A Run-time Reconfigurable System for Adaptive High Performance Efficient Computing

Chuan Hong, Khaled Benkrid, Nazrin Isa and Xabier Iturbe
System Level Integration Group, School of Engineering, The University of Edinburgh, UK
{C.Hong, K.Benkrid, M.N.Isa, X.Iturbe}@ed.ac.uk

Abstract— Field programmable hardware gives electronic systems the ability to be reconfigured at run time. This allows electronic systems to be more efficiently customized on demand and on-the-fly depending on user requirements and environmental changes. This paper presents a run-time reconfigurable system that allows computing tasks to adjust their sizes in response to current available resources, optimizing the overall performance by maximally exploiting all the resources on the chip. In particular, we present a novel run-time task assembler, which assembles tasks with desired parameters on-the-fly, together with an efficacious run-time task placer to rapidly configure tasks at optimum locations. The system is demonstrated with a dynamic programming-based pairwise sequence alignment application. Real hardware implementation result shows that our run-time reconfigurable system optimizes resource usage on the fly by ~ 3x, while matching the performance of carefully hand-crafted static solutions.

Index Terms— Reconfigurable Computing, Adaptive Hardware, High Performance Computing, Bioinformatics

I. INTRODUCTION

FIELD programmable hardware gives electronic systems the ability to change their function and configuration after manufacture. By enhancing configurability, the reconfiguration overhead can be reduced in terms of both speed and area, which makes rapid run-time reconfiguration possible [1]. This allows electronic systems to be flexibly reconfigured with desired parameters on-the-fly, therefore hardware resources are more efficiently utilized, by trading off performance with hardware resources and power consumption [2].

However, due to the constraints imposed by increasingly heterogeneous hardware, most existing approaches lack flexibility in terms of the size of reconfigurable tasks and the size of Partial Reconfiguration (PR) regions. Most commonly, such approaches use predefined tasks synthesized off-line [3] [4] [5]. As a result, only a limited number of tasks are available. Moreover, memory space is wasted to accommodate multiple versions of bitstreams. In addition, predefined fixed PR regions whereby boundaries have to be statically fixed to accommodate static communication ports (Bus Macros or Proxy Logic) [6] [7], result in wasted resources since PR regions have to cater for the largest reconfigurable task possible.

To circumvent the above shortcomings, we present a run-time reconfigurable system, which not only supports on-line task generation, but also allows tasks to be placed at arbitrary positions with no boundary concerns. This is achieved by a novel run-time task assembler and a run-time task placer as central components of our reconfigurable system. The task assembler takes into the consideration both user requirements and currently available resources to select hardware resources and assemble them to a full application task on-the-fly; whereas the task placer is responsible for rapidly placing tasks at an optimum position within minimum time overhead by applying state-of-art 2D-packing algorithm with compressed bitstream. The communication between tasks and the host uses a hybrid network, which is based on a high bandwidth bus and an internal configuration port. Our run-time reconfigurable system gives considerably higher flexibility to allow for high performance computing systems to autonomously adapt to user demands and current resource availability. For instance, our system can support scalable multi-user, multi-tasking applications whereby resources can be dynamically managed in respect of user requirements and hardware availability.

We demonstrate our approach in the context of a bioinformatics sequence alignment application [8], in which a query sequence needs to be compared with a database using a pipeline of Processing Elements (PEs). In this context, the number of pipeline stages can be adjusted depending on currently available resources and user requirements. Compared with previous reconfigurable approaches [9] [10], our contribution includes 1) finer grained task scalability and flexible task customization supporting for dynamic multi-user multi-tasking applications; and 2) higher resource usage efficiency and better overall system performance, achieved by adjusting task size with current resource availability.

The remainder of the paper is organized as follows. Section II first outlines the overall proposed run-time reconfigurable system. Then, the run-time task assembler and run-time placer are respectively presented in section III, IV. The sequence alignment application is then demonstrated in section V, with implementation results given in section VI. Finally, conclusions are drawn in section VII.

II. SYSTEM ARCHITECTURE

In our proposed system, the whole chip area is divided into two partitions: a static region for the run-time reconfigurable system and a PR region for application tasks. The latter is not constrained by predefined boundaries, which allows hardware tasks to be swapped in/out in a time multiplexed fashion. Fig.1 depicts our proposed system architecture, in which 4 tasks with
different pipeline stages are placed and executed in their PR region. The static run-time reconfigurable system consists of a host API, a bitstream library, a task assembler and a task placer. The host API is connected in an open network and uses TCP/IP protocols to communicate with clients. After receiving requests from clients, the task information (including task type and required performance) is passed to the run-time task assembler. Depending on the user requirement, the task assembler first decides the type of basic processing elements (PEs) and their constructions, e.g. pipeline stages, then reads from the bitstream library to assemble a full application task and pass it to the task placer. If current resources are insufficient for the task, i.e. task size is larger than maximum available free area on the chip, the task assembler will resize the task to fit within the current available resources using a folding mechanism. The task placer analyzes the currently available chip resources to attempt an optimum position for the task. The optimum position is calculated using state-of-art 2D packing algorithms, which gives more compact placements with less fragmentation.

To allocate tasks, the placer uses the Internal Configuration Access Port (ICAP) with compressed configuration mode to reduce the configuration time overhead. After a task is placed on the chip, the placer will update the information of current available resource (maximum available resource size and type), which is then feedback to task assembler to be used for assembling next upcoming task. The communication between tasks and the host uses a hybrid network, which consists of a high bandwidth bus (high throughput) and the ICAP (low throughput). Tasks requiring high throughput communication are connected to the bus, whereas tasks with low external interaction can be placed anywhere and use ICAP for input/output data. The ICAP based communication utilizes the configuration port to write/read data to/from tasks through the configuration layer, which maintains a routing-less PR region [11]. In our real hardware implementation, the host API is implemented on a Xilinx MicroBlaze microprocessor; whereas the task assembler and task placer are implemented separately on two Xilinx PicoBlaze processors.

III. RUN-TIME TASK ASSEMBLER

The run-time task assembler assembles pre-synthesized PEs to generate functional application tasks. To enable the inter-communication between separately synthesized PEs, an inter-PE Bus Macro (BM) is integrated before synthesis. This is a module that has been manually placed and routed using particular wires. The BM constrains the PEs to use specified wires for both inputs and outputs. Therefore, the PEs are able to communicate with each other as long as the same wires are shared between the previous PE’s output and the next PE’s input within the pipeline. Fig.2 gives an example of the inter-PE communication in the case of a Xilinx Virtex-5 FPGA.

The FPGA resources are firstly divided by vertically aligned clock regions, and each clock region is divided by horizontally aligned columns, including Configurable Logic Block (CLB) columns and Block RAM (BRAM) columns [12]. The short wires are homogeneously distributed between two adjacent CLBs. In this example, PEs are area-constrained within two CLB columns. The horizontal short wires (direct lines) are shared between PE1 and PE2, which exclusively connect two CLBs. To direct the output signal from a flip-flop or a Look-Up-Table (LUT) to the specific wires (direct lines), the output signal hops among Programmable Interconnections Points (PIPs) within a switch box before terminating at direct lines. Since the switch boxes routings are highly regular in each column despite resource heterogeneity, the direct lines can be used between any two columns, which give a constant propagation delay. Xilinx Virtex5 FPGAs have 6 direct lines between each two adjacent CLBs, which provide sufficient bandwidth for regular tasks.

In our proposed run-time system, a task is generally composed by a task header, task body, task tail and dummy columns. The task header is usually implemented in a BRAM column which contains the input data for the task. The task body consists of pipelined PEs, whose length is determined by user requirement and current available resources. The task

![Fig. 1. Illustration of our proposed run-time reconfigurable system](image1)

![Fig. 2. Inter-PE communication illustration](image2)
The EAC algorithm will find the maximum length of PEs to meet both user requirement and current resource availability, otherwise the task will wait in the task queue until previously placed task is finished and removed from the chip. The task tail outputs processed data, which could be implemented using BRAM or CLB depending on the data size. According to the resource type information received from task placer, the dummy columns are used to cope with heterogeneities within the FPGA chip, where for instance CLB columns in Virtex-5 FPGAs are intercepted by other types of columns e.g. BRAM, DSP and IOB. To pass data from a previous PE to the next PE without any data processing, the dummy columns (bypasses) are separately synthesized to directly connect their inputs to their outputs using switch boxes only. Fig.3 gives the steps to generate a customized application task by dynamically assembling PEs from the bitstream library.

IV. RUN-TIME TASK PLACER

Since assembled tasks are expected to have a high variation in their sizes, chip resources could be severely fragmented if tasks are randomly placed. To floor-plan the chip resources and place tasks at an optimum location, a run-time task placer is developed, using state-of-art 2D-packing algorithm and compressed bitstream to reduce the configuration overhead in both area and time, respectively.

A. EAC Placement Algorithm

In our proposed system, the PR is not constrained with communication routings, therefore tasks can be arbitrarily allocated anywhere on the chip. In such context, the FPGA chip resources can be modeled as a 2-dimensional rectilinear grid, where 2D-packing algorithms can be applied. To compute the optimum location within a short period, we have presented Empty Area Compact (EAC) algorithm in [13]. The EAC uses two matrices to represent current chip resource usage. The first matrix is called “Shape Matrix”, in which the occupied cells (CLBs) are marked as zero, and all other cells are incrementally scored in the horizontal direction. The second matrix is named “Area Matrix”, in which each score is the maximum

\[ \text{Area Matrix} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 & 3 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \]

Fig. 4. Illustration of EAC placer

\[ \text{Shape Matrix} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 & 3 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \]

A “Row MER” column and a “Chip MER” are attached to show the Maximum Empty Rectangle (MER) at each row and for the whole chip. Fig.4 gives an example of the two matrices, the five grayed cells are the CLBs occupied by previously placed tasks. The size of the dash-circled area is 3×2, with a width of 3 and area of 6, labeled in the bottom-right cell in Shape Matrix and Area Matrix respectively. An upcoming task is compared with both matrices to pre-select all possible locations. After that, each possible location is scored and the position with the minimum cost is determined as the best location to place the task. Simulations show that the EAC algorithm achieves 25% improvement in task acceptance rate compared to previously developed KAMER and Vertex List-based algorithms [13]. After a task is placed, the task placer will update the chip resource information and then feedback the size and resource type of the chip MER to the task assembler for assembling next task.

B. Compressed Bitstream Configuration

Configuring a task onto the chip requires writing bitstream to the configuration port, e.g. the internal Xilinx ICAP port in Virtex-5 FPGAs which is a 32-bit wide, usually clocked at 100MHz [12]. The bitstream is composed by frames, which represent the minimum configuration unit and consist of 41 32-bit words per frame. The frame number required for each type of resource is given in Table I, and Fig.5 depicts the bitstream alignment for one CLB column in Virtex-5. Conventionally, 41-word frames are written sequentially, which consumes 41 cycles for each frame configuration; hence the configuration time is:

\[ T_{cfg} = T_{init} + (41 \times N_{frame}) \times f_{clk} \]  

(2)

Where \( T_{cfg} \) is the whole configuration time, \( T_{init} \) is the initialization time, \( N_{frame} \) is the number of frames and \( f_{clk} \) is the ICAP clock frequency. However, the configuration speed can be significantly improved if the bitstream consists of a number of identical frames. The identical frames can be replicated to another frame address with 2 more cycles (see Fig.6); therefore the time needed to configure identical tasks can be reduced to:

\[ T_{cfg} = T_{init} + (41 + 2 \times N_{frame}) \times f_{clk} \]  

(3)

The latter is called compressed bitstream configuration, which was originally invented by Xilinx to reduce the bitstream size. In our proposed system, this technique is applied not only for

![Fig. 5. Bitstream and frame alignment of one CLB column in Virtex-5](image-url)

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>XILINX FPGA FRAME NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLB</td>
</tr>
<tr>
<td>Virtex4</td>
<td>22</td>
</tr>
<tr>
<td>Virtex5</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>XILINX FPGA FRAME NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLB</td>
</tr>
<tr>
<td>Virtex4</td>
<td>22</td>
</tr>
<tr>
<td>Virtex5</td>
<td>36</td>
</tr>
</tbody>
</table>

Fig. 5. Bitstream and frame alignment of one CLB column in Virtex-5
Fig. 6. Steps for normal and compressed configuration

speeding up partial reconfiguration, as pipelines often heavily reuse identical PEs, but also for blanking finished tasks (replicating zero frames), so that power consumption is reduced. When multiple identical configurations are required, the compressed configuration mechanism achieves a maximum of ~20x speed-up compared with conventional sequential configuration.

V. CASE STUDY IMPLEMENTATION: PAIRWISE BIOLOGICAL SEQUENCE ALIGNMENT

A. Pairwise Biological Sequence Alignment

In bioinformatics, sequence alignment is used to identify the regional/global similarities between two biological sequence e.g. DNA, RNA or protein sequences. To achieve the optimum alignment, dynamic programming based sequence alignment algorithms such as the Smith-Waterman algorithm [8] are widely used to find and score the best alignment between a query sequence and database sequences. The affine gap penalty based Smith-Waterman (S-W) algorithm introduced by GOTOH [14] in 1982, is widely used today since it is more biologically realistic than the linear gap penalty model. It uses three matrices calculated as shown below:

\[
F(i, j) = \max \begin{cases} 
F(i-1, j-1) + s(x_i, y_j) \\
F(i-1, j) + s(x_i, y_j) \\
F(i, j-1) + s(x_i, y_j)
\end{cases} 
\]

Where \(F(i,j)\) is the best score so far; \(I_s(i,j)\) and \(I_s(i,j)\) are the scores aligned to gaps with residue \(x_i\) and \(y_j\) respectively; \(S(x_i, y_j)\) is the probabilistic score of substituting \(x_i\) with \(y_j\) or vice-versa; \(d\) is the penalty associated with opening a gap, and \(e\) is the penalty associated with extending a gap. The probabilistic score for substituting \(x_i\) with \(y_j\) or vice-versa uses a substitution matrix e.g. BLOSUM50 [8] based on a particular biological model.

B. Hardware Pipeline & PE Folding

The S-W algorithm can be implemented in hardware using pipelined PE arrays for higher performance [15]. The PE array (of length \(N\)) is mapped to query residues (one-to-one), and the database sequence (of length \(M\)) is shifted into the PE array. After the entire database sequence is shifted through the PE array, the highest alignment scores among the database sequence are retained as they depict the more biologically related sequences to the query sequence. However, limited hardware resources in practice rarely allow for one-to-one mapping between PEs and query residues; hence the available resources need to be reused using folding.

Fig. 7 gives an example whereby only three PEs can be fitted in hardware for six-residue query sequence. The three PEs are first assigned with the first three query residues (e.g. ‘P’ ‘A’ ‘R’). After a database sequence is shifted in, the three PEs are then folded with the next three residues (e.g. ‘W’ ‘D’ ‘C’). The results from the first fold are buffered into a FIFO to be used as input in the second fold, hence then need for an input multiplexer.

We have designed a reconfigurable PE architecture which allows for the online generation of scalable pipelines with low time and area overheads [16]. The architecture of a single PE is presented in Fig. 8. Each PE has four inputs \((F(i,j-1), I_s(i,j-1), I_s(i,j), x_i)\), four outputs \((F(i,j), I_s(i,j), I_s(i,j), x_i)\), and a 3-clock cycle latency. The substitution values for a particular query residue are stored in one LUT inside the PE, whereas another LUT in the PE is configured with the next folding substitution values.

C. Integration with our Run-Time Reconfigurable System

In our pairwise biological sequence alignment implementation, each PE uses 2 CLB columns (see Fig. 9). The top 14 switchboxes are used for inter-PE communications, in which the left-side 7 are used for inputting data from the
previous PE and the right-side 7 are used for outputting values to the next PE. Each switchbox has 6 direct lines, which together give a 42-bit wide input, as for the output. At the bottom of each PE, a feedback link (5-bit wide) is integrated to allow the last PE (task tail) to feed its output back to the first BRAM for the next folding.

Our pairwise sequence alignment core was implemented on a Xilinx Virtex5 LX110T FPGA (see Fig.10). The run-time reconfiguration system resides in the static region, leaving all other resources free to be allocated to application tasks. Application tasks can be scheduled and allocated at different positions by the run-time reconfiguration system. For instance, Fig.10 shows two different PE pipeline arrays (21 and 9 PEs in length respectively) running in parallel for two different query sequences (of arbitrary length). In our sequence alignment case study implementation, the PE array starts from a BRAM column (task head), which caches the database sequence from external memory. In the target FPGA chip (Xilinx Virtex5 LX110T FPGA), there are 8x4 BRAM columns, giving 32 possible positions to allocate a task. A high-bandwidth bus is implemented in the leftmost area of the chip for fast update of the first BRAM columns’ content. Since alignment scores have relatively small sizes, the ICAP is used for reading task results.

The chip has 50x8 CLB columns, which allows a maximum of eight 25-long PE pipelines. To further increase the number of pipeline stages for a single task, a PE array can be routed across clock regions by using a previously developed snake strategy [17], allowing for a 200-long PE pipeline.

VI. RESULTS ANALYSIS

The performance of our hardware implementation was tested on an Alpha Data ADM-XRC-5LX board, which has a Xilinx XC5VLX110T FPGA chip. The performance of PE is tested using UniProtKB/TrEMBL database and the real results are compared with other software and hardware implementations. Results show that our dynamically online-generated tasks have zero-penalty on the PE performance, compared with carefully hand-crafted static solutions [16]. TABLE II shows that when the maximum pipeline stage (25 PEs) is used, the task achieves a ~30x speed-up, compared with software SSEARCH35 running on Intel(R) Quad Core 64-bit Q8300. TABLE III gives the result compared with other two hardware approaches. In order to compare performance regardless of the technology of the device, such as gate delay and propagation delay, the normalized speed-up is obtained by:

\[
\text{SpeedUp}_{\text{normalized}} = \frac{\text{SpeedUp}_{\text{run}}}{\text{Device Delay}} \times \text{Device Delay}
\]

Where SpeedUp_{run} is the real speed improvement, Number_{LC} is the number of consumed logic cells, which is proportional to the occupied area, and Device Delay is the propagation delay of the particular device. Result shows that the normalized speed is improved by 1.53x and 2.4x, compared with other two hardware approaches respectively [10] [18].

Table IV gives the performance of task placer, including the time spent on making location decisions and configuring different size of tasks. The average time for configuring one application task (including placement algorithm execution and hardware reconfiguration times) is less than 150 µs, and a task can be removed within 40 µs. In terms of area, the whole run-time reconfigurable system consumes 1210 slices and 6 BRAMs, which is just 7% and 5% of the total resources on XC5VLX110T FPGA, respectively.

A real-time multi-user, multi-tasking test example is given in TABLE V, in which the overall system run-time performance is improved by adjusting task size (number of folds) with the current available chip resources. In this example, six task sets (φ1 - φ6) are used and each task set contains a number of tasks. All the tasks are identically used for P02652 database scanning, but with different user requirements on their performances, namely the number of maximum folds, or the minimum number of PEs. In another word, tasks requiring better performance cannot be folded too many times. During the test, tasks are requested from the user every 36 or 18 seconds, with a different maximum fold number, which is a random number ranging from Fold_{min} to Fold_{max}. For example, in set φ1, a task is requested every 36 second from the user, and the number of allowed folds (maximum fold number) for each task is a random number ranging from 4 to 32. Starting from 4 folds (the minimum folds limited by the XC5VLX110T chip size), the
folding number is doubled (i.e. task size is halved) every time the available resource is not enough for the current folds. If a task cannot be placed with its maximum fold number, it will wait in the task queue until any previous task be finished and removed from the chip. After one hour, the number of competed tasks ($N_{\text{finished\_tasks}}$) is recorded and the completion rate ($R_{\text{finished\_tasks}}$) is calculated. Since task size can be adjusted and fit into smaller resource slots, more chip resources are used at the same time, and less resource stays in idle. Benefiting from this, the task finishing rate is increased by ~3x compared with tasks without using folding adjustment (see Fig.11).

### TABLE II

<table>
<thead>
<tr>
<th>Query Accession</th>
<th>Length</th>
<th>PE</th>
<th>Fold</th>
<th>Execution Time (s)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>P02652</td>
<td>100</td>
<td>25</td>
<td>4</td>
<td>303</td>
<td>9416</td>
</tr>
<tr>
<td>Q0H3V2</td>
<td>200</td>
<td>25</td>
<td>8</td>
<td>599</td>
<td>17160</td>
</tr>
<tr>
<td>Q8NC42</td>
<td>400</td>
<td>25</td>
<td>16</td>
<td>1215</td>
<td>35992</td>
</tr>
</tbody>
</table>

### TABLE III

<table>
<thead>
<tr>
<th>Ref</th>
<th>Device</th>
<th>Resource Ratio</th>
<th>Device Delay Ratio</th>
<th>Raw Speed-up</th>
<th>Normalized Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>[18]</td>
<td>XC2V6000</td>
<td>0.69</td>
<td>0.23</td>
<td>x4.58</td>
<td>x1.53</td>
</tr>
<tr>
<td>[10]</td>
<td>XC2V6000</td>
<td>0.49</td>
<td>0.23</td>
<td>x5.13</td>
<td>x2.40</td>
</tr>
</tbody>
</table>

1 Resource Ratio = LC's consumed by our approach / LC's consumed by Ref.
2 Device Delay Ratio = XC5VLX110 delay / XC2V6000 delay

### TABLE IV

<table>
<thead>
<tr>
<th>Process</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding Location (Average Time)</td>
<td>18 μs</td>
</tr>
<tr>
<td>Configure PE Header</td>
<td>68.38 μs</td>
</tr>
<tr>
<td>Configure 1 PE Body</td>
<td>2.98 μs</td>
</tr>
<tr>
<td>Configure whole task (10x PE)</td>
<td>111.18 μs</td>
</tr>
<tr>
<td>Configure whole task (25x PE)</td>
<td>134.22 μs</td>
</tr>
<tr>
<td>Update Data Input</td>
<td>26.56 μs</td>
</tr>
<tr>
<td>Blank whole task (25x PE)</td>
<td>36.32 μs</td>
</tr>
</tbody>
</table>

### TABLE V

<table>
<thead>
<tr>
<th>Task set</th>
<th>$\phi_1$</th>
<th>$\phi_2$</th>
<th>$\phi_3$</th>
<th>$\phi_4$</th>
<th>$\phi_5$</th>
<th>$\phi_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fold_out</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>32</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Fold_in</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$N_{\text{comput}}$</td>
<td>100</td>
<td>100</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>$R_{\text{comput}}$</td>
<td>Every 36 s</td>
<td>Every 36 s</td>
<td>Every 18 s</td>
<td>Every 18 s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using folding adjustment

- $N_{\text{failed\_tasks}}$ = 91/91% = 73/73% = 57/57% = 93/46.5% = 78/39% = 59/29.5% |
- $R_{\text{failed\_tasks}}$ = 32/32% = 32/32% = 32/19% = 32/19% = 32/19%  |

Not using folding adjustment

- $N_{\text{failed\_tasks}}$ = 32/32% = 32/32% = 38/38%  |
- $R_{\text{failed\_tasks}}$ = 32/32% = 32/32% = 19% = 19% = 19%  

Chip size: 50 x 8 CLB columns, Test duration: 1 hour (3000s) Query accession: P02652, Folding starting number: 4

Fig. 11 Performance improvement by using folding adjustment

### VII. CONCLUSION

In this paper, we presented a run-time reconfigurable system which allows application tasks to adapt their sizes depending on current available resources. The system is demonstrated in the context of a sequence alignment application. Results show that tasks can be generated on the fly with minimal time and area overhead, and the overall efficiency is improved by running multiple tasks for multiple users in parallel in a way that is adaptable to currently available hardware resources and user requirements. As for future works, the system adaptability can be extended to give response to other environmental changes, such as heat, power, and emerging faults for low-power and fault-tolerant applications.

### REFERENCES


