of the RC6 processor

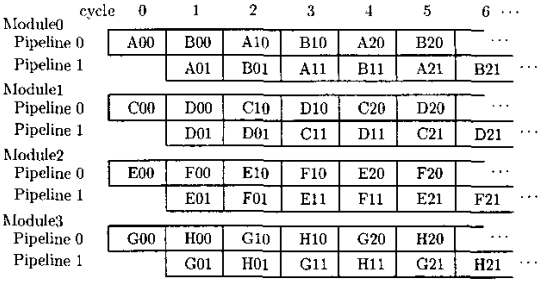

Figure 6. Pipeline processing of the RC6 processor

while Pipeline 1 processes variable length rotation. By overlapping the processes for two blocks, multipliers could efficiently be used in each clock cycle.

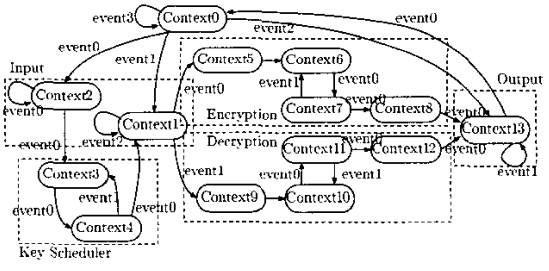

Figure 7. Context transition diagram for the RC6 processor

In this implementation, the total process is divided into 14 contexts as shown in Figure 7. After initialization in Context 0, Context 1 and 2 are used for stream data input, and data is distributed into VMEM modules. In Context 3 and 4, key scheduling is done and round key S is obtained, and encryption/decryption is started. Context 5, 6, 7, and 8 are for encryption. Pipelines shown in Figure 6 are implemented in these contexts. Context 9, 10, 11, and 12 are for decryption and they are similar to that of the encryption pipeline. The encryption/decryption

result is transferred outside the chip in Context 13.

Table 2. Required resources for the RC6 processor

CONTEXT	ALU	DMU	FFU	RFU	PE
0	5	17	27	5	27
1	27	147	137	3	147
2	8	5	0	3	8
3	20	9	24	2	24
4	32	10	25	2	32
5	32	72	72	2	72
6	162	137	136	2	162
7	164	137	136	3	164
8	32	2	40	0	40
9	99	73	168	2	168
10	163	137	160	2	163
11	161	136	160	2	161
12	128	64	168	0	168
13	10	116	138	16	138
MAX	**164**	**147**	**168**	**16**	**168**

Unlike the previous design example, this design uses all eight Tiles of the DRP-1 chip utilizing the resources fully. As shown in Table 2, more than 60% of the resources, including PEs and wiring modules, are used in Context 6, 7, 11, and 12.

3.3 Discrete Cosine Transform (DCT)

Discrete Cosine Transform is a widely used transformation method used in JPEG and various types of MPEGs.

Here, DCT used in JPEG coder is implemented. In this implementation, the program is written in an extended C language, and the configuration of DRP-1 is generated with the DRP compiler.

The steps of DCT are as follows:

- The input data stream is stored in a 8×8 two dimensional matrix.
- A row of the matrix is accessed in order, and fixed coefficients (A-C) are multiplied with the sums. That is, SUM0 = d[i][0]+d[i][7], SUM1 = d[i][1]+d[i][6], SUM2 = d[i][2]+d[i][5], and SUM3 = d[i][3]+d[i][4]. Then, intermediate results (Z0, Z2, Z4, and Z6) can be obtained as follows:

```
Z0 = A*SUM0 + A*SUM1 + A*SUM2 + A*SUM3;
Z2 = B*SUM0 + C*SUM1 - C*SUM2 - B*SUM3;
Z4 = A*SUM0 - A*SUM1 - A*SUM2 + A*SUM3;
Z6 = C*SUM0 - B*SUM1 + B*SUM2 - C*SUM3;
```

- Fixed coefficients (D-G) are multiplied with difference of accessed data (SUB0-SUB3), that is, SUB0 = d[i][0]-d[i][7], SUB1 = d[i][1]-d[i][6], SUB2 = d[i][2]-d[i][5], and SUB3 = d[i][3]-d[i][4], and intermediate results (Z1, Z3, Z5, and Z7) can be obtained as follows:

```
Z1 = D*SUB0 + E*SUB1 + F*SUB2 + G*SUB3;
Z3 = E*SUB0 - G*SUB1 - D*SUB2 - F*SUB3;
Z5 = F*SUB0 - D*SUB1 + G*SUB2 + E*SUB3;
Z7 = G*SUB0 - F*SUB1 + E*SUB2 - D*SUB3;
```

- The shifted results are stored into the row of a matrix.
- A column of the matrix is accessed in order.
- Similar computation to those of rows is applied to the column.

Since the coefficients (A-G) are fixed during computation, the above multiplication can be reduced into shift-and-add operation by the DRP compiler. However, if the stream data is stored in a single matrix and is accessed in order, the parallelism is strictly limited. So, in this implementation, the stream data is stored in eight vectors (d0-d7) each of which is assigned to independent VMEMs. In this policy, row order access (d0[i]-d7[i]) can be done in parallel, but column order access (di[0]-di[7]) must be serially done. Like most embedded memories, VMEM in DRP-1 has a clock cycle delay from addressing to data read out. We minimize the execution cycles by accessing the VMEMs in a pipelined fashion.

First, in order to enhance parallelism, computation for vector-j (column direction) is divided into the following steps:

Step 1. Addressing dj[0]

Step 2. sum0 = sub0 = dj[0],
Addressing dj[7]

Step 3. Z0=A*(sum0+dj[7]),
Z1=D*(sub0-dj[7]), ...,
Z7=G*(sub0-dj[7]),
Addressing dj[1]

Step 4. sum1 = sub1 = dj[1],
Addressing dj[6]

Step 5. Z0+=A*(sum1+dj[6]),
Z1+=E*(sub1-dj[6]), ...,
Z7+=F*(sub1-dj[6]),
Addressing dj[2]

⋮

Step 8. sum3 = sub3 = dj[3],
Addressing dj[4]

Step 9. Z0+=A*(sum3+dj[4]),
Z1+=G*(sub3-dj[4]), ...,
Z7+=D*(sub3-dj[4])

Step 10. Store shifted Z0 to dj[0]

Step 11. Store shifted Z1 to dj[1]

⋮

Step 17. Store shifted Z7 to dj[7]

Note that, the above 17 steps can be applied to every vector in parallel. However, steps which require a lot of PEs are only 4 steps: 3, 5, 7, and 9, while others require only memory access and simple computation. So, we implemented the above steps for vector-j in a pipelined manner as shown in Figure 8. Since two vectors are accessed with a clock delay, the

number of required PEs are mostly equalized in each step. Although the parallelism is degraded in the final 8 clocks for storing data, a part of this sequence can be overlapped with the process for stream output.

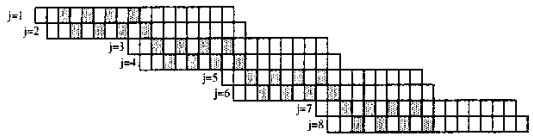

Figure 8. Pipelined execution for column direction

Figure 9 shows context transition diagram for DCT. Each context are represented with a white circle whose area is relative to the required number of PEs, and the figure in a circle shows the number. The black circles represent the required clock cycles for each contexts. The usage of PEs are kept nearly equal throughout the contexts including the computation in the column direction.

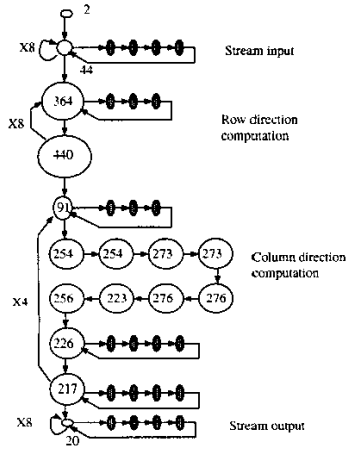

Figure 9. The context transition diagram for DCT

The execution clock frequency is about 15MHz, relatively low, but the total execution speed for computation is 5.9μsec. Including stream input/outputs, all 16 contexts, the maximum supported on the DRP-1, were used. Figure 3 shows the resources necessary for this implementation. No multipliers were used and the maximum number of required PEs in a context was 440.

TI's DSP can execute the same C code in 15.5μsec, and the performance gain of DRP-1 is about 2.6 times.

Table 3. Required resource for DCT

CTXT	ALU	DMU	FFU	RFU	PE
0	0	2	0	2	2
1	10	44	0	7	44
2	154	364	16	2	364
3	153	440	16	25	440
4	91	86	0	34	91
5	141	254	0	49	254
6	133	254	0	53	254
7	155	273	0	53	273
8	147	273	0	53	273
9	155	276	0	53	276
10	147	276	0	53	276
11	118	223	0	45	223
12	103	256	0	59	256
13	215	226	0	113	226
14	217	215	0	124	217
15	20	20	0	2	20
MAX	**217**	**440**	**16**	**124**	**440**

3.4 IMDCT in MP3 Decoder

The last example is acceleration of an MPEG-1 Audio Layer III (MP3) decoder. This implementation assumes a chip consisting of a simple embedded CPU and the DRP, and full size DRP-1 with eight Tiles are used. Figure 10 shows the process flow of the MP3 decoder[4]. Inverse Modified Discrete Cosine Transform (IMDCT) is selected for the DRP execution, since it occupies the largest portion of the MP3 decoding process, or 40% of total execution time according to profiling.

The width of the bus between the core processor and the DRP-1 is assumed to be 32-bits, and the block transfer of 576×32-bit sample data is done from the RAM to the DRP-1. After finishing the operation in the DRP-1, a wave data of the same size is transfered back to the RAM.

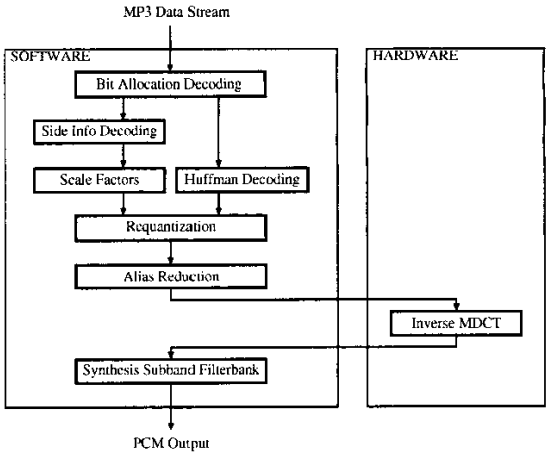

Figure 10. MP3 decoder software/hardware co-execution

All 16 contexts are used in this implementa-

tion. Figure 11 shows the IMDCT process flowchart. Eleven contexts in the DRP-1 are used for the IMDCT process as shown in Figure 11. Most of them require multipliers and adders. In addition to those eleven contexts, five contexts are used for data distribution and collection: *Input*, *Memory Read*, *Memory Write*, and *Output*. The state transition is depicted in Figure 12. DRP-1 part executes 18 sample data per iteration – 32 iterations execute 576 sample data.

We adopt a policy of "move the logics instead of data." The sample data is transfered to VMEMs in *Input* and stored in PE's registers in *Memory Read*. The eleven logic contexts switch in to act on the data, while the data is moved as little as possible. By limiting the movement of data to within the PE, the data path is folded fourth dimensionally reducing the critical path as well as the wiring resource demand. The results are written back to VMEMs in *Memory Write* and the data is transfered back to the processor in *Output*.

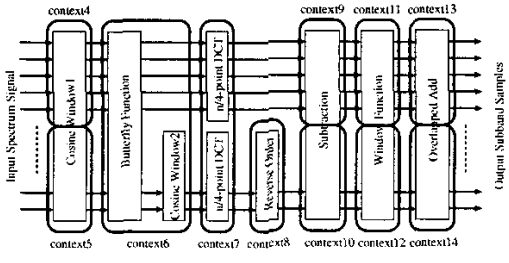

Figure 11. Context layout for IMDCT

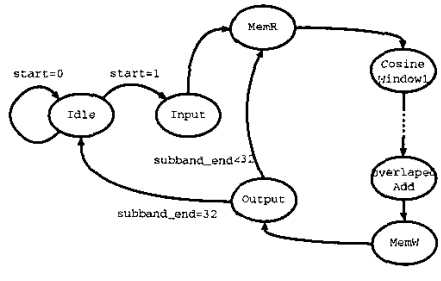

Figure 12. State transition diagram for IMDCT

Table 4 shows the required resource for each context. Context 3 (distribution of data stream) and Context 15 (concatenation) required the most PEs in realizing multiplexors/de-multiplexors.

The maximum operating frequency of the decoder is 36MHz. For 576 sample data input, the DRP-1 executes the IMDCT in 103 μsec.

4 Evaluation

We evaluate the performance of the DRP-1 by (1) its relative performance against the competing architectures, and (2) the amount of resources used in realizing the application.

Table 4. Required resources for IMDCT

CTXT	ALU	DMU	FFU	RFU	MUL	PE
0	3	0	6	0	0	6
1	5	12	9	3	0	12
2	7	9	0	19	0	19
3	62	222	0	152	0	222
4	21	39	0	74	8	74
5	21	39	0	74	8	74
6	77	58	31	163	8	163
7	152	149	93	172	8	172
8	53	4	8	77	0	77
9	100	12	20	86	0	100
10	40	0	3	73	0	73
11	57	39	0	110	8	110
12	57	39	0	110	8	110
13	21	39	0	74	8	74
14	21	39	0	85	8	85
15	210	124	118	194	0	210
MAX	**210**	**222**	**93**	**194**	**8**	**222**

4.1 Performance Comparison

The applications implemented on the DRP-1 were tested against a high-end embedded processor MIPS64, a Texas Instruments (TI) DSP TMS320C6713, and Pentium III and Pentium 4 processors. MIPS64 is a four-way in-order super scalar processor which runs at 500MHz. 32KB L1 instruction cache and 32KB L1 data cache are provided along with the 512KB L2 cache. TI's VLIW style DSP TMS320C6713 is tested on the evaluation board TMS320C6713 DSK that operates at 225MHz. 4KB L1 instruction cache and 4KB data cache are provided along with the 256KB L2 cache. Code Composer Studio Ver. 2.20.05, a development tool exclusively for the TI DSPs, are used to compile the program.

Applications ran on the Pentiums and the MIPS64 were compiled with gcc 2.95.3 with -O3 optimization option.

The summary of the performance and required number of contexts is shown in Table 5. Note that applications on the DRP-1 performed better than other platforms that operate at higher frequencies. In most stream applications, it is the workload, not complexity, that determines the overall performance. Although DRP-1 operates at much lower frequency than the other architectures, the data path architecture together with pipeline executions and parallel processing led to the speed up.

4.2 Resources Used

The maximum number of resources needed for each application are shown in Tables 1, 2, 3, and 4. Notice that no application is implemented in one

Table 5. Performance comparison

Application	Platform (No. of Tiles, Contexts)	Freq.	Throughput
α blender	DRP-1 (1,5)	38 MHz	145.0 fps
w/anti-aliasing	Pentium4	2.53 GHz	48.1 fps
(256*256)	DSP	225 MHz	8.2 fps
RC6	DRP-1 (8,14)	32 MHz	829.4 Mbps
(encryption)	MIPS64	500 MHz	144.5 Mbps
	DSP	225 MHz	96.7 Mbps
RC6	DRP-1 (8,14)	32 MHz	829.4 Mbps
(decryption)	MIPS64	500 MHz	137.6 Mbps
	DSP	225 MHz	94.7 Mbps
DCT	DRP-1 (8,16)	15 MHz	173.5 Mbps
	DSP	225 MHz	66.0 Mbps
IMDCT	DRP-1 (8,16)	36 MHz	329 Mbps
	Pentium III	800 MHz	267 Mbps
DWT	DRP-1 (8,10)	57 MHz	650 fps
(80*80)	Pentium 4	1.70 GHz	230 fps
(filter length=3)	Pentium III	800 MHz	320 fps
Viterbi	DRP-1 (8,12)	33 MHz	3.3 Mbps
decoder	Pentium 4	2.40 GHz	300-600 bps

context but a sequence of contexts which are time-multiplexed in one clock.

Except for the DCT and the Viterbi Decoder, less than half of the PEs available on the DRP-1 were consumed while giving throughput better than that of the competing architectures. Although the operating frequencies of the DRP-1 were less than 20% of other architectures, better performance were achieved because numerous pipelines were mapped onto the DRP-1 simultaneously. These pipelines processed vast amounts of data in parallel thus resulting in high throughput.

It is the heavy workload of stream applications that gives the DRP-1 the edge against other architectures.

5 Feasibility of Our Design Techniques

We analyze the five implementation techniques that we believe are useful in programming dynamically reconfigurable devices and discuss their feasibility below.

1. The division of target application into each context should be carefully designed so as to not degrade parallelism. Pipeline processing of the RC6 processor and the DCT implementation proves this point.

2. For efficient parallel processing, data stream must be divided into small fragments and stored in the separate distributed memory modules. An example of this is the DCT implementation where data fragmentation/separation enabled pipelined executions.

3. When there are dependencies between processes within an application, pipeline execution of independent data streams are efficient as shown in Figure 6 in the RC6 implementation.

4. Limited resources (i.e. multipliers) can be efficiently used by the scheduling of the pipeline execution as in the RC6 processor shown in Figure 6.

5. Because it is faster to reconfigure the processor than to move the data within the processor, keeping the data distributed around multiple PEs and "modifying" the logic around the data is substantially better than continuously moving the data to the individual PEs. IMDCT implementation for the MP3 decoder shows this point. Without this technique, it would have taken more time to achieve the same results.

6 Conclusion

In this paper, we have implemented and evaluated several stream applications on the prototype chip DRP-1. Our implementation shows that one single Tile of DRP can outperform that of a high-end processor. With more complex applications, all eight Tiles can be utilized to achieve performance that can surpass those of Pentium III/4.

Through our implementation, we have extracted and proposed programming techniques that could be applied to any dynamically reconfigurable processor. By using these techniques, a programmer can gain more out of the dynamically reconfigurable processors.

References

[1] E.Capsi, M.Chu, R.Huang, J.Yeh, J.Wawrzynek, and A.DeHon. "Stream Computations Organized for Reconfigurable Execution (SCORE)." Proc. FPL2000, pp. 605-615, 2000.

[2] John Daemen and Vincent Rijmen. "AES Proposal: Rijndael," http://csrc.nist.gov/encryption/ eas/rijndael/Rijndael.pdf, 1999.

[3] IPFlex Inc. DAP/DNA-2. http://www.ipflex.co.jp

[4] Szu-Wei Lee, "Improved Algorithm for Efficient Computation of the Forward and Backward MDCT in MPEG Coder," IEEE Transactions on Circuits and Systems II: Analog and Digital Processing, vol. 48, issue 10, pp. 990-994, 2001.

[5] M.Motomura, "A Dynamically Reconfigurable Processor Architecture," Microprocessor Forum, Oct. 2002.

[6] QuickSilver Technology. Adaptive Computing Machine (ACM). http://www.qstech.com

[7] PACT XPP Technologies. The XPP White Paper. http://www.pactcorp.com

[8] H.Schmit and et al., "PipeRench: A Virtualized Programmable Datapath in 0.18 Minron Technology," Proc. of IEEE CICC, 2002.

[9] R. Sidney, R. Rivest, M. Robshaw, and Y. L. Yin, "The RC6 Block Cipher," The First Advanced Encryption Standard (AES) Candidate Conference, Jun. 1998.

[10] SO/IEC JTC 1/SC 27. http://www.din.de/ni/sc27/