VLSI Design, 2002 Vol. 14 (3), pp. 239–248

Taylor & Francis
Taylor & Francis Group

An Algorithm Visualization Tool on the Reconfigurable Mesh

JACIR L. BORDIM*, TATSUYA HAYASHI and KOJI NAKANO

Department of Electrical and Computer Engineering, Nagoya Institute of Technology, Showa, Nagoya 466-8555, Japan

(Received 1 February 2000; Revised 2 October 2000)

Many parallel algorithms on the reconfigurable mesh have been developed so far. However, it is hard to understand the behavior of these parallel algorithms, mainly because the bus topology dynamically changes during the execution of an algorithm. In this work, we present the visual mesh system (*VMesh*), a tool for visualizing algorithms on the reconfigurable mesh. The main objective of the VMesh is to provide a comprehensive environment for algorithm visualization and development. The VMesh has shown to be a valuable tool for studying and understanding the behavior of parallel algorithms on the reconfigurable mesh.

Keywords: Algorithm visualization; Reconfigurable mesh; Parallel algorithms; Field programmable gate arrays

INTRODUCTION

The newest generation of field programmable gate arrays (FPGA) provide a more robust platform for the development of reconfigurable computers by implementing some key features, such as fast runtime reconfiguration, partial reconfiguration, reprogrammability, and the ability to read the internal state of a device.

On the dynamic reconfigurable computers, many efficient parallel algorithms have been proposed [1,6,7,11,15,16]. However, the operations of such algorithms are not easily understood since the bus topology dynamically changes during the algorithm execution. In general, algorithms are often difficult to understand when viewed in their traditional textual form. This difficulty becomes even greater when considering parallel algorithms. A graphical simulation of an algorithm can expose some properties that might otherwise be obscure in the textual description.

Algorithm visualization (or animation) enables the users to explore the operations of an algorithm. It provides facilities for users to view and interact with their algorithms, by giving ways to control the execution of an algorithm through a graphical display. Algorithm visualization has been used in several areas of computer science, such as teaching, algorithm development, and analysis.

In this work, we propose the visual mesh system (*VMesh*), an algorithm visualization tool on the reconfigurable mesh. The motivation of our work arises primarily from the difficulty of understanding the behavior of the algorithms devised on the reconfigurable mesh. Since the processors internal connections are dynamically changing during the execution of an algorithm, it is difficult to determine which connection is being used by each processor. From the same reason, the occurrence of data collision, which happens when more than one processor attempts to write data on the same bus at the same time, is not easily observed. In addition, when data is traversing among processors, it is hard to identify the data held by each processor during the execution of an algorithm.

To the best of our knowledge, two different visualization systems on the reconfigurable mesh are known: a visualization system for algorithms on PARBS developed by Miyashita *et al.* [12] and the simulator presented by Steckel *et al.* [17]. Both the systems provide facilities to visualize the execution of an algorithm trough a graphical display. However, these systems possess some drawbacks. Miyashita's tool does not provide ways to visualize the algorithm code being executed, which would facilitate the task of finding errors and make improvements on the algorithm.

In the simulator developed by Steckel, it is required to write an algorithm in assembler instruction. This makes the coding task time-consuming and the implementation of more sophisticated algorithms may not be feasible

*Corresponding author. E-mail: jace@maple.elcom.nitech.ac.jp

ISSN 1065-514X print/ISSN 1563-5171 online © 2002 Taylor & Francis Ltd
DOI: 10.1080/10655140290011041

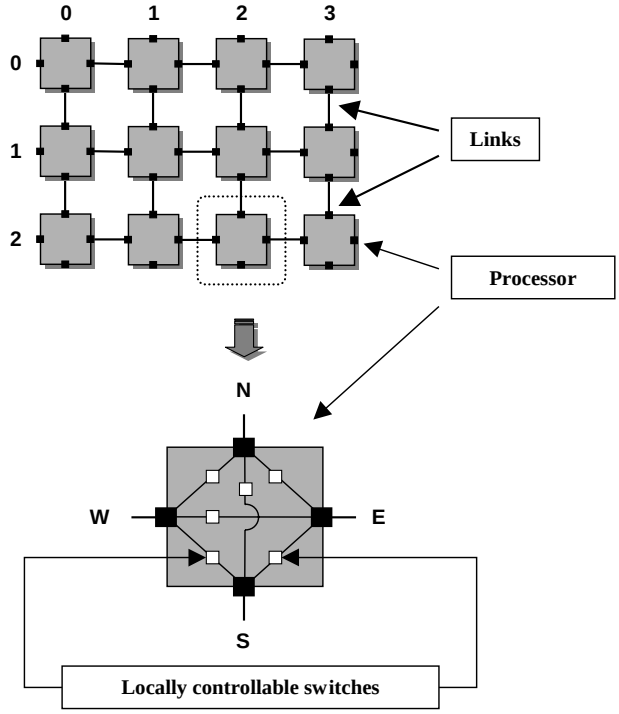

FIGURE 1 A reconfigurable mesh of size 3×4.

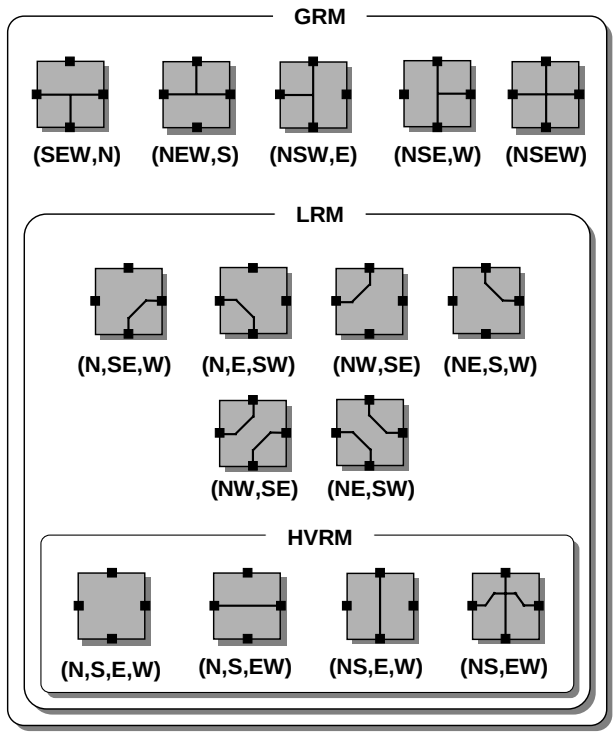

FIGURE 2 Possible port connections of a processor accordingly to the GRM, LRM and HVRM models.

within a reasonable time. Moreover, during the execution of an algorithm it does not provide ways to visualize the data being held by each processor. Neither of the aforementioned systems provide more detailed information regarding the execution of the algorithm.

The VMesh provides facilities to monitor a parallel algorithm in action on the reconfigurable mesh. Among other features, our tool warns the user whenever a data collision occurs, allows the user to visualize the data held by each processor, and shows the algorithm code being executed and the number of active and used connections. In addition, our tool supplies some statistical information to assist users to analyze and compare the performance of similar algorithms. Another important characteristic of our tool is that it uses a high level programming interface which facilitates the implementation of the algorithm.

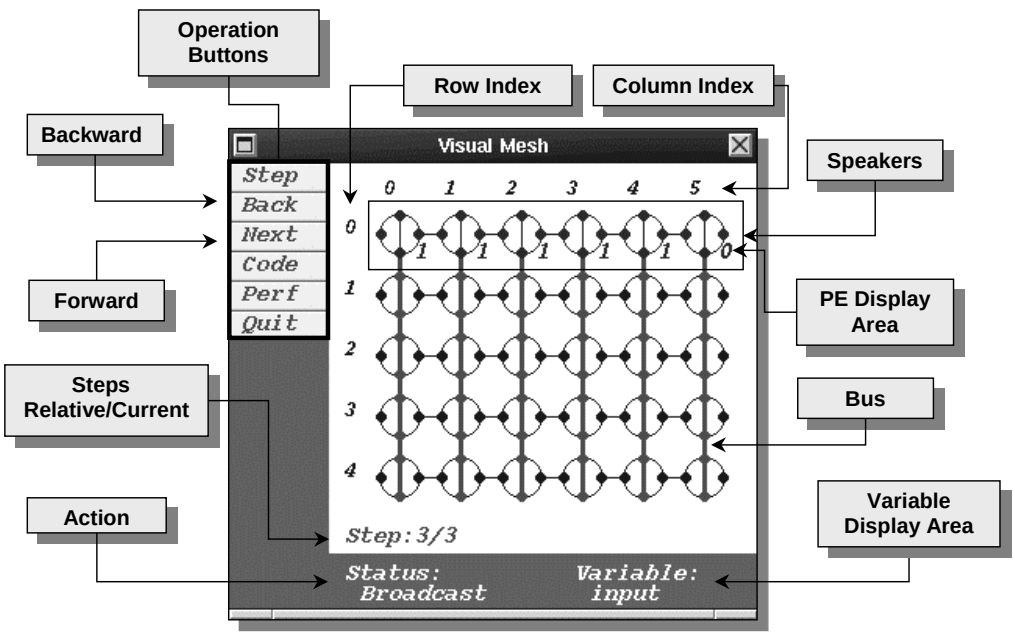

FIGURE 3 The VMesh main window.

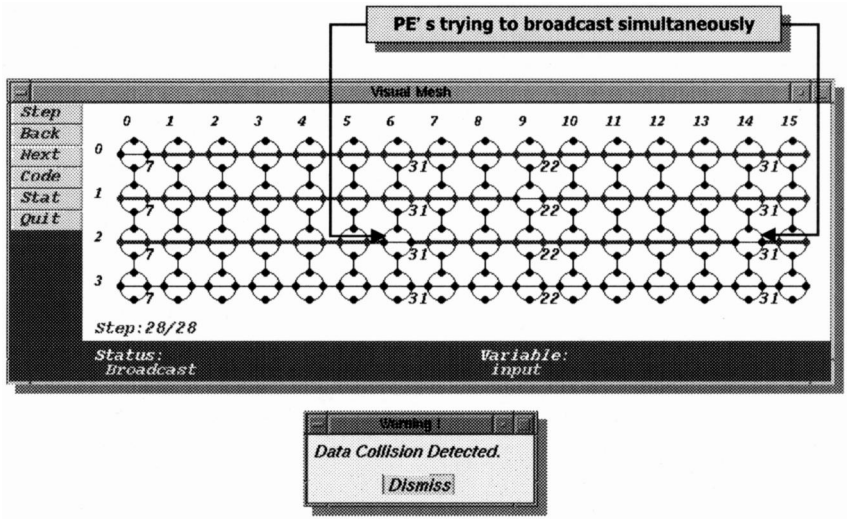

FIGURE 4 Data collision warning message.

The organization of this paper is as follows: the second section briefly describes the reconfigurable mesh model. The VMesh system organization and features are covered in the third section. In fourth section, we present some algorithms implemented on the VMesh, and fifth section concludes the paper.

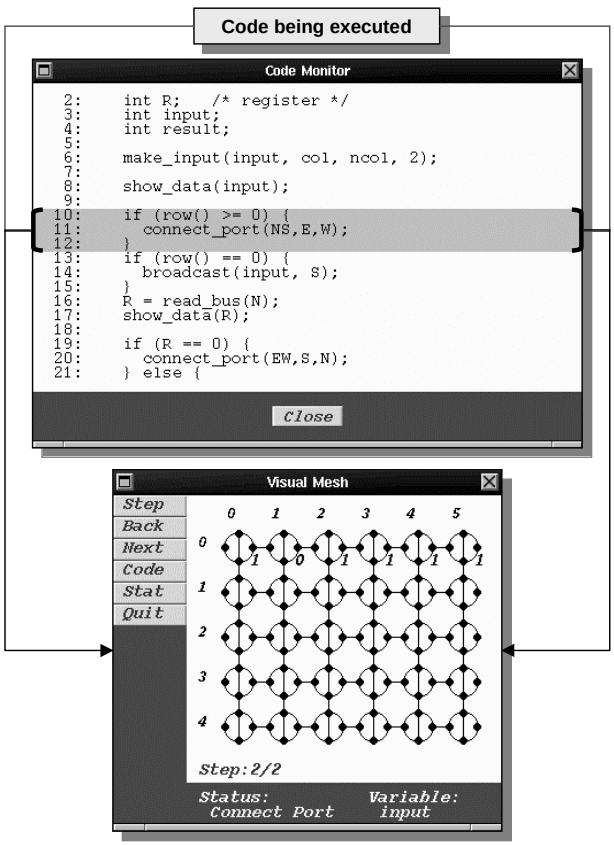

FIGURE 5 The VMesh code monitor window.

RECONFIGURABLE MESH

A reconfigurable mesh (RM, for short) of size $m \times n$ consists of mn, SIMD processors positioned in a two-dimensional grid with m rows and n columns. An RM processor (PE), is identified by a unique index (i, j), $(0 \leq i \leq m-1,\ 0 \leq j \leq n-1)$, therefore the processor with index (i, j) is denoted by PE(i, j). Each PE is connected to at most four neighbors through fixed links which are attached to the processor via an interface, called port. As shown in Fig. 1, each processor has four ports, denoted by North (N), South (S), East (E), and West (W) ports.

The possible combinations of external ports in a processor change according to the particular architectural model proposed. Several reconfigurable mesh models have been proposed in the literature [4,5,7,13,19]. These models vary according to their switch capabilities. The most popular two-dimensional array models are the horizontal–vertical reconfigurable mesh (HVRM Model), the linear reconfigurable mesh (LRM Model), and the general reconfigurable mesh (GRM Model). In the HVRM Model, switches may change the configuration of the network so that buses of different lengths are formed horizontally along rows and vertically along columns. Thus, a single bus cannot change directions by using both horizontal and vertical connections. In the LRM Model, a bus may consist of any connected path of edges, not only vertical nor only horizontal. In the GRM Model, the configuration of buses is composed by any partition of the network into edge-disjoint subgraphs. Figure 2 shows all possible port connections of the models mentioned above which are obtained by opening and closing the proper processor switches.

On the reconfigurable mesh, each connected component formed by links and internal connections constitutes a *bus*. When a bus configuration is established, processors that are attached to the same bus can communicate with each other by broadcasting data on the common bus.

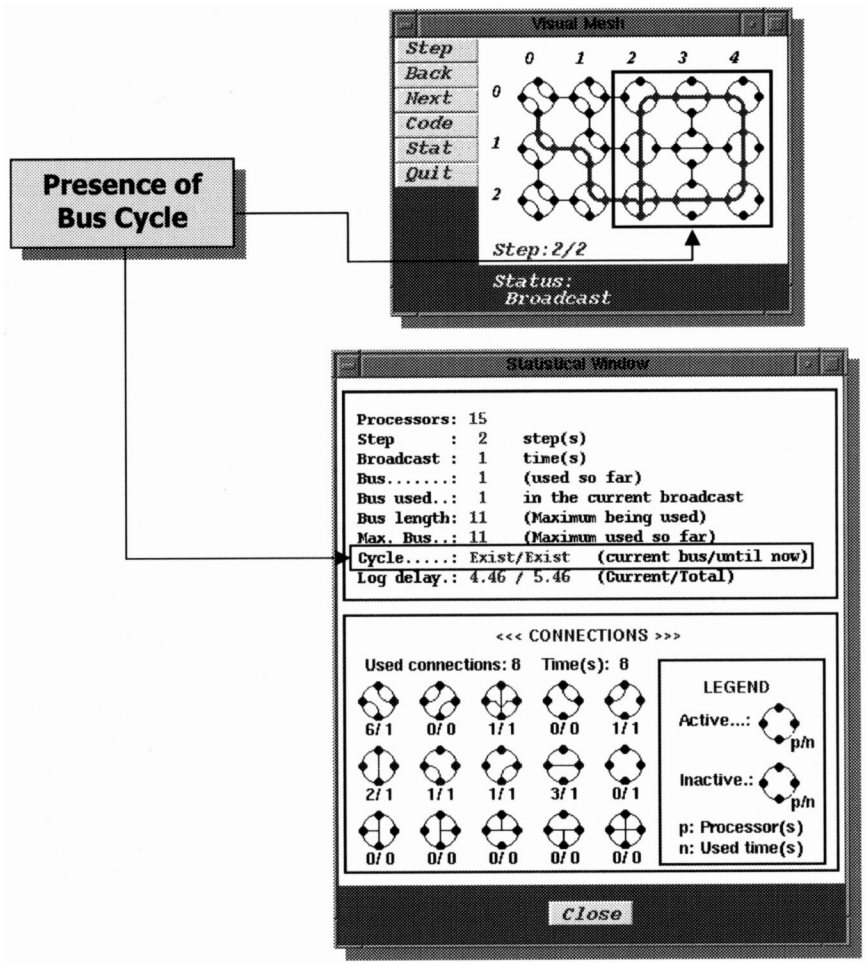

FIGURE 6 The presence of a bus cycle is shown in the statistical window.

Regarding the broadcast delay, Miller *et al.* [14] considered two computational models: the *Constant Time Propagation* and the *Logarithmic Time Propagation*. In the constant time propagation model, the information placed on a reconfigurable bus is assumed to propagate throughout the mesh in constant time. In the logarithmic time propagation model, the propagation time is assumed to be $O(\log n)$, where n is the maximum number of switches in a minimum path switch between two processors connected to the bus.

A single time step on a reconfigurable mesh is composed of the following four phases:

1. Phase 1, Change the configuration of a reconfigurable bus system by connecting or disconnecting its internal ports.
2. Phase 2, Send data to each port. When two or more processors attempt to send the data along the same bus, a collision occurs and the data being transmitted will be discarded by all processors connected to the bus.
3. Phase 3, Receive data from each port. Some or all processors connected to the bus read the data sent in phase 2.
4. Phase 4, A constant-time local computation is done by each processor

These four phases are executed synchronously by all processors, that is, no processor executes a phase until all processors have finished the previous one. Usually, the computation time of an algorithm is evaluated by counting the number of iterations of these four phases during the execution of the algorithm.

VMESH SYSTEM

This section covers the VMesh, its organization and features. The VMesh is written in C language using GNU C compiler and X-Window to implement its graphical capabilities. Thus, it can be easily ported to most Unix-based workstations.

The VMesh interface was designed aiming simplicity and usability. It allows the animation of an algorithm without requiring any visualization code, thus allowing the user to focus only on the implementation of the algorithm.

The VMesh system consists of three windows, the *main window*, the *code monitor*, and the *statistical window*. The

Functions	Description
connect_port()	Switches the PEs internal ports
reset_port()	Reset PEs internal ports
broadcast()	Broadcast data on the bus through the selected port
read_bus()	Read bus from the selected port
show_data()	Shows the specified data held by the selected PE
hide_data()	The data being shown is hidden
make_input()	Data is placed in the selected PEs
row()	Select the PEs of a particular row
col()	Select the PEs of a particular column

FIGURE 7 Predefined available functions.

visualization of the algorithms takes place in the main window, which is composed of a two-dimensional grid of processors, six buttons, and a step counter. At the bottom of the main window, the current task being executed and the variable being used are displayed. The task being executed can assume one of the following states: reading the bus, showing data, or broadcasting data. Figure 3 shows a snapshot of the main window and points out its features. The code monitor and statistical window can be accessed from the main window. The code monitor displays the algorithm code which is being executed, while the statistical window shows statistical information on the number of connections, buses, and broadcasts used by the algorithm.

Prefix Sum

```
void main () {
  int r_bus;
  int input;
  int result;

  make_input(input, 0, col, ncol, 2);
  show_data(input);
  if (row() >= 0) {
    connect_port(NS);
  }
  if (row() == 0) {
    broadcast(input, S);
  }
  input = read_bus(N);
  show_data(input);
  if (input == 0) {
    connect_port(EW);
  } else {
    connect_port(NE, SW);
  }
  hide_data();
  if (row() == 0 && col() == 0) {
    broadcast(1, W);
  }
  r_bus  = read_bus(E);
  show_data(r_bus);
  connect_port(NS);
  if(r_bus  == 1){
    broadcast(row(),N);
  }
  result = read_bus(N);
  show_data(result);
  reset_port();
  hide_data();
}
```

FIGURE 8 Prefix sum algorithm code.

The VMesh system can be configured to run either with a specified number of columns and rows, or with a specified number of processors. Each processor can be switched into any of the 15 internal connections depicted in Fig. 2. Therefore, our tool can simulate the HRVM, LRM, as well as the GRM model. On the VMesh, processors can communicate with each other by broadcasting data on the common bus. A bus is established by switching the internal connections into the desired pattern. Our tool shows the bus path with a thick red line and those processors which are broadcasting data (i.e. *speakers* of the bus) in blue. This feature aids the user to identify the processors that are broadcasting data, the buses being used and also the path in which data is traversing. The VMesh also allows the users to visualize the data being handled by each processor during the execution of an algorithm. The data held by each processor is shown in the PE display area as depicted in Fig. 3.

Whenever a data collision occurs, that is, when two or more processors attempt to broadcast data on the same bus, a warning message is displayed and the data being transmitted is discarded by all processors connected to that bus. The significance of such feature comes from the fact that when writing an algorithm it is difficult to identify the presence of data collision. Figure 4 shows the occurrence of a data collision when two processors on the third row (at 6th and 14th columns) attempt to broadcast data on the same bus.

The execution of an algorithm on the VMesh is done step by step. A step is defined as the execution of a code block. Thus, the number of steps is not related to the number of lines, but to the number of actions performed by the algorithm. An “if” statement, for instance, is executed in a single step. When the step button is pushed, the operations corresponding to that step are animated in the main window and the step counter is updated. After its execution, the statistical information regarding that step is updated in the statistical window as well as in the statistical output file. Our tool also provide ways to run an algorithm forward or backward, thus, allowing the visualization of previous steps. For this reason, two-step counters are maintained, one corresponding to the current step and another to the relative step. When the user go

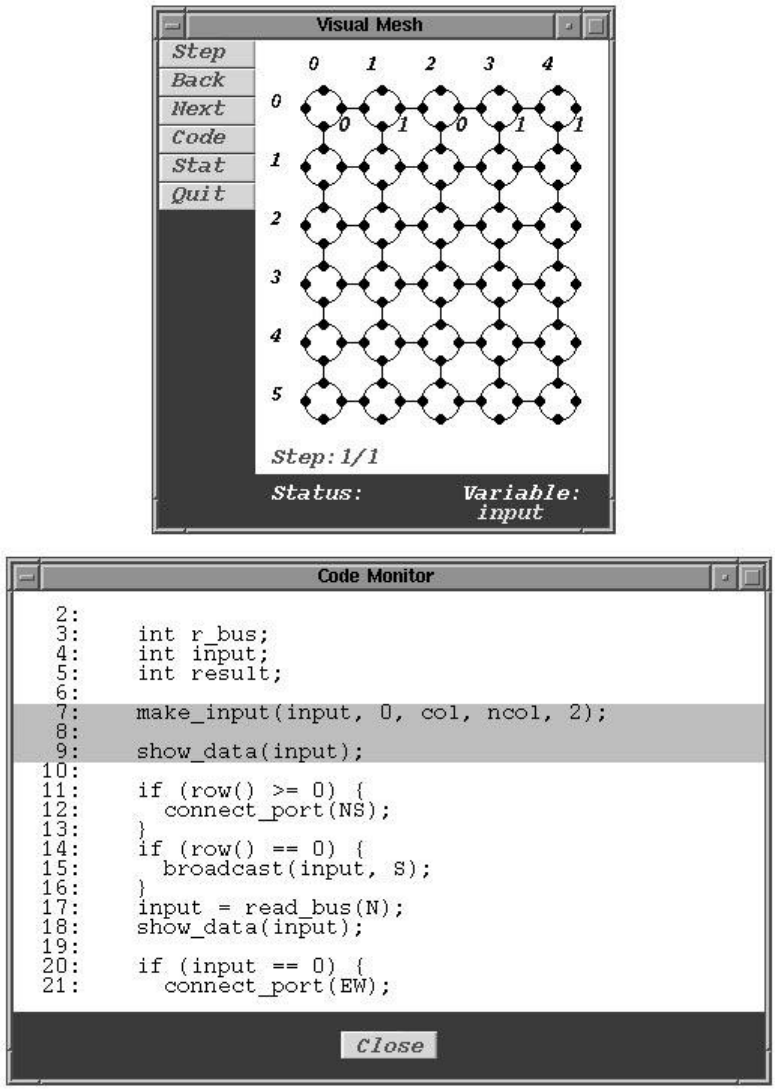

FIGURE 9 The binary values are placed in the first row of the mesh.

backwards in the execution, the relative step is updated while the current step counter remains unchanged.

The code monitor shows the algorithm source code and highlights the line(s) which are being visualized in the main window. Such feature facilitates the task of finding errors and assists the user in making improvements on the algorithm. The code monitor is shown in Fig. 5.

The statistical window gathers the information retrieved from the main window after executing a step. This statistical information can help users to compare the performance of related algorithms. Below, a description of the information displayed on the statistical window is given (see Fig. 6).

- *Number of broadcasts*: Shows how many times processors have driven data into a bus.
- *Step being executed*: Shows the step to which the statistical information refers.
- *Bus length*: The maximum number of switches a datum have traversed in the current broadcast, that is, the maximum number of switches from one end of the bus to the other.
- *Maximum bus length*: The maximum bus length used by the algorithm up to the current step.
- *Connections*: Shows how many times each connection has been used, which connections are active (i.e. being used in the current step), and the number of processors using each connection.
- *Log-time delay*: As mentioned in "Reconfigurable mesh" section, two different propagation delay models are commonly found in the literature, the constant time propagation model and the logarithmic time propagation model. The log-time delay is a logarithmic function of the bus length, which represents the time taken by a datum to travel from one end of the bus to the other in the logarithmic time

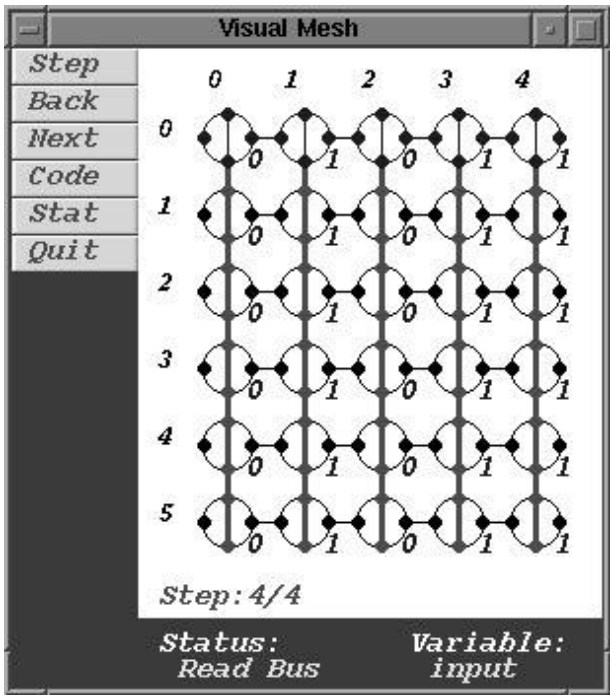

FIGURE 10 Processors read their North port to receive the broadcasted data.

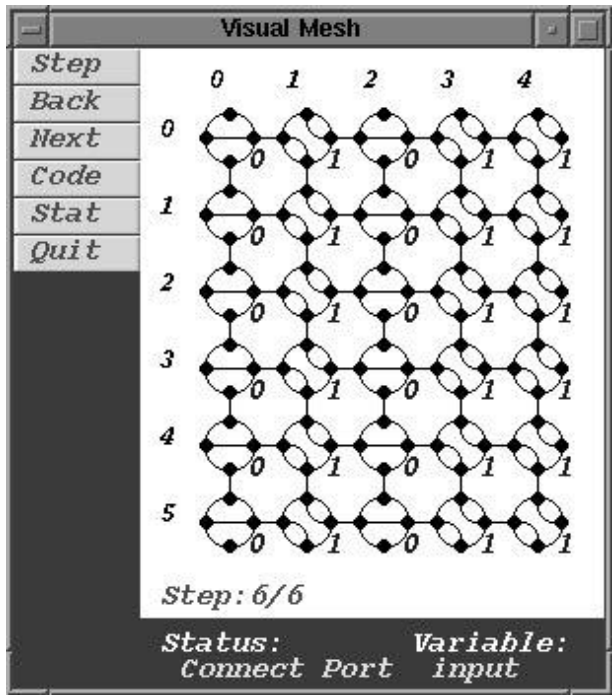

FIGURE 11 Connections (NE, SW) and (EW) are switched.

propagation model.

- *Bus cycles*: Our tool detects the presence of bus cycles, as depicted in Fig. 6. Such information about the bus shape can be useful in deciding the underlying bus model to be used. For example, the reconfigurable mesh that allows bus cycles can compute the *minimum spanning tree* problem in constant time, but no efficient algorithm is known on the reconfigurable mesh without them.

The visualization of an algorithm on the VMesh can easily achieved by executing the following steps: (i) Write an algorithm in a C-like language. To assist the user in this task, some predefined functions are available and can be used within the program. Figure 7 provides a list of the available functions along with a brief description. (ii) Translate the algorithm into a C-program. This is achieved by using a *translator*, which reads the user algorithm and generates the visualization code to be executed by the VMesh. (iii) Compile and link the translated program with the VMesh modules, thus producing the executable file.

As we have shown, the VMesh provides ways to control the execution of an algorithm and allows the visualization of its actions. Thus, our tool assists the user to better understand the behavior of parallel algorithms on the reconfigurable mesh. The statistical information supplied may aid users to analyze and compare the performance of similar algorithms. Therefore, we believe that our system can serve both as an instructional aid and as a research instrument.

ALGORITHMS

In order to demonstrate the VMesh system, we present the animation of the *prefix sum* algorithm, which is one of the most fundamental algorithms devised on the reconfigurable mesh. Subsequently, some other algorithms that have been implemented on the VMesh are shown.

Prefix Sum of a Binary Sequence

The prefix sum algorithm can be used to sum values, solve problems in the field of image processing, graph theory, and so forth. The prefix sums problem can be solved in constant time using $(n+1)\times n$ processors on the reconfigurable mesh [18]. The problem is stated as follows. For n given binary values $(a_0, a_1, \ldots, a_{n-1})$, each a_i $(0 \le i \le n-1)$ is given to the processors in the first row of the RM. After executing the prefix sums, each PE(i,j) $(0 \le i \le n,\ 0 \le j \le n-1)$ knows the value of $(a_0 + a_1 + \cdots + a_j)$.

In order to visualize the algorithm, we begin by defining the mesh size in which the algorithm will be executed. The VMesh used for this illustration has six rows and five columns of processors. Figure 8 shows the complete source code of the prefix sum algorithm used in this animation. Some local variables are defined in the first lines of the algorithm source code. These variables will be used for data storage throughout the algorithm execution.

The input binary values are randomly generated by the *make_input*() function. This function is also used to place the input values in the first row of the VMesh.

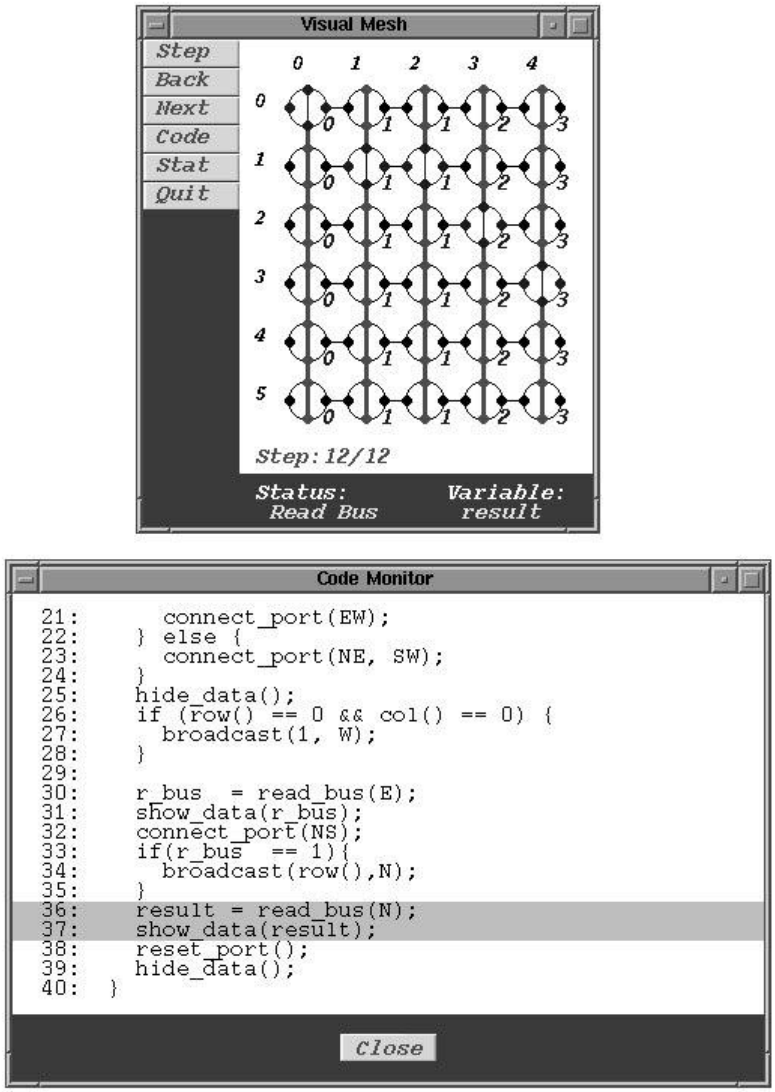

FIGURE 12 The prefix sums output.

The execution of the first step is shown in the main window and the lines being executed are shown highlighted in the code monitor as depicted in Fig. 9. The function *show_data*(*input*) in line 9 verifies which processor has received the *input* value and then shows it in the PE display area. The name of the variable being used is also shown at the bottom right of the main window. Thus, the user can easily identify the values being held by each processor during the execution of the algorithm.

Next, the input is distributed to the processors which are attached to the same column. This is done by having all processors connecting their North–South ports, thus creating a bus on each column of the mesh. The function *connect_port*(NS, E, W) is used to switch over to the proper connections. Then, each processor at the topmost row of the mesh broadcasts the input value on its South port by using the *broadcast_data*(*input*, S) function. All processors read their North port to receive the value sent in the previous step. These values are read by the *read_bus*(N) function and stored in a local variable. Figure 10 shows the step in which processors have just read their North port. Note that those processors which have broadcasted the data are depicted in blue and the buses are depicted in red.

Those processors which have received 0 (zero) in the previous step connect their East–West ports (*connect_port*(EW)), otherwise they connect their North-East and South-West ports (*connect_port*(NE, SW)) as shown in Fig. 11.

The processor located at the uppermost left corner of the mesh (i.e. PE(0,0)) broadcasts 1 on its West port. Those processors that succeed in receiving 1 on their East port learn that their row position correspond to the solution of the prefix sum problem. The prefix sum output is shown

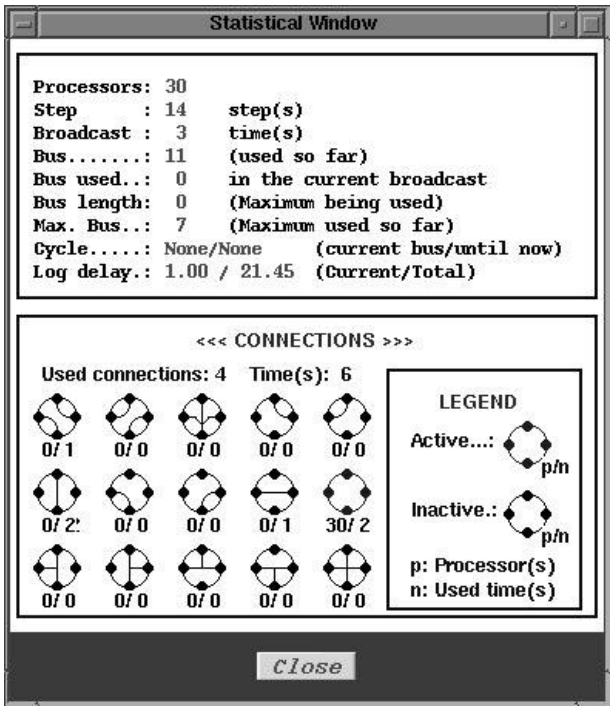

FIGURE 13 The prefix sums statistical information.

in Fig. 12. Recall that the statistical window gathers the information retrieved from the main window after the execution of a step. The statistical information of the prefix sum algorithm, after the execution of the final step, is shown in Fig. 13 and summarized in Fig. 14.

As we have shown through this animation of the prefix sums algorithm, the VMesh provides a comprehensive environment for developing, debugging, and visualizing algorithms.

Algorithms Implemented on the VMesh

In this section we give a brief description of some algorithms that have been implemented on the VMesh. Figure 14 gathers the statistical information obtained from these algorithms.

- The *prefix w-remainders* [8] is an algorithm on an $(w+1)\times 2n$ RM. Given n binary values $(a_0, a_1, \ldots, a_{n-1})$, it computes a sequence $(x'_0, x'_1, \ldots, x'_{n-1})$ of integers, where $x'_i = (a_0 + a_1 + \cdots + a_i) \bmod w$. Each a_i $(0 \le i \le n-1)$ is given to PE$(2i, 0)$. After executing the algorithm, each PE$(2i, 0)$ knows the value of x'_i.
- The *maximum number* [3] is an algorithm on an $n \times n$ RM. Given n integer numbers $(a_0, a_1, \ldots, a_{n-1})$, it computes the maximum number of the sequence.
- The *leftmost determining* [9] is an algorithm on an $(1 \times n)$ RM. Given n binary values $(a_0, a_1, \ldots, a_{n-1})$, it computes the leftmost element whose value is 1.
- The *ranking* [18] is an algorithm on an $n \times n$ RM. Given n integer numbers $(a_0, a_1, \ldots, a_{n-1})$ stored in the first column of the mesh, it computes the rank of an element stored in processor PE(0,0).
- The *sorting* [18] is an algorithm on an $n \times n^2$ RM. Given n numbers $(a_0, a_1, \ldots, a_{n-1})$ stored in the first column of the mesh, it sorts these numbers in non-decreasing order.
- The *minimum spanning tree* [19] is an algorithm on an $ne \times e$ RM. Given a weighted graph G with n vertices and e edges, it computes the minimum spanning tree.
- The *tree embedding* [13] is an algorithm on an $1 \times n$ RM. Given n values $(a_0, a_1, \ldots, a_{n-1})$ distributed one per processor, it computes the sum of the n values in a tree-like manner.
- The *bit counting* [13] is an algorithm on an $(n+1) \times n$ RM. Given a sequence of n bits $(a_0, a_1, \ldots, a_{n-1})$, it

Performance of the Agorithms

Algorithm	RM Size	Broadcasts	Buses	Bus Length	Log-time Delay	Cycle	Connections
Prefix Sums	5x6	3	11	7	21.45	None	4
Prefix W-Remainders	3x8	4	17	12	24.58	None	8
Maximum of *n* Items	5x5	3	15	4	20	None	3
Leftmost Determining	1x6	2	4	5	15.32	None	2
Ranking	5x5	3	11	6	22.58	None	4
Sorting	4x16	7	40	15	48.74	None	4
Minimum Spanning Tree	16x4	8	64	18	57.23	Exist	5
Tree Embedding	1x8	3	7	4	19	None	2
Bit Counting - XOR	5X5	2	6	6	14.58	None	4

FIGURE 14 Performance of the algorithms implemented on the VMesh.

counts the number of 1-bits of the input sequence.

CONCLUSION

VMesh is an interactive visualization tool that allows the animation of parallel algorithms on the reconfigurable mesh. Our tool provides a sophisticated graphical output which enables the visualization of the operations executed by an algorithm. The VMesh uses a high level input programming interface that facilitates the algorithm implementation. As shown in previous section, the implementation of an algorithm on the VMesh is simple and straightforward and hence we were able to implement a number of algorithms on it.

The statistical information supplied by the VMesh can assist the users to analyze and compare the performance or related algorithms. These statistical informations can also be used to choose the best algorithm to solve a given problem. Therefore, we believe that the VMesh can serve both as an instructional aid and as a research instrument.

References

[1] Alnuweiri, H.M. (1994) "Constant-time parallel algorithms for image labeling on a reconfigurable network of processors", *IEEE Transactions on Parallel and Distributed Systems* **March**, 320–326.

[2] Alnuweiri, H.M. (1994) "A fast reconfigurable network for graph connectivity and transitive closure", *Parallel Processing Letters* **4**, 105–115.

[3] Akl, S.G. (1997) Parallel Computation: Models and Methods (Prentice Hall, Englewood Cliffs, NJ).

[4] Ben-Asher, Y., Gordom, D. and Schuster, A. (1991) "The power of reconfiguration", *Journal of Parallel and Distributed Computing* **13**, 139–153.

[5] Ben-Asher, Y., Gordom, D. and Schuster, A. (1995) "Efficient self-simulation algorithms for reconfigurable arrays", *Journal of Parallel and Distributed Computing* **30**, 1–22.

[6] Lin, R., Olariu, S., Schwing, J. and Zhang, J. (1992) "Sorting in O(1) time on a reconfigurable mesh of size n × n", *Parallel Computing: From Theory to Sound Practice, Proceedings of EWPC'92*, 16–27.

[7] Nigam, M. and Sahni, S. (1994) "Sorting n numbers on $n \times n$ reconfigurable meshes with buses", *Journal of Parallel and Distributed Computing* **23**(1), 37–48.

[8] Nakano, K., Masuzawa, T. and Tokura, N. (1991) "A sub-logarithmic time sorting on a reconfigurable array", *IEICE Transactions* **E-74**(11), 3894–3901.

[9] Nakano, K. (1994) "An efficient algorithm for summing up binary values on a reconfigurable mesh", *IEICE Transactions, Fundamentals* **E77-A**(4), 652–657.

[10] Nakano, K. (1997) "Constant time algorithms on the reconfigurable meshes", *Journal of IPSJ* **38**(11), 1019–1025.

[11] Nakano, K. (1995) "A bibliography of published papers on dynamically reconfigurable architectures", *Parallel Processing Letters* **5**(1), 111–124.

[12] Miyashita, K., Shimizu, Y. and Hashimoto, R. (1998) "A visualization system for algorithms on PARBS", *Proceedings of International Conference on Computers and Information Technology (ICCIT)*, 215–219.

[13] Maresca, M. and Li, H. (1989) "Connection autonomy in SIMD computers: a VLSI implementation", *Journal of Parallel and Distributing Computing* **7**, 302–320.

[14] Miller, R., Prasanna Kumar, V., Reisis, D.I. and Stout, Q.F. (1988) "Meshes with reconfigurable buses", *MIT Conference on Advanced Research in VLSI*, 163–178.

[15] Miller, R. and Stout, Q.F. (1988) "Efficient parallel convex hull algorithms", *IEEE Transactions on Computers* **37**(12), 1605–1618.

[16] Schnorr, C.P. and Shamir, A. (1985) "An optimal algorithm for mesh connected computers", *Proceeding of the 18th ACM Symposium on Theory of Computing, Berkeley, CA* **May**, 255–263.

[17] Steckel, C., Middendorf, M., ElGindy, H. and Schmeck, H. (1998) "A simulator for the reconfigurable mesh architecture", 1998's IPPS/SPDP Workshops, Lecture Notes in Computer Science (Springer, Berlin) **Vol. 1388**, pp 99–104.

[18] Wang, B.F., Chen, G.H. and Lin, F.C. (1990) "Constant time sorting on a processor array with a reconfigurable bus system", *Information Processing Letters* **34**(4), 187–192.

[19] Wang, B.F. and Chen, G.H. (1990) "Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems", *IEEE Transactions on Parallel and Distributed Systems* **1**(4), 500–507.

Authors' Biographies

Jacir L. Bordim received the BE degree from Passo Fundo University, Brazil in 1994, and ME degree from Nagoya Institute of Technology, Japan in 2000. He is currently a PhD candidate at the Department of Electrical and Computer Engineering at Nagoya Institute of Technology, Japan. His research interests include parallel algorithms, software engineering and computer architecture.

Koji Nakano received the BE, ME, and PhD degrees from Osaka University, Japan, in 1987, 1989, and 1992, respectively. From 1992 to 1995, he was a research scientist at Advanced Research Laboratory, Hitachi Ltd. Since 1995, he has worked at Nagoya Institute of Technology, Japan. His research interests include parallel algorithms and architecture, computational complexity, and graph theory.

Tatsuya Hayashi received the BE degree from Waseda University, Japan, in 1960. He worked for Fujitsu from 1960 to 1993. Since 1993, he has been a professor of the Department of Electrical and Computer Engineering at Nagoya Institute of Technology, Japan. His research interests include software engineering, natural language processing, and computer architecture.