Real-time Design Constraints in Implementing Active Vibration Control Algorithms

Mohammed Alamgir Hossain
Department of Computing, School of Informatics, University of Bradford, Bradford BD7 1DP, UK

Mohammad Osman Tokhi*
Department of Automatic Control and Systems Engineering, The University of Sheffield, Sheffield S1 3JD, UK

Abstract: Although computer architectures incorporate fast processing hardware resources, high performance real-time implementation of a complex control algorithm requires an efficient design and software coding of the algorithm so as to exploit special features of the hardware and avoid associated architecture shortcomings. This paper presents an investigation into the analysis and design mechanisms that will lead to reduction in the execution time in implementing real-time control algorithms. The proposed mechanisms are exemplified by means of one algorithm, which demonstrates their applicability to real-time applications. An active vibration control (AVC) algorithm for a flexible beam system simulated using the finite difference (FD) method is considered to demonstrate the effectiveness of the proposed methods. A comparative performance evaluation of the proposed design mechanisms is presented and discussed through a set of experiments.

Keywords: Algorithm analysis and design, active vibration control (AVC), flexible beam system, real-time control, memory management.

1 Introduction

The analysis and design of algorithms is currently a subject of widespread interest among researchers and scientists. Accordingly a new scientific subject emerged during the 1960s and quickly become established as one of the most active fields of study and an important topic in computer and systems engineering. The reason for this sudden interest in the study of algorithms is not difficult to trace; it was the fast and successful development of digital computers and their uses in many different areas of human activity, which led to the construction of a great variety of computer algorithms. In many cases, analysis of algorithms has led to the revaeling of completely new algorithms that are faster than all available algorithms. In general, however, the goal of algorithmic analysis is to obtain sufficient understanding of the relative merits of complicated algorithms so as to provide useful information to someone undertaking an actual computation.

In practice, more than one algorithm exists to solve a specific problem. Depending on the formulation, an algorithm can be evaluated numerically in different ways. As computer arithmetic is of finite accuracy, different results can evolve, depending on the algorithm used and the way it is evaluated. On the other hand, the same computing domain could offer different performances due to variation in the algorithm design and in turn source code implementation. The choice of the best algorithm for a given problem and for a specific computer is a difficult task and depends on many factors, for instance, data and control dependencies of the algorithm, regularity and granularity of the algorithm and architectural features of the computer domain[1,2].

The ideal performance of a computer system demands a perfect match between machine capability and program behaviour. Program performance is turnaround time, which includes disk and memory accesses, input and output activities, compilation time, operating system overhead, and central processing unit (CPU) time. In order to shorten the turnaround time, one can reduce all these time factors by minimising runtime memory management, efficient partitioning and mapping of the program, and selecting an efficient compiler for specific computational demands to enhance the performance. Compilers have a significant impact on the performance of a system. This means that some high-level languages have advantages in certain computational domains, and some have advantages in others. The compiler itself is critical to the performance of the system as the mechanism and efficiency of taking a high-level description of the application and transforming it into a hardware dependent implementation,

Manuscript received November 8, 2005; revised January 12, 2006.

*Corresponding author. E-mail address: O.Tokhi@sheffield.ac.uk
differs from compiler to compiler\textsuperscript{[3,4]}.

Performance is also related to program optimisation facility of the compiler, which may be machine dependent. The goal of program optimisation is, in general, to maximise the speed of code execution. This involves several factors, such as minimisation of code length and memory accesses, exploitation of parallelism, the elimination of dead code, in-line function expansion, loop unrolling and the maximum utilisation of registers. Optimisation techniques include vectorization using pipelined hardware and parallelisation using simultaneously multiprocessors\textsuperscript{[5]}.

The performance demands of modern real-time signal processing and control applications have motivated the development of advanced special-purpose and general-purpose hardware architectures. However, developments within the software domain have not occurred at the same pace and/or level as in the hardware domain. Therefore, although advanced computing hardware with significant levels of capability is available in the market, these capabilities are not fully utilised and exploited at the software level. Efficient software coding is essential in order to exploit special hardware features and avoid the associated shortcomings of an architecture. There has been a substantial amount of effort devoted to this area of research over the last decade\textsuperscript{[6-8]}.

It is essential for the enhanced performance of a computing domain that a characteristic matching between the computing requirements of an algorithm and computing capabilities of that computing domain is made. Moreover, the source code and corresponding memory management facility of a computing domain play an important role in its overall performance when implementing an algorithm. This further includes the memory access time required during the execution of a program code. Some special-purpose digital signal processing (DSP) devices, for example the Texas Instruments TMS320 devices, incorporate instructions, at the assembly language level, that allow the executing commonly occurring operations in digital filtering applications together, such as multiply, add and shift. Such facilities attempt to minimise the memory access time and hence enhance the performance of the processor\textsuperscript{[9,10]}.

This paper addresses the issue of algorithm analysis, design and software coding for real-time active control systems. A number of design methodologies are proposed for the real-time implementation of an AVC algorithm. The proposed methodologies are exemplified and demonstrated with FD simulation algorithm of a flexible beam system within the framework of AVC. Finally, the comparative performance of the proposed design mechanisms are presented and discussed through a set of experimental investigations.

2 AVC algorithm

Consider a cantilever beam system with a force \( U(x, t) \) applied at a distance \( x \) from its fixed (clamped) end at time \( t \). This will result in a deflection \( y(x, t) \) of the beam from its stationary position at the point at which the force has been applied. In this manner, the governing dynamic equation of the beam is given by

\[
\mu^2 \frac{\partial^4 y(x, t)}{\partial x^4} + \frac{\partial^2 y(x, t)}{\partial t^2} = \frac{1}{m} U(x, t)
\]

where \( \mu \) is a beam constant and \( m \) is the mass of the beam. Discretising the beam in time and length using central FD methods, a discrete approximation to (1) can be obtained as\textsuperscript{[11,12]}:

\[
Y_{k+1} = -Y_{k-1} - \lambda^2 SY_k + \frac{(\Delta t)^2}{m} U(x, t)
\]

where \( \lambda^2 = [(\Delta t)/(\Delta x)^4] \mu^2 \) with \( \Delta t \) and \( \Delta x \) representing the step sizes in time and along the beam respectively,

\[
Y_{k+1} = \begin{bmatrix} y_{1, k+1} \\ y_{2, k+1} \\ \vdots \\ y_{n, k+1} \end{bmatrix}, \quad Y_j = \begin{bmatrix} y_{1, k} \\ y_{2, k} \\ \vdots \\ y_{n, k} \end{bmatrix}, \quad Y_{j-1} = \begin{bmatrix} y_{1, k-1} \\ y_{2, k-1} \\ \vdots \\ y_{n, k-1} \end{bmatrix}
\]

and \( S \) is a penta-diagonal matrix, given (as \( n = 20 \), for example) as:

\[
S = \begin{bmatrix} a & -4 & 1 & 0 & 0 & \cdots & 0 \\ -4 & b & -4 & 1 & 0 & \cdots & 0 \\ 1 & -4 & b & -4 & 1 & \cdots & 0 \\ 0 & 1 & -4 & b & -4 & 1 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & 1 & -4 & b & -4 \\ \cdots & \cdots & \cdots & 0 & 1 & -4 & c & -2 \\ \cdots & \cdots & \cdots & 0 & 0 & 2 & -4 & d \end{bmatrix}
\]

where \( a = 7 - \frac{7}{\lambda^2} \), \( b = 6 - \frac{2}{\lambda^2} \), \( c = 5 - \frac{2}{\lambda^2} \) and \( d = 2 - \frac{2}{\lambda^2} \).

Equation (2) is the required relation for the simulation algorithm, characterising the behaviour of the cantilever beam system, which can be implemented on a digital computer easily. For the algorithm to be stable it is required that the iterative scheme described in (2), for each grid point, converges to a solution. It has been shown that a necessary and sufficient condition for stability satisfying this convergence requirement is given by \( 0 < \lambda^2 \leq 0.25 \).\textsuperscript{[12]}

A schematic diagram of an AVC structure is shown in Fig.1. A detection sensor detects unwanted (primary) disturbance. This is processed by a controller to generate a cancelling (secondary, control) signal so as to achieve cancellation at the observation point. The
objective in Fig. 1 is to achieve total (optimum) vibration suppression at the observation point. Synthesising the controller on the basis of this objective yields

$$C = \left[1 - \frac{Q_1}{Q_0}\right]^{-1}$$

(3)

where $Q_0$ and $Q_1$ represent the equivalent transfer functions of the system (with input at the detector and output at the observer) when the secondary source is off and on respectively.

Fig. 1 Active vibration control structure

To investigate the nature and real-time processing requirements of the AVC algorithm, it is divided into two parts, namely control and identification. The control part is tightly coupled with the simulation algorithm, and both are described in an integral manner as the control algorithm. The simulation algorithm will also be explored as a distinct algorithm. Both of these algorithms are predominately matrix based. The identification algorithm consists of the parameter estimation of the models $Q_0$ and $Q_1$, and the calculation of required controller parameters according to equation (3). However, the nature of the identification algorithm is completely different when compared with the simulation and control algorithms\textsuperscript{10}. Therefore, for reasons of consistency, only the simulation and control algorithms are considered in this investigation.

3 Algorithm design

3.1 Beam simulation algorithm

The beam simulation algorithm is of regular iterative type. In implementing this algorithm on a sequential vector processor a performance better than with any other processor can be expected. The algorithm processes floating-point data, which is computed within a small iterative loop. Accordingly, performance is further enhanced if the processor has internal/external data cache and instruction cache, and built-in maths co-processor etc.

The simulation algorithm in (2) can be expressed, for example, as computing the deflection of segments 8 and 16, as in Fig. 2, assuming no external force is applied at these points.

It follows from the above that computation of deflection of a segment at time step $t$ can be described as in Fig. 3. It should also be noted that computation of the deflection of a particular segment is dependent on the deflection of six other segments. These heavy dependencies could be major causes of performance degradation in real-time sequential computing, due to memory access time. On the other hand, these dependencies might cause significant performance degradation in real-time parallel computing due to interprocessor communication overheads.

To explore this issue, a number of design mechanisms for the beam simulation algorithm were developed in a real-time performance context. Seven designs for a simulation algorithm were developed and tested in a set of experiments\textsuperscript{5,14}. These designs are considered here for further investigation in an AVC framework. Algorithms with different designs are described in Figs. 3~13.

3.1.1 Beam algorithm-1: Shifting of data array

Algorithm-1 was adopted from a previously reported work\textsuperscript{5}. The algorithm is shown in Fig. 4. It can be seen that complex matrix calculations are performed within an array of three elements, each representing information about the beam position at different instants of time. Following these calculations, the memory pointer is shifted to the previous pointer time step before the next iteration. This technique of shifting the pointer does not contribute to the calculation efforts and is therefore a program overhead. Other algorithms were further deployed to address this issue.

3.1.2 Beam algorithm-2: Array rotation

Algorithm-2 incorporates design suggestions made

```
y[8][8] ← y[8][6]+lumsq*(y[6][7]-4*y[7][7]+b*y[8][7]-4*y[9][7]+y[10][7]);
y[16][16] ← y[16][14]+lumsq*(y[14][15]-4*y[15][15]+b*y[16][15]-4*y[17][15]+y[18][15]);
```

Fig. 2 Calculation of deflection of segments 8 and 16 (where, lumsq is lambda square)
Fig. 3 Data dependencies for computation of deflection of each segment

Loop {
  //Step 1
  y0[0] = -y0[0] - lamsq * (a*y0[1] - 4*y1[1] + y2[1]);
  y1[0] = -y1[0] - lamsq * (-4*y0[1] + b*y1[1] - 4*y2[1] + y3[1]);
  ...
  y18[0] = -y18[0] - lamsq * (y16[1] - 4*y17[1] + c*y18[1] - 2*y19[1]);
  y19[0] = -y19[0] - lamsq * (2*y17[1] - 4*y18[1] + d*y19[1]);
  //Step 2: Shifting memory locations
  y0[0] = y0[1]; y0[1] = y0[2]; y1[0] = y1[1]; y1[1] = y1[2];
  ...
  y18[0] = y18[1]; y18[1] = y18[2]; y19[0] = y19[1]; y19[1] = y19[2];
}

Fig. 4 Design of Algorithm-1

Fig. 5 Design of Algorithm-2
Loop \{ 
for (i=0; j<1000; j++) \{
\}
\} // Shifting memory locations

Fig. 6 Design of Algorithm-3

Loop \{ 
\}
\} // Shifting memory locations

Fig. 7 Design of Algorithm-4

Loop \{ 
\} // Step 1
\} // Step 2
\} // Step 3

Fig. 8 Design of Algorithm-5
Loop {
  // Step 1
  y0[0] = y0[0] - lamSq * (a*y0[1] - 4*y1[1] + y2[1]);
  y1[0] = y1[0] - lamSq * (-4*y0[1] + b*y1[1] - 4*y2[1] + y3[1]);
  ...
  y18[0] = y18[0] - lamSq * (y16[1] - 4*y17[1] + c*y18[1] - 2*y19[1]);
  y19[0] = y19[0] - lamSq * (2*y17[1] - 4*y18[1] + d*y19[1]);
  // Step 2
  y0[1] = y0[1] - lamSq * (a*y0[0] - 4*y1[0] + y2[0]);
  y1[1] = y1[1] - lamSq * (-4*y0[0] + b*y1[0] - 4*y2[0] + y3[0]);
  ...
  y18[1] = y18[1] - lamSq * (y16[0] - 4*y17[0] + c*y18[0] - 2*y19[0]);
  y19[1] = y19[1] - lamSq * (2*y17[0] - 4*y18[0] + d*y19[0]);
}

Fig. 9 Design of Algorithm-6

Fig. 10 Re-calculting in 2 time steps

Fig. 11 Block representation of Algorithm-2

Fig. 12 Block representation of Algorithm-6

Fig. 13 Design of Algorithm-7
A listing of Algorithm-2 is given in Fig. 5. In this case, each loop calculates three sets of data. Instead of shifting the data of the memory pointer (that contains results) at the end of each loop, the most current data is directly recalculated and written into the memory pointer that contains the older set of data. Therefore, re-ordering of the array in Algorithm-1 is replaced by recalculcation. The main objective of this design effort is to achieve better performance by reducing dynamic memory allocation and in turn memory pointer shift operation. Thus, instead of using a single code block and data-shifting portion, to calculate the deflection, as in Algorithm-1, three code blocks, are used with the modified approach in Algorithm-2. It is worth noting that in Algorithm-2, the overhead of Algorithm 1 due to memory pointer shift operation is eliminated and every line of code is directed towards the simulation effort.

3.1.3 Beam algorithm-3: Large array and less frequent shifting

In Algorithm-1 shifting memory pointers was required at each iteration. Algorithm-3 was developed as an attempt to reduce the number of memory pointer shifting instructions and thereby decrease program overhead. An array of 1000 elements was considered for each beam segment. This array size was chosen rather arbitrarily, but was small enough to allow easy allocation of these monolithic memory blocks within typical hardware boundaries. Fig. 6 shows how the array was utilised in Algorithm-3. Shifting occurs at the end of every thousandth iteration, rendering the overhead produced at this stage negligible. However, array positions are indirectly referenced via variable, accessed at run-time, which in turn lead to an overhead. Of far greater concern to program performance is the fact that large data structures need to be dealt with. Therefore, the internal data cache struggles to handle large amount of data.

3.1.4 Beam algorithm-4: Nested loops and shifting

Algorithm-4 incorporates merely a minor modification of Algorithm-1, as shown in Fig. 7. The aim in this algorithm is to contain the number of instructions inside the main loop, and therefore, reduce the instruction size of the program. This was accomplished by nesting secondary loops inside the main iterations. Complex substitutions need to be carried out to determine which matrix elements need to be referred to when performing ongoing calculations. A moderate amount of overhead resulting from these necessary substitutions was anticipated. The benefits of this algorithm include quicker compilation, greater flexibility in respect of the number of segments (which possibly changes at run-time) and a fixed number of program instructions in the main loop as segment sizes increase. The likelihood of cache misses in the instruction cache was significantly reduced.

3.1.5 Beam algorithm-5: Nested loops and array rotation

Fig. 8 shows a listing of Algorithm-5, in which the new methods of Algorithm-4 were applied with the concepts of Algorithm-2. Three distinct calculation runs are performed during each iteration, but instead of listing the instructions for each segment separately, nested loops are used to limit the number of instructions (source code lines) in the main program loop. The benefits of employing this technique are identical with those listed in the description of Algorithm-4. However, it possesses the same disadvantage of overhead produced by the complex substitutions required.

3.1.6 Beam algorithm-6: Two-element array rotation

Algorithm-6 is shown in Fig. 9. This makes use of the fact that access to the oldest time segment is only necessary during re-calculation of the same longitudinal beam segment. Hence, it can directly overwritten with a new value, as shown in Fig. 10.

Figs. 11 and 12 show simplified flow diagrams of Algorithm-2 and Algorithm-6, respectively. The conventional re-calculation algorithm in Fig. 4 requires three memory segments in the time domain. In contrast, Algorithm-6 is optimised for the particular discrete mathematical approximation of the governing physical formula, exploiting the previously observed features.

It should be noted that this particular algorithm is not suitable for applications for which the previous assumption does not hold. This technique gives a major performance advantage over the conventional rotation method, in particular when the number of beam segments is increased.

3.1.7 Beam algorithm-7: Nested loops two-element array and rotation

Algorithm-7, as shown in Fig. 13, is based on improvements achieved in Algorithm-6. Additionally, the notion of nested loops was incorporated. The advantages and disadvantages of this approach were identified earlier and remain true for this particular algorithm.

3.2 Control algorithm
As mentioned earlier, the AVC algorithm consists of a beam simulation algorithm and control algorithm. For simplicity the control algorithm in (3) can be rewritten as a difference equation as in Fig. 14 (Hossain, 1995), where \( b_0, b_1, \cdots, b_4 \), and \( a_0, a_1, \cdots, a_3 \) represent controller parameters. The arrays \( y_{12} \) and \( y_c \) denote input and controller output, respectively. It should be noted that the control algorithm shown in Fig. 14 has similar design and computational complexity to one of the beam segments described and discussed in the beam simulation of Algorithm-1.

4 Implementation and results

The AVC algorithms based on seven different methods of the beam simulation and control algorithms were implemented with a similar specification using the C programming language on a uniprocessor computing domain for similar specification[7]. It is worth mentioning that seven different forms of the AVC algorithm were implemented based on the seven different forms of the beam simulation algorithm. Therefore, the AVC algorithm Alg-1 design form consists of beam simulation Algorithm-1 and the control algorithm in Fig. 14. Similarly, AVC algorithm Alg-2 is formed by combining the beam simulation Algorithm-2 and the control algorithm in Fig. 14 and so on. Therefore, seven different forms of AVC algorithm were implemented, tested and verified. It is worth noting that a fixed number of iterations (250 000) were considered in implementing all of the algorithms, for reasons of consistency.

To explore the controller performance of these design mechanisms, all the seven forms of the AVC algorithm were implemented for 20 segments. Although the AVC algorithm was designed in different forms, the resultant outcomes of all of these forms were maintained the same. Figs. 15~17 show the performance of the AVC algorithm using Alg-1. Fig. 15 shows beam fluctuation before cancellation while Fig. 16 shows the corresponding fluctuation after cancellation. These diagrams demonstrate the capabilities and dynamic behaviour of the resultant controller. This is further demonstrated in Fig. 17, which shows the auto-power spectral density at the end point of the beam before and after cancellation. As mentioned earlier, the main objective of this investigation is to maintain the same processing output with different forms of the algorithms, so as to demonstrate comparative real-time computing performance in implementing the AVC algorithm. Therefore, the performance of the other forms of the AVC system are not included here to avoid duplication.

To explore the comparative real-time computing performance of the design mechanisms, all seven forms of the AVC algorithm were implemented for 20 segments. The execution time performance of the algorithms relative to Alg-1 is shown in Table 1. It can be observed that Alg-3 was the slowest among all the algorithms. In contrast, Alg-2 performs best among all the design mechanisms. Alg-6 performed better than Alg-1, but was slower than Alg-2. It can also be observed that Alg-4 is almost 2.5 times slower than Alg-1. This is further demonstrated in Fig. 18, where Alg-3 was not incorporated due to its poor performance compared to the other algorithm designs. It can be seen that Alg-4 performed the worst among the six design mechanisms of algorithm shown in Fig. 18.

To explore the performance of the design mechanisms further, all designs of the algorithm, except Alg-3, were implemented with different numbers of segments. Fig. 19 depicts comparative performance of Alg-1 and Alg-2 for 20 to 200 segments. It can be seen that the execution time for both algorithms increases almost linearly when increasing the number of segments. It can also be seen that Alg-2 performs better throughout except in the 100 segments case.

Fig. 20 shows the comparative real-time performance in implementing Alg-6 and Alg-7. It can be seen that Alg-6 performs better throughout. It can also be seen that the performance variation of Alg-6 compared to the Alg-7 is not linear, and it performs best for the 80 segments case. Table 2 presents further details to demonstrate the performance of all the different designs of the AVC algorithm relative to Alg-1.

Table 2 shows the performance ratio of the different forms of the algorithm relative to Alg-1. It can be seen that Alg-4 performed worst throughout, and that the transition towards weaker performance occurred in AVC Alg-6 halfway between the transitions of Alg-1 and Alg-2. In spite of being outperformed by Alg-1 in a narrow band of around 100 segments, Alg-6 offers the best performance overall. Therefore, the design mechanism employed in Alg-3 can offer potential advantages in real-time control applications.
\[ yc[n] = b_0 \cdot y_{12}[n] + b_1 \cdot y_{12}[n-1] + b_2 \cdot y_{12}[n-2] + b_3 \cdot y_{12}[n-3] + b_4 \cdot y_{12}[n-4] - (a_0 \cdot yc[n-1] + a_1 \cdot yc[n-2] + a_2 \cdot yc[n-3] + a_3 \cdot yc[n-4]); \]

//Shift data array
\[ y_{12}[n-4] = y_{12}[n-3]; y_{12}[n-3] = y_{12}[n-2]; y_{12}[n-2] = y_{12}[n-1]; y_{12}[n-1] = y_{12}[n]; \]
\[ yc[n-4] = yc[n-3]; yc[n-3] = yc[n-2]; yc[n-2] = yc[n-1]; yc[n-1] = yc[n]; \]

Fig. 14 Design outline of the control algorithm (data array shifting method)

![Fig. 15 Fluctuation of the beam along the length before cancellation](image1)
![Fig. 16 Fluctuation of the beam along the length after cancellation](image2)

![Fig. 17 Auto-power spectral density at the end point before and after cancellation](image3)

![Fig. 18 Execution time in implementing different algorithms](image4)

Table 1 Relative performance of the different designs as compared to the Alg-1 (‘X’ represents 2, 3, 4, · · ·, 7)

| Ratio         | Alg-1 | Alg-2 | Alg-3 | Alg-4 | Alg-5 | Alg-6 | Alg-7 |
|---------------|-------|-------|-------|-------|-------|-------|-------|
| Alg-X/Alg-1   | 1     | 0.67  | 157   | 2.46  | 1.65  | 0.83  | 1.48  |
5 Concluding remarks

An investigation into the analysis, design, software coding and implementation of algorithms so as to reduce the execution time, and in turn enhance the real-time performance of the algorithm, has been presented within the framework of the real-time implementation of an active control algorithm. A number of design approaches were proposed and demonstrated for a control algorithm for a flexible beam. The same resultant outcomes with different forms in implementing AVC algorithm were maintained, so as to demonstrate comparative real-time computing performances. It was observed that execution time, and in turn the performance of an algorithm, varies with different approaches in a real-time implementation context. It is also shown through investigations that a design based on reduced instructions provides linear performance, although in most cases these are slower. In contrast, designs leading to large number of instructions cause non-linear transitions at certain stages when internal built-in instruction cache were unable to handle the load. It is worth mentioning that such transitions within the control algorithms, considered occur with the computation of different numbers of segments. Therefore, identification of suitable source code design and implementation mechanism for best performance is a challenge. As a whole, the proposed approaches can have a significant impact on the design and real-time implementation of real-time control algorithms.

References

[1] A. U. Thoeni. Programming Real-Time Multicomputers for Signal Processing. Prentice-Hall, Hertfordshire, UK, 1994.
[2] M. O. Tokhi, M. A. Hossain. CISC, RISC and DSP Processors in Real-Time Signal Processing and Control. Journal of Microprocessors and Microsystems, vol. 19, no. 5, pp. 291–300, 1995.
[3] G. Bader, E. Gehrke. On the Performance of Transputer Networks for Solving Linear Systems of Equation. Parallel Computing, vol. 17, no. 12, pp. 1397–1407, 1991.
[4] M. O. Tokhi, M. A. Hossain, M. J. Baxter, P. J. Fleming. Heterogeneous and Homogeneous Parallel Architectures for
Real-Time Active Vibration Control. *IEE Proceedings-D: Control Theory and Applications*, vol. 142, no. 6, pp. 1–8, 1995.

[5] M. O. Tokhi, M. A. Hossain, M. J. Baxter, P. J. Fleming. Tokhi. Performance Evaluation Issues in Real-Time Parallel Signal Processing and Control. *Journal of Parallel and Distributed Computing*, vol. 42, pp. 67–74, 1997.

[6] B. N. Bershad, D. Lee, T. H. Romer, J. B. Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, ACM Press, San Jose, California, pp. 158–170, 1994.

[7] B. Clader, C. Krantz, S. John, T. Austin. Cache-Conscious Data Placement. In *Proceedings of Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, ACM SIGARCH SIGOPS SIGPLAN and the IEEE Computer Society, San Jose, California, pp. 139–149, 1998.

[8] K. Hwang. Advanced Computer Architecture – Parallelism, Scalability, Programmability. McGraw-Hill, California, USA, 1993.

[9] S. Mcfailing. Program Optimization for Instruction Caches. In *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'89)*, ACM Press, Boston, MA, pp. 183–191, 1989.

[10] M. A. Hossain. Digital Signal Processing and Parallel Processing for Real-Time Adaptive Noise and Vibration Control. Ph.D. dissertation, Department of Automatic Control and System Engineering, The University of Sheffield, UK, 1995.

[11] P. K. Kourmoulis. Parallel Processing in the Simulation and Control of Flexible Beam Structure System. Ph.D. dissertation, Department of Automatic Control and Systems Engineering, The University of Sheffield, UK, 1990.

[12] G. S. Virk, P. K. Kourmoulis. On the Simulation of Systems Governed by Partial Differential Equations. In *Proceedings of IEE Control-88 Conference*, pp. 318–321, 1988.

[13] M. O. Tokhi, M. A. Hossain. Self-Tuning Active Control of Noise and Vibration. In *Proceedings of IEE - Control Conference-94*, Coventry, UK, vol. 1, pp. 263–278, March 21-24, 1994.

[14] U. Kabir, M. A. Hossain, M. O. Tokhi. Reducing Memory Access Time in Real-Time Implementation of Signal Processing and Control Algorithms. In *Proceedings of International Federation of Automatic Control (IFAC) Workshop on Algorithms and Architectures for Real-time Control (AARFC’2000)*, Palma de Mallorca, Spain, pp. 15–18, May 15-17, 2000.

Mohammed Alamgir Hossain received his M.Sc from University of Dhaka (Bangladesh) in 1984 and Ph.D. from University of Sheffield, UK, in 1995. He has worked at several academic institutions and is currently employed as Lecturer at the Department of Computing, the University of Bradford, UK.

His main research interests include intelligent control, high-performance computing for real-time signal processing and control (HPC) and network congestion control.

Mohammad Osman Tokhi obtained his B.Sc. degree in Electrical Engineering from Kabul University, Afghanistan, in 1978 and Ph.D. from Heriot-Watt University, UK, in 1988. He has worked at several academic and industrial establishments and is currently employed as Reader at the Department of Automatic Control and Systems Engineering, the University of Sheffield, UK.

His main research interests include adaptive/intelligent and soft computing techniques for modelling and control of dynamic systems, high-performance computing for real-time signal processing and control (HPC), and biomedical applications of robotics and control.