Serial Simulation of Reconfigurable Mesh, an Image Understanding Architecture*

M. Manzur Murshed†
Computer Sciences Lab, Research School of Information Sciences & Engg.
The Australian National University, Canberra ACT 0200, Australia
E-mail: murshed@cslab.anu.edu.au

Richard P. Brent
Oxford University Computing Laboratory, Oxford, OX1 3QD, England
E-mail: Richard.Brent@comlab.ox.ac.uk

Abstract

There has recently been an interest in Reconfigurable Mesh (RM) because of its simplicity and dynamic nature. Among many applications, RM can be viewed as a part of a powerful image understanding architecture for supporting real time image understanding applications. This paper defines a programming model for the general RM which is expressed by means of a programming language, called RPC (Reconfigurable Parallel C). The paper also presents a serial simulator, RMSIM (Reconfigurable Mesh SIMulator), which permits to study RPC programs written for a subnetwork of 3-D RM, known as mesh of meshes, where buses can be formed only in 2-D space. RMSIM is an easy-to-use simulator capable of simulating any RPC program in different axis-orientations within restricted regions. To enhance the debugging facilities, RMSIM is equipped with a snapshotter to generate LaTeX pictures of any planar segment of the simulated mesh at any step of program execution.

Keywords: *Reconfigurable mesh; Image processing; Simulation; Parallel programming; Programming language*

1 Introduction

It is well-known that interprocessor communications and simultaneous memory accesses often act as bottlenecks in present-day parallel machines. Bus systems have been introduced recently to a number of parallel machines to address this problem. Examples include the Bus Automaton [17], the Reconfigurable Mesh (RM) [14, 20], the Content Addressable Array Parallel Processor (CAAPP) [22], and the Polymorphic Torus [12]. A bus system is called *reconfigurable* if it can be dynamically changed according to either global or local information.

Among many applications, RM can be viewed as a part of a powerful image understanding architecture for supporting real time image understanding applications and research in knowledge-based computer vision [22]. Recently, constant time algorithms have been developed on RM for a number of image processing problems [4, 6, 7, 8, 14, 15].

Research on reconfigurable mesh has concentrated mainly on the development of computation models and on the implementation of experimental systems. The experimental systems YUPPIE [10, 11] and GCN [18] have mainly focused on the efficient implementation of the hardware supporting reconfigurability, and have not produced any programming model.

The lack of a programming model, in particular, makes the development of RM algorithms very difficult; the algorithms are usually formulated as sequences of steps involving the manipulation of switches that control reconfiguration and neither automatic validation nor simulation can be done. On the contrary, a programming model supporting a high level language and a simulator would allow for the automatic validation of the algorithms and for the automatic performance evaluation of the programs.

Maresca [12] has expressed his concern that the general RM is so flexible and powerful that it has turned out to be impossible to derive high level programming models preserving such flexibility and power. Maresca, therefore, has pruned the flexibility and power of the general RM in defining a new reconfigurable mesh architecture, Polymorphic Processor Arrays (PPA), for which a programming model has been proposed as a basis for the design of a parallel programming language, called PPC (Polymorphic

*A preliminary version of this work appeared in [16].
†Corresponding author.

Parallel C), and a compiler/simulator has been implemented [12].

PPC is a well defined programming model but it addresses only the issues concerning PPA. Recently, Ben-Asher *et al.* [1, 2, 3] have proposed a systematic approach in expressing RM algorithms where each step of an algorithm is divided into four substeps in sequence as discussed in Section 2. This has motivated us to define a programming model of the general RM and to write a serial simulator, RMSIM (Reconfigurable Mesh SIMulator) to support the model. The RM programming model is expressed by means of a programming language, called RPC (Reconfigurable Parallel C). RMSIM, written in ANSI C, can simulate a subnetwork of a 3-dimensional reconfigurable mesh known as *mesh of meshes* (Section 2). This simulator has an in-built interpreter to execute RPC programs. The interpreter is capable of executing a program in different axis-orientations within restricted regions. In defining RPC, we have concentrated on making the effort of transforming the algorithms into equivalent programs straightforward and easy. To aid in debugging, RMSIM is capable of generating LaTeX picture of any planar segment of the mesh, at any step, while executing a program.

This paper is organized as follows. In the next section we present the computational model of RM. Section 3 defines a programming model for RM. Issues of serial accessing of processing elements are addressed in Section 4. In Section 5 we briefly describe the debugging facilities of RMSIM. Section 6 concludes the paper.

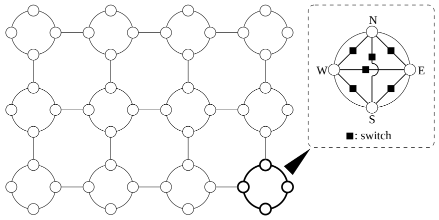

Figure 1: A 3×4 reconfigurable mesh.

2 RM Computational Model

The Reconfigurable MESH (RMESH) [14] and the Processor Array with a Reconfigurable Bus System (PARBS) [20] gained wide acceptence in the literature as RM computational models. Li and Stout [9] have demonstrated that PARBS is probably more powerful than RMESH by showing the performance difference of the computation of exclusive-or (XOR) on PARBS and RMESH models. We, therefore, consider PARBS to be the RM computational model in the rest of this paper.

The reconfigurable mesh is primarily a two-dimensional mesh of processors connected by reconfigurable buses. In this parallel architecture, a processor element (PE) is placed at the grid points as in the usual mesh connected computers. Each PE is connected to at most four neighbouring PEs through fixed bus segments connected to four I/O ports **E** & **W** along dimension x and **N** & **S** along dimension y. These fixed bus segments are the building blocks of larger bus components which are formed through switching, decided entirely on local data, of the internal connectors (see Figure 1) between the I/O ports of each PE. The possible fifteen interconnections of I/O ports through switching are shown in Figure 2. The connection patterns are represented as $\{p_1, p_2, \ldots\}$, where each of p_i represents a group of switches connected together such that $\bigcup_{\forall i} p_i = \{\mathbf{N}, \mathbf{E}, \mathbf{W}, \mathbf{S}\}$. For example, $\{\mathbf{NS}, \mathbf{E}, \mathbf{W}\}$ represents the connection pattern with ports **N** and **S** connected and ports **E** and **W** unconnected. Like all bus systems, the behaviour of RM relies on the assumption that the transmission time of a message along a bus is independent of the length of the bus [3].

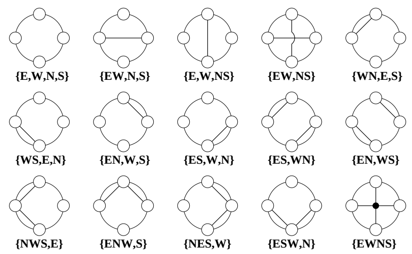

Figure 2: Possible fifteen internal connections between the four I/O ports of a PE.

A reconfigurable mesh operates in the single-instruction multiple-data (SIMD) mode. Besides the reconfigurable switches, each PE has a computing unit with a fixed number of local registers. A single time step of an RM is composed of the following four substeps:

BUS substep. Every PE switches the internal connectors between I/O ports by local decision.

WRITE substep. Along each bus, one or more PEs on the bus transmit a message of length bounded by the bandwidth of the fixed bus segments as well as the switches. These PEs are called the *speakers*. A bus is called to be in *ambiguous* state when there are more than one speakers. In the most common *exclusive write* model, the *ambiguous* state is considered to be an *error* state. The *common write* model [15, 4] allows multiple processors broadcasting simultaneously to the same bus so long as they all broadcast the same message. Otherwise, the *ambiguous* state is considered to be an *error* state. In both the models it is assumed that the *error* state is detectable by the processors and the bus carries arbitrary value. The most powerful *concurrent write* model [23] also allows multiple processors broadcasting simultaneously to the same bus and the bus always carries the *wired-or* of all the messages.

READ substep. Some or all the PEs connected to a bus read the message transmitted through.

COMPUTE substep. A constant-time local computation is done by each PE.

Reconfigurable meshes of higher dimension can be constructed in a similar way. A number of interesting algorithms [3, 5, 13, 19, 21] have appeared in the literature for multi-dimensional RMs. In most of the cases [3, 5, 13, 21], 3-D RM with two additional ports **U** and **D** along dimension z was considered because of a subnetwork of it, known as *mesh of meshes*, where buses can be formed only in 2-D space. In a mesh of meshes, not all the six ports, $\mathbf{N}, \mathbf{S}, \mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{D}$, can participate simultaneously in configuring dynamic interconnections. Only the ports on the same plane can be interconnected and all the possible 15 interconnections among each set of ports $\{\mathbf{E}, \mathbf{W}, \mathbf{N}, \mathbf{S}\}$ on XY-plane, $\{\mathbf{E}, \mathbf{W}, \mathbf{U}, \mathbf{D}\}$ on XZ-plane, and $\{\mathbf{N}, \mathbf{S}, \mathbf{U}, \mathbf{D}\}$ on YZ-plane are allowed. To overcome the implementation issues, optimal simulation of higher dimensional RMs by 2-D ones is addressed in [19].

3 RM Programming Model

RMC is an extension of ANCI C. In fact, RM-SIM is implemented as an ANCI C library preserving Object Oriented Programming (OOP) principles through data hiding. Like C, RPC is designed to be small. RPC programs, therefore, are assumed to be dependent extensively on library programs which are not the intregal part of the core of the language.

3.1 Initialisation and Basic Assumptions

Initialisation of the RM system to be simulated is carried out by calling the function `SetGlobalDim()` in the `main()`. `SetGlobalDim()` sets size `Nx` × `Ny` × `Nz` of the simulated RM, `regN` number of registers per PE, and bus write mode (`exclusive|common|concurrent`).

To locate a PE into the simulated mesh of meshes, 3-D Cartesian coordinates are used. Let $PE_{x,y,z}$ denote the PE at the coordinate (x, y, z). In every

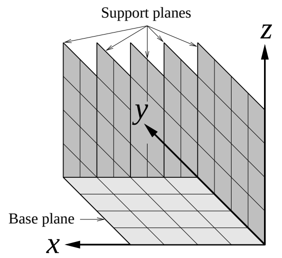

Figure 3: *XY_Z* axis-orientation.

RPC program, the XY-plane is assumed to be the base plane while the planes perpendicular to it act as the supporting planes. Let this axis-orientation be called *XY_Z* as shown in Figure 3. Possible five other axis-orientations *YX_Z*, *YZ_X*, *ZY_X*, *ZX_Y*, and *XZ_Y* play important role in program reusability as discussed in Section 3.5.

An RPC program also assumes that it will be executed in a restricted region of the simulated mesh defined by the constants `Sx`, `Sy`, `Sz`, `Ex`, `Ey`, `Ez`, which includes only the processing elements $PE_{x,y,z}$, $\min(\texttt{Sx},\texttt{Ex}) \leq x \leq \max(\texttt{Sx},\texttt{Ex})$, $\min(\texttt{Sy},\texttt{Ey}) \leq y \leq \max(\texttt{Sy},\texttt{Ey})$, $\min(\texttt{Sz},\texttt{Ez}) \leq z \leq \max(\texttt{Sz},\texttt{Ez})$. This assumption also helps in program reusability which is discussed in Section 3.5.

```
::Program_Name

S::  Start_of_Program Statement;

G::  beGin_of_Every_Step Statement;
B::  Bus Statement;
W::  Write Statement;
R::  Read Statement;
C::  Compute Statement;
F::  Finish_of_Every_Step Statement;
 .
 .
 .
E::  End_of_Program Statement;
```

Figure 4: RPC program layout in execution sequence.

3.2 Program Layout

The layout of an RPC program is given in Figure 4. This layout also shows the sequence of execution. The `S::`, `G::`, `C::`, `F::`, and `E::` statements are optional. Every `B::` statement starts a step and the `G::(F::)` statement is executed at the beginning(end) of each step. The statements can be *simple* or *complex*. As in C, a complex statment is recursively defined as a sequence of simple or complex statements enclosed in curly braces.

3.3 Data Structure

RPC assumes data width of the simulated mesh to be the `sizeof(double)`. The registers are considered as `double` variables and the bandwidth of the buses and ports are assumed to be the data width. It is programmer's responsibilty to interpret the content of a register otherwise, if other simple data types e.g. `int`, `char`, etc. are to be used. In a similar way complex data types can also be handled. Rigisters are indexed from 0 to $\texttt{regN}-1$. To enforce data hiding, registers are accessed only through the following two functions:

- `SetReg(reg, val)` – sets the content of register `reg` with `val`.
- `GetReg(reg, val)` – gets the content of register `reg`.

3.4 Data Communication

Data communication in RPC is supported by four primitives, available in the form of the following functions:

- `Bus(set_1, set_2, ...)` – selects the connection pattern {`set_1`, `set_2`, ...} as the interconnection of I/O ports. There can at most be six parameters and each parameter is a string of characters drawn from the alphabet $\{\mathbf{E},\mathbf{W},\mathbf{N},\mathbf{S},\mathbf{U},\mathbf{D}\}$ such that $\bigcup_{\forall i} \texttt{set\_}i = \{\mathbf{E},\mathbf{W},\mathbf{N},\mathbf{S},\mathbf{U},\mathbf{D}\}$ and $\forall i : \forall j \neq i : \texttt{set\_}i \cap \texttt{set\_}j = \{\}$. As RMSIM simulates only a mesh of meshes, it carries no meaning if a parameter contains characters from all the three sets $\{\mathbf{E},\mathbf{W}\}$, $\{\mathbf{N},\mathbf{S}\}$, and $\{\mathbf{U},\mathbf{D}\}$.
- `Write(prt, val)` – writes the `val` on the bus connected to the port `prt`.
- `Read(prt)` – reads the content of the bus connected to the port `prt`.
- `Error(prt)` – returns 1 if the bus connected to the port `prt` is in *error* state; returns 0 otherwise.

3.5 Program Reusability

Like many other programming languages, RPC is capable of using previously written programs into a new program through *subroutine calls*. Many RM algorithms appeared in the literature containing references to other published algorithms. This simplified the description of those algorithms significantly. The capability of reusing programs enables a programmer to convert those algorithms into programs in similar straightforward fashion.

Program reusability in RPC is supported by the function `Call(Prog_Name, Requested_Axis_orientation, sx, ex, sy, ey, sz, ez )`. Reusability of program through this function depends on the following key issues:

Call Stack – It is assumed that each PE is equipped with an internal stack to preserve the values of `Sx`, `Sy`, `Sz`, `Ex`, `Ey`, `Ez` when a subroutine call is made to another program.

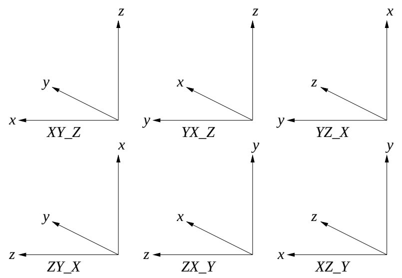

Figure 5: Possible six axis-orientations.

Axis-orientation Mapping – It has already been mentioned in Section 3.1 that while a program is written, RPC assumes *XY_Z* to be the axis-orientation. But during a subroutine call a program can be executed in any of the possible six axis-orientations as shown in Figure 5. RPC allows nesting of subroutine calls and the axis-orientation is selected accordingly. Suppose the current axis-orientation and the `Requested_Axis_orientation` be *YZ_X* and *XZ_Y*, RPC will then select *YX_Z* as the resultant axis-orientation.

The example RPC program (Figure 6) of computing the ranks of N distinct numbers on an $N \times N \times N$ mesh of meshes exemplifies the axis-orientation mapping. Let the number n_i be stored in $PE_{i,0,0}$, $0 \leq i < N$. Now a row broadcast after a column broadcast is done to distribute the numbers in the XY-plane so that $PE_{i,j,0}$, $0 \leq i,j < N$, receives the pair (n_i, n_j) and then produces 1, if $n_i > n_j$, or 0, otherwise. Ranks are now computed by simply `Call`ing the program `PrefixSum` in *YZ_X* axis-orientation in every YZ-planes to add the comparison values along each column.

Region Mapping – Enhancement in the power of program reusability through axis-orientation mapping cannot be realised completely if no way is allowed to `Call` a program in a restricted area of the mesh of meshes e.g. in the previous example of computing ranks each `Call` of the program `PrefixSum` uses a specific YZ-plane rather than using the entire mesh of meshes. RPC assumes that a program will be executed in a restricted region of the simulated mesh defined by the constants `Sx`, `Sy`, `Sz`, `Ex`, `Ey`, `Ez` as mentioned in Section 3.1. Hence, defining a restricted region in program `Call`ing is as simple as assigning `Sx` = `sx`, `Ex` = `ex`, and so on.

```
::PrefixSum

B:: Bus("NS", "E", "W", "U", "D", "");
W:: if(y == Sy && z == Sz)
      Write(N, GetReg(0));
R:: Read(N, 0);

B:: if(GetReg(0) == 0)
      Bus("EW", "N", "S", "U", "D", "");
    else Bus("WN", "ES", "U", "D", "", "");
W:: if(x == Sx && y == Sy && z == Sz)
      if(GetReg(0) == 0)
        Write(E, 99);
      else Write(N, 99);
R:: Read(E, 1);

B:: if(z == Sz)
      Bus("NS", "E", "W", "U", "D", "");
W:: if(z == Sz && GetReg(1) == 99)
      Write(S, y-Sy);
R:: if(y == Sy && z == Sz)
      Read(S, 1);

::Rank

B:: Bus("NS", "E", "W", "U", "D", "");
W:: if(y == Sy && z == Sz)
      Write(N, GetReg(0));
R:: Read(N, 0);

B:: Bus("EW", "N", "S", "U", "D", "");
W:: if(x == y && z == Sz)
      Write(E, GetReg(0));
R:: Read(E, 1);
C:: { SetReg(2, GetReg(0));
      if(GetReg(0) > GetReg(1))
        SetReg(0, 1);
      else SetReg(0, 0); }

B:: ;
W:: ;
R:: ;
C:: if(y == Sy && z == Sz) {
        Call( PrefixSum, YZ_X, Ey, Sy, Sz,
              Ez, x, x );
        SetReg(0, GetReg(2));
    }
```

Figure 6: An RPC program to compute the ranks of some distinct numbers [21].

4 Serial Accessing of PEs in RMSIM

RPC assumes that a program will be executed in all the processors in parallel. But being a serial simulator, RMSIM assumes the following order in accessing PEs sequentially:

Loop along 3rd axis
    Loop along 2nd axis
        Loop along 1st axis

Actual axes to be considered in place of 1st, 2nd, and 3rd axes are resolved according to the current axis-orientation. In *ZX_Y* axis-orientation 1st, 2nd, and 3rd axes are taken as z, x, and y axes. The step of each loop is either 1 or -1 depending on `Sx`'s and `Ex`'s. For example, the region defined in Figure 7 will generate the following loop structure if the current axis-orientation is *XY_Z*:

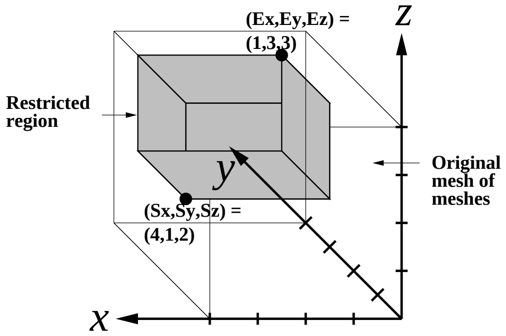

Figure 7: A restricted region in a $5 \times 5 \times 5$ mesh of meshes.

for $z = 2$ to 3 step 1
    for $y = 1$ to 3 step 1
        for $x = 4$ to 1 step -1

5 Debugging in RMSIM

RMSIM generates run time error codes while interpreting a program if problem occurs. Besides this standard technique, RMSIM is also equipped with a snapshotter which can be used to generate LaTeX picture of the bus configurations, along any plane of the simulated mesh of meshes, at any step of program execution. The generated pictures are scalable and can show the content of the registers of each PE.

6 Conclusion

This paper defines a programming model for the general RM which is expressed by means of a programming language, called RPC (Reconfigurable Parallel C). The paper also presents a serial simulator, RMSIM (Reconfigurable Mesh SIMulator), which can simulate a subnetwork of 3-D RM known as mesh of meshes. Besides simulating, RMSIM provides a program interpreter which can execute RPC programs in any possible axis-orientation within an enclosed region. RMSIM can generate snapshots of any planar segment of the simulated mesh in LaTeX picture format.

Technical Reference

This software is available by anonymous ftp: `cslab.anu.edu.au` in the directory `/pub/Manzur/RMSIM` (free distribution).

References

[1] Yosi Ben Asher, Dan Gordon, and Assaf Schuster. Efficient self-simulation algorithms for reconfigurable arrays. *Journal of Parallel and Distributed Computing*, 30:1–22, 1995.

[2] Y. Ben-Asher, K. J. Lange, D. Peleg, and A. Schuster. The complexity of reconfiguring network models. *Information and Computation*, 121:41–58, 1995.

[3] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster. The power of reconfiguration. *Journal of Parallel and Distributed Computing*, 13:139–153, 1991.

[4] Prabir Bhattacharya. Connected component labeling for binary images on a reconfigurable mesh architecture. *Journal of Systems Architecture*, 42:309–313, 1996.

[5] Yen-Cheng Chen and Wen-Tsuen Chen. Constant time sorting on reconfigurable meshes. *IEEE Transactions on Computers*, 43:749–751, 1994.

[6] Kuo-Liang Chung and Horn-Yi Lin. Hough transform on reconfigurable meshes. *Computer Vision and Image Understanding*, 61:278–284, 1995.

[7] Ju-Wook Jang, Heonchul Park, and Viktor K. Prasanna. A fast algorithm for computing a histogram on reconfigurable mesh. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:97–106, 1995.

[8] Jing-Fu Jeno and Sartaj Sahni. Reconfigurable mesh algorithms for the hough transform. *Journal of Parallel and Distributed Computing*, 20:69–77, 1994.

[9] H. Li and Q. F. Stout. Reconfigurable SIMD massively parallel computers. *Proc. IEEE*, 79:1345–1351, 1991.

[10] Hungwen Li and Massimo Maresca. Polymorphic-torus network. *IEEE Transactions on Computers*, 38:1345–1351, 1989.

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

[12] Massimo Maresca. Polymorphic processor arrays. *IEEE Transactions on Parallel and Distributed Systems*, 4:490–506, 1993.

[13] Mark S. Merry and Johnnie Baker. A constant time sorting algorithm for a three dimensional reconfigurable mesh and reconfigurable network. *Parallel Processing Letters*, 5:401–412, 1995.

[14] Russ Miller, V. K. Prasanna Kumar, Dionisios I. Reisis, and Quentin F. Stout. Data movement operations and applications on reconfigurable VLSI arrays. In *Proc. International Conference on Parallel Processing*, pages 205–208, 1988.

[15] Russ Miller, V. K. Prasanna-Kumar, Dionisios I. Reisis, and Quentin F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42:678–692, 1993.

[16] M. Manzur Murshed and Richard P. Brent. RM-SIM: a serial simulator for reconfigurable mesh parallel computers. Technical Report TR-CS-97-06, Joint Computer Science Tech. Report Series, The Australian National University, April 1997.

[17] J. Rothstein. Bus automata, brains, and mental models. *IEEE Trans. Syst. Man Cybern*, 18:522–531, 1988.

[18] D. B. Shu and J. G. Nash. The gated interconnection network for dynamic programming. In S. K. Tewsburg et al., editors, *Concurrent Computations*, pages 645–658. Plenum, New York, 1988.

[19] Ramachandran Vaidyanathan and Jerry L. Trahan. Optimal simulation of multidimensional reconfigurable meshes by two-dimensional reconfigurable meshes. *Information Processing Letters*, 47:267–273, 1993.

[20] Biing-Feng Wang and Gen-Huey Chen. 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*, 1:500–507, 1990.

[21] Biing-Feng Wang, Gen-Huey Chen, and Ferng-Ching Lin. Constant time sorting on a processor array with a reconfigurable bus system. *Information Processing Letters*, 34:187–192, 1990.

[22] C. C. Weems et al. The image understanding architecture. *Internat. J. of Comput. Vision*, 2:251–282, 1989.

[23] Charles C. Weems et al. The image understanding architecture. *International Journal of Computer Vision*, 2:251–282, 1989.