This is a repository copy of *Parallelising and Developing Control Algorithms for Heterogeneous Architectures*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/80059/

Monograph:
Baxter, M.J., Tokhi, M.O. and Fleming, R.F. (1995) Parallelising and Developing Control Algorithms for Heterogeneous Architectures. Research Report. ACSE Research Report 597 . Department of Automatic Control and Systems Engineering

Departme
AUTOMA
CONTROL
and SYSTEMS
ENGINEERING
UNIVERSITY of SHEFFIELD

PARALLELISING AND DEVELOPING CONTROL ALGORITHMS FOR HETEROGENEOUS ARCHITECTURES

M J Baxter, M O Tokhi and P J Fleming

Department of Automatic Control and Systems Engineering,
The University of Sheffield, Mappin Street, Sheffield, S1 3JD, UK.

Tel: + 44 (0)114 2825136.
Fax: + 44 (0)114 2731 729.
E-mail: O.Tokhi@sheffield.ac.uk.

August 1995

Baxter, MJ, Tokhi MO and Fleming PJ

Abstract

System developers have found that exploiting parallel architectures for control systems is challenging and often the resulting implementations do not provide the expected performance advantages over traditional uni-processor solutions. This paper presents a generic method and a suite of design tools for the implementation of control algorithms on parallel architectures. These tools allow a developer to translate a control system algorithm into efficient executable code, via a highly automated route, for a class advanced parallel architectures. The tools are demonstrated and discussed by developing several case-study algorithms to full implementations with an emphasis on the problematic areas leading to performance degradation common to parallel systems.

Key words: *Heterogeneous architectures, parallel processing, real-time control.*

Baxter, MJ, Tokhi MO and Fleming PJ

CONTENTS

LIST OF TABLES AND FIGURES

1 Introduction

The computational performance of sequential processors is no longer increasing at the explosive rate seen over the last two decades. Bounded by the physical constraints of manufacture and operation, it is unlikely that substantial increases will be observed in the future. Following this, a performance gap is emerging between the current hardware and todays demanding algorithms. In search of alternatives, parallel processing has been adopted by many as a solution for a variety of demanding applications. For example, recently available parallel processors such as the Inmos Transputer and the Texas Instruments TMS320C40 (C40) have generated considerable excitement in the field of control systems.

The key feature that distinguishes parallel processors from conventional processors are their dedicated communication hardware and links. These hardware features allow the construction of co-orporating networks of processors with the links providing inter-processor communication. Theoretically the processing power is scalable, additional processors can be introduced into a network to provide the computational performance required. However, as many engineers have discovered, this is far from the truth. The burden of inter-processor communication ultimately prevents any further improvement in the computational power by adding more processors. In addition, the communication overheads are often further increased by a poorly designed or inappropriate algorithm structure.

This is a particular problem for the control systems domain. Here, algorithms are often constructed from a large number of computationally undemanding tasks with, in comparison, a significant communication demand. The combination of these characteristics and the fast cycle times required often lead to parallel processing being intractable for algorithms in this form. In practice single processor solutions are often favourable for development simplicity and for the elimination of inter-processor communication.

Addressing the imbalance of computation and communication demand has been found to be challenging as a number of highly interacting issues are responsible for this performance degradation. However, tractable parallel solutions for control algorithms can

be afforded with careful consideration during the development of a parallel implementation. For example, the use of well-understood modern control theory permits the restructuring of an algorithm whilst preserving its transfer properties, thus allowing the algorithm to be tuned to the hardware (Baxter et.al, 1994a). Later, in the development, the algorithm can be further tuned within the software engineering domain.

This paper will continue with an introduction to the generic design approach used here and the design tools. Subsequently the case-study algorithms used throughout this paper are described, followed by a discussion of the class of target architectures considered. A discussion of the design tools follows, with particular attention to development difficulties and how the design tools are used to address them. Several implementations are developed and discussed, at various stages of development, within this section to illustrate the points raised. Finally the paper ends with the concluding remarks.

2   A generic approach and the design tools

A number of complex and interacting issues need to be addressed if the development of a parallel system is to be successful. To effectively exploit parallel architectures the algorithms hardware demands and the available hardware resources must be closely matched. An approach to this involves the profiling of the application algorithm to identify the demands, and the characterisation of each of the processors available to build the architecture (Ghafoor, 1993). This information can then be used to distribute the algorithm, in an efficient manner, to an appropriate configuration of processors. However, in practice this is difficult to achieve as the profiling of an algorithm must be performed with respect to certain types and quantity of computation, for example, degree of vectorism or the floating-point operations (FLOPs) required. These are metrics associated with the software domain and cannot be easily applied within the application domain. Moreover, the hardware characterisation is also difficult to achieve. This, effectively, requires extracting an accurate model of the processor behaviour by using benchmarking techniques, which are known to provide only a rough estimate of performance (Dongarra et.al, 1987).

2

The above approach, however, can be simplified providing the application algorithms are primarily constructed from a small number of computational block types. For example transfer functions, state-space equations, integrators, gains and adders are all commonly found in control system algorithms. Digital signal processing algorithms are also, in general, built from a small number of blocks and this characteristic can be found in a broad range of application domains. An efficient distribution of tasks to processors can then be based on advance knowledge of the execution times of these task sets. These requirements can be determined by benchmarking techniques that will be described later.

This approach has been encapsulated within the in-house developed suite of design tools used for the basis of this research investigation (Bass et.al, 1994). Figure 1 illustrates the development cycle the design tools are based upon. Here, an application oriented editor, in this case Simulink (The Mathworks, 1992a), provides an entry method for the algorithm into the design tools. Although conventionally Simulink is used to simulate dynamic systems it has also been found useful for the purpose of an editor. This editor is augmented by a set of transformation tools (Tts) (Baxter et.al, 1994a). These TTs allow the algorithm to be restructured, while preserving the transfer properties, in order to tune the algorithm to the available hardware. The algorithm is then parsed and imported into the software domain where it is represented as a data-flow diagram (DFD). The algorithm in this form is architecture independent. Each node in the DFD represents a block of the algorithm. The mapper subsequently appends an annotation to the DFD, of the task to processor mapping and determines the task execution order. From this information the code generator creates the source code for the implementation which is then compiled. The code generator utilises a library of code fragments, described as templates, which represent the typical building blocks of the application domain for the generation of the source code. The final stage is the performance analysis which assesses whether the implementation fulfils the original specification. Although the design tools greatly simplify the development process through automation, the user can intervene at many stages in order to tune the system performance.

3   Case study algorithms

Two control algorithms are considered in this paper for the illustration of the tools and techniques described. The first is the VAP flight control law, a linear MIMO algorithm. This algorithm has been entered into Simulink as can be seen in Figure 2. The inputs are pitch rate (q), barometric height error (hb), vertical acceleration(h) and airspeed error (ua). The outputs are elevator rate demand (nd) and throttle demand (vd). The second algorithm is the Turnip flight control law, also a linear MIMO algorithm as can be seen in Figure 3 and is used later in this paper. These algorithms are the starting point for the development, and will appear in different forms at several stages during implementation.

It should be noted that neither of the algorithms are sufficiently demanding to warrant a parallel approach. However, they are of sufficient complexity, possessing significant cross-coupling terms, to be a non-trivial exercise for a parallel implementation.

4   The target architecture

There are a wide variety of architecture classes that can be described as parallel, for example these can range from systolic arrays to clusters of workstations. This research programme is primarily focused on heterogeneous architectures, where a number of closely coupled and computationally disparate processors are integrated into a single network via point-to-point communication links. This is in contrast with the more conventional homogeneous architectures that employ only identical processing elements, an example of this would be a network of T8 transputers. The inherent nature of a heterogeneous architecture can provide three significant advantages over other solutions: performance, fault-tolerance (through hardware diversity) and cost. The computational demands of algorithms are often found to be heterogeneous and the use of a heterogeneous architecture may release a significant improvement in the real-time performance. Furthermore, the computational demands of modern control and digital signal processing (DSP) algorithms are increasing rapidly and may outpace the performance advancement of conventional hardware solutions. It is here that heterogeneous architectures may become an efficient and

4

cost effective method of providing such performance. However, the use of a heterogeneous architecture introduces further complexity in the design when compared to a homogeneous system.

A heterogeneous architecture can be constructed from various types of processors. The system here is developed for control and DSP applications with the real-time performance being of prime importance. The hardware, as illustrated in Figure 4, incorporates three C40s and a transputer configured as a fully connected network. Serial to parallel converters connect the transputer links with the C40 comports, here off-the-shelf hardware is used for simplicity. The hardware is hosted by a Sun workstation. A homogeneous language approach using 3L Parallel C for both processor types is used (3L, 1991,1992). Here the source code can be compiled for either processor type and thus reduces the complexity of mixed languages.

5   Design tools and implementation

This section discusses the development of parallel implementations of the case-study algorithms described earlier. The tools used at each stage of development will be described, with concentration on many of the issues that must be addressed to avoid unnecessary performance degradation. The starting point of the development is the VAP flight control law and the fixed heterogeneous architecture described earlier. The VAP was entered into Simulink in preparation for the development. Based on the previously outlined development approach the next stage for consideration are application domain transformations. Later the variants of the VAP control law and the Turnip control law will be discussed for further illustration of the techniques described.

5.1 Application domain transformations

Control system algorithms are typically constructed out of a number computational blocks with data flowing between them. This can be viewed as a data-flow diagram. The usual approach to implementing such an algorithm on a parallel architecture is through

algorithmic parallelism. Here, each computational block is assigned to a processor of the architecture and the communications between blocks are routed via communication channels, either on-chip or through hardware links to other processors.

The application algorithm, however, would normally have been developed without consideration of the target hardware. The implementation of the algorithm in this form may lead to significant degradation of the expected performance. This is mainly due to the large communication demand of the algorithm and contrastingly poor performance of the communication infrastructure of the hardware. However, other characteristics such as task to processor matching are also significant, especially in heterogeneous architectures. Many developers have found that recasting an algorithm into an alternative structure can lead to significant performance gains. This can be performed by application domain transformations that restructure an algorithm into a form intended to be more appropriate for the target hardware whilst preserving its transfer characteristics. Therefore, consideration of algorithm transformation should be an essential phase of any parallel implementation.

Obviously, transformations of an application algorithm are application domain dependent. However, modern control theory has a wide range of well understood mathematical techniques for the manipulation of control algorithms. Some of these have formed the basis of the available transformations within the design tools. The TTs augment the Simulink simulation environment, and Matlab is used with the Control System Toolbox for many of the transformation operations (The Mathworks, 1990, 1992b).

The TTs are driven using a graphical user interface (GUI) as can be seen in Figure 5. The TTs also performs the graphical restructuring of the algorithm in Simulink. To perform a transformation the user simply clicks on the appropriate blocks to be transformed in the Simulink block diagram, selects the transform options and clicks on `Transform' in the graphical user interface. The Simulink diagram is then redrawn, incorporating the transformations. Using the original Simulink diagram, which gives a useful display of the structure and complexity of the algorithm, recasting will give an immediate indication of the suitability of the recast algorithm.

Although many typical control system transforms have been identified and a software toolbox constructed, an open structure has been used to allow the user to integrate other transforms as they are required. The transforms that are supported by the toolbox currently are:

- *Representation Domain*: transforms from transfer function to state-space, and vice-versa.

- *Parallelisation*: single large grained tasks can be decomposed into several finer-grain parallel tasks.

- *Serialisation*: single large tasks can be decomposed into a serial chain of blocks, useful for partitioning a block containing different types of calculation so that it can be mapped across several types of processor.

- *Minimisation*: a group of blocks that have complex interconnections can be reduced into a single larger block.

- *Time Domain*: conversions from the continuous to the discrete time domain, and vice versa.

Although performing the transformations is clearly straightforward, the nature of what constitutes a good algorithm structure is not well understood. However, several issues should be considered during transformation and the first is granularity. Consider the VAP algorithm in Figure 2. The diagram contains a significant number of computationally simple blocks, such as gains and additions and integrators with complex interconnections. This results in a large quantity of communication in comparison to the computation. The granularity of the tasks in this case are described as fine-grained. Alternatively, if the computation was large when compared to the communication it would be described as coarse-grained. Granularity can be defined formally for both task and processor as (Maguire, 1991)

$$\text{Task granularity} = \frac{\text{computational requirements}}{\text{communication requirements}} \qquad (1)$$

$$\text{Processor granularity} = \frac{\text{computational performance}}{\text{communication performance}} \quad\quad (2)$$

For task granularity, equation (1), computational requirements are the number of FLOPs in each cycle of the task, and the communication requirements are the quantity of communications for each cycle of the task. It is essential, of course, that both of these requirements are based on the same word length. Using the above equation a measure of granularity for each block can be determined. These measures of granularity give the first indication of how profitable a parallel processing approach may be for the algorithm in hand. The measures may show the algorithm is unsuitable for a parallel implementation, and this can then lead to the restructuring of the algorithm into a form that is more suitable. For processor granularity a similar expression arises, equation (2). The computational performance comprises processor FLOPs and the communication performance comprises the data rate transfer of the communication links. This assumes that FLOPs are the main contributors to the processing time.

Table 1 shows the task granularity of selection of the blocks in the VAP algorithm of Figure 2. However, task granularity is only really useful when a measure of processor granularity is also available. For example, assume the hardware granularity is unity and constant throughout the processors in the architecture. This value of unity means that FLOPs are executed at the same rate as communication, however note that it is usually found that communications and computation do not occur concurrently. If the above tasks were mapped onto this architecture, it would be expected that for task `G1' that the processor would spend three times more time computing than communicating. However, for the finer-grained tasks such as Ge, a gain block, the time spent communicating is twice that of computing and would be expected to perform poorly in terms of execution time.

Consider Figure 6, this represents the first transformation performed with one simple operation of the TTs, it is the independent path transfer function representation of the original VAP algorithm of Figure 2. Clearly the number of blocks has been significantly reduced and this has resulted in a very significant reduction in the communications. It is also noticeable that all the transfer function blocks can now be executed concurrently,

8

improving the parallelism the algorithm expresses. The computational demands of each block has increased, this has effectively increased the task granularity which may be found to be advantageous. Consider the task granularities in Table 2, they are significantly larger than those of the original algorithm.

As a further illustration to this technique, consider a further transformation, here into a highly parallelised variant of the last transform, as can be seen in Figure 7. Here the resulting transfer function tasks can all be executed in parallel, with a final summation. This algorithm is highly parallel, and contains a number of fine-grained tasks as the computational requirements are small but have significant communications. This would in general be inappropriate for many architectures as the communications would become burdensome. The task granularities of this algorithm can be seen in Table 3, and the values do not vary so wildly as those of the preceding algorithms showing the algorithm has become more homogeneous.

The transforms reported here have retained the transfer function representation,. However, the algorithm may also benefit transformation into alternative representations, such as state-space for example which ultimately results in matrix computation. This will change the method by which the algorithm is evaluated and can be used to tune , for example, the regularity of an algorithm to a particular processor. Regularity is a term used to describe the degree of uniformity in the execution thread of the computation. For example the evaluation of a state-space equation is very regular, whereas for a transfer function it is less regular. If a vector processor was available, such as the powerful Intel 80i860, the regular matrix structure of state-space blocks would be a particularly appropriate representation. Algorithm transformation has a strong relationship with partitioning. Partitioning is the phase at which an algorithm is cut into smaller parts suitable for the target architecture. Algorithm transformation therefore permits more flexible parallelisation strategies as a number of starting points are available, the original algorithm and its transformations.

The true effectiveness of these transformations can be seen later in this paper after executable codes have been developed. However, as demonstrated here transformations

can be used to restructure the algorithm and tune the task granularities whilst preserving the transfer characteristics.

5.2 *Software domain representation and algorithm development*

Once the system developer is happy with the algorithm structure within the control system domain, the algorithm can be translated into the software domain using the design tools. A powerful computer aided software engineering (CASE) tool, called Software through Pictures (StP), is used here. This is the environment used to further develop the algorithm. Here it is represented as a data-flow diagram as can be seen in Figure 8, for the independent path variant of the VAP control law of Figure 6. The blocks of the simulink diagram are converted into nodes, and the connectivity into directed arcs. This one to one relationship is conceptually simple, efficient and preserves the structure of the algorithm.

The transfer into the software domain permits the subsequent development phases to be applied to the algorithm, namely the mapping, scheduling and code-generation. The subsequent stage to be addressed, in this design approach, is the combined mapping and scheduling of the nodes of the task graph to processors in the architecture and will be described simply as mapping. Currently, the target architecture is specified textually, however, future developments will lead to a more flexible specification of the architecture. The final phase is then the generation of source code and configuration files that can be compiled into an executable code for the architecture.

5.2.1 Mapping tools

Mapping is the allocation of the tasks that represent the parallelised algorithm onto the processors in the architecture. Mapping is probably the most critical stage in the entire implementation, as a badly mapped algorithm is unlikely to be well matched to the hardware and ultimately leads to the degradation of the real-time performance.

Mapping has been found to be computationally hard (NP-complete) (Shirazi et.al, 1990). Therefore, to obtain an efficient mapping a heuristical approach must be used. Although a significant amount of consideration has been given to the development of

mapping heuristics for homogeneous architectures by numerous researchers (Thoeni, 1994), the development of mapping heuristics that take account of the special characteristics of heterogeneous architectures is still in its infancy. Only a few research groups have addressed the issues of heterogeneity in mapping (Baxter et.al, 1994b; Menasce et.al, 1991a,b; Tripathi and Menasce, 1992). However, it is clear that to minimise the run time overheads of an algorithm, the mapping must be performed off-line (static mapping). This reduces the performance penalty that a dynamic mapping policy would introduce, and appropriate when the tasks composing the algorithm are not required to be dynamically reallocated or the task precedence changed during run-time.

The heterogeneous mapping problem is simple in concept. A data-flow diagram (DFD) represents the algorithm to be mapped across a network graph of the parallel architecture, where each node of the DFD must be mapped to a single node of the network graph. A mapping is considered good if it results in a low cycle time of the algorithm, and bad if it is high. Unfortunately the upper and lower bounds of the cycle time are difficult to estimate. Thus, only measures of relative goodness can be made. However, there are several important facets of this problem that influence the value of goodness.

Firstly, since the parallel architecture is heterogeneous it is found that the execution time of a task will vary depending upon which processor type it is mapped on. Therefore, some tasks may execute faster on processor type `X' than on type `Y'. However, for other tasks the faster processor may be type `Y'. This disparity can be explained in terms of how well-matched the task-processor pair are, for example, a task with a significant proportion of matrix computation may be best matched to a vector processor. A naive approach to the heterogeneous mapping problem could therefore be to place each task on the processor that achieves the minimum task execution time.

However, there are other factors that influence the cycle time of the algorithm. The most important of these, as described earlier, is communication time. It has been found that the interprocessor communication overheads in real-time parallel systems often dominate the actual computing time. The other extreme approach to mapping could be to map all tasks onto a single processor, thus reducing the inter-processor communication time to

11

zero. This is obviously a poor policy as there is no exploitation of any inherent parallelism and this would, in general, lead to a large cycle time of the algorithm. Also, the tasks would be constrained to the restrictive computational resources of a single processor type.

Therefore, there is a trade-off between the time advantage of mapping tasks to the most appropriate processor and the potentially devastating overhead of communications that may be incurred. Furthermore, it is also important to note that a DFD has precedence. Tasks cannot be executed until their predecessors have completed. The mapping must take account of the precedence to minimise processor idling while tasks are waiting to receive input or to send output data. Also of significance are interference costs. These are the costs (which are permitted to be negative) that occur when the combination of several tasks on a processor affect their expected execution times. These effects arise from architectural features such as instruction pipes, cache memory and multi-level memory.

In this design method the task execution order(schedule) is determined by simple rules, based on the precedence of the tasks and this is performed off-line. The scheduling is performed as an integral part of the mapping process.

The alternative mapping policies used in the design tools are discussed below.

5.2.2 The mapping heuristics

The design tools currently integrate two distinct heterogeneous mapping methods. The first is a simple heuristic drawn from previous research and the second is an experimental and in-house designed genetic algorithm (GA) approach (Baxter et.al, 1995). The mapping algorithms require a measure of execution time and two methods are used to calculate this: a simulation model and the use of the hardware in the loop. The available permutations of algorithms using these methods are:

- The modified Menasce (MM) heuristic.
- A simple genetic algorithm (GA) with a model architecture.
- A simple GA with the hardware-in-the-loop.

The modified Menasce approach This is a simple heuristic that constructs a mapping in a single pass. This is based on average task execution times for each processor, the communication costs and the precedence of the tasks.

This approach is an extension to the one of the heuristics developed by Menasce et al, who have presented a methodology for systematically building static heuristic processor mapping algorithms (Menasce et.al, 1991a; Tripathi and Menasce, 1992). A mapping algorithm was considered to be composed of two parts: an envelope and a heuristic. The envelope is responsible for selecting, at each step, the subset of tasks (called the task domain) and the subset of processors (called the processor domain) to be considered by the heuristic. The heuristic selects a task and processor pair from the task and processor domains. A simple deterministic simulation of the execution of a parallel job is performed, and assumes that task execution times are deterministic and equal to their estimated average values. In this manner, a heuristic is used to select the appropriate task and processor pair from the corresponding domain. These pairs are selected for mapping, at each instance of the simulation process, until either the task domain or the processor domain are empty. At this point the simulation clock is set to the time of the next event: the completion of the executing task. Then task domain is updated, based on the precedence of the tasks, and the processor domain based on the set of processors which are not executing any task at this time of the simulation process. This iterative procedure ends when all tasks have been mapped and scheduled, meaning that they have all been executed during this simulation. The strengths of this approach include accounting for the computational demand of the tasks and the precedence of the data-flow diagram. However, the initial implementation of the algorithm was found to perform poorly as it did not take account of the communication overheads. The algorithm was subsequently modified to consider the implications of communication and thus resulted in the MM algorithm.

The genetic algorithm approaches Genetic algorithms (GAs) are optimisation algorithms They are also engineering metaphors for the natural processes that are termed evolution (Goldberg, 1989). It is the simplicity of the central evolutionary mechanisms, of which

there are three, that make GAs so effective. These operators are: selection, crossover and mutation.

GAs employ a population of individuals. Each individual is represented by a data structure (a chromosome in biological terms) that encodes the parameters that are to be tuned by the optimisation process. Therefore, each individual represents a single point in the search space (a solution). The above operators are iteratively applied to each generation of individuals. The first operation is called selection. This will choose individuals for mating based on their objective value. The objective value represents how good that solution is and usually requires some form of function to be evaluated. The second operation, crossover, will take these pairs and exchange elements of the data structures. The crossover may result in the progeny having a higher or lower objective value as compared to the parents. The final operation, mutation, will periodically modify parts of the data structure of an individual and therefore making it represent a different point in the search-space. Mutation drives the GA into exploring alternative areas of the search-space.

Unlike many traditional optimisation techniques GAs do not require a knowledge of the search-space. Only an objective value for each individual is required. GAs have been found to be robust and particularly appropriate when the shape of the search-space is not well understood. This makes GAs an appropriate optimisation method for the heterogeneous mapping problem.

The same GA is used for both these approaches. The difference being the objective function they use. The first objective function was a model of the architecture, whereas the second integrated the real hardware giving more accurate values of the cycle time. The GA used a simple encoding strategy, where the chromosome length was equal to the number of tasks in the algorithm. The value of each element of the chromosome corresponds to the processor to which that task is mapped to.

A selection of attributes need to be specified for the GA, which are optimisation problem dependent. The number of individuals in each generation was set at 50. This value is loosely based on the length of the chromosome (i.e. the number of tasks) which for the application algorithms to be mapped was up to 34. Linear ranking and a selection pressure

14

of 2 were used. The selection strategy was stochastic universal sampling with a generation gap of unity, meaning the whole population is replaced at each generation. The crossover was the standard dual point crossover with a rate of 0.7. Finally, the mutation rate was set to 0.7 per individual. This GA can be used with both the model and the hardware-in-the-loop. Thus resulting in two variations in approach.

5.2.3 Benchmarking

An accurate determination of the load of each task on each processor is used by two of the mapping policies investigated. Moreover, the communication costs across the links in the architecture is also required. The process of determining these costs is described here as benchmarking. The mapping tools assume the execution times of each of the tasks are deterministic, and unaffected by the interactions of other tasks on the same processor. However, it has been found that several tasks on a single processing element can lead to competition of architectural resources. These are described as interference costs. Interference costs are not well understood but can be attributed to on-chip memory or cache contention, for example.

Communication benchmarking   Two types of communication links are used in the experimental heterogeneous architecture, transputer to C40 and C40 to C40. The implemented algorithms exclusively use a 32 bit representation of data. Therefore, the message passing capability of the links were benchmarked using 32 bit packets. The benchmarking source code was compiled by the 3L Parallel C compilers for both processors. The results are shown in Table 4. The performance of both communications is in the order of 1Mbit/sec for 32 bit sized packets. The manufacturer specified communication performance of these devices is in stark contrast to these results. The C40 links are quoted to be capable of 160Mbits/sec for a 50MHz device (Texas Instruments, 1991). The C40 to transputer links are known to be constrained by the parallel to serial converter that electrically interfaces these devices and would be expected to achieve a data rate of around 8Mbits/sec. However, the software also has an implication on this

performance. The 3L Parallel C for the C40 introduces a run-time micro-kernel which orchestrates all communications of that processor. This results in severe degradation of the communication performance for short messages, typical of control systems, which will affect both these results.

Code template benchmarking   The code template library represent the building blocks of the algorithms. Each of the code templates were integrated into a timing harness and executed on both processors. The benchmarking results shown in Table 5 are a small, but representative, number of the code templates that have been designed.The TMS320C40 appears to dominate the performance of the transputer in simple execution times. However, consider the unexpected disparity in the timing of the `Medium Transfer Function' library module. In this case the C40 is slower than the Transputer. This can be explained by register, on-chip memory or cache contention occurring on the C40 more readily than on the transputer. This is typical of the complex computational characteristics of a heterogeneous architecture. Such disparities are often not well understood.

The communication times in Table 4 are comparable with the measured execution times in Table 5, for some blocks the communication times are significantly larger clearly demonstrating the massive overheads communication can have on the execution time of an algorithm. The mapper must consider the cost of placing tasks on alternative processors against the cost of placing the modules on the same processor which may lead to a module being computationally mis-matched.

5.2.4 Architecture execution model and hardware-in-the-loop

The mapping heuristics rely on a value of cycle time of the complete algorithm in order to determine a solution. As introduced the algorithms either use a model of the architecture execution behaviour or the hardware within the optimisation loop. The model utilises the communication and code template benchmarking results that have been calculated in advance of the mapping phase. The model then only requires the DFD and a mappng in order to calculate an execution time. The benchmarking results are measures of average

16

performance of only single tasks, and the model cannot infer additional costs due to interference of tasks mapped to the same processor. The model can determine an execution time instantaneously.

The experimental architecture can also provide a measure of cycle time. However, this requires the code generator to create the source code for the implementation, this then must be compiled and run. Obviously, this does take significantly more time than that of the model. A clear trade-off results between accuracy and speed for the two methods of determining cycle time.

5.2.5 Source code generation and the template library

The subsequent stage in the development is the generation of an executable code for the specified architecture. An in-house developed code-generation tool is used for this. The code generator is the tool used to transfer the algorithm from the mapped and scheduled DFD to an executable code. It is recognised that algorithms from a particular application domain are often built from a small number of computational block types. This characteristic has been exploited by the code generator. A library of reusable code templates has been designed, that represent the most common building blocks of the application domain. For example, in the control systems domain, blocks representing transfer functions and state-space equations are characteristic. The library is extendible allowing the integration of additional templates. However, it currently supports the common building blocks of control algorithms such as transfer functions, integrators, adders and gains. In addition, there is also support for fault-tolerant mechanisms in the implementation through the use of voter blocks, for example. The library can easily be extended to allow other classes of algorithms to be implemented. For example, libraries of code templates for implementing active noise control and adaptive control algorithms are being considered.

The code generator plucks the code templates from the library that represent the blocks of the algorithm, and assembles these code fragments with the addition of software harnesses into complete source code files ready for compilation. This is based on the results

of the mapping and scheduling phase which has annotated each block of the algorithm with a processor of the architecture. The code templates extracted from the library are then instantiated with the parameters of that instance of the block in the algorithm. Subsequently the instantiated templates are integrated into source code files, one for each processor. Furthermore, the code generator also generates the configuration files and 'make' files for automating the compilation phase. The generated source code is 3L Parallel C, although other generators are available for homogeneous networks of Transputers generating Inmos ANSI C using the same code template library.

The mechanism for building the complete source code is relatively straightforward, however several issues must be considered to preserve the transfer properties of the algorithm. These are both related to numerical representation. For example, the analogue to digital converters in the final architecture generate integers as output. However, the controller would usually use floating point representation. The code generation must provide the conversion routines for this. The use of heterogeneous architectures introduce related problems. For example, the transputer uses formats based on the IEEE floating-point representation, whereas the C40 uses the Texas Instruments format. Again conversion routines must be integrated. Other common problems, such as round-off, truncation and overflow must also be considered.

5.3 Implementation results

At this stage the real-time performance of the implemented algorithms is finally assessed, and the results can be seen in Table 6. The results show both the simulated cycle time derived from the model, and the measured cycle time of the implemented algorithms. The TTs were used to restructure the VAP control algorithm into the independent path VAP, and the resulting cycle time is half that of the original. The tuning of the parallelism and task granularities are responsible for this significant enhancement. Tuning the algorithm within the application domain should be explored further. In particular, recasting into alternative representations should be examined. For example, transforming the algorithms

into state-space is potentially profitable for architectures employing processors with vector (e.g. Intel 80i860) or zero overhead looping (e.g. C40) facilities.

The simulated cycle times have been calculated by the architecture execution model, and when compared to the actual measured cycle times a clear disparity is observed. For the VAP and turnip algorithms this has misled the mapping heuristic into identifying poorer solutions than when the hardware-in-the-loop is used. This proves the model is not accurate and should be developed further. However, modelling execution behaviour is challenging as it also requires an understanding of interference costs between tasks on the same processor. The use of the hardware-in-the-loop does provide vastly improved results, although the evaluation time of an implementation requires compiling the complete source code of the implementation and this takes a significant amount of time. If the system was increased in complexity by using a large number of processors and tasks this approach may be infeasible and modelling may be the only solution.

The mapping algorithms are shown to vary significantly. The increased complexity of the heterogeneous mapping optimisation problem is clearly challenging. The MM approach fairs particularly badly for the algorithms with a larger number of tasks when compared to the GA approaches. However, the use of the hardware-in-the-loop for the GA improved the performance significantly. The heterogeneous mapping problem needs to be investigated further.

6   Conclusion

This paper has introduced a generic method for the implementation of control algorithms onto parallel architectures. A suite of design tools have been developed to automate many stages of the development and these have been discussed for the implementation of several algorithms. The tools provide a highly automated route from algorithm to implementation, permit tuning by the user and provide a method of rapid prototyping of parallel implementations. Throughout the development of the implementations, the paper has focused on and addressed many of the development problems that can lead to a detriment

to the real-time performance of the implementation. The use of application domain transformation has been shown to provide real-time performance advantages in the final implementation. Several mapping policies have been explored and the use of the hardware-in-the-loop within the optimisation loop is shown to significantly improve the optimisation process in terms of identifying a good distribution of tasks. However, the large time spent compiling may render it less useful approach for larger systems.

7   Acknowledgements

8   References

BASS, J. M., BROWNE, A. R., HAJJI, M. S., MARRIOT, D. G., CROLL, P. R. and FLEMING, P. J. (1994). Automating the development of distributed control software, *IEEE Parallel and Distributed Technology*, **2**, (4), pp. 9-19.

BAXTER, M. J., TOKHI, M. O. and FLEMING, P. J. (1994a). Parallelising algorithms to exploit heterogeneous architectures for real-time control systems, *Proceedings of the IEE Control `94 Conference*, **2**, pp. 1266-1271.

BAXTER, M. J., TOKHI, M. O. and FLEMING, P. J. (1994b). Heterogeneous architectures for real-time control systems: Design tools and scheduling issues, *Preprints of the 12th IFAC Workshop on Distributed Computer Control Systems*, pp. 89-94.

BAXTER, M. J., TOKHI, M. O. and FLEMING, P. J. (1995). Mapping real-time control algorithms onto heterogeneous architectures, *Preprints of the IFAC Workshop on Algorithms and Architectures for Real-time Control*, pp. 563-568.

DONGARRA, J., MARTIN, J. L. and WORLTON, J. (1987). Computer benchmarking: paths and pitfalls, *IEEE Spectrum*, July, pp. 38-43.

GHAFOOR, A. (1993). A distributed heterogeneous supercomputing management system, *IEEE Computer*, June, pp. 78-86.

GOLDBERG, D. E. (1989). Genetic algorithms in search, optimisation and machine learning, Addison-Wesley.

MAGUIRE, L. P. (1991). *Parallel architectures for Kalman filtering and self-tuning control*, PhD thesis, The Queen's University of Belfast, UK, pp. 50-57.

MENASCE, D. A., DA SILVA PORTO, S. C. and TRIPATHI, S. K. (1991q). *Static heuristic processor assignment in heterogeneous multiprocessors*, University of Maryland at College Park, Technical Report UMIACS TR-91-131 and CS TR-2765.

MENASCE, D. A., SAHA, D., DA SILVA PORTO, S. C., ALMEIDA, V. A. F, and TRIPATHI, S. K. (1991b). Static and dynamic processor scheduling disiplines in heterogeneous parallel architectures, University of Maryland at College Park, Technical Report UMIACS TR-91-162 and CS TR 2807.

SHIRAZI, B., WANG, M. and PATHAK, G. (1990). Analysis and evaluation of heuristic methods for static task scheduling, *Journal of Parallel and Distributed Computing*, **10**, pp. 222-232.

TEXAS INSTRUMENTS (1991). *TMS320C4x User's Guide*, Texas Instruments Inc.

THE MATHWORKS (1990). *Control system toolbox*, The Mathworks Inc.

THE MATHWORKS (1992b). *MATLAB reference guide*, The Mathworks Inc.

THE MATHWORKS (1992a). *SIMULINK user's guide*, The Mathworks Inc.

THOENI, U. A. (1994). *Programming real-time multicomputers for signal processing*, Prentice-Hall.

TRIPATHI, S.K. and MENASCE, D. A. (1992). Scheduling issues in heterogeneous multiprocessor systems, *Transputers `92*, pp. 1-16.

3L (1991). *Parallel C user guide for the INMOS transputer*, 3L Ltd.

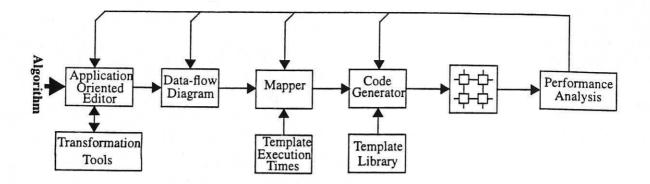3L (1992). *Parallel C user guide for the Texas Instruments TMS320C40*, 3L Ltd.
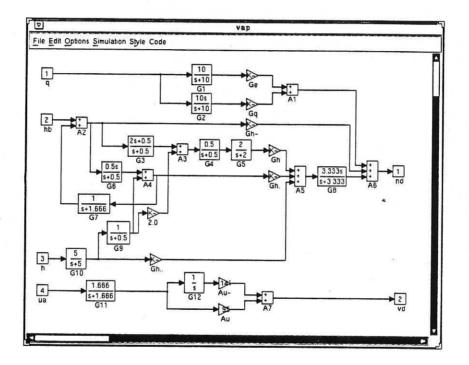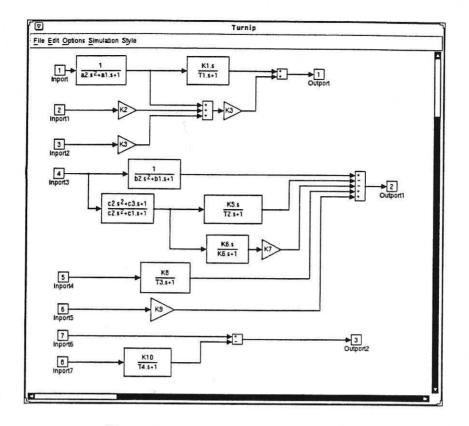
Figure 1: Heterogeneous System Development Cycle

Figure 2: VAP flight control law entered into Simulink
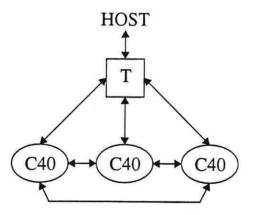
Figure 3: Turnip helicopter flight control law

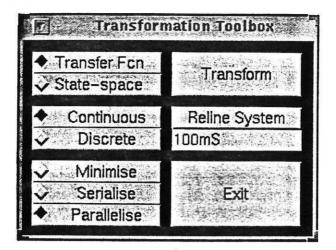**Figure 4: The topology of the experimental heterogeneous architecture**

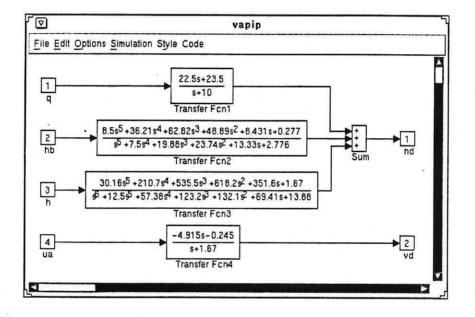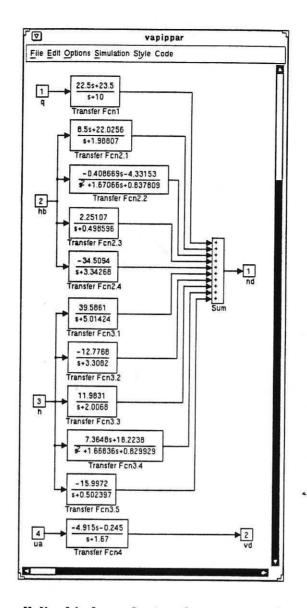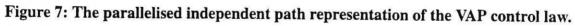**Figure 5: The Transformation Toolbox graphical user interface.**

Figure 6: Algorithmic transformation of the VAP flight control law

Figure 7: The parallelised independent path representation of the VAP control law.
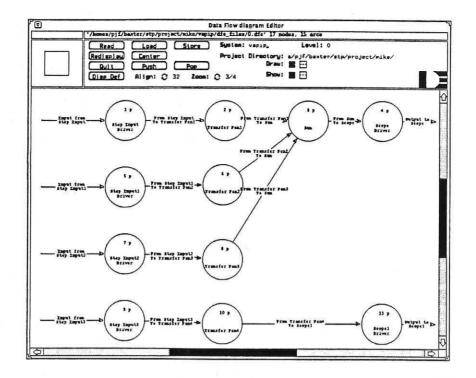
Figure 8: The independent path VAP flight control law within Software through Pictures

Block Name	FLOPs	Comm's (Words)	Task Granularity
G1	6	2	3
Ge	1	2	0.50
A4	1	3	.33
G8	10	2	5

Table 1: Granularity calculations for a selection of blocks in the VAP control law

Block Name	FLOPs	Comm's (Words)	Task Granularity
Transfer Fcn1	10	2	5
Transfer Fcn2	44	2	22
Transfer Fcn3	46	2	23
Transfer Fcn4	10	2	5

Table 2: Granularity calculations for the independent path representation of the VAP control law

26

Block Name	FLOPs	Comm's (Words)	Task Granularity
Transfer Fcn1	10	2	5
Transfer Fcn2.1	10	2	5
Transfer Fcn2.2	14	2	7
Transfer Fcn2.3	6	2	3
Transfer Fcn2.4	6	2	3
Transfer Fcn3.1	6	2	3
Transfer Fcn3.2	6	2	3
Transfer Fcn3.3	6	2	3
Transfer Fcn3.4	14	2	7
Transfer Fcn3.5	6	2	3
Transfer Fcn4	10	2	5

Table 3: Granularity calculations for the parallelised independent path representation of the VAP control law

Link Type	Time for Message-passing (μsec) (length = 32 bit)	Data rate of Message-Passing (Mbits/sec)
T8 and C40	24	1.333
C40 and C40	38	0.842

Table 4: Communication Benchmarking Results

Library Module	Time on 25MHz T805 Transputer (μS)	Time on 40 MHz TMS320C40 (μS)	Ratio of Execution Times(Transputer/C40) (μS)
Simple Transfer Function	11.21	5.21	2.15
Medium Transfer Function	36.01	41.81	0.86
Complex Transfer Function	114.08	67.61	1.69
5 Input Summer	2.52	0.74	3.41
3 Output Demultiplexer	1.21	0.55	2.57
Gain	1.65	0.47	3.51

Table 5: Library Module Execution Time Benchmarking

27

Algorithm	Mapper	Simulated Cycle Time(μS)	Measured Cycle Time(μS)
VAP Flight Control Law	MM	290.93	354.00
	GA+model	113.62	281.00
	GA+hard.	240.76	239.00
Independent Path VAP Flight Control Law	MM	117.27	128.00
	GA+model	117.27	127.00
	GA+hard.	117.27	120.00
Turnip Helicopter Flight Control Law	MM	230.08	299.00
	GA+model	111.00	195.00
	GA+hard.	327.87	168.00

Table 6: Algorithm cycle times

28