

**A Multiprocessing System-on-Chip Framework
Targeting Stream-Oriented Applications**

by

Darcy Philip Cook

A Thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
In partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada

Copyright © 2010 by Darcy Philip Cook

Abstract

Over the past decade, the processing speed requirement of embedded systems has steadily increased. Since faster clocking of a single processor can no longer be considered to increase the processing speed of the system (due to overheating and other constraints), the development of multiprocessors on a single chip has stepped up to meet the demand. One approach has been to design and develop a multiprocessing platform to handle a large set of homogeneous applications. However, this development has been slow due to the intractable design space, which results when both the hardware and software are required to be adjustable to meet the needs of the dissimilar applications. A different approach has been to limit the number of targeted applications to be similar in some sense. By limiting the number of targeted applications to a cohesive set, the design space can become manageable. This thesis proposes a framework for a multiprocessing system-on-chip (MPSoC), consisting of a cohesive hardware and software architecture intended specifically for problems that are stream-oriented (e.g., video streaming). The framework allows the hardware and software to be customized to fit a specific application within the cohesive set, while narrowing the design space to a manageable set of design parameters. In addition, this thesis designs and develops an analytic model, using a discrete-time Markov chain, to measure the performance of an MPSoC framework implementation when the number of concurrent processing elements is varied. Finally, a chaotic simulated annealing algorithm was developed to determine an optimal mapping and scheduling of tasks to processing elements within the MPSoC.

Acknowledgements

I would like to thank the following individuals and groups who were instrumental in providing advice and support throughout the development of this thesis.

- My advisor Dr. K. Ferens for his guidance and support in the development of this thesis and throughout my entire graduate studies.
- Dr. A.S. Alfa, Dr. W. Kinsner, and Dr. P. Thulasiramin who each were helpful in guidance and support of my projects within each of their courses that together make up a large portion of the work in this thesis.
- The Xilinx corporation for donating the Xilinx ISE and Xilinx EDK development tools that were used for the experimental implementation of the MPSoC described in this thesis.
- Parker Hannifin, formerly Vansco Electronics, my former employer, who provided financial support and provided the flexibility to allow me to pursue graduate studies while being employed for them.
- Bristol Aerospace, my current employer who provided the flexibility to allow me to continue my graduate studies while being employed for them.
- My parents for their encouragement to continue the pursuit of higher education.
- My kids (Nolan, Grace, Hannah, and Rebekah) for sharing their dad with his graduate studies.
- Most of all I would like to thank my wife, Susan, who has been a parent-and-a-half to make up for her husband being half-a-parent while pursuing his own dreams.

TABLE OF CONTENTS

Abstract	i
Acknowledgements	ii
List of Figures	vi
List of Tables	ix
List of Abbreviations and Acronyms	x
List of Symbols	xii
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Thesis Statement and Objectives.....	3
1.3 Organization of Thesis	4
Chapter 2 Background and Related Work	6
2.1 General Background on Computing Frameworks.....	6
2.2 Models of Computation.....	8
2.2.1 Overview of Models of Computation	8
2.2.2 Synchronous Dataflow Model of Computation	9
2.3 Parallel Processing Architectures.....	13
2.4 Related Work in Multiprocessing System Frameworks.....	17
2.5 Thesis Contributions to the Field of Study.....	23
Chapter 3 Multiprocessing System-on-Chip Framework Description.....	25
3.1 MPSoC Framework Architecture.....	25
3.1.1 Hardware Architecture.....	25
3.1.2 Software Architecture	36
3.1.3 Hybrid Pipeline Scheduling	40
3.2 MPSoC Experimental Implementation	47
3.2.1 Soft Processor Implementation	48
3.2.2 Task Controller Implementation	48
3.2.3 Task Controller Interface Peripheral Implementation	48

3.2.4	Memory Controller Implementation	49
3.2.5	Memory Controller Interface Peripheral Implementation	51
3.2.6	Global Memory Implementation.....	51
3.2.7	Snoopy	52
3.2.8	Snoopy Interface Peripheral.....	56
Chapter 4	Analytical Model for the Multiprocessing System-on-Chip Framework.....	58
4.1	Motivation for an Analytical Model of the MPSoC Framework	58
4.2	Related Work in Analytical Models of Multiprocessing Systems	59
4.3	Performance Metrics	61
4.4	Analysis Inputs	62
4.5	System Design Assumptions.....	63
4.5.1	Types of Memory Requests	63
4.5.2	Distribution of Memory Requests.....	64
4.5.3	Memory Service Distribution	65
4.5.4	Task Execution Times.....	70
4.5.5	Memory Access Control	71
4.6	Description of the MPSoC Analytical Model	71
4.6.1	Partitioning the Processing Period	72
4.6.2	Analyzing a Partition	74
4.7	Example using the MPSoC Analytical Model	90
Chapter 5	Task Allocation Optimization	98
5.1	Motivation for Task Allocation Optimization.....	98
5.2	Overview of Combinatorial Optimization Techniques Applied to Task Allocation Problems.....	99
5.3	Application of Simulated Annealing to the Task Allocation Problem.....	103
5.4	Application of Chaotic Simulated Annealing to the Task Allocation Problem	106
5.4.1	Continuous Function Chaotic Simulated Annealing.....	106
5.4.2	Mapping a Chaotic Variable in the Solution Space	109
5.4.3	CSA1 Method	109
5.4.4	CSA2 Method	111

5.4.5	CSA3 Method	113
5.5	Comparison of SA and CSA Results.....	115
5.5.1	Time Series Analysis	119
5.5.2	Power Density Spectrum.....	124
5.6	Task Allocation Optimization Conclusions	127
Chapter 6	Experimental Results.....	129
6.1	Experimental Application	129
6.2	Experimental Implementation and Results	132
Chapter 7	Conclusions and Future Work.....	141
7.1	Thesis Conclusions.....	141
7.2	Recommendations for Future Work.....	144
7.3	Thesis Contributions	146
References	151
Appendix A	159
A.1.	Xilinx Tool Development Flow	159
A.2.	Simulation of the MPSoC Framework Implementation.....	164
A.3.	Xilinx University Program Virtex-II Pro Development Board	166
A.4.	Experimental Setup	167
Appendix B	169
B.1.	Matlab Implementation of the Analytical Model	169
B.2.	Implementation of the Analytical Model and Optimization Algorithm in the C Programming Language.....	169

LIST OF FIGURES

Fig. 2-1. An example of a dataflow graph.	10
Fig. 2-2. An example of processing tasks from a dataflow graph in a pipeline.....	12
Fig. 3-1. Block diagram of MPSoC hardware architecture.....	26
Fig. 3-2. Interface between the Task Controller and soft processor	30
Fig. 3-3. Interface between the memory controller and a single processor.	34
Fig. 3-4. Interface between the global memory, the memory controller, a single processor.	35
Fig. 3-5. A flowchart of the general program structure.	37
Fig. 3-6. A flowchart of the head processor program structure.	39
Fig. 3-7. Traditional pipeline with variable task time.....	43
Fig. 3-8. Dynamic pipeline scheduling with variable task time.....	45
Fig. 3-9. Filling the pipeline with hybrid pipeline scheduling.....	46
Fig. 3-10. Implementation of the task controller interface peripheral controller.....	49
Fig. 3-11. Implementation specific interface to global memory.....	50
Fig. 3-12. Implementation specific memory controller interface peripheral.	51
Fig. 3-13. Snoopy interface peripheral.....	57
Fig. 4-1. Distribution of service time for $k=5$ and $p_e=0.25$	67
Fig. 4-2. Distribution of service time for $k=10$ and $p_e=0.5$	67
Fig. 4-3. Memory distribution of the individual memory types: (a) memory type 1, (b) memory type 2, (c) memory type 3. The probability distribution for a global memory access made up of a combination of the three memories is shown in (d).....	70
Fig. 4-4. First window analyzed in the processing period.	73
Fig. 4-5. Second window analyzed in the processing period.....	73
Fig. 4-6. State transition diagram for processor i	75
Fig. 4-7. Initial task arrangement in the processing period.....	92

Fig. 4-8. Probability distributions of Markov chains representing memory access in first partition.	93
Fig. 4-9. Task arrangement in processing period after first partition analyzed.	94
Fig. 4-10. The ideal processing period is compared with the actual processing period. The ideal processing period does not consider the effect of memory interference, where the actual processing period does consider memory interference.	96
Fig. 5-1. Example of a down movement of task D.	104
Fig. 5-2. Histogram of values generated by Logistic Map (left) and New Chaotic Map (right), from [MiHu04].	108
Fig. 5-3. Histogram of perturbations for CSA1.	111
Fig. 5-4. Chaotic variable vs. algorithm iteration for CSA2.	113
Fig. 5-5. Histogram of perturbations for CSA2.	113
Fig. 5-6. Chaotic variable vs. algorithm iteration for CSA.	115
Fig. 5-7. Histogram of perturbations for CSA3.	115
Fig. 5-8. Initial task solution for experiment #1.	119
Fig. 5-9. Initial task solution for experiment #2.	119
Fig. 5-10. Processing period calculated at each iteration for experiment #1.	119
Fig. 5-11. Number of perturbations at each iteration for experiment #1.	120
Fig. 5-12. Period calculated at each iteration for experiment #3.	121
Fig. 5-13. Period calculated at each iteration for experiment #2.	122
Fig. 5-14. Processing period calculated at each iteration for experiment #4.	123
Fig. 5-15. Power density spectrum of experiment #3 results.	125
Fig. 6-1. A dataflow graph representing the green screen application.	130
Fig. 6-2. Processing period results for the ideal value, the experimental measurement, the calculation by the proposed analytical method, and the calculation by the simple method (saturated bandwidth from [SLOW07]).	134
Fig. 6-3. Speedup of the memory intensive application for two, three, and four processors.	136

Fig. 6-4. Individual measured task execution times for the memory intensive application.	137
Fig. 6-5. Execution time vs. the number of processors for the computationally intensive application.	138
Fig. 6-6. Speedup vs. the number of processors for the computationally intensive application.	139
Fig. 6-7. Processor idle time of each processor in the one, two, three, and four processor cases.	140
Fig. A - 1. A Screenshot of the design summary page within the Xilinx ISE 10.1.03 tool	161
Fig. A - 2. A screenshot of a VHDL module Open within the Xilinx ISE 10.1.03 tool.	161
Fig. A - 3. A screenshot of the peripheral interface GUI within the Xilinx EDK tool. ..	163
Fig. A - 4. A screenshot of the four Microblaze processor block diagram generated by the Xilinx EDK tool.	163
Fig. A - 5. A screenshot of the Xilinx EDK tool with a software application for one of the processors open.	164
Fig. A - 6. A screenshot of a simulation test bench for the task scheduler module.	165
Fig. A - 7. A screenshot of a simulation of the task scheduler hardware module.	166
Fig. A - 8. Xilinx University Program Virtex-II Pro Development Board from [Xili05]	167
Fig. A - 9. Block diagram of MPSoC experimental test setup.	168
Fig. A - 10. Screenshot of serial output received by HyperTerminal running on the PC, with data sent from the XUPV2P board.	168
Fig. B - 1. Screenshot of output from chaotic simulated annealing program.	170

LIST OF TABLES

Table 4-1. Example task parameters.....	90
Table 4-2. Task parameters after first partition is analyzed.....	94
Table 4-3. Task parameters after analyzing the entire processing period.....	95
Table 5-1. Task characteristics for experiments 1 and 2.....	116
Table 5-2. Task characteristics for experiments 3 and 4.....	117
Table 5-3. Experimental final processing periods.	124
Table 6-1. Experimentally determined task parameters of the memory intensive application.....	133
Table 6-2. Task allocations for 2, 3, and 4 processors.....	133

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application Specific Integrated Circuit (pg. 6)
BOA	Bayesian Optimization Algorithm (pg. 22)
CSA	Chaotic Simulated Annealing (pg. 102)
DFG	Dataflow Graph (pg. 9)
DMA	Direct Memory Access (pg. 22)
DRAM	Dynamic Random Access Memory (pg. 63)
DSP	Digital Signal Processor (pg. 6)
EDK	Embedded Development Kit (pg. 159)
EEPROM	Electrically Erasable Programmable Read Only Memory (p. 63)
FFT	Fast Fourier Transform (pg. 19)
FIFO	First-In, First-Out (pg. 11)
FPGA	Field Programmable Gate Array (pg. 5)
FPU	Floating Point Unit (pg. 48)
GA	Genetic Algorithm (pg. 22)
Geo	Geometric Distribution (pg. 75)
GPU	Graphic Processing Unit (pg. 6)
GUI	Graphical User Interface (pg. 162)
HNN	Hopfield Neural Networks (pg. 102)

ISE	Integrated Synthesis Environment (pg. 159)
JPEG	Joint Photographic Experts Group (pg. 1)
JTAG	Joint Test Action Group (pg. 159)
kB	kilobyte (pg. 51)
MHz	Megahertz (pg. 52)
MIMD	Multiple Instruction, Multiple Data (pg. 28)
MMU	Memory Management Unit (pg. 48)
MPEG	Moving Picture Experts Group (pg. 1)
MPSoC	Multiprocessing System-on-chip (pg. i)
NoC	Network-on-Chip (pg. 14)
NP	Nondeterministic Polyminal (pg. 98)
NUMA	Non-Uniform Memory Access (pg. 25)
OPB	On-chip Peripheral Bus (pg. 48)
PC	Personal Computer (pg. 132)
PH	Phase Type Distribution (pg. 75)
PSO	Particle Swarm Optimization (pg. 99)
QEA	Quantum-inspired Evolutionary Algorithm (pg. 99)
RAM	Random Access Memory (pg. 35)
RGB	Red Green Blue (pg. 129)

SA	Simulated Annealing (pg. 99)
SDF	Synchronous Data Flow (pg. 4)
SDRAM	Synchronous Dynamic Random Access Memory (pg. 63)
SRAM	Static Random Access Memory (pg. 63)
TS	Tabu Search (pg. 99)
UMA	Uniform Memory Access (pg. 25)
USB	Universal Serial Bus (pg. 166)
VHDL	VHSIC Hardware Description Language (pg. 147)
VHSIC	Very High Speed Integrated Circuit (pg. xii)
XUPV2P	Xilinx University Program Virtex-II Pro (pg. 47)
YUV	Colour space based on luminance and chrominance (pg. 129)

LIST OF SYMBOLS

$\%$	percent (pg. 93)
$0_{(ij)}$	a zero filled i by j matrix (pg. 76)
α_i	probability of a memory request from task i (pg. 72)
α_n	attenuation variable in CSA1 algorithm (pg. 109)
β_{np}	number of tasks a given processor has assigned (pg. 43)
β	starting vector in the memory service time probability transition matrix (pg. 66)
β_c	damping factor in Mingjun and Huanwen algorithm (pg. 107)
β_d	damping parameter in CSA1 algorithm (pg. 109)
β_s	spectral exponent in the $\frac{1}{f\beta_s}$ term (pg. 124)
γ	parameter in new chaotic map (pg. 107)
ΔE	difference between the pipeline processing periods of successive iterations in the SA and CSA algorithms (pg. 105)
ε_i	error between iterative calculations of φ_i for processor i (pg. 85)
ε_r	error limit for iterative process of finding $f_{1,1}^{(n)}$ (pg. 83)
ζ_i	probability that processor i will have to wait for memory access when it has a pending memory request (pg. 88)
η	parameter in new chaotic map (pg. 107)

θ_i	ratio of the task executed in the analyzed partition window to the total execution time of the task (pg. 89)
θ_{prev_i}	ratio of the amount of task i that was analyzed in previously analyzed partitions to the total task execution time (pg. 89)
λ	number of pipeline stages (pg. 42)
λ_i	probability of accessing memory type i in global memory (pg. 69)
μ	parameter in the logistic chaotic equation (pg. 107)
π_i	steady state probability vector for processor i (pg. 87)
$\sigma_i^{(n)}$	probability of processor i requesting access to memory within n time quanta (pg. 82)
v_i	vector representing start of service in the vacation probability transition matrix for processor i (pg. 75)
$\varphi_i^{(n)}$	probability that there will be a memory request from processor i when it finishes vacation (pg. 80)
A_j	matrix representing transitions between states of processor j in the vacation process (pg. 78)
B_j	matrix representing transitions from states where the memory controller is serving processor j in the vacation process (pg. 78)
D_β	fractal spectral dimension (pg. 124)

$f_{x,y}^{(n)}$	probability of first visiting state y from state x in a Markov chain at the n^{th} time quantum (pg. 82)
$f(T_n)$	cooling schedule function for SA and CSA algorithms (pg. 106)
G^*	global minimum task arrangement for SA and CSA algorithm (pg. 105)
G_{current}	initial task arrangement for SA and CSA algorithm (pg. 105)
G_{new}	latest task arrangement for the SA and CSA algorithms (pg. 105)
J	number of unsuccessful iterations in the SA and CSA algorithms before the search is restarted from the best known solution (pg. 106)
k	number of states in the memory service probability transition matrix (pg. 66)
k_b	constant analogous to Boltzmann's constant used in SA and CSA algorithms (pg. 106)
K_{DS}	total number of data sets to be executed by tasks in the DFG (pg. 44)
m	number of states in the vacation probability transition matrix (pg. 75)
mod	the modulus operator (pg. 78)
M_L	list of preceding tasks for each task in the DFG (pg. 44)
M_{mem}	number of different types of memory in a multiple memory system (pg. 68)
$\text{num}_{\text{perturbs}}$	number of perturbations in CSA1 algorithm (pg. 110)
N_L	list of tasks assigned to a processor (pg. 43)

N	number of active processors (pg. 71)
N_{mem}	average number of memory requests made during a task execution (pg. 64)
$N_{\text{no_mem}}$	number of time quanta during a task's execution without a memory request (pg. 64)
p_e	probability of moving to the next stage in a negative binomial process (pg. 66)
perturb_max	maximum number of perturbations in CSA algorithms (pg. 110)
perturb_min	minimum number of perturbations in CSA algorithms (pg. 110)
P_i	probability transition matrix for processor i (pg. 76)
Q	list of the number of times each task in a DFG has been executed (pg. 44)
S	matrix representing the phases of service in the service time probability transition matrix (pg. 66)
S^0	vector representing the end of service in the service time probability transition matrix (pg. 66)
$\text{tanh}()$	hyperbolic tangent function (pg. 107)
t_p	pipeline processing period (pg. 12)
t_p^*	global minimum pipeline processing period (pg. 105)
$t_{\text{partition_new}}$	new time of a pipeline processing window after considering memory access waiting times (pg. 89)

$t_{\text{remaining_new_i}}$	time remaining in a task to be analyzed after analysis of a single time window (pg. 89)
$t_{\text{remaining_old_i}}$	time remaining in a task to be analyzed after analysis of a single time window without considering memory access wait times (pg. 89)
$t_{\text{total_task_i}}$	total time remaining for task i that is yet to be analyzed (pg. 89)
t_s	system clock period (pg. 63)
T	transpose matrix operation (pg. 66)
T_{max}	maximum temperature parameter for SA and CSA algorithms (pg. 105)
T_{min}	minimum temperature parameter for SA and CSA algorithms (pg. 105)
T_p	pipeline processing period (pg. 42)
$v_{x,y}$	the probability of transitioning from state x in a Markov chain to state y (pg. 82)
V_i	matrix representing the phases of service in the vacation probability transition matrix for processor i (pg. 75)
V_i^0	vector representing the end of service in the vacation probability transition matrix for processor i (pg. 75)
V_i'	full probability transition matrix representing the vacation of processor i (pg. 81)
z_m	chaotic variable at the m^{th} iteration (pg. 107)

Chapter 1

INTRODUCTION

1.1 Motivation

Optimizing the hardware and software of a computing system to fit the characteristics of a particular problem can be very beneficial in achieving performance increases of the computing system. However, the complexity of a design increases substantially for a system with less fixed design parameters. There is a great deal of benefit in using a framework for a computing system that fixes certain parameters that do not need to change to suit a particular application or class of applications. The framework allows the designer to focus the design effort on parameters that can significantly affect the performance of an application under development. Fixing certain parameters within a framework reduces the design complexity, but also reduces the design options available. Narrowing the scope of the applications to a particular class of application can allow the framework to be customized to suit the characteristics that are common throughout the class of applications.

Stream-oriented applications are a class of applications that can be well represented by dataflow graphs [BaGo05]. These applications usually have a large amount of data that needs to be processed by a relatively small number of tasks, with particular data dependencies between the tasks. Stream-oriented applications are common in multimedia applications such as JPEG and MPEG-4 encoders and decoders. The framework proposed in this thesis is specific to the stream-oriented class of applications.

Implementation of an MPSoC design can be very time consuming, and therefore analysis of the design before implementation is beneficial to determine if the desired performance (determined by some performance criteria that is specific to the problem) can be achieved, and to determine how the design could be changed to achieve better results. The use of a framework allows for more accurate analysis of a design, because the particular features of the framework can be modeled within the model of the design used for analysis. An analysis model of design also allows many variations of the design to be considered using optimization algorithms, resulting in automation of the design optimization.

This thesis is motivated by the need to develop complex computing applications in a short period of time through the use of a framework that reduces the design space, analysis that evaluates a proposed solution before implementation, and automated optimization of the solution. Therefore, this thesis strives to address the following research questions:

- Does the narrowing the scope of applications to a particular class of application result in an MPSoC framework that can be optimized for both performance and development time?
- Can an analytical model of an MPSoC system be used to predict performance of the system before implementation?
- Can combinatorial optimization techniques be used to automate the mapping and scheduling of tasks in an MPSoC system?

1.2 Thesis Statement and Objectives

The purpose of this thesis is to present an effective MPSoC framework that reduces the complexity of design for parallel computing systems targeted towards stream-oriented applications.

In the context of this purpose, the following thesis objectives are identified in order to address the research questions specified in the previous section:

- Definition and justification of the class of applications to which the framework targets;
- Definition of the components that make up the framework architecture, including the components that are fixed for all implementations of the framework and the components that require customization for each application;
- Discussion of the framework architecture in comparison to alternative architectures to demonstrate the strength and weaknesses of the framework architecture;
- Development of an analytical model, based on discrete-time Markov chains, that can be used to evaluate the performance of an implementation of the framework given application specific performance criteria;
- Automated task allocation of an implementation of the MPSoC framework for application specific performance criteria using combinatorial optimization algorithms, specifically chaotic simulated annealing, and the analytical model as a cost function to evaluate potential solutions;

- Verification of the objectives mentioned above through experimental results.

1.3 Organization of Thesis

This thesis consists of seven chapters. The organization of the remaining chapters is described below.

Chapter Two provides some background information and related work in the areas of computing models, parallel processing architectures, hardware/software partitioning, and computing frameworks. The chapter begins by giving a general overview of computing frameworks and the benefits of narrowing a framework to a specific class of applications. Then an overview of computing models is given, including the computing model chosen for the MPSoC framework, namely synchronous dataflow (SDF). An overview of parallel processing architectures is given in the context of the strengths and weaknesses for processing applications using a SDF computing model. This is followed by an overview of related work in the area of multiprocessing system frameworks. This background information provides the context for the remainder of the thesis by explaining some of the key concepts related to the MPSoC framework development and discussing the current state of the research in the field. The chapter is concluded by stating the unique contributions made by this thesis to the field of study.

Chapter Three describes the Multiprocessing System-on-Chip framework. This chapter describes the hardware and software architectures of the MPSoC and the scheduling scheme for the MPSoC. This includes detailed descriptions of the components that make up the framework, definition of the components that are fixed within the framework, and identification of the components that require customization for each

application of the framework. The implementation specific features of the MPSoC framework that are used for the experimental demonstration are described to show how the MPSoC framework could be implemented.

Chapter Four introduces the analytical model used to evaluate the performance of an implementation of the MPSoC framework. The analytical model is a stochastic model based on discrete-time Markov chains that can be used to evaluate the performance of an implementation of the framework for several different performance criteria.

Chapter Five introduces automated optimization techniques using chaotic simulated annealing. A brief overview of combinatorial optimization including discussion of alternative techniques is given followed by a detailed description of the simulated annealing and chaotic simulated annealing methods proposed for optimization of an implementation of the MPSoC framework. The simulated annealing and chaotic simulated annealing are compared and analyzed using experimental results.

Chapter Six describes the experiments used to verify and evaluate the MPSoC framework and the analytical model. The MPSoC was implemented on a Xilinx Virtex-II Pro FPGA [Xili07] using an example video processing application to demonstrate the benefits and tradeoffs of the framework and verification of the analytical model.

Finally, Chapter Seven gives concluding remarks about the MPSoC framework, analytical model, and optimization techniques, with discussion of the direction of future related research. The paper is concluded with a summary of the contributions made by this thesis. The contributions are expanded from those highlighted in Chapter Two to give a comprehensive list of the contributions made to the field of study.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 General Background on Computing Frameworks

For many years, data and signal processing applications have been implemented with either general purpose microprocessors or application specific integrated circuits (ASIC). General purpose microprocessors offered the benefit of being customizable for applications through the use of software without having to make application specific changes to hardware, but, generally, the hardware is not optimized specifically for the application. ASICs, on the other hand, have the benefit of being hardware customizable, specifically for the application to maximize performance, although at the price of very high development costs. In the past twenty years or so there have been many devices developed as an attempt to bridge the gap between customization and optimal performance. Digital Signal Processors (DSP), for example, have the ability to bridge the gap between general purpose microprocessors and ASICs for some types of applications. DSPs implement common signal processing functions in hardware blocks, which increases performance [Lee88b] [Lee89]. Software is then used for implementing the portions of the applications that are not common to most DSP applications. The cost of the increased performance of DSPs is that the scope of the applications to which they are well-suited is narrowed to only signal processing applications. Graphic processing units (GPU) also provide increased performance over the general purpose microprocessor and still allow for customization through software, but at the cost of narrowing the scope of applications to graphic processing applications or applications that have features similar

to graphics processing problems. Many scientific computing applications also benefit from the GPU architectures because of similar features of the problems [KEGS09].

Over the past decade, the multiprocessing system-on-chip (MPSoC) has been established as a unique branch of evolution in computer architecture that targets application specific embedded systems [WoJM08]. A wide variety of MPSoCs have been developed to provide solutions in networking, multimedia, signal processing, communications, and other applications. With the prevalence of field programmable gate arrays (FPGAs), designers can customize hardware with much lower development costs compared to ASIC development. However, development of complex applications entirely in hardware can still be a very large development effort. Soft processors are microprocessor cores that can be implemented in configurable hardware logic of an FPGA to provide software configurability benefits of a general purpose microprocessor (GPM), and, also, can easily be designed to have hardware features of GPM (such as particular peripherals, floating-point units, instruction pipelining) added or removed to fit a specific application. The development of soft processors has resulted in an increase of application specific multiprocessing systems ([CoHJ07], [JoLi96], [BeBB08], [RSJK05]). The design space for implementing a multiprocessing system-on-chip using soft processors is very large, and the development time for such a system, where the goal is to customize the system to maximize performance of a specific application, can be prohibitively long. Therefore, in the same way as the DSP and the GPU provide optimized platforms for specific classes of applications, there is a benefit in the development of a multiprocessing system-on-chip framework that reduces the design space for particular classes of applications. This can result in a solution that is more

computationally efficient than a general solution, and more flexible than an application specific solution. A class of problems can be defined by a common computing model. The design of the multiprocessing system can then be optimized to specifically consider only problems well-suited to particular computing model can allow for better performance of the system compared to a generalized parallel processing system.

2.2 Models of Computation

2.2.1 Overview of Models of Computation

A model of computation is a formal description of the computational behaviour of an application that is independent of the detailed design. A model of computation is composed by notation and by the rules for computation behaviour [BaGo05]. It is useful to express an application in terms of a model of computation to determine the structure of the application independent of the implementation, because an analysis of the computational model can result in better informed decisions. For example, it allows a designer to choose the implementation hardware and software that best suits the application, rather than forcing an application into hardware and software that is not necessarily the best fit.

Some common models of computation are finite state machines [Moor56][Meal55], discrete event models [BaGo05], synchronous/reactive models [BeBe91], Statecharts [Hare87], Petri nets [Reis85], and dataflow models [Lee91]. Each of these models of computation is not mutually exclusive from one another, and often an application can be well represented using several models of computation [LeSa98]. The

rules of a computational model specify a finite set of possible operations, and, as a result, certain types of models are better suited to certain problems than others.

Stream-oriented applications are data-dominated, meaning that they generally have large sets of data that need to be processed through a relatively small number of tasks, which do not change based on the data to be processed. Each task is executed for each data set processed. This differs from control-oriented applications where the tasks to be executed may differ depending on the data being processed. Computing models that determine the task to be executed based on some conditions (such as state machines, Petri-nets, etc.) are well-suited to these control applications. Dataflow models, on the other hand, are well-suited to applications where the order of tasks being executed does not change based on the data to be executed. Since stream-oriented applications are data dominated, and the MPSoC framework proposed in this thesis is targeting stream-oriented applications, the model of computation chosen for the MPSoC framework proposed in this thesis is a dataflow computing model.

2.2.2 Synchronous Dataflow Model of Computation

Stream-oriented applications have relatively large data sets which can be processed by a set of tasks, either serially or concurrently. Dataflow computing of stream-oriented applications is best modeled by a dataflow graph (DFG) [BaGo05]. A DFG has nodes that represent computational steps (i.e., tasks) in an algorithm, and edges that represent the movement of data. The DFG is unidirectional with no loops. All of the preceding tasks previous to a current task must have completed processing the data set

before the current task can begin processing the same data set. The DFG does allow for parallel computations, where data dependencies do not exist.

Fig. 2-1 shows an example of a DFG. As indicated, task C can execute at the same time that task E is being executed, but task H cannot start executing (on the same data set) until task D, F, and G are all finished. Multiple datasets can be processed through the DFG. This allows for the possibility of pipelining data through the graph, which would allow a new data set to enter the graph before the current data set(s) have left the graph. However, the succeeding data set can only be processed by tasks that have finished processing the preceding data set.

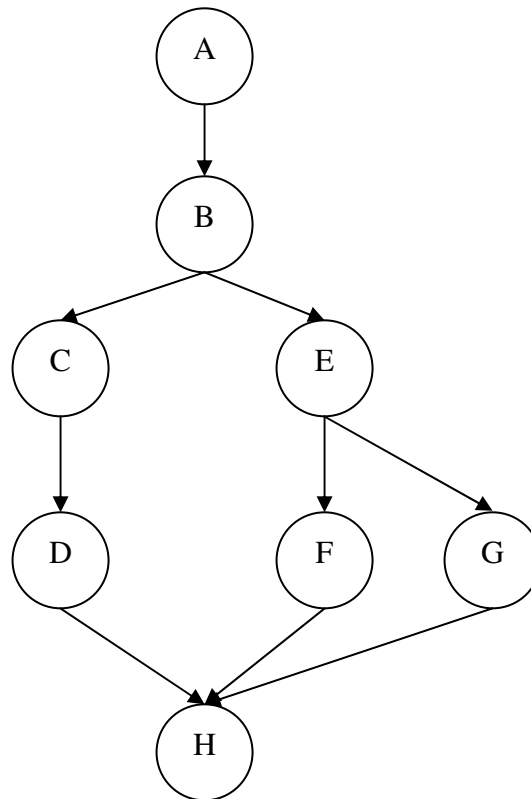


Fig. 2-1. An example of a dataflow graph.

A special case of data flow computing is synchronous dataflow (SDF). SDF has tasks that are represented by a DFG, where the tasks process a fixed amount of data in a

given processing period when the pipeline becomes full [LeMe87]. The tasks are statically scheduled (i.e., scheduled at design time). In the case where the task execution times are fixed, this produces a predictable processing period and therefore a predictable total program execution time.

Homogeneous SDF is a special case of SDF that places a limitation on each node such that it can only produce or consume one chunk of data per iteration (the size of a “chunk” of data is implementation specific). This limitation is often placed to limit buffer sizes between processing units implementing each node in the DFG, for hardware architectures where processing units are connected in a point-to-point manner with FIFO buffers between them. The MPSoC framework proposed in this thesis does not rely on specific buffers between processing units, and therefore the MPSoC framework is valid for both homogeneous and non-homogeneous SDF applications.

Fig. 2-2 shows an example of how the DFG in Fig. 2-1 can be processed using a SDF computing model by three processors in a pipelined manner. In this example it takes five processing periods to process one set of data, but a new set of data can start processing each processing period. This means that after the pipeline is full (i.e. after 5 processing periods in this case), a data set can be completed every single processing period rather than every five periods. So, in the example below, when the pipeline is full the processors would execute the following tasks:

- Processor 1 would process task A from data set $x+4$, then task C from data set $x+2$, and then task F from data set $x+2$.

- Processor 2 would process task B from data set $x+1$, then task G from data set $x+2$, and then task H from data set x .
- Processor 3 would process task E from data set $x+2$ and then task D from data set $x+1$.

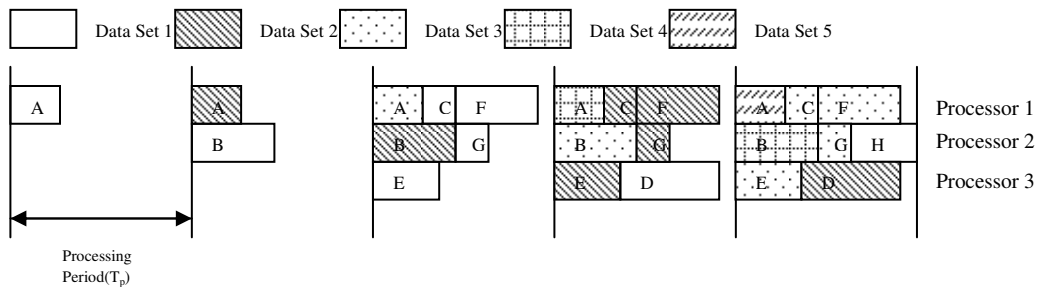


Fig. 2-2. An example of processing tasks from a dataflow graph in a pipeline.

For a large number of data sets, the average processing time of a data set will be the processing period (t_p), as shown in Fig. 2-2. There will be some additional time at the beginning to fill the pipeline, and some time at the end to empty the pipeline, but for large data sets this time is negligible. Therefore, to process a data set as fast as possible, the processing period should be reduced to as short a time as possible. Adding more processors allows for more tasks in the DFG to be executed in parallel which would ideally decrease the average processing time. However, there is a system cost to adding more processors. There is usually a particular performance increase that is needed in order to justify adding additional processors.

Even if a system has a fixed number of processors, there are many ways that the tasks can be arranged to result in different system behaviour. The number of pipeline stages, the allocation of a task to a pipeline stage, and the allocation of a task to a processor are all parameters of the system that can be varied to change the behaviour of the system, and can affect system performance.

2.3 Parallel Processing Architectures

The field of parallel processing architecture is a large area of study. There is a significant amount of literature discussing tradeoffs of different parallel processing architectures. This includes areas as diverse as supercomputing architectures to embedded systems-on-chip. This section will give a brief overview of the differences between possible parallel processing architectures within the context of application using a SDF computing model within systems-on-chip. The large research area of parallel computing architectures for supercomputing applications or interconnected stand-alone computers will not be discussed since it is out of the scope of this thesis. Considering that the SDF computing model divides an application into a number of tasks and describes the dataflow between the tasks within the DFG, any parallel computing architecture customized towards an SDF computing model will have a number of processors that are assigned a certain number of tasks, either statically (at design time), or dynamically (at run time), and there will need to be an interconnect network between the processors to share data as described by the DFG. The parallel processing architecture is mostly defined by the interconnection network that determines how information is passed from one processor to another. The three basic types of communication are:

1. Routed network communication where there is not a direct path from each processor to every other processor and information may be routed through intermediate processors. This includes many variations of a network such as mesh, binary tree, hypertree, and hypercube networks [Quin04].

2. Point-to-point communication where any processor can communicate with any other processor over dedicated communication medium.
3. Bus-based communication where any processor can communicate with any other processor over a shared communication medium.

Routed networks are often beneficial for multiprocessing systems that are passing a relatively small number of messages compared to the data being processed. Routed networks within systems-on-chip have been explored extensively for many different types of applications, often being referred to as a Network-on-chip (NoC) [LOCX05] [LuDM09]. However, for applications where a large amount of data needs to be sent relative to the data being processed, the overhead associated with routing the data can become significant. Stream-oriented problems inherently have a large amount of data that needs to be passed from one node to another in the DFG, meaning there must be a large amount of data routed from one node to another. An advantage of the SDF computing model in a multiprocessing system with routed network, is that the routes can be fixed at design time, and do not have to be dynamically created while the application is running because the tasks are statically scheduled (i.e. tasks are assigned to processors at design time). A disadvantage of the SDF computing model in a multiprocessing system with routed network is that the requirement to route data between processors essentially results in an extra DFG task, which would assign the forwarding function to intermediate processors that are not processing application data. Since there are generally a relatively small number of tasks in the DFG for a stream-oriented problem, the routed network does not offer any great advantage over the point-to-point network.

The major advantage of point-to-point communication is that there are no communication delays (due to communication channels being used for other communication) and no routing delays. The major disadvantage of point-to-point communication is that there needs to be a dedicated communication channel between each processor. In the case of the SDF computing model, the number of communication channels can be reduced to replicate the communication between tasks as shown in the DFG. Basically, the topology of the routed data is determined by the edges in the DFG. This is only the case if the tasks are allocated to microprocessors at design time (statically scheduled). One consideration of the point-to-point network is the size of the communication buffers. The communication buffers between processors must be customized to fit the data to be sent between each task through one iteration of the DFG multiplied by the number of pipeline stages. This means that the hardware is designed specifically for the data pipeline; therefore, there can be no dynamic adjustment of task scheduling or pipeline staging. While the point-to-point network may be an effective MPSoC architecture for applications using the SDF computing model, it requires a significant amount of hardware customization specific to the application DFG, or a fully connected point-to-point network, which is usually inefficient for more than three processors.

Bus-based networks have the advantage that any processor can communicate with any other processor; however, there is a disadvantage that the shared communication channel can become the bottleneck for the system. This is especially true if one processor is sending a message to another processor when the other processor is busy and, as a result, the shared communication channel is tied up waiting for the receiving processor to

become free to receive the message. This problem can be alleviated through the use of data buffers, where the data passed is stored in a buffer to be accessed by the receiving processor when it becomes available. Even with data buffers, there needs to be a method for each processor to know when there is data to be received. The data buffers can be extended to a global memory, where each processor does not communicate with each other processor directly; rather, it stores the output data from each task in the DFG to a global memory. Then the next task (or tasks) in the DFG that depend(s) on this data can access it from global memory, without needing to know which processor provided the data. A disadvantage of the bus-based network is the need for a method to signal the validity of data for processing. Tasks in a routed network or point-to-point network can determine when data is valid to process by when the data arrives from a preceding task. However, in a bus-based network the data can always be read even if it is not valid, so a method to determine validity of the data is required. Valid data can be marked with a stored flag for every data set (which would require additional memory), or a task scheduler can be used to signal when the data is available for tasks to execute.

The major benefit of a bus-based global memory MPSoC using a SDF computing model is that the basic hardware architecture does not have to be significantly customized to fit the application. Another advantage of the bus-based global memory system is that the tasks do not necessarily need to be statically scheduled, since the hardware architecture does not have to change to match the mapping and scheduling of the tasks to the processors. This thesis uses a hybrid pipeline scheduling technique for the MPSoC framework (described in detail in Section 3.1.3). This technique has statically scheduled

tasks, but allows for a dynamic pipeline stage and period, and is only possible with a bus-based global memory MPSoC architecture.

2.4 Related Work in Multiprocessing System Frameworks

This section gives an overview of some of the related work that has been done in the field of application specific MPSoC architectures and MPSoC frameworks.

Comparison of the related work to this thesis is made to highlight the contributions made in this work.

There has been much work in the area of customizing MPSoC architectures by using soft processors to fit the characteristics of an application. In these works, a very specific application is generally chosen, and an MPSoC hardware and software architecture is proposed to best suit the application. These papers show the benefit of an MPSoC for solving specific applications, but require a large amount of effort to customize a system from scratch for a specific application. One example of the development of an MPSoC for a specific application can be found in [RSJK05] where a soft multiprocessor system is developed and implemented in a Xilinx Virtex-II Pro FPGA to forward IPV4 packets. Their system was analyzed and compared with network processors (ASICs) designed for the same function. Another example can be found in [BeBB08] where an MPSoC system is developed in a Xilinx Virtex-II Pro FPGA using the Xilinx MicroBlaze soft processor [Xili08b]. Their system was designed specifically for a real-time control application. The MicroBlaze processors were connected directly together and passed messages between each other in a daisy-chained connection scheme. The approach taken in this thesis differs from papers like [RSJK05] and [BeBB08] in that

an MPSoC framework is developed that is not specific to an application, but rather to a class of applications.

In [LOCX05], a design methodology of MPSoCs, which are specific to real-time vision processing applications, is proposed. This paper targets a class of applications (defined as the class of real-time vision processing applications), rather than a specific application, but differs from a framework in that it does not specify features of the design that are common to application class, but rather proposes a method of development of MPSoC for applications that fit into this class. Two example architectures are chosen to demonstrate how the evaluation of the architectures could be achieved. One of the example architectures is a crossbar NoC, and the other is a bus-based architecture. The design method explores heterogeneous processors and heterogeneous memory systems. The allocation of tasks to processors is done in a heuristic fashion based on broad generalizations of the application, without any rigorous analysis and without consideration of inter-processor communications. [LOCX05] differs from the work in this thesis in that a framework, rather than a design methodology, is proposed. However, just as [LOCX05] sets out to define a design methodology and ends up defining portions of an MPSoC architecture, the definition of a framework with an analysis and optimization technique in this thesis, also largely maps out the design methodology to be used for implementation of the framework. The method of analysis and the task allocation optimization proposed in this thesis are more structured than the methods proposed in [LOCX05].

In [SLOW07], a heterogeneous MPSoC architecture for specific embedded computer vision algorithms is proposed. In this case, the number and type of processors in the architecture are not chosen, but left up to the designer. The algorithm is organized in an optical flow graph (essentially a dataflow graph). The processors are connected to a bus-based global memory. A simplistic calculation was used to determine the required global memory bandwidth of the system. This considers only the limiting case and ignores the complexities of differing tasks' memory access probabilities. [SLOW07] is similar to this thesis in that an MPSoC framework is defined; however, [SLOW07] defines the framework for a specific algorithm, rather than a broad class of applications, as was done in this thesis. This thesis also proposes a detailed analysis method and task allocation optimization technique as part of the framework, where these significant design activities are only touched on lightly in [SLOW07].

In [LuDM09] the performance of a cluster based MPSoC is evaluated. This paper describes an MPSoC framework where several processors are organized in clusters and routing of messages is done through a hierarchal bus-based NoC. Each processor in a cluster connects to a local memory bus to communicate with other processors in its cluster by writing to the cluster-local memory. A single processor in each cluster is responsible for routing messages to other clusters by writing to a global memory. A round-robin arbitration scheme is used to determine access to the local and global memories. Performance of the architecture was demonstrated with the implementation of a matrix-multiplication problem, and a Fast Fourier Transform (FFT) problem. The mapping of the algorithm to the processors and the scheduling of the tasks is not discussed, and is assumed to be done in a heuristic fashion specific to the application. The

architecture proposed in [LuDM09] is well-suited to divide-and-conquer type problems, i.e., problems where the data is divided up, and can be processed individually and reassembled. While there is a benefit to this architecture for divide-and-conquer types of problems, a shortcoming of the paper is that it does not mention that these are the types of problems that are well-suited to the hierarchal bus NoC, but rather alludes that the MPSoC architecture proposed is a good general solution. The example applications given are a matrix multiplication problem and an FFT problem. Both are problems that can divide up the data and be processed individually and then reassembled, so that the results are predictably good.

In [WuPe04], a design methodology for the development of a bus-based MPSoC specific to applications described in a DFG is proposed. This is specific to algorithms that are to be optimized for latency. Timing constraints are assigned to levels of the DFG, and tasks are scheduled in the order that they appear in the DFG. Access to global memory is done on a priority basis, based on the timing constraints assigned in the DFG. This paper states that the tasks are mapped to processors by analysis of the system, but does not state specifically what that analysis is, and so it is presumed to be based on some knowledge of the designer about the application tasks. A system controller is used to schedule tasks that are assigned to processors, but there is no mention of the scheduling technique used.

[WuPe04] has elements that are very similar to this thesis; for example, the framework based on applications that are described by DFGs, the bus-based hardware architecture, and the use of a hardware task scheduler. However, in this thesis, an analytical tool is developed to analyze the performance of the system in detail before implementation, and a method for optimization of task allocation is proposed, where these are left up to the

judgement of the designer in [WuPe04]. In addition, a novel method of scheduling is proposed in this thesis that is specific to the MPSoC framework, where the scheduling technique is not addressed in [WuPe04].

In [CoHJ07], a method was developed to produce a latency-optimized MPSoC, given a throughput constraint. The method targets applications that are stream-oriented and throughput constrained. The method requires the hardware architecture to consist of homogeneous processors that are connected together through point-to-point communication. Tasks are pipelined to exploit parallelism using static pipeline scheduling. A heuristic algorithm is used for mapping tasks to processors and scheduling the tasks. The main goal of the algorithm is to minimize communication between processors by mapping tasks that communicate with each other to the same processor where possible. One trade-off in this method of mapping is that it depends on homogeneous processing systems, since no consideration is made as to whether the tasks mapped to a given processor are well-suited to the strengths of this processor. The MPSoC framework proposed in this thesis is similar to [CoHJ07] in that applications that are represented in a DFG and are stream-oriented are targeted. However, the hardware architectures differ since a bus-based global memory architecture was used in this thesis, allowing for the hybrid pipeline scheduling proposed in section 3.1.3, whereas a point-to-point communication method was used in [CoHJ07], which means the scheduling must be static. Also, the MPSoC framework developed in this thesis can be used in both homogeneous and heterogeneous multiprocessor systems and homogeneous or heterogeneous global memories, while the architecture described in [CoHJ07] is specific to only homogeneous multiprocessing systems.

A method for mapping tasks organized in a DFG to processors within an MPSoC using a Bayesian Optimization Algorithm (BOA) is presented in [TBCP09]. [TBCP09] assumes a bus-based global memory MPSoC architecture. The instructions for each task are loaded from global memory into a processor's local memory during run time. This allows for tasks to be scheduled dynamically; however, this also requires a DMA (Direct Memory Access) controller to move data and a dedicated processor to be used only for scheduling of tasks. A simple analytical model is created to evaluate a mapping and scheduling solution with a task graph. Each task is divided into two categories, communication tasks and operational tasks. Communication tasks represent the time that instructions and data are communicated to a processor, and operational tasks represent the processing time of each task. It is assumed that all instructions and data are communicated at the beginning and end of each operational task. This model does not allow for the possibility of communication interspersed with processing during the execution of the task. Each communication task is characterized according to the amount of data communicated, and the communication medium delays; each operational task is characterized according to the execution time. A task graph showing the communication and operational tasks is created that is specific to a particular mapping, and from this, the performance of the task mapping can be determined. A BOA algorithm is used as an optimization algorithm to determine a good task mapping. BOA is a probabilistic model building genetic algorithm, a particular type of Genetic Algorithm (GA), where the standard mutation and crossover operators have been replaced by the construction and the sampling of a Bayesian network. The results are then validated with an experimental implementation of using Xilinx MicroBlaze processors and PowerPC processors in a

Xilinx Virtex-II Pro FPGA. The work in [TBCP09] is very similar in its objectives to this thesis, since in both cases, an analytical model of an application mapped and scheduled on an MPSoC is developed, and an optimization algorithm is used to automate the optimization of the task allocation. The major differences are that the architecture proposed in [TBCP09] is not proposed as an MPSoC framework, but rather as a very specific implementation of an MPSoC. Also, the analytical method used to evaluate particular solutions was overly simplistic in that all communication to global memory must occur at the beginning and end of a task, rather than allow for communication interspersed within the execution of a task. The analytical method proposed in this thesis allows for interspersed communication and operations. The analytical model proposed in this thesis also considers characteristics of the global memory arbitration method.

2.5 Thesis Contributions to the Field of Study

This thesis makes unique contributions to the MPSoC field of study. Through the review of related work discussed in Section 2.4, it is evident that there has been much research done in the area of MPSoCs. While other research has resulted in proposed MPSoC frameworks and task mapping and scheduling methods, these are generally treated as separate research problems, which do not typically integrate the development of the framework with analytical and optimization tools. A benefit of an MPSoC framework, in general, is to reduce development effort. In addition, associated analytical models to predict the performance of the system under different configurations, as well as optimization techniques can significantly reduce the development effort. This thesis is unique in that it addresses a more comprehensive picture; it develops a flexible

multiprocessing framework, which is integrated and supported by an analytical tool to measure the effect on performance under different numbers of multiprocessors, and, furthermore, a tool to determine an optimal number of processors. Other unique contributions of this thesis include the proposed hybrid pipeline scheduling (Section 3.1.3), the analytical model of the MPSoC using discrete-time Markov chain (Chapter 4), and the application of chaotic simulated annealing to the task allocation optimization specific to the proposed MPSoC framework (Chapter 5).

Chapter 3

MULTIPROCESSING SYSTEM-ON-CHIP FRAMEWORK DESCRIPTION

3.1 MPSoC Framework Architecture

The proposed MPSoC framework consists of both hardware and software architectures that may each be individually configured to fit a specific application. The following section defines the hardware and software architectures that make up the MPSoC framework, including defining which components of the hardware and software are fixed within the framework and which components are configurable to fit a specific application.

3.1.1 Hardware Architecture

The hardware architecture (Fig. 3-1) is a multiple processor system with a uniformly accessible global shared memory. This means that each of the processors in the system can access the global memory space to share data, and all processors have an equal average global memory access time. The terms uniform memory access (UMA) and non-uniform memory access (NUMA) are often used to categorize memory architectures in parallel processing systems [Quin04]. The MPSoC framework proposed in this thesis may be a UMA system or NUMA system depending on the implementation. While access to global memory is uniform, each processor has its own local memory as well. In the case where homogeneous processors are using the MPSoC, then the system is a UMA memory architecture. However, if the MPSoC framework is implemented with

heterogeneous processors where different processors have different local memory access times, the system would be categorized as a NUMA memory architecture.

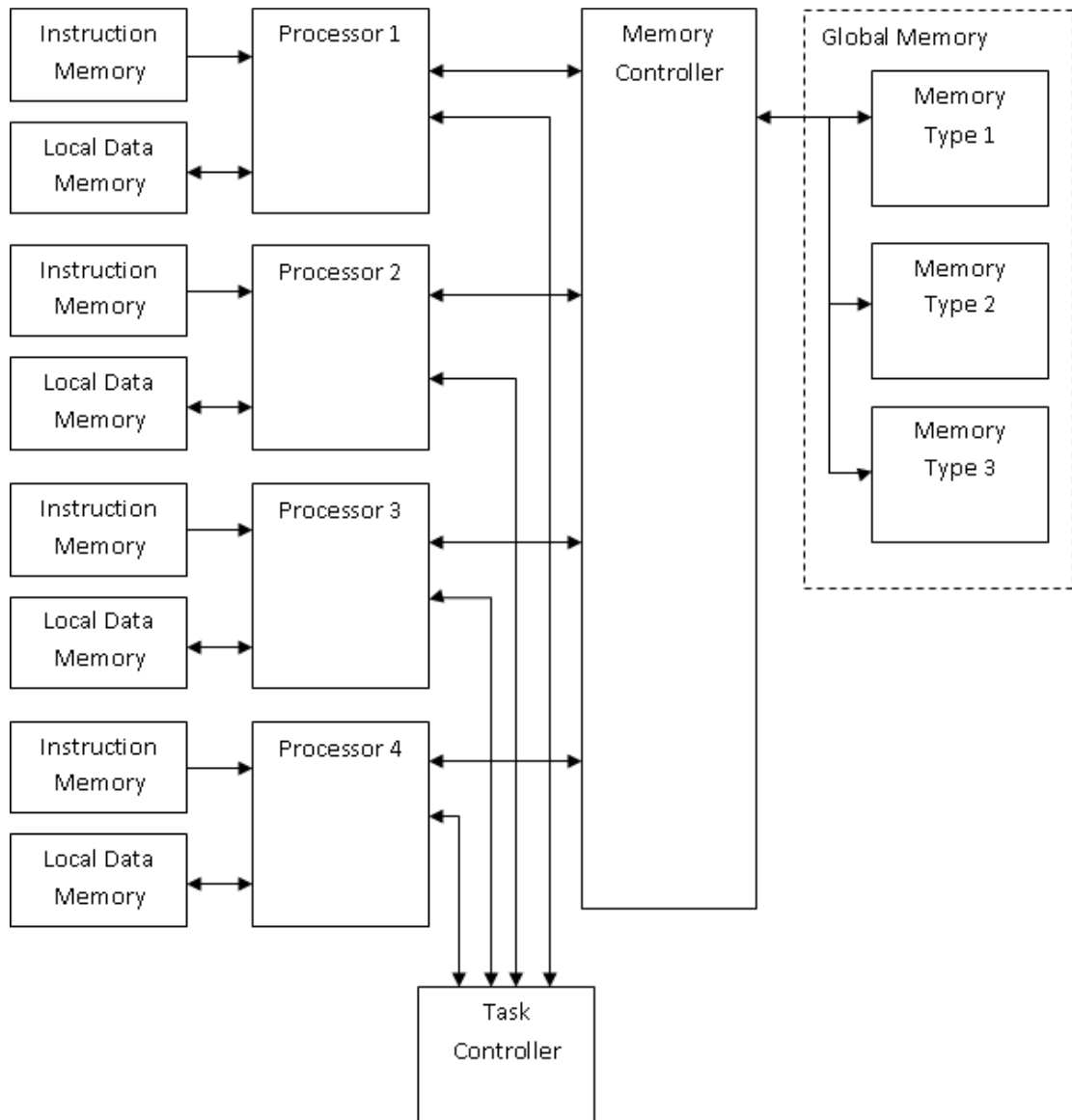


Fig. 3-1. Block diagram of MPSoC hardware architecture.

Fig. 3-1 shows the hardware architecture of the MPSoC. The main components of the system are the soft processors (including local data and instruction memories), global shared memory, memory controller to control access to the global memory, and task controller. The diagram shows four processors and three memory types; however, the

number of processors and the number of different memory types that make up global memory can vary depending on the needs of the application.

3.1.1.1 Soft Processor

A soft processor is implemented in configurable logic within an FPGA. The framework described here is not specific to any particular soft processor. The MPSoC framework does not require a homogeneous multiprocessor system, meaning that all the processors in the system do not have to be the same type. The main benefit of a homogenous multiprocessor system is the simplicity of the design. If the same type of processor is used for all the processors in the system, the main design decisions are determining the number of processors to include in the system, and the scheduling and allocation of tasks to the processors. A design with a heterogeneous multiprocessor system is more complicated because, in addition to the design decisions for a homogeneous system, it also needs to be determined which different types of processors should be included, how many of each kind, and what the differences are between the processors. Differences between the processors in a heterogeneous multiprocessing system can be relatively small, such as different peripherals or local memory sizes, or can be very large, such as entirely different processing architectures. While a heterogeneous processing system can be more complicated, there can be major advantages to a heterogeneous processing system for some applications. The advantage is most significant tasks with particular characteristics can be mapped to processors that are well-suited to tasks with those characteristics. For example, if it was the case that a few tasks in an application required significant digital signal processing, but the other tasks did not

require this processing, then it would be advantageous to have a Digital Signal Processor (DSP) as one of the processors in this system to execute the digital signal processing tasks, but have other processors better suited to the other tasks in the application. This would result in a better performance solution than using DSPs for all tasks or using general processors for all tasks.

Each of the processors requires its own local data and instruction memory, which allows each processor to run an independent program. Within Flynn's taxonomy, this system is defined as a multiple instruction, multiple data (MIMD) system [Flynn72]. The local instruction memory contains the entire program for each of the processors, and the local data memory is used for calculations while executing tasks.

All processors in the system will be assigned a subset of the DFG tasks to be executed, and the execution of the tasks will be scheduled by the task controller. One of the processors in the MPSoC framework is assigned additionally as the head processor. The head processor is used to execute any serial code that may be required before and/or after the DFG tasks are executed. For example, in an image compression application, the head processor may need to open a file before the DFG tasks commence; furthermore, upon completion of the DFG tasks, the head node may be required to close the file. Opening and closing of the file only need to happen once respectively, so they do not fit within the DFG. However, during execution of the DFG, there is no distinct difference between the head processor and the other processors in the system.

3.1.1.2 Task Controller

The task controller is the scheduler for the system. It has specific knowledge of the application, so it is customized for each application. The task controller is implemented entirely in hardware, independent of the soft processors. This allows the task switching overhead to be much less than it would be if the scheduling was done in software by a processor. The DFG task dependencies are entered as parameters in the task controller hardware description language. Also, the task allocation for each processor in the system is specified in the hardware description of the task controller. This allows the task controller to control which task should be executed on which processor.

The task controller uses the following signals to control the execution of the tasks on the processors. There is a set of these signals for each processor in the system.

- `executing` – Each processor has an “executing” signal that is sent to the task controller. This signal is set by the processor when it is currently executing the most recently assigned task.
- `start_exec` – A “start_exec” signal is sent from the task controller to each processor. This signal is used to notify a processor when it should start executing a task.
- `DFG_node` – A “DFG_node” bus is sent from the task controller to each processor. This bus is timed with the “start_exec” signal, so that they are both active together.
- `iteration_num` – An “iteration_num” bus is sent from the task controller to each processor. The width of the “iteration_num” bus is implementation

specific; it is not part of the generic MPSoC framework definition. How the `iteration_num` is used by each processor is specific to the function of the executing task. For example, it may be used as an index into global memory to specify the location of the current data set.

- `proc_done` – A “`proc_done`” signal is sent to each processor. This signal is used when all the data sets that are to be executed by a processor are finished. This is used to signal to a processor that all of its work is done.

The following diagram shows a block diagram of the interface between the task controller and one of the processors in the system. This interface is repeated between the task controller and each processor in the system.

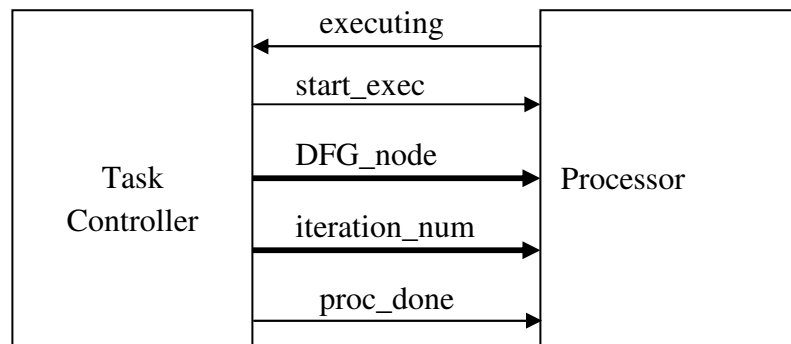


Fig. 3-2. Interface between the Task Controller and soft processor

The head processor will also have a “`start_task_control`” signal that is used to enable the task controller. This is not repeated for all processors, but is only available to the head processor. This signal is made active when the task controller should start operating. The purpose of this is to allow the head processor to execute pre-DFG tasks before the DFG tasks are executed, and then execute post-DFG tasks that may be required after the DFG tasks are finished executing.

The system designer determines which tasks are executed by which processor, but the task controller will automatically determine how many pipeline stages are required by using the task dependencies to determine which task can be executed. The pipelining technique is a novel approach that can result in dynamic pipeline stages. This pipeline scheduling method is a hybrid between static pipeline scheduling and dynamic pipeline scheduling. The hybrid pipeline scheduling technique is discussed in more detail in 3.1.3.

3.1.1.3 Task Controller Interface Peripheral

The task controller interface peripheral is a custom peripheral for each processor that is used to make the signals between the task controller and the processor available to the software executing on the processor. The generic MPSoC framework requires that a task controller interface peripheral exists, but the detailed design of the peripheral is implementation specific because it depends on the type of processor used in the system. Each processor in the system will have its own task controller peripheral to interface with the task controller.

3.1.1.4 Memory Controller

The memory controller is used to control each processor's access to global memory. The interface between each processor and the memory controller is an asynchronous interface. This means that there are a few handshaking signals that are required for both the memory controller and the processor to acknowledge communication between them. The asynchronous interface allows the memory controller to operate at an entirely different clock speed than each of the processors. This is significant for heterogeneous systems, where there may be several processors running at

different clock speeds. Since the memory controller has an asynchronous interface with the processors, the different clock speeds of the different processors are supported.

Only one of the processors in the system can access global memory at a time. The memory controller controls accessibility to global memory to ensure that more than one processor does not try to access global memory at the same time. The memory controller determines which processor can have access to global memory by individually polling each processor in the system to see if it is requesting to access global memory. If a processor that is currently being polled is requesting access to global memory, then the memory controller will service the memory request. When the memory request is finished being served, the memory controller immediately moves its attention to the next processor in the system to see if it is requesting memory access. The memory controller will only wait one clock cycle for each processor to have a memory request, and if there is no memory request, the next processor is polled. Therefore, if there are four processors in the system and none of them is currently requesting memory access, the memory controller will cycle through each of the processors in four clock cycles.

Each processor has the following control signals that are used to interface with the memory controller:

- Address bus – This bus specifies the address of the location in global memory that the processor would like to access. The size of the memory is implementation specific and is not defined as part of the generic MPSoC framework.

- Data bus – This is a bidirectional bus that contains the data that is to be read from or written to global memory.
- mem_request – This signal is active when the processor wants to access global memory. This is the signal that the memory controller polls to determine if there is an active memory request.
- read_write – This signal is used to specify whether the requested memory access is a read or a write. The signal is low if a read is requested, and high if a write is requested.
- ack_in – This is an acknowledge signal sent from the processor to the memory controller. This is used as part of the handshaking scheme used to acknowledge to the memory controller that data read from the global memory was received by the processor.
- ack_out – This is an acknowledge signal sent from the memory controller to the processor. This is used by the memory controller to signal to the processor that the memory request has been served.

Fig. 3-3 is a block diagram showing the interface between the memory controller and one processor. This same interface is repeated between the memory controller and each processor in the system.

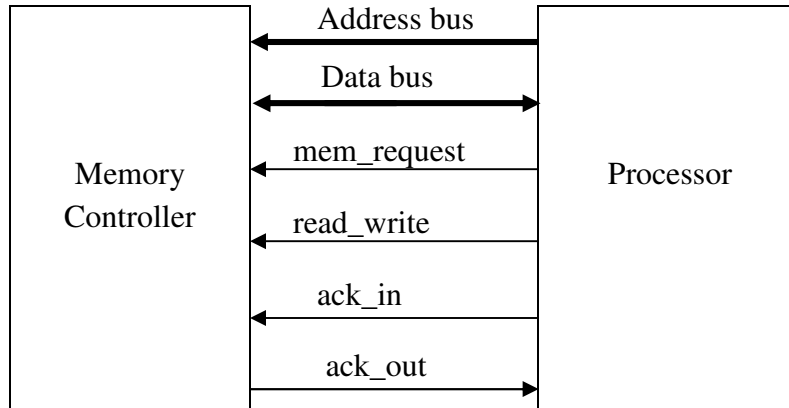


Fig. 3-3. Interface between the memory controller and a single processor.

The memory controller also interfaces with the global memory. The memory controller can be configured to interface with any type of memory, or even several different types of memory in the same system. The interface between the memory controller and the global memory is implementation specific and it is not defined as part of the generic MPSoC framework. Since the interface to the global memory is implementation specific, both synchronous and asynchronous memory interfaces could be supported. Fig. 3-4 expands on Fig. 3-3 to show the interface between the memory controller and global memory. There is only one interface between the memory controller and global memory; it is not repeated as is the case with the interface between the processor and the memory controller.

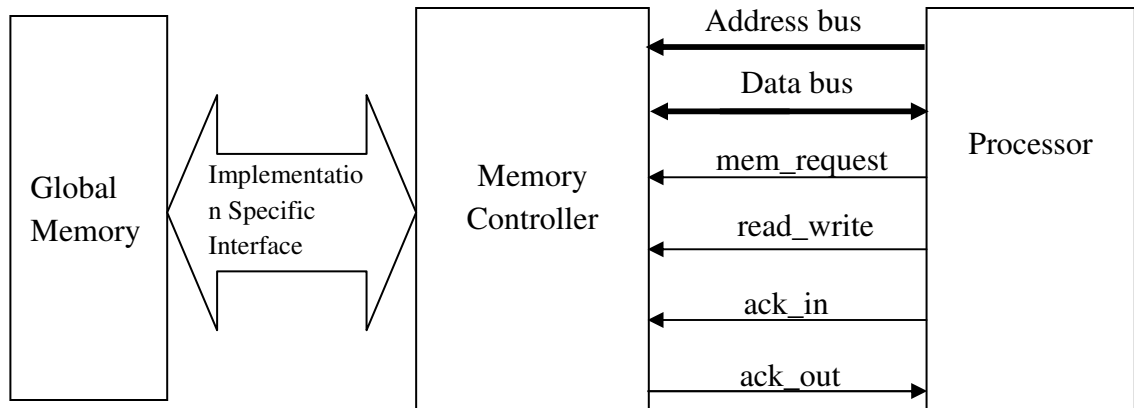


Fig. 3-4. Interface between the global memory, the memory controller, a single processor.

3.1.1.5 Memory Controller Interface Peripheral

The memory controller interface peripheral is a custom peripheral for each processor that is used to allow the software running in the processor to access global memory over the asynchronous global memory interface. The generic MPSoC framework requires that a memory controller interface peripheral exists, but the detailed design of the peripheral is implementation specific because it depends on the type of processor used in the system. Each processor in the system will have its own memory controller peripheral to interface with the memory controller.

3.1.1.6 Global Memory

The generic MPSoC framework requires that a global memory exists, but the type and size of the global memory is implementation specific. Some applications may require a large amount of data in which case an external RAM chip would be appropriate. For applications that do not require a significant amount of memory, the memory blocks built into the FPGA may be sufficient. The global memory could also be made up of several different types of memories, depending on the needs of the application and the memory available in the target system.

3.1.2 Software Architecture

The software architecture is defined by the MPSoC framework. The software architecture describes the structure of the program to be run on each processor. This structure varies slightly between the head processor and the other processors.

3.1.2.1 *General Processor Program Structure*

The general program structure for any processor in the system that the following steps:

1. Run any initialization code that may be required for processor initialization.
This is specific to the particular processor used.
2. Continuously poll the “start_exec” flag and the “proc_done” flag sent by the task controller until either is active. If the “start_exec” flag is active go to step 3, if the “proc_done” flag is active exit the program.
3. Read the “DFG_node” bus to determine which task should be executed.
4. Read the “iteration_num” bus to determine the task iteration number.
5. Set the “executing” flag active.
6. Execute the assigned task.
7. Set the “executing” flag inactive and go to step 2.

The only step listed in the above algorithm that is specific to the application is step 6. This means that all of the other steps in the general program structure are implemented with code that is specific to the processor used, and would not have to be

rewritten for different applications. A flowchart of the general program structure is shown in Fig. 3-5.

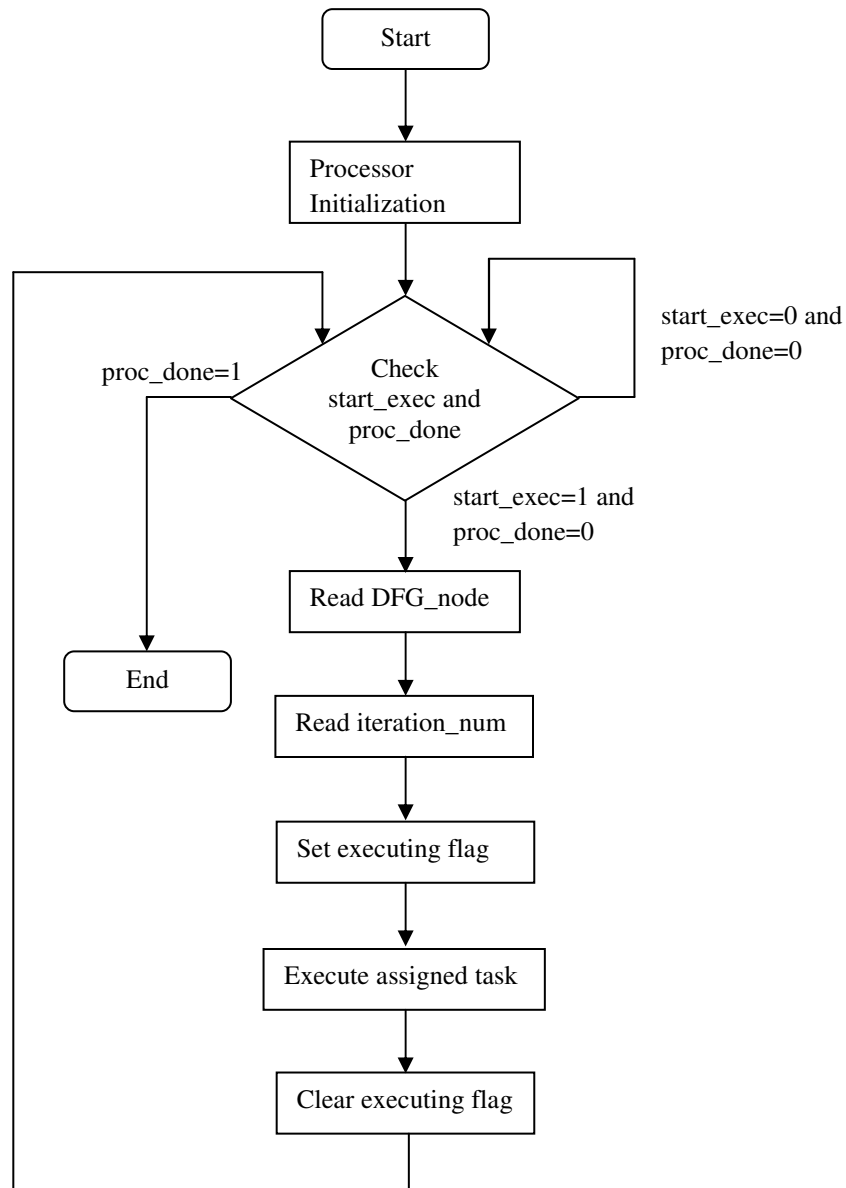


Fig. 3-5. A flowchart of the general program structure.

3.1.2.2 Head Program Structure

The head processor program structure is almost identical to the general processor program structure with the exception that there may be some pre-DFG processing and post-DFG processing required, and the head processor is responsible for initializing the task controller to start DFG task scheduling. Fig. 3-6 shows the a flowchart of the software structure for the head processor.

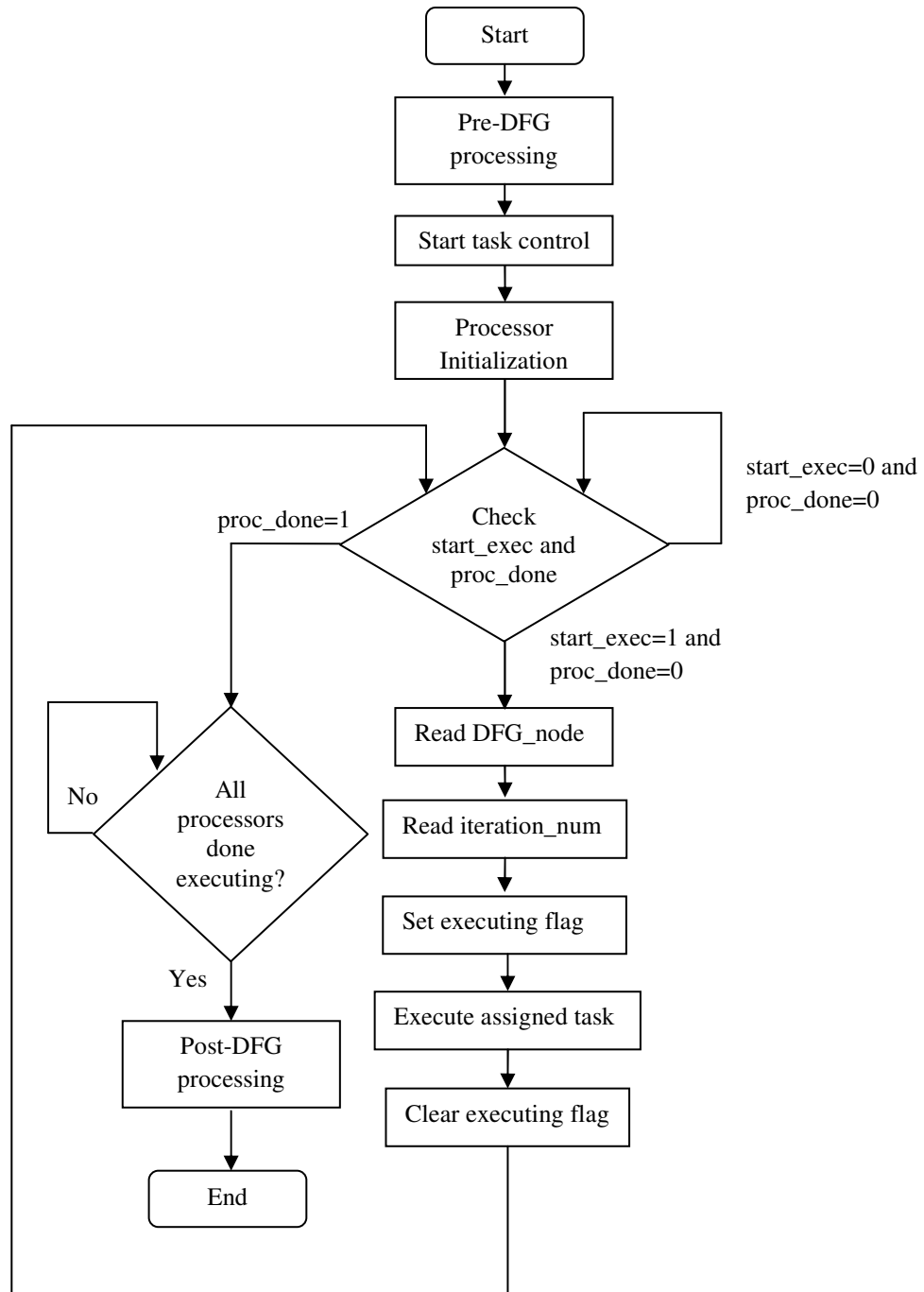


Fig. 3-6. A flowchart of the head processor program structure.

3.1.2.3 Software Global Memory Access

The memory controller ensures that only one processor can access global memory at a time. However, the memory controller does not control the execution of software;

therefore, when a memory request is made it is essential to have a software strategy in place to determine what happens when the program is waiting to access memory. The software strategy for global memory accesses defined by the MPSoC framework is to halt program execution until the memory access is finished. This means that, when a global memory access is requested, the program will wait until the memory access is finished before continuing execution. This is a simple and safe strategy that ensures that the expected values exist in memory. However, the safety and simplicity of this strategy comes at the price of extended processing time. The program is essentially frozen when a memory request is made and other processors are accessing memory. A more complex strategy could be developed to allow for a memory request to be made and then continue with other processing while the memory request is being serviced. This type of strategy would need to have methods to ensure the validity of data. This is an area of possible improvement that could be added to the MPSoC framework in the future.

3.1.3 Hybrid Pipeline Scheduling

The overall functionality of the task controller is described in section 3.1.1.2, but the details of pipeline scheduling algorithm implemented by the task controller are discussed in this section. The task controller implements a novel pipeline scheduling algorithm that is a hybrid between traditional static pipeline scheduling and dynamic pipeline scheduling. Static pipeline scheduling involves allocating the tasks to the processors at design time, and also determining the schedule in which the tasks will be executed. This approach is used for synchronous dataflow execution, which is demonstrated in section 2.2.2. Dynamic pipeline scheduling involves dynamically

assigning tasks to available processors during run-time. Dynamic scheduling has been applied to many different fields of study, and implemented with many variations [LePa95]. Most notably there have been many variations of dynamic pipeline scheduling applied to instruction pipelining within computer architectures. True dynamic scheduling has not been successfully applied to task scheduling because the overhead involved in determining optimal solutions during run-time is too large to be feasible. Dynamic scheduling is difficult because the system needs to be analyzed in real-time to determine an optimal task scheduling and mapping. Chapters Four and Five in this thesis discuss the analysis and optimization of the MPSoC framework for a particular application, but the analysis and optimization proposed in this thesis is intended to be performed at design time. In order for dynamic scheduling to be successful, an analytical model and optimization technique that could be performed in real-time by the task scheduler would be required. Hybrid algorithms that apply some elements of static pipeline scheduling and some elements of dynamic scheduling have been proposed and have been applied to task scheduling with some success [HaLe97]. The pipeline scheduling algorithm implemented as part of the proposed MPSoC framework is a novel hybrid pipeline scheduling algorithm that implements elements of both static and dynamic pipeline.

The element of static pipeline scheduling that applies to this hybrid method is that the allocation of tasks to processors is done at design time. The tasks are also executed in the order that was determined at design time, but the entire list of tasks may not be executed in each pipeline period. The number of pipeline stages are not predetermined at design time or fixed throughout execution, as would be the case with static scheduling.

Static pipeline scheduling defines a fixed pipeline period (T_p) that is repeated. Within each pipeline period, each task has a particular time slot assigned to it in which the task is executed. Each processor has a defined set of tasks to be executed in each pipeline period. When processing first starts, the first λ pipeline periods are required to fill the pipeline, where λ is the number of pipeline stages. This means that some tasks may not be executed in the first few pipeline periods. This is because the results of the initial tasks must be available for any tasks that depend on those results. This pipeline filling is shown in Fig. 2-2, where all the tasks are not executed until the fifth processing period. However, a static pipeline period can only be fixed if the execution times of the tasks are fixed. Typically there is a variation in the execution times of the tasks, due to causes such as different branch paths that can be taken in the program execution, or variable memory access times. Because of the variable execution times, there may be times where the pipeline period is extended because one task cannot start to execute until a task it depends on has finished executing. The strategy of inserting delays to account for variable task execution time is called quasi-static scheduling and was first proposed by Lee [Lee88a]. The following figure shows an example of this, using the DFG example from Fig. 2-1 and the pipeline task allocation shown in Fig. 2-2. This figure shows the ideal pipeline period on the left, and a pipeline period on the right where task E had a longer than expected execution time and the quasi-static scheduling was applied. All other task execution times are the same.

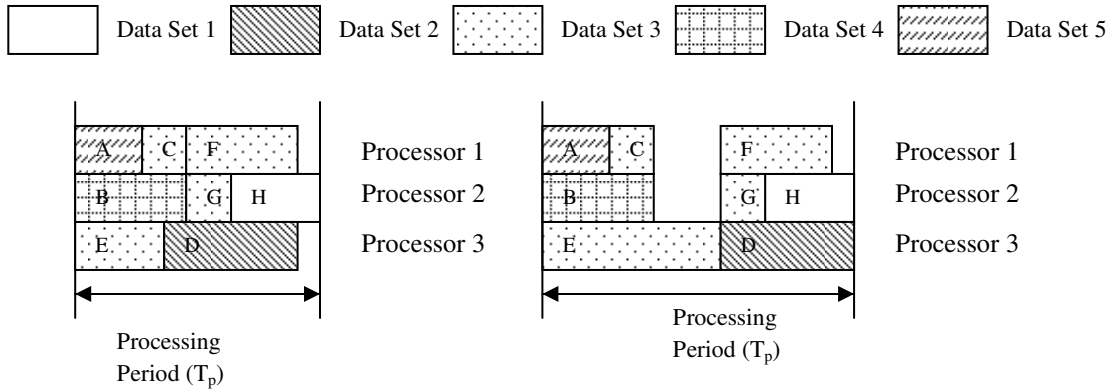


Fig. 3-7. Traditional pipeline with variable task time.

Since tasks E, F and G are all executing on the same data set in the same period and tasks F and G depend on the results of task E, Processor 1 and Processor 2 have to wait until task E is finished before F and G can execute. This causes Processor 1 and Processor 2 to be idle when they could otherwise be executing a task.

The new pipeline scheduling algorithm developed for the task controller in the MPSoC framework reduces the idle times significantly by changing the number of pipeline stages to ensure that, if there are any tasks assigned to a processor that can be executed, they are executed, rather than waiting for other tasks to finish. There is no longer a global processing period (T_p); now each processor will have its own processing period, that is independent of the other processors.

The following parameters for each processor are given to the task controller during initial design of the system:

- β_{np} – This is the number of tasks the processor has assigned to it.
- N_L – This is a list of tasks that the processor has been assigned to execute. The i^{th} task in the list will be denoted as $N_L[i]$.

- M_L – A list of the preceding tasks for each task in the DFG. The preceding tasks for a specific task are all of the tasks that directly output into the specific task. The j^{th} preceding task for the i^{th} task in the list will be denoted by $M[i][j]$.
- K_{ds} – The total number of data sets to be executed by the tasks in the DFG.

The task controller also keeps a list for each processor containing the number of times that each task in the DFG has been executed. This list will be called Q , where $Q[i]$ is the number of times that the i^{th} task has been executed.

Given the above input parameters the dynamic pipeline scheduling algorithm for each processor in the system can be described as follows:

```
// assign the first task in the list that has not completed all of its iterations
pipeline_index ← minimum pipeline index i where Q[ NL[i] ] < Kds

// while there are still tasks that have not finished their iterations
while Q[ NL[i] ] < Kds for any task i that is in NL
{
    // if the predecessors to the current task have more iterations complete than the current task, then
    // execute this task, otherwise skip over it to the next task
    if Q[ NL[pipeline_index] ] < Q[ ML[ NL[pipeline_index] ][j] ] for all j then
    {
        Allow task NL[pipeline_index] to be executed
        Wait until task NL[pipeline_index] is finished executing
    }
    // move to the next task, and if the end of the list is reached, then start again at the beginning
    pipeline_index ← pipeline_index + 1
    if pipeline_index = βnp then
    {
        pipeline_index ← minimum pipeline index i where Q[ NL[i] ] < Kds
    }
}
}
```

This algorithm is essentially scheduling each task assigned to a processor in order of the list N_L , until it either gets to the end of the list, or it runs into a task that cannot be scheduled because it is waiting for one of its preceding tasks to finish executing with data from the same data set. When the task controller runs into a task that cannot be executed because its predecessors have not yet finished executing tasks from the same data set, then that task is skipped, and the next task in the list is examined. When the task controller reaches the end of the list, it goes back to the beginning of the list, or to the first task in the list that has not already finished executing all of its data sets when the task at the beginning of the list has finished all of its data sets.

With this algorithm, a processor will never sit idle if there are any tasks in its list that can be executed (i.e. tasks that are not held up by other task dependencies). Using the example shown in Fig. 3-7, the hybrid pipeline scheduling algorithm would adapt to the variable length of task E as shown in Fig. 3-8. Now instead of Processors 1 and 2 waiting idle until task E is finished, the next data set of A and C are executed on Processor 1, and the next data set for task B is executed on Processor 2. This essentially increases the number of pipeline stages for the system, but will result in a shorter total execution time for the entire system.

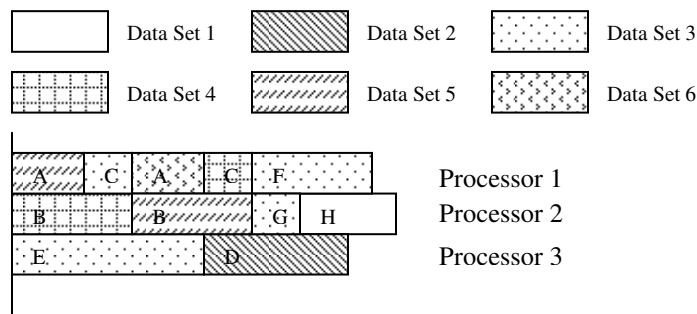


Fig. 3-8. Dynamic pipeline scheduling with variable task time.

This scheduling algorithm also has advantages when the pipeline is initially filling up because the pipeline period is not fixed, thus eliminating the effect of wasted execution time when there are no tasks to be executed. This can be seen by the next figure that shows what would happen on initial task execution if hybrid pipeline scheduling was used instead of static pipeline scheduling for the example from Fig. 2-2.

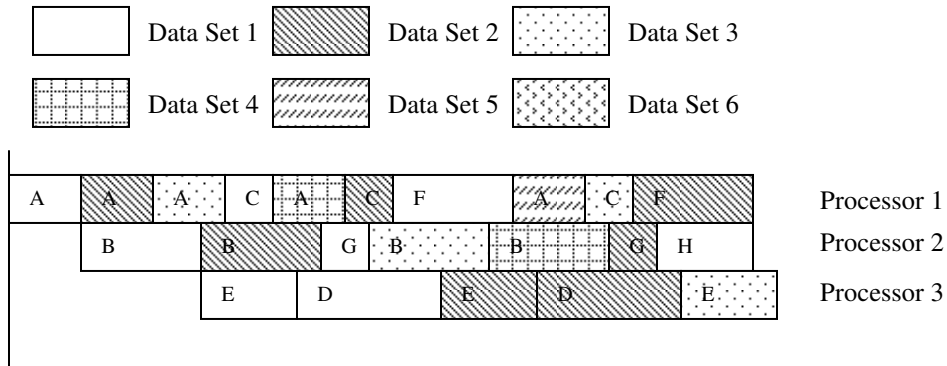


Fig. 3-9. Filling the pipeline with hybrid pipeline scheduling.

This hybrid pipeline scheduling causes much less processor idle time than (quasi)static pipeline scheduling. In this example, it can be seen that tasks without any dependencies (like task A) will tend to get farther ahead of other tasks with dependencies in the number of data sets executed compared to static pipeline scheduling. However, this is not really a problem when the goal is to finish all tasks as quickly as possible because, when A finishes all of its data sets, it will just stop being scheduled, and the other tasks will catch up. This could become an issue for applications where latency is critical. Latency is defined as the time it takes for a data set to be executed by all nodes in the DFG, from the time that the first task in the DFG executes using the data set, to the time that the last task in the DFG executes, using the same data set. In this case, the hybrid pipeline scheduling algorithm proposed here could make the system latency larger than

with traditional pipelining. An example in which delays would not be desirable would be with a video conferencing application, where delays could impact the flow of the conversation due to problems with the synchronization of the audio and video that may result if the number of pipeline stages in the DFG gets too large. However, in applications where the maximum throughput is the goal, and latency is not significant, then the hybrid pipeline scheduling algorithm will always perform at least as good as (quasi)static pipelining, and, most often, it will perform better.

The hybrid pipeline scheduling that has been implemented in this model is really only feasible in this MPSoC framework because of the global shared memory. The global shared memory allows the number of pipeline stages to grow indefinitely because the data that the following dependant tasks require is not stored in a temporary buffer that can overflow, but rather in global memory. In the case of the point-to-point communication used in [CoHJ07], the hybrid pipeline scheduling algorithm used here could cause the communication buffers to overflow. In that hardware architecture, the data passed from task to task is stored in communication buffers that are sized according to the dependency distance between the tasks, which is determined at design time.

3.2 MPSoC Experimental Implementation

In order to demonstrate the MPSoC framework developed, the framework was implemented in a Xilinx Virtex-II Pro FPGA, on the Xilinx XUPV2P development platform [Xili08a]. This section explains how each of the application specific components was designed for the MPSoC framework implementation.

3.2.1 Soft Processor Implementation

The soft processors used for the implementation of the MPSoC framework were Xilinx MicroBlaze [Xili08b] soft processors. The MicroBlaze processor is a Reduced Instruction Set Computer (RISC) with 32-bit instructions. The MicroBlaze processor is optimized for use within Xilinx FPGAs, and is designed to be easily customized to fit a variety of different applications. A number of features can be added to the processor to increase functionality, or removed to reduce FPGA resources, such as the floating point unit (FPU), local memory bus interface, hardware divider and multiplier, a memory management unit (MMU), and many other features.

3.2.2 Task Controller Implementation

For the MPSoC framework implementation, the task controller was implemented using the hybrid scheduling method described in section 3.1.3. The DFG_node bus was implemented as an 8-bit bus sent to each processor, which means that the DFG is limited to 256 tasks. For other implementations where more tasks are required, the width of the DFG_node bus could be increased. The iteration_num bus was implemented with a 16-bit bus that specifies the number of times that the current task has been executed previously. The 16-bit bus width means that the number of iterations is limited to 65535; if more iterations are required, the width of the bus could be increased.

3.2.3 Task Controller Interface Peripheral Implementation

The task controller interface peripheral between the task controller interface peripheral and the processor is the Xilinx On-chip Peripheral Bus (OPB) [Xili06]. The OPB is a standard bus interface provided by Xilinx for the MicroBlaze soft processors.

The OPB can be used to interface with a memory-mapped peripheral of the MicroBlaze processor. The task controller interface peripheral allows the signals sent by the task controller to be read and written by addressing particular registers that can be accessed by software execution. This peripheral is the interface between the task controller hardware, and the software executing on the MicroBlaze processors.

Fig. 3-10 extends Fig. 3-2 to show how the task controller interface peripheral interfaces with the task controller and the MicroBlaze processor core, for one processor. These interfaces are repeated for each processor in the system.

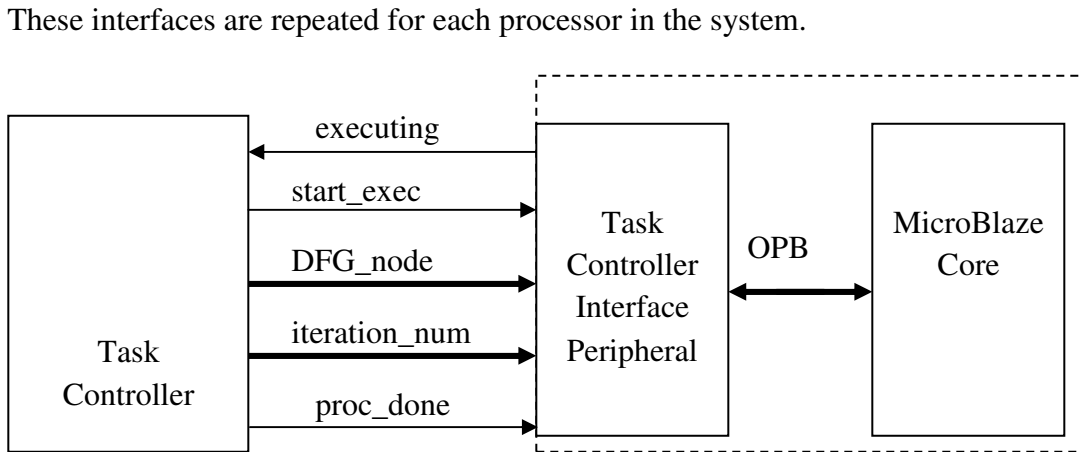


Fig. 3-10. Implementation of the task controller interface peripheral controller.

3.2.4 Memory Controller Implementation

In the MPSoC framework implementation, the interface between the memory controller and the global memory is a synchronous interface. This means that the memory controller provides a clock signal to the global memory, which is acted upon on each rising edge of the clock. A synchronous interface is generally faster than an asynchronous interface, but a common clock signal is required for both the memory controller and the global memory. The signals between the memory controller and the global memory are as follows:

- Address bus – This bus specifies the address in global memory that is currently being addressed.
- Data bus – This is a bidirectional bus that contains the data read from and written to global memory.
- read_write – This signal is used to specify whether the current memory access is a read or a write. The signal is low if a read occurs, and high if a write occurs.
- chip_enable – This signal is used to specify whether a memory access is currently active. If this signal is high, then the address, data, and read write signals are valid, and the memory should be accessed. If this signal is low, the memory is inactive.
- clock – This is the shared clock signal between the memory controller and the global memory. The global memory evaluates the signals on every rising edge of the clock signal, and acts accordingly.

Fig. 3-11 expands on Fig. 3-4 to also show the implementation specific signals used in the MPSoC framework implementation.

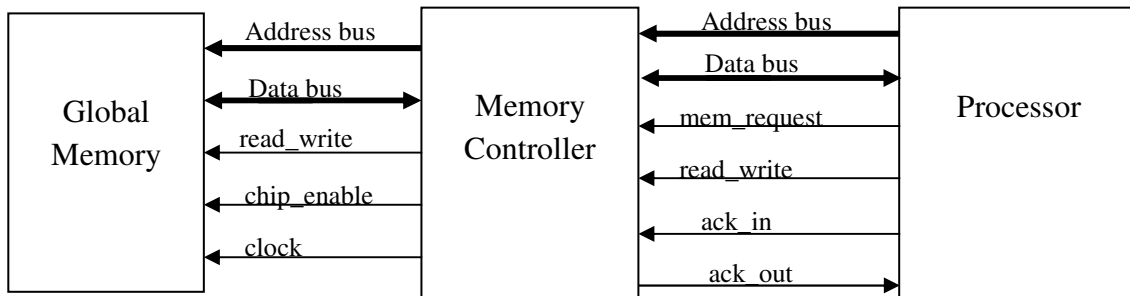


Fig. 3-11. Implementation specific interface to global memory.

3.2.5 Memory Controller Interface Peripheral Implementation

In the MPSoC implementation, the interface between the memory controller interface peripheral and the processor is the Xilinx OPB, just as was done with the task controller interface peripheral. Using the OPB allows the peripheral to be a memory-mapped peripheral of the MicroBlaze processor. The memory controller interface peripheral allows the signals sent by the memory controller (which were described in the previous section) to be read and written, by addressing particular registers that can be accessed by software execution. This peripheral is the interface between the memory controller hardware, and the software executing on the MicroBlaze processors.

Fig. 3-12 extends Fig. 3-3 to show how the memory controller interface peripheral interfaces with the memory controller and the processor core for one processor.

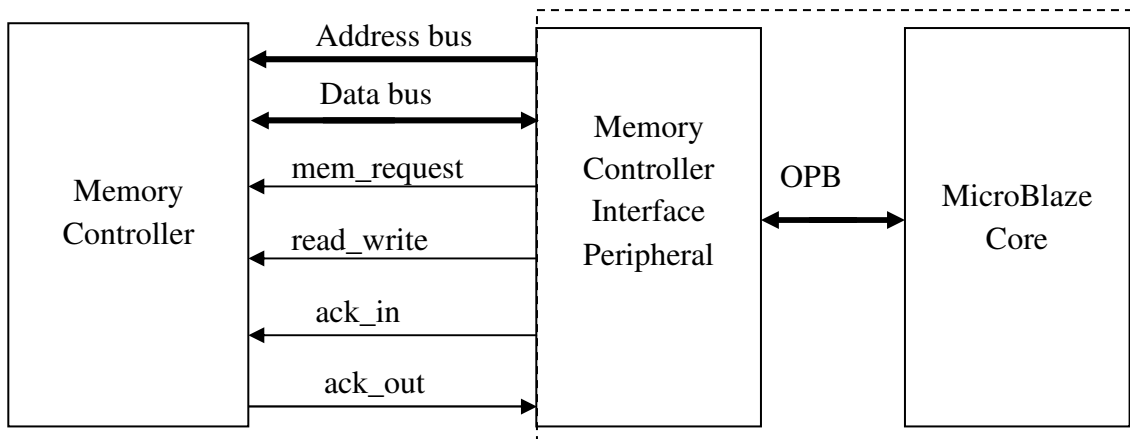


Fig. 3-12. Implementation specific memory controller interface peripheral.

3.2.6 Global Memory Implementation

The global memory in the MPSoC implementation was made up of eight 2 kB block RAMs within the Virtex-II Pro FPGA, combined together to create a 16 kB global memory. This is a very small global memory compared to what would normally be used

for most applications; however, this was limited to the amount of memory available within the Virtex-II Pro FPGA. An external memory could be mapped to the FPGA to allow for a larger global memory; but, for the sake of the experiments needed to verify the MPSoC, this was not necessary.

3.2.7 Snoopy

In order to evaluate the system performance, several metrics are required. The snoopy block was developed to monitor the internal system signals and measure the parameters that were interesting in evaluating the system performance. The snoopy block measures these parameters by monitoring the control signals that are sent between the various system components. The snoopy block does not add any functionality to the system; it only measures system performance metrics. Therefore, it is not required by the generic MPSoC framework. However, it can be used for any implementation of the MPSoC framework because the signals monitored to measure system performance metrics are all defined as part of the generic MPSoC framework.

Since snoopy is implemented entirely in hardware, the timing measurements are accurate to the minimum clock period. In the MPSoC implementation, snoopy used a 100 MHz clock reference, which means that all timing measurements were accurate to within 10 ns. The following sections discuss the parameters that snoopy calculates, and the signals used to derive those measurements.

3.2.7.1 Total Program Execution Time

The total program execution time is the time from when the DFG tasks are first executed, to the time when the last DFG task is executed. This does not include any pre-

DFG or post-DFG task execution that may be done by the head processor. This time is determined by the snoopy block by counting the number of clock cycles from the time that the “start_task_control” signal is set active by the head processor until the time when all of the “proc_done” signals are set, which indicates that all processors are finished all of their DFG tasks.

3.2.7.2 Processor Memory Waiting Time

The processor memory waiting time is the total time that each processor spends waiting for a memory request to be served. This is the sum of the time for each individual memory access by a particular processor. There is a separate total memory waiting time that is kept for each individual processor. These times are determined by the snoopy block by keeping a cumulative count (separate count for each processor) of the number of clock cycles that the “mem_request” signal for each processor is active.

3.2.7.3 Number of Task Memory Accesses

The number of task memory accesses is the number of memory accesses that are made by each task in the DFG. This is a separate count that is kept for each task in the DFG. The snoopy block determines the number of memory accesses for each task by monitoring the “mem_request” signal from each processor to the memory controller and the “DFG_node” buses sent from the task controller to each processor. When a “mem_request” signal for a particular processor first becomes active, the value on the “DFG_node” bus for that processor is read by snoopy to determine which DFG task is being executed by the processor requesting the memory request. The count of the memory requests for that particular node is then incremented.

3.2.7.4 Total Task Execution Time

The total task execution time is the total time spent executing a particular task in the DFG. This is the sum of the times it took to execute particular tasks, each time it was executed. There is a separate total task execution time that is kept for each individual task in the DFG. The snoopy block determines the total task execution time for each task by monitoring the “executing” signal from each processor to the task controller and the “DFG_node” buses sent from the task controller to each processor. When an “executing” signal for a particular processor is active, a cumulative count is kept for the task that is read on the “DFG_node” bus for that processor.

3.2.7.5 Processor Idle Time

The processor idle time is the amount of time that each processor is not executing a task in the DFG. This does not include the time during pre-DFG or post-DFG execution, only the time that the task controller is actively scheduling DFG tasks. There is a separate processor idle time that is kept for each individual processor. The snoopy block determines these times by keeping a cumulative count (separate count for each processor) of the number of clock cycles that the “executing” signal for each processor is inactive. The count is only kept if the “start_task_control” signal is active, and at least one of the “proc_done” signals is not active. These means that even if a processor is finished all of its tasks in the DFG, but another processor is not finished its tasks, the processor idle time is incremented. It might seem strange to count the time after all a processor’s tasks are finished as idle time, but this time can be significant in determining load balancing between processors. For example, if one processor finishes all of its tasks well before another processor is finished, its idle time might be significantly larger. This

would then flag to the system designer that a different pipelining task allocation may be appropriate to better balance the processor loading.

3.2.7.6 System Parameter Communication

It is important to have a method to communicate the system metrics that are collected by the snoopy block, so that the information can be viewed by the system designer to evaluate the system. This information is communicated to the head processor, so that the head processor can read this information and display it through whatever means required by the system. The head processor typically would read and report the parameters from the snoopy block as part of the post-DFG processing. The signals that the snoopy block uses to communicate the parameters to the head processor are as follows:

- info_addr – This is a 4-bit address bus specifying which parameter should be communicated. The addresses are as follows:
 - 0: Lower 32-bits of Total Execution Time
 - 1: Upper 32-bits of Total Execution Time
 - 2: Lower 32-bits of Processor Memory Waiting Time
 - 3: Upper 32-bits of Processor Memory Waiting Time
 - 4: Number of Task Memory Accesses
 - 5: Not Used
 - 6: Lower 32-bits of Total Task Execution Time
 - 7: Upper 32-bits of Total Task Execution Time
 - 8: Lower 32-bits of Processor Idle Time

- 9: Upper 32-bits of Processor Idle Time
- 10 to 15: Not Used
- `proc_task_addr` – This is an address that specifies the processor or task whose information is required. Whether this address is specifying a processor number, or a DFG task number, depends on the current `info_addr`. If the `info_addr` address is 0 or 1, then the total execution time is reported. Since this is not specific to any processor or task, the `proc_task_addr` is ignored. If the `info_addr` is 2, 3, 8, or 9, then the `proc_task_addr` is specifying a processor number, because those `info_addr` addresses are specifying processor specific parameters. If the `info_addr` address is 4, 6, or 7, then the `proc_task_addr` is specifying a DFG task number, because those `info_addr` addresses are specifying task specific parameters.
- Data – This is a 32-bit data bus that contains that data specified by the addresses on the `info_addr` and `proc_task_addr` buses.

There is no need for acknowledge, read/write, or enable signals for the head processor to interface with the snoopy block, because all of the information is read-only. Therefore, the data bus can always be active, and it is up to the host processor to ensure that, when it reads the data bus, the `info_addr` and the `proc_task_addr` is valid.

3.2.8 Snoopy Interface Peripheral

The snoopy interface peripheral is a custom processor peripheral that was created to interface the snoopy block with the head processor. The snoopy block is not part of the generic MPSoC framework; therefore, the snoopy interface peripheral is also not part of

the generic MPSoC framework, but rather implementation specific. Only the head processor has a snoopy interface peripheral. In the MicroBlaze implementation that was created for this project, the interface between the memory controller interface peripheral and the processor is the Xilinx OPB just as the Xilinx OPB was the interface between the task controller interface peripheral and the memory controller interface peripheral. Using the OPB allows the peripheral to be a memory-mapped peripheral of the MicroBlaze processor. The snoopy interface peripheral allows the signals sent by the snoopy block (which were described in the previous section) to be read and written by addressing particular registers that can be accessed by software execution. This peripheral is the interface between the snoopy hardware, and the software executing on the head processor.

The following block diagram shows how the snoopy interface peripheral interfaces with the snoopy block and the head processor core.

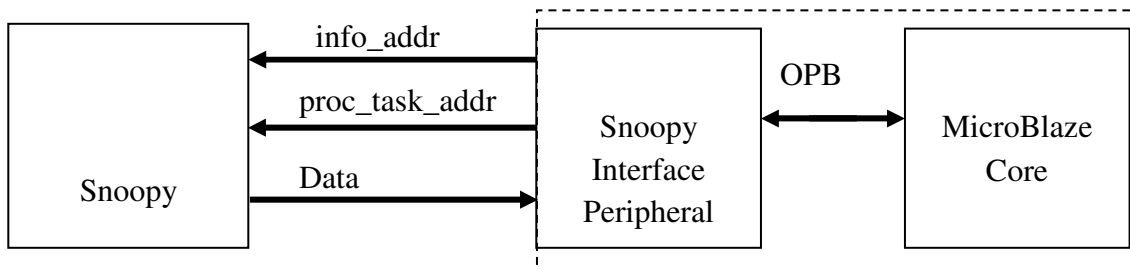


Fig. 3-13. Snoopy interface peripheral.

Chapter 4

ANALYTICAL MODEL FOR THE MULTIPROCESSING SYSTEM-ON-CHIP (MPSOC) FRAMEWORK

4.1 Motivation for an Analytical Model of the MPSoC Framework

While the MPSoC framework introduced in the previous section narrows the design space by fixing key portions of the hardware and software architecture to fit the chosen computing model, there are still many design decisions that can be made for a particular application. Assuming that an application has been structured in a DFG, some key decisions are:

- How many processors should be applied to a given problem?
- Which tasks should be assigned to which processor?
- What task schedule order should be used?
- In the case of heterogeneous multiprocessing systems, should certain tasks be assigned to certain processors to achieve better performance?

These design decisions can have a significant effect on the performance of a system. It is therefore beneficial to have a method to estimate the performance of the system before implementation. This results in the option of trying many more permutations of a particular application in a shorter period of time, as opposed to spending the time to implement each permutation and then measuring the performance of the system. An analytical model of the system is required to estimate the performance of a system without implementation.

The fundamental trade-off that occurs with a multiprocessing system that has a global shared memory is that the addition of more processors can result in more tasks being executed in parallel, but can also result in more contention of global memory. Therefore, it is important to determine how many processors can be added to a system before the costs of additional processors outweigh the benefits.

4.2 Related Work in Analytical Models of Multiprocessing Systems

There have been several studies in the past that have analyzed memory interference in a shared memory multiprocessor system. However, these studies have focused on multiprocessing systems without defining a computing model. This typically results in analysis methods that can be applied to a wider range of problems, but at the cost of less accurate analysis. Since the model of computing was not considered in these studies, assumptions could not be made about the contents of global shared memory, and therefore it had to be assumed that the global shared memory could contain both data and instructions. In the case where global memory contains both data and instructions, a single bus global memory multiprocessing system is inefficient because there is very high probability of memory interference, which counteracts the benefits of executing tasks in parallel. Therefore, in these studies more complex architectures are analyzed to reduce memory interference, such as cache systems, multiprocessing systems with multiple memories either connected in a crossbar network [Bhan75] [DaSe96] [MuAM87] [Nade88] [SeDe79], or with multiple buses [DaSe96] [JoLi96] [PaMi98]. The MPSoC framework architecture discussed in Chapter 3 specifies that the global shared memory contains only data and not instructions. This allows for the analytical model to be

specifically tailored to global memory containing only data, resulting in a more accurate analysis for the MPSoC.

Another benefit of tailoring the analytical model specifically to the MPSoC framework, compared to a more general analytical model, is that each specific task can have its own probability of global memory request that is independent of the other tasks in the system. In previous studies, which are more general analyses, the assumption was made that all processors request access to shared memory with the same probability throughout all execution, and across all processors. Therefore, memory requests were often modeled as a Bernoulli process with a fixed probability for discrete time analysis [Bhan75] [JoLi96] [DaSe96] [MuAM87] [SeDe79], or as a Poisson process with a fixed probability for continuous time analysis [Nade88]. This thesis takes a different approach because the shared memory contains only data and not instructions. This means that the probability of requesting access to shared memory depends on the specific task being executed, since the function of a task will determine the frequency at which data in shared memory needs to be accessed. In this thesis, memory requests are modeled as a Bernoulli process. Accordingly, each task in the DFG has its own probability of requiring a global memory request. As a result, the probability of a memory request is allowed to change for a particular processor, as the processor switches from executing one task to another.

The analytical model presented in this thesis also differs from previous work in how memory service time is modeled. Most of the previous studies assume that the memory service time is constant [Bhan75] [JoLi96] [MuAM87] [ReVa99] [SeDe79];

however, there are some that assume a Bernoulli process [DaSe96] or Poisson process [Nade88] for memory service. This thesis recognizes that the memory service time is dependent on the specific memory type used, and not on the hardware architecture itself. Therefore, the memory service time is represented by a phase type distribution that can be adjusted to fit the characteristics of the specific memory used in an application. This also allows for the flexibility in using several different types of memory that together make up global memory. A negative binomial process is suggested as a phase type distribution that can be easily adjusted to fit the characteristics of a specific memory.

4.3 Performance Metrics

There are many different possible performance metrics that can be considered for a given system, such as processor load balancing, system throughput time, system latency, and system resources (for example, number of processors and/or cost of processors). The performance evaluation of a system can involve any of these performance metrics, and significance of each of the performance metrics will determine effectiveness of a particular implementation of the MPSoC. The most common performance metric is system throughput time, that is, the time required to finish processing all data sets. An increased performance is often measured by a reduction of the system throughput time. However, in some cases system throughput time is not the most important performance metric. The latency of a system can be very important for stream-oriented applications. For example, in the case where the application is a video conferencing system, it is not only important to have all of the data sent and processed, but for the latency of the system to be such that there are not any noticeable delays in

two-way communication. The analytic model proposed in the following sections is not particular to any one performance metric, and can be used to evaluate an implementation of the MPSoC against many different performance metrics. However, the analytical model is a stochastic model that relies on the probabilities of memory access times, probabilities of global memory requests, and average task execution times. Therefore, the model is well-suited to determine the typical expected performance of an MPSoC framework implementation, but will not necessarily predict the worst case performance of an MPSoC framework implementation.

4.4 Analysis Inputs

In order to analyze a particular implementation of the MPSoC framework, the following implementation parameters need to be known about the system before analysis:

- The number of processors in the system.
- The number of pipeline stages in a system.
- The order of execution of each task in the system, and the task allocation to processors.
- The probability of a memory access request for each task that is being executed. This probability may differ between different tasks.
- The execution time of each task on its assigned processor if it does not have to wait for memory access (i.e., the time it would take to execute if it was the only task being executed).

- The memory access time probability distribution. This is the probability mass function showing the probability of a memory access taking a certain amount of time once a processor is given access to the memory.
- The system clock period (t_s). This is used as the sample period of the system, which will usually be chosen to be the instruction execution time for the processors. The system time quanta will be considered the minimum time quanta for the discrete Markov chain that is used to analyze memory access.

4.5 System Design Assumptions

There are a number of simplifications that are assumed when analyzing any complex system. Without these simplifications, an analysis would be intractable. The goal is to make assumptions that simplify the system to the point where it can be analyzed, but not simplify it so much that it does not resemble the real world system. The following section describes the major assumptions that were made for the analytical model presented in this thesis.

4.5.1 Types of Memory Requests

There are two main types of memory requests, which are reads and writes. For many types of memory technologies the time it takes to read data from memory is very similar to the time it takes to write data to memory, but for other types of memory the read time and the write time can vary significantly. For example, the read and write times are usually very similar in volatile types of memory (e.g. SRAM, SDRAM, DRAM), and the read and write times often differ significantly in non-volatile types of memory (e.g. EEPROM, Flash, hard disk). The assumption made in this thesis is that the read and write

times do not differ significantly, and so there is no distinction made between reads and writes to memory. This means that the memory analyzed will more closely match a real world volatile type memory, and that the results may not be valid for a memory where the read and write times vary significantly.

4.5.2 Distribution of Memory Requests

The rate and repeatability at which memory requests occur can vary significantly between different tasks in the DFG. Some processes may have memory requests that occur at very specific periods, while other processes may have batch memory requests, and then large times where there are no requests. The specific distribution of the memory requests for a given task cannot be determined without knowing what the task is. Even if the task is known, the distribution can be difficult to predict because it may depend on the data being processed. In this case, the interest is in solving the problem generally, without knowledge of the specific distribution, so the assumption is made that the memory requests are made according to a Bernoulli process with probability α_i , where i is the particular task in the DFG that is being executed. This means that the probability of a memory request can differ between tasks. By using the average number of memory requests made during its execution (N_{mem}) and the number of time quanta without memory requests (N_{no_mem}), the probability of a memory request for a given task could be determined with the following equation:

$$\alpha = \frac{N_{mem}}{N_{mem} + N_{no_mem}} \quad (1)$$

In the case where the processors can cache the data read previously from global memory into their local memories, the probability of a memory request can be reduced. This can be accounted for in the model in many different ways. In the most simplistic case, the probability of a cache hit for a particular task can be determined, and then the probability of a global memory request can then be adjusted to account for the cache by multiplying the probability of a memory request by the probability that the data does not already exist in cache (i.e. 1 minus the cache hit probability). More complex analysis of the effect of cache in the system can be made, for example, by considering multiple cache memories with different characteristics, or shared cache memories; however, this is beyond the scope of this thesis.

4.5.3 Memory Service Distribution

Some types of memories have deterministic service times, where a memory read/write completes in a known and certain number of clock cycles. Other types of memories have non-deterministic service times, in which case, a probability distribution may be used to predict memory read/write service times. Moreover, different types of non-deterministic memories will have different memory service probability distributions. The memory access time is modeled with a phase type distribution that can be tailored to fit the probability mass function of the global memory access times.

4.5.3.1 Model of a Single Memory Type

A negative binomial phase type distribution can be used to generate a memory access time probability distribution to fit most types of memories. Often the only parameters given for read or write time of a memory are the maximum read/write time,

and the typical read/write time. These parameters can be used as shaping guidelines in the model. For example, a memory's mean service time can be used to represent the highest probability of service time, and the tail of the distribution can be cut-off by the maximum read/write time. The probability of service occurring after the maximum service time is assumed to be very small and to never occur (this probability may be available from the memory manufacturer, and could vary for different types of memory). There are two parameters that can be varied to shape the distribution of the memory service to closely match a particular memory; these are the number of stages in the negative binomial distribution (k), and the probability of moving to the next stage (p_e). The memory service phase type distribution has representation $(\beta, S), k$. The vector β represents the starting phase of the phase type distribution, and will be of the form:

$$\beta = [1 \ 0 \ 0 \ \dots \ 0] \quad (2)$$

β will have k elements in it.

The matrix S represents the phases of service. It is a square matrix of order k and will be of the form:

$$S = \begin{bmatrix} 1-p_e & p_e & & & \\ & 1-p_e & p_e & & \\ & & \dots & \dots & \\ & & & 1-p_e & p_e \\ & & & & 1-p_e \end{bmatrix} \quad (3)$$

The vector S^0 represents the end of service. It will have k elements in it and will be of the form:

$$S^0 = [0 \ 0 \ \dots \ 0 \ p_e]^T \quad (4)$$

Since there are k stages in the negative binomial distribution, the minimum service time will be $k*t_s$ (where t_s is the minimum time quanta). The probability that service will be finished at time $i*t_s$, where $i \geq k$ is given by:

$$p_i = \binom{i-1}{k-1} p_e^k (1-p_e)^{(i-k)} \quad (5)$$

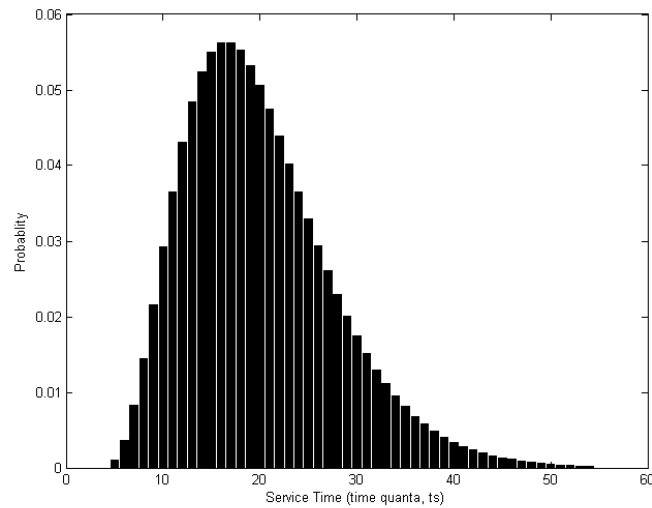


Fig. 4-1. Distribution of service time for $k=5$ and $p_e=0.25$.

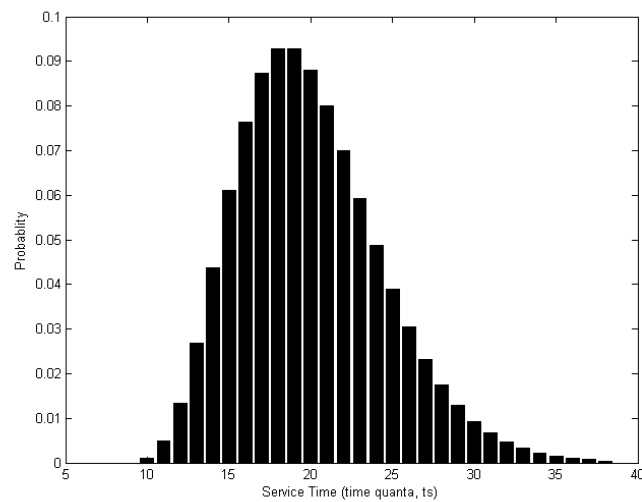


Fig. 4-2. Distribution of service time for $k=10$ and $p_e=0.5$.

Two examples of the service time distribution for particular values of k and p_e are shown in Fig. 4-1 and Fig. 4-2. These two examples show how the variance of the distribution can change with a different k and p_e while maintaining the same mean service time. As the number of phases in the distribution increases, the variance of the distribution decreases. This phase type distribution can also be used to represent a memory type that has a deterministic service time by setting p_e to 1, and k to the number of time quanta in which service will be finished.

4.5.3.2 Model of Multiple Memory Types

A global memory may consist of several different types of memories, each with its own memory access time probability distribution. The global memory probability distribution will then be a combination of these probability distributions. To create a phase type distribution to model M_{mem} different types of memory, the phase type distribution of each memory type is required; also, the probability of accessing each of the different types of memories must be known. Given that the matrix representing the phases of service for the i^{th} memory type (as described by (3)), S_i , the matrix that represents the phases of service for the combined memories is determined to be:

$$S = \begin{bmatrix} S_1 & & & \\ & S_2 & & \\ & & \dots & \\ & & & S_{M_{mem}} \end{bmatrix} \quad (6)$$

Where M is the total number of different memory types. The vector S^0 represents, which represents the end of service, can be made of each of the individual end of service vectors (as described in (4)). It will be of the form:

$$S^0 = \begin{bmatrix} S_1^0 \\ S_2^0 \\ \dots \\ S_{M_{mem}}^0 \end{bmatrix} \quad (7)$$

Finally, the vector representing the starting phase of the phase type distribution differs from the form shown in (2). In this case the vector considers the probability of each of the types of memories being accessed, and the starting phase vector will be in the following form (the notation $0_{(i,j)}$ will be used to represent an i by j matrix full of zeros):

$$\beta = [\lambda_1 \quad 0_{(1,k_1-2)} \quad \lambda_2 \quad 0_{(1,k_2-2)} \quad \dots \quad \lambda_{M_{mem}} \quad 0_{(1,k_{M_{mem}}-2)}] \quad (8)$$

The β vector, the S^0 vector and the S matrix together make up the phase type distribution with representation $(\beta, S), k$, where:

$$k = \sum_{i=1}^{M_{mem}} k_i \quad (9)$$

As an example, suppose three different memory types are used, where the probabilities of accessing each memory type when global memory is accessed are: $\lambda_1=0.3, \lambda_2=0.4, \lambda_3=0.3$. Each of the memory types are modeled with a negative binomial phase type distribution with the following parameters:

- Memory Type 1: $k_1=5$ and $p_{e1}=0.25$ (the memory access probability distribution is shown in Fig. 4-1)
- Memory Type 2: $k_2=10$ and $p_{e2}=0.25$
- Memory Type 3: $k_3=5$ and $p_{e3}=0.5$

Fig. 4-3 (a), (b), and (c) show the probability distribution resulting from each of the three memories if they were the only memories in the system. Fig. 4-3 (d) shows the

resulting probability distribution of the combination of the memories when they are all used to form global memory and the probabilities of access each are as described above.

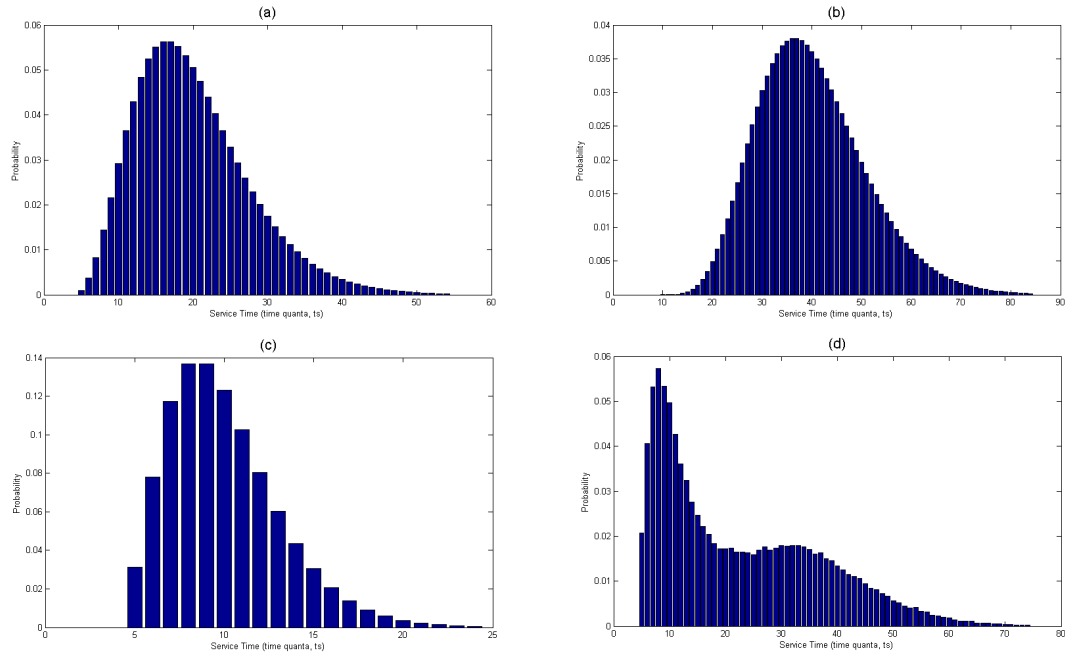


Fig. 4-3. Memory distribution of the individual memory types: (a) memory type 1, (b) memory type 2, (c) memory type 3. The probability distribution for a global memory access made up of a combination of the three memories is shown in (d).

4.5.4 Task Execution Times

For the purposes of this analysis, it is assumed that the processing time for each task is fixed when run on a single processor (i.e. no global memory interference effects). In practice, this assumption is not strictly true when there are a number of branch statements within a task that can result in varied execution times based on the data that is being processed. However, for the purposes of determining the overall execution time, the average task execution time for each task can be measured in a single processor system and used in the analysis. If the variance of the real task execution times is relatively small, the assumption of a fixed task execution time will not cause a significant

difference in the overall execution time predicted by the proposed model compared to the actual system execution time.

4.5.5 Memory Access Control

In order to determine which processor gets access to the memory when there is a request by more than one processor, it is necessary to have a control algorithm that is implemented by the memory controller. The memory access control method is defined as part of the MPSoC framework (described in section 3.1.1.4). This memory access control method is a polling algorithm that cycles through the active processors to see if there is a pending memory request. The controller spends one time quantum (t_s) checking each processor for memory requests. So, it will take $N * t_s$ to check all processors for memory requests if there are no active requests, where N is the number of active processors. If there is a memory request for a processor that is currently being polled, then the memory request is serviced. When the memory request is finished being served, the memory controller will wait with the current processor for one more time quanta to allow for another memory request to be made. If no memory request is made in the next time quantum, then the memory controller moves to the next processor to check it for memory requests. However, if another memory request is made by the processor that just finished a memory access in this one time quantum, then the memory controller will service the new memory request without checking the other processors in between.

4.6 Description of the MPSoC Analytical Model

The goal of the analysis is to determine the amount of time that each processor must typically wait when it has a memory request, while a memory request of another

processor is currently being serviced. From this information the execution time of the process can be extended to represent the average execution time of a task in the DFG when considering the time waiting for memory access. This allows for the tools to compare different implementations of the MPSoC framework.

4.6.1 Partitioning the Processing Period

One of the complexities of analyzing this system is that each processor could execute several different processes, and each process could have its own independent memory request probability (α_i). This means that each processor may not have a fixed memory request probability, but rather the memory request probability will change with each task that is executed by the processor. The time at which a processor switches from one task to another task is independent from the task switching times of other processors in the system. In order to get around this difficulty, the processing period is partitioned into windows where the tasks of the DFG that are being executed do not change for all processors. The results of this analysis is then used to adjust the processing period to account for memory access waiting times in the window that was analyzed. The next window is then analyzed, until the entire processing period is analyzed. This is an iterative approach to the problem.

For example, if the system that was described earlier was to be analyzed, the processing period with a full pipeline would look as shown in Fig. 4-4. The first window that would be analyzed would be the section where tasks A, C, and D overlap. The system is then analyzed to determine how much time those sections of A, C, and D are lengthened when the time waiting for memory access is considered.

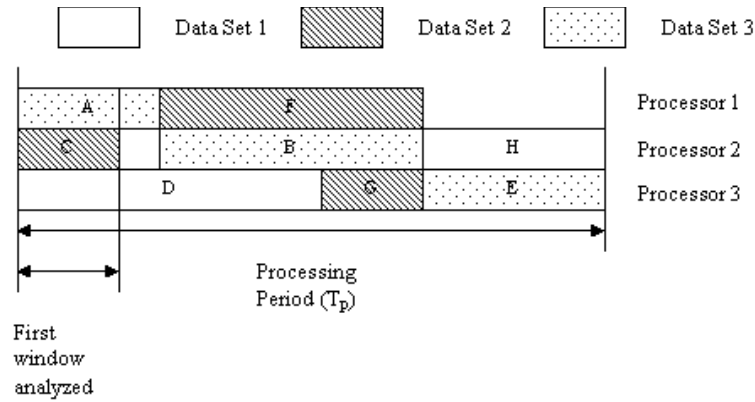


Fig. 4-4. First window analyzed in the processing period.

Suppose that after analyzing the first window, the time for task A was extended (due to memory access time). Consequently, the processing period was now adjusted to compensate for Task A’s extension, as shown in Fig. 4-5.

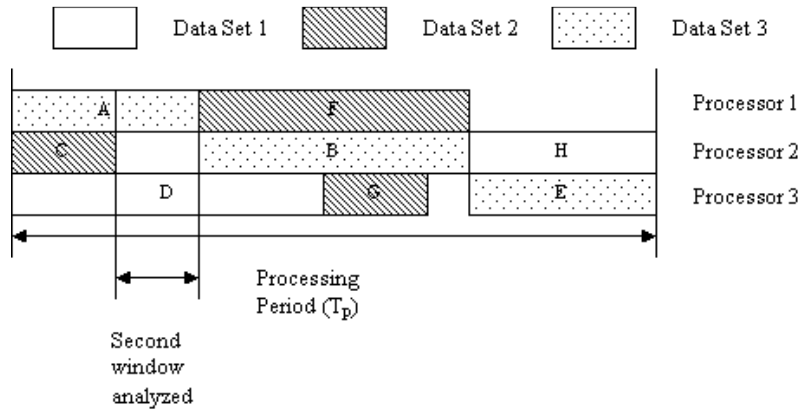


Fig. 4-5. Second window analyzed in the processing period.

It can be seen that the overall processing period is now extended because the processing time of task A was extended; B must come after A is finished; and E must come after B is finished. This change also created a gap between task G and E on Processor 3, which is now an additional window that must be analyzed, that was not a window before the first iteration. The second window that is to be analyzed is shown in Fig. 4-5. This window consists of task A on Processor 1 and task D on Processor 3. Processor 2 does not execute during this window, so the system is analyzed as if there are

only two processors for this window. After the second window is processed, the processing period may change again. This process is repeated until the entire processing period is analyzed, at which point the entire processing period will be adjusted to take into consideration the memory access waiting times.

4.6.2 Analyzing a Partition

4.6.2.1 Model of Each Processor's Memory Service

In order to analyze any particular partition of the processing period to determine the amount of time that each processor waits for memory access while another processor is being served, the memory requests by each processor are modeled with a discrete time Markov chain. The memory requests for each processor can be modeled as a queue of length 1. That is, there is either a memory request, or there is not a memory request. There cannot be more than one memory request per processor at the same time.

From the point of view of each processor in the system, there are four states that the memory controller can be in at any given time. The first state occurs when the memory controller is polling the current processor to see if it has a memory request. The second state occurs when the memory controller is in the midst of servicing a memory request from the current processor. The third state occurs when the memory controller is servicing other processors and the current processor does not have a pending memory request. The fourth state occurs when the memory controller is servicing other processors and the current processor has a pending memory request. When the memory controller is servicing processors other than the current processor, the memory controller is said to be on vacation with respect to the current processor. This means that the memory controller

is on vacation (with respect to the current processor) in the third and fourth states. When the system only contains a single processor, there is no time spent in states 3 and 4 (the vacation states), because there are no other processors to service. Therefore, the execution time of additional tasks due to a processor waiting for memory access in a multiprocessor system is determined by the amount of time spent in state 4. The goal of the proposed analysis method is to determine the amount of time spent in state 4, from which the effect of memory interference on the task execution time can be determined. Fig. 4-6 shows the state transition diagram for processor i , with the probabilities of changing states shown on the transition edges. This illustrates the transitions from state to state as defined by the probability transition matrix in (10).

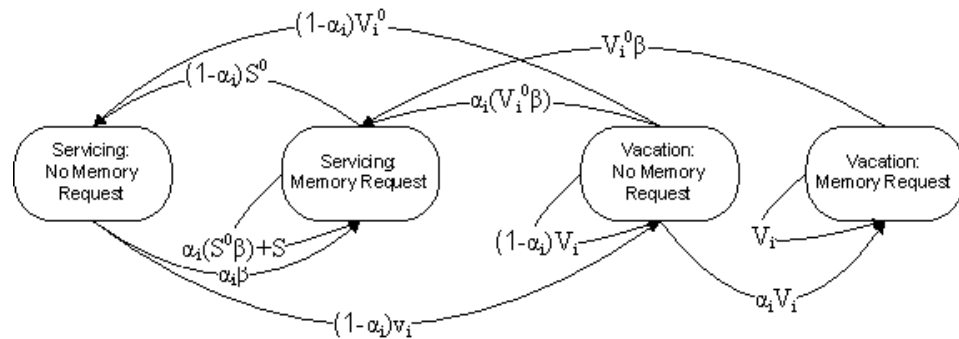


Fig. 4-6. State transition diagram for processor i .

There are several ways that the described system could be modeled with a discrete time Markov chain. The method chosen was to model each memory request queue individually as a Geo/PH/1 system with PH vacations. This means that the arrival of the memory access requests is a Bernoulli process with arrival probability α_i (for processor i), service is a negative binomial phase type with representation $(\beta, S), k$, and there are vacations with representation $(v_i, V_i), m$, meaning v_i is a vector describing the probability of starting in each phase of V_i , V_i represents the phases of the vacation, V_i^0 is the vector

representing the end of the vacation, and m is the order of the square matrix V_i . The system is considered to be on vacation when the memory controller is serving other processors. The probability transition matrix for a processor i can then be represented as shown below. The notation $0_{(ij)}$ will be used to represent an i by j matrix full of zeros.

$$P_i = \begin{bmatrix} 0 & \alpha_i \beta & (1 - \alpha_i) V_i & 0_{(m1)} \\ (1 - \alpha_i) S^0 & \alpha_i (S^0 \beta) + S & 0_{(mk)} & 0_{(mk)} \\ (1 - \alpha_i) V_i^0 & \alpha_i (V_i^0 \beta) & (1 - \alpha_i) V_i & \alpha_i V_i \\ 0_{(1m)} & V_i^0 \beta & 0_{(mm)} & V_i \end{bmatrix} \quad (10)$$

The first row in the above matrix represents the state in which there are no memory requests by the processor, but the memory controller is checking to see if there are any memory requests. The probability of remaining in this state for the next time quanta is 0, because either no memory request will arrive and the memory controller will then go on vacation (i.e. serve other processors), or a memory request will arrive and the memory request will begin being served.

The second row in the above matrix represents the states when a memory request is being processed. The system will remain in this state in the next time quanta if memory service does not finish, or if service finishes but another memory request is made immediately. If service finishes and another memory request is not made immediately then the system will return to the first state where the memory controller will wait one time quanta for another memory request, and if one does not arrive the memory controller will serve other processors. The second state consists of a number of sub-states, where the number of sub-states is k , which is the order of the memory service matrix S .

The third row of the above matrix represents the case where the system is on vacation (memory controller is serving other processor's memory request) and there are currently no memory requests for processor i . The system can go to any other state from this state. If vacation ends (i.e. the memory controller has finished serving other processors) and no memory request arrives, then the system will go to the first state to wait one time quanta for a memory request to arrive. If vacation ends and a memory request does arrive, then the system will go to the second state where the memory request will be served. If vacation does not end and no memory request arrives, then the system will remain in the third state. If vacation does not end and a memory request arrives, then the system will go to the fourth state. The third state consists of a number of sub-states, where the number of sub-states is m , which is the order of the vacation matrix V_i .

The fourth row of the above matrix represents the case where the system is on vacation and there is currently a pending memory request. In this case, no more memory requests can arrive, which means the system can either remain in the current state if vacation does not end, or it will go to the second state if vacation does end, where the pending memory request will be served. The fourth state consists of a number of sub-states, where the number of sub-states is m , which is the order of the vacation matrix V_i .

4.6.2.2 Vacation Model

For a given problem the arrival probability, α_i , for each processor in the partition that is being analyzed is known, and the service of the memory, $(\beta, S)_k$, is also known; therefore, in order to analyze the probability transition matrix for each processor's memory service, the vacation process, $(v_i, V_i)_m$, is the only remaining unknown process.

The form of the matrix A_j representing transitions between states when the memory controller is serving processor j is shown below.

$$A_j = \begin{bmatrix} 0 & \alpha_j \beta \\ (1 - \alpha_j) S^0 & \alpha_j (S^0 \beta) + S \end{bmatrix} \quad (13)$$

The first row in the above matrix represents the state when there are 0 memory requests. In this case the processor will only remain in service if a memory request arrives, in which case the memory request will begin being served. If no memory request arrives then the memory controller will go on to serve the next processor (this transition is represented in matrix B_j).

The second row represents the state when there is 1 memory request that is currently being served. If service finishes and no new memory request arrives, the state represented by the first row (0 memory requests) is entered. The state remains the same if either service does not finish, or if it finishes but a new memory request arrives to start a new memory request service.

The form of the matrix B_j representing transitions between states when the memory controller is moving from serving processor j to serving the next processor (which is $(j+1) \bmod N$) is shown below.

$$B_j = \begin{bmatrix} (1 - \varphi_{(j+1) \bmod N}) (1 - \alpha_j) & \varphi_{(j+1) \bmod N} (1 - \alpha_j) \beta \\ 0_{(1k)} & 0_{(kk)} \end{bmatrix} \quad (14)$$

The first row of this matrix represents the state when there are 0 memory requests for processor j . The second row represents the state when there is 1 memory request for processor j . Since the processor will never start a vacation when there is a pending memory request that can be serviced, the probability of starting to serve the next

processor when there is one memory request is 0, which is why the second row consists of zeros. When there are 0 memory requests the memory controller will start to serve the next processor, but it could transition to the state where there is no pending memory request for the next processor, or it could transition to the state where there is a pending memory request for the next processor, depending on whether a memory access request has arrived for the next processor since it was last served. The parameter φ_j represents the probability that a memory request is made by processor j from the time its vacation starts to the time that its vacation ends. This means that $1 - \varphi_j$ represents the probability that there are no memory requests made by processor j in the time that its vacation starts, to the time its vacation ends. The first entry in the first row of matrix B_j represents the transition from serving processor j to serving the next processor (processor $(j+1) \bmod N$) when there are no memory requests pending for processor $(j+1) \bmod N$. The second entry in the first row of matrix B_j represents the transition from serving processor j to serving the next processor (processor $(j+1) \bmod N$) when there is one memory request pending for processor $(j+1) \bmod N$.

The vector that represents the start of vacation for processor i is v_i , and it can be represented as follows:

$$v_i = [(1 - \varphi_{(i+1) \bmod N}) \quad \varphi_{(i+1) \bmod N} \beta \quad 0 \quad \dots \quad 0] \quad (15)$$

The first entry in the vector represents the transition to serving the next processor (processor $(i+1) \bmod N$) when there is no pending memory request for the next processor. The second entry in the vector represents the transition to serving the next processor when there is one pending memory request. The rest of the vector is filled with zeros.

The vector that represents the transitions when the vacation of processor i ends is given by V_i^0 , and it can be represented as follows:

$$V_i^0 = \begin{bmatrix} 0 \\ \dots \\ 0 \\ 1 - \alpha_{((i+N-2) \bmod N)+1} \\ 0_{(1k)} \end{bmatrix} \quad (16)$$

This shows that the vacation for processor i finishes after processor $((i+N-2) \bmod N)+1$ (which is the previous processor to i in the cycle) was being serviced, but now has 0 memory requests, and no new memory request arrived.

The parameter φ_i is defined as the probability that a memory request occurs for processor i while processor i is on vacation. To determine this value, the amount of time spent in the vacation process needs to be known. The amount of time that the memory controller is on vacation relative to processor i can vary, and will not typically be a fixed number. The probability of the vacation process ending in a particular number of time quanta, needs to be determined for all time quanta amounts in which the probability is significant. The first step in calculating these probabilities is to create a new Markov chain with a probability transition matrix V_i' by combining v_i , V_i , and V_i^0 , as shown below:

$$V_i' = \begin{bmatrix} 0 & v_i \\ V_i^0 & V_i \end{bmatrix} \quad (17)$$

Starting in state 1 of V_i' , the system will transition to the sub-matrix V_i by the probabilities in the starting vector v_i ; it will then sojourn within V_i until it returns to state 1 by the probabilities defined in V_i^0 . Since the vacation process with representation

(v_i, V_i) , m starts vacation through v_i and ends vacation through V_i^0 and sojourns within V_i during vacation, the vacation time is the same as the time that it takes to return to state 1 of V_i for the first time when starting from state 1. The parameter $f_{x,y}^{(n)}$ is defined as the probability of first visiting state y from state x in a Markov chain at the n^{th} time quantum. The following result has been shown to be valid [Rose71], given the number of states in the Markov chain is m , and the probability of transitioning from any state y to any state x is $v_{x,y}$.

$$f_{x,y}^{(n+1)} = \sum_{z=1}^m v_{x,z} f_{z,y}^{(n)} - f_{y,y}^{(n)} v_{x,y} \quad \text{for } n \geq 1 \quad (18)$$

This also means that $f_{x,y}^{(1)} = v_{x,y}$ for any x and y . The probability of finishing the vacation process in n time quantum can then be calculated by using (18) to calculate $f_{1,1}^{(n)}$ in the Markov chain represented by V_i , which is the probability of first returning to state 1 starting from state 1 in n time quanta.

The probability of processor i requesting access to memory in one time quantum was previously given as α_i . This can then be used to calculate the probability of processor i requesting access to memory within n time quanta, defined as $\sigma_i^{(n)}$, with the following equation:

$$\sigma_i^{(n)} = \alpha_i \sum_{h=1}^n (1 - \alpha_i)^{h-1} \quad \text{for } n \geq 1 \quad (19)$$

Equations (18) and (19) can now be used to calculate the probability that the vacation for processor i will end in n time quanta and that there will be a memory request made by processor i during that vacation. This probability is defined as $\varphi^{(n)}$ and is calculated by:

$$\varphi_i^{(n)} = \sigma_i^{(n)} f_{1,1}^{(n)} \text{ for } n \geq 1 \quad (20)$$

Therefore, the probability that a memory request occurs for processor i while processor i is on vacation is given by:

$$\varphi_i = \sum_{n=1}^{\infty} \varphi_i^{(n)} \quad (21)$$

It is not practical to use (21) to determine φ_i since this equation involves an infinite sum. It can be shown that if the probability of requesting access to memory for each of the processors is less than 1 (i.e., $\alpha_i < 1$ for all i), then the probability of eventually finishing a vacation is 1. This means that state 1 of V_i ' is a recurrent state for which the following equation holds true [IsMa76]:

$$\sum_{n=1}^{\infty} f_{1,1}^{(n)} = 1 \quad (22)$$

This fact can be used to determine a practical limit to the sum in (21) by choosing some acceptable error limit ε_r where:

$$\sum_{n=1}^r f_{1,1}^{(n)} = 1 - \varepsilon_r \quad (23)$$

Equation (23) gives an upper limit, r , to the sum in (22), where the probability of the vacation ending in more than r time quantum is considered insignificant. The smaller the error, ε_r , the larger the value of r , which means that more accuracy in the calculation of φ_i will come at the cost of increased computational overhead. Once the value of r is calculated, φ_i can be approximated by:

$$\varphi_i = \sum_{n=1}^r \varphi_i^{(n)} \quad (24)$$

The calculation of φ_i is shown in the form of a pseudocode function below.

```

function  $\varphi_i = \text{calc\_phi}(V_i', \alpha_i)$ 
  -- order_  $V_i'$  is the order of the square matrix  $V_i'$ 
  order_  $V_i' \leftarrow \text{get\_order}(V_i')$ 
  n  $\leftarrow 1$ 
  F  $\leftarrow V_i'$  -- F is the matrix holding the probability of first
    --moving from each state to each other state of  $V_i'$  for
    -- the current value of n, where  $f(1,1)$  is the
    -- probability of finishing the vacation in n time
    -- quantum
  sum  $\leftarrow F(1,1)$ 
  --  $\sigma_i$  is the probability of a memory request occurring in n
  -- time quantum for the current value of n (Equation 15)
   $\sigma_i \leftarrow \alpha_i$ 
   $\varphi_i \leftarrow \sigma_i * F(1,1)$  -- Equation 16

  -- check to see if the error limit has been reached
  while (sum < 1 -  $\epsilon_r$ )
    --this next for loop calculates the next value of F as
    -- shown by Equation 14
    for x=1 to order_  $V_i'$ 
      for y=1 to order_  $V_i'$ 
        total  $\leftarrow 0$ 
        for z=1 to order_  $V_i'$ 
          total  $\leftarrow \text{total} + V_i'(x,z) * F(z,y)$ 
        end (for)
        F_new(x,y)  $\leftarrow \text{total} - F(y,y) * V_i'(x,y)$ 
      end (for)
    end (for)
    n  $\leftarrow 2$ 
    F  $\leftarrow F\_new$ 
    sum  $\leftarrow \text{sum} + F(1,1)$ 
     $\sigma_i \leftarrow \sigma_i + (1 - \alpha_i)^{n-1}$  -- Equation 15
     $\varphi_i \leftarrow \varphi_i + \sigma_i * F(1,1)$  -- Equation 16 and 17
  end (while)
end (function)

```

The above algorithm shows a method to calculate φ_i that depends on the probabilities in the probability transition matrix V_i' . However, many of the probabilities in the matrix V_i' also depend on the value of φ_i . In order to overcome this paradox, an iterative approach is taken. The iterative algorithm is outlined below.

```

for i=1 to N
   $\varphi_i \leftarrow \alpha_i$   -- initialize all  $\varphi_i$  parameters
   $\varepsilon_i \leftarrow 1$   -- initialize all  $\varepsilon_i$ 
end (for)

while (| $\varepsilon_i$ | > 10-12 for any i) -- check for convergence
  for i=1 to N
     $\varphi_{i\_old} \leftarrow \varphi_i$  -- save the last  $\varphi_i$  parameter, because a
    -- new one will be calculated
    -- create the  $V_i'$  matrix the based on the latest  $\varphi_i$ 
     $V_i' \leftarrow \text{build\_vacation\_process}(\varphi_i, i)$ 
    -- calculate the new value of  $\varphi_i$ 
     $\varphi_i = \text{calc\_phi}(V_i', \alpha_i)$ 
     $\varepsilon_i \leftarrow \varphi_{i\_old} - \varphi_i$  -- calculate the error between the
    -- current  $\varphi$  and the last one
  end (for)
end (while)

```

The above algorithm first assigns an arbitrary value to φ_i for all i , $1 \leq i \leq N$. In this case the value for φ_i is assigned α_i . While any value between 0 and 1 can be assigned and the algorithm will still work, using a value that is closer to the actual final value will result in faster convergence. Since φ_i is the probability that there will be a memory request while processor i is on vacation and α_i is the arrival probability, in general, the larger α_i is, the larger φ_i will be, which is why α_i is used as a starting guess.

An error value (ε_i) is kept for each processor. This is the difference between the latest value of φ_i and the value of φ_i that was calculated previously to the latest value.

When all of the error values are less than 10^{-12} , then this means that the algorithm has converged to a final value of φ_i for all i . The error values are initialized to 1 at the beginning to ensure that the while loop is entered the first time. Then the vacation matrices for each processor i (V_i, V_i^0, v_i) are built using the current value of φ_i for all i . Then a new value of φ_i for all i is calculated using the pseudocode function `calc_phi`. The difference between the new values of φ_i and the previous values of φ_i are calculated. This difference is checked to see if it is less than 10^{-12} for all i . If the error is smaller than the limit, then the values of φ_i have converged to the final values; otherwise the process needs to be repeated. Once the final values of φ_i are determined, there are no longer any unknowns, so the vacation process is fully defined.

This algorithm depends on convergence of the φ_i values. If the values of φ_i do not converge then the algorithm would continue indefinitely; therefore, it would not be stable. Explicit proof of convergence for this algorithm is not offered in this thesis; however, an argument for proof of convergence could be made that is similar to the proof of the stability of token passing rings made by Georgiadis and Szpankowski in [GeSz92].

4.6.2.3 Determining the Memory Access Waiting Probability

Now that the vacation process for each processor is defined, the probability distribution of the discrete time Markov chain that models the memory accesses of each processor can be determined. The amount of time that is spent waiting for access to the memory when the memory controller is currently serving another processor can be determined from the probability distribution.

First, the vacation process, $(v_i, V_i)_m$, for each processor i is built using the determined values of φ_j for all $1 \leq j \leq N$, as shown with (11), (12), (13), (14), (15), and (16). Then, the vacation process is used to build the probability transition matrix for the Markov chain representing the memory accesses of processor i , as shown with (10). Then, the steady state probability vector π_i can be calculated for the Markov chain. The steady state probability vector for P_i can be calculated by solving for $\pi_i = \pi_i P_i$ [Nels95]. The steady state probability vector represents the probability of being in any given state of P_i in a steady state condition.

The fourth block row of P_i shown in (10) represents all of the states where processor i has a pending memory request, but the memory controller is currently serving other processors. Therefore, the sum of the probabilities in the steady state vector that represent the states in the fourth block row of the P_i shown in (10) is the probability that the processor has a pending memory request, but the memory controller is serving another processor. Each of the rows in the matrix shown in (10) is made up of several sub-rows of which the number depends on the number of states in the memory service process, k . The first row shown in (10) is actually only one row. The second row represents k actual rows. The third and fourth rows are each made up of m sub-rows, where m is the order of the vacation process. The order of the vacation process is also dependent on the order of the memory service, and can be determined by the following equation:

$$m = (k + 1)(N - 1) \quad (25)$$

There is one block row in the vacation for each processor, except the processor who the vacation process is defined for, so there is $N-1$ block rows in the vacation. Each block row in the vacation is made up of $k+1$ rows, one row for the case where the currently serviced processor has 0 memory requests, and k rows for the case where the currently serviced processor has 1 memory request that is being serviced.

This means that the sum of the last m items in the steady state vector π_i will be the probability that processor i will have to wait for memory access when it has a pending request because the memory controller is currently serving another processor. The variable ζ_i is defined as the probability that processor i will have to wait for memory access when it has a pending request because the memory controller is serving other processors, and can be calculated as follows:

$$\zeta_i = \sum_{h=2+k+m}^{1+k+2m} \pi_i[h] \quad (26)$$

The notation $\pi_i[h]$ means the item h in vector π_i (where the first item in the vector is item $\pi_i[1]$).

4.6.2.4 *Adjusting the Partition to Account for Waiting Time*

Now that ζ_i for each processor can be calculated, these values can be used to adjust the length of the partition of the processing period that is being analyzed so it can be adjusted to account for the time that each processor spends waiting for memory access.

The partition ends when the first task that is executing in the partition ends. In order to determine which task ends first, the end time of each of the tasks is calculated

taking into consideration the memory access waiting time. The end time of each task can be calculated with the following equation:

$$t_{remaining_new_i} = \frac{t_{remaining_old_i}}{1 - \zeta_i} \quad (27)$$

Where $t_{remaining_old_i}$ is the time remaining in the task execution without considering the memory access wait times, from the beginning of the current partition being analyzed. The task that has the smallest $t_{remaining_new_i}$ is the task which will end first, and therefore this is the new partition time, defined as $t_{partition_new}$.

The other calculation that needs to be done in order to adjust the processing period to be able to analyze the next partition is to determine how much of the task processed on each of the processors is done within the analyzed partition. The ratio of the task executed in the partition time to the total execution time of the task that is executed on processor i is defined as θ_i . Given the total execution time of the task executed by processor when the memory access waiting time is not considered, $t_{total_task_i}$, the value of θ_i can be calculated by:

$$\theta_i = \frac{t_{partition_new}}{t_{total_task_i}} (1 - \zeta_i) \quad (28)$$

Given the ratio of the task that was analyzed in previously analyzed partitions, θ_{prev_i} , the remaining time that needs to be analyzed for each task can then be calculated by the following equation:

$$t_{task_remaining_i} = t_{total_task_i} - t_{total_task_i} (\theta_i + \theta_{prev_i}) \quad (29)$$

After the amount of each task that is remaining is calculated the entire processing period can be updated (as the example in Fig. 4-5), and the next partition can be analyzed.

4.7 Example using the MPSoC Analytical Model

This section will use an example to show how the analytical model described in the previous sections can be used to determine the pipeline processing period when the effect of memory contention between processors sharing global memory is considered. In the example the memory service will be modeled by a negative binomial process, with $k=5$ and $p_e=0.25$. The method of implementation is described in appendix B. The memory service time distribution is as shown in Fig. 4-1. The DFG for the tasks executed on the processors is as shown in Fig. 2-1. There are four processors in the system that can execute tasks. In this example, the four processors are homogeneous, meaning that each of the tasks will take the same amount of time to execute on any of the processors. The parameters of the tasks are as shown in the following table:

Table 4-1. Example task parameters.

Task	Executing Processor	Initial Task Execution Time	Probability of Memory Access Request (α_i)	Initial Start Time in Processing Period	Initial End Time in Processing Period
A	1	500	0.03125	0	500
B	2	1000	0.02	0	1000
C	3	2000	0.015	0	2000
D	2	500	0.025	1000	1500
E	4	500	0.015	0	500
F	1	2000	0.05	500	2500
G	4	1000	0.025	500	1500
H	2	1000	0.005	1500	2500

The scheduling of tasks for the purpose of the example was done using a heuristic algorithm proposed in [BaGa99] which is a variation of the list-scheduling algorithm originally proposed in [Hu61]. The algorithm for scheduling is as follows:

The algorithm is as follows:

1. For every node in the DFG, determine the longest completion time from that node to any finishing node. This will give a list of priorities, with the longest completion time having the highest scheduling priority (in the case of a tie, the priority can be chosen arbitrarily).
2. Take the node with the highest priority that has not yet been scheduled and schedule it to a processor in a slot with the earliest possible completion time. Do not consider which other nodes need to be finished before the current one can execute at this point.
3. Remove the scheduled node from the list.
4. Go to step 2 until the list is empty.

This is a greedy algorithm that does not necessarily result in the optimal solution, especially since the effect of memory contention is not considered. The next chapter discusses task scheduling for the MPSoC in greater detail. The arrangement of the tasks results in a processing period (t_p) of 2500 before the memory access waiting times are considered. This is the ideal time, as if every processor could access the memory whenever it was requested. The timing diagram for this system, before analyzing the memory access wait times, is as shown in Fig. 4-7.

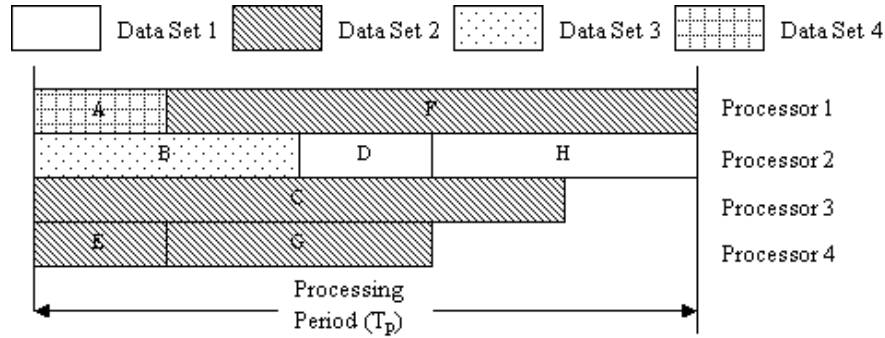


Fig. 4-7. Initial task arrangement in the processing period.

The first partition that is considered is from time 0 of the processing period to time 500. In this window, processor 1 is executing task A, processor 2 is executing task B, processor 3 is executing task C, and processor 4 is executing task E. Since all processors are active, $N=4$ for analyzing this window. Equation (21) can be used to determine the order of the vacation process for any of the processors. The order of the vacation process, m , will be 18. This means that the order of the matrix P_i for each processor i will be 42. After executing the iterative algorithm used to calculate φ_i as outlined in section 4.2.2, the values converge to: $\varphi_1=0.4297$, $\varphi_2=0.3489$, $\varphi_3=0.2964$ and $\varphi_4=0.2964$.

Now that the values of φ_i are calculated, the steady state probability distribution for the Markov chain, representing the memory accesses for each of the processors, can be calculated. Graphs showing these probability distributions are shown in Fig. 4-8.

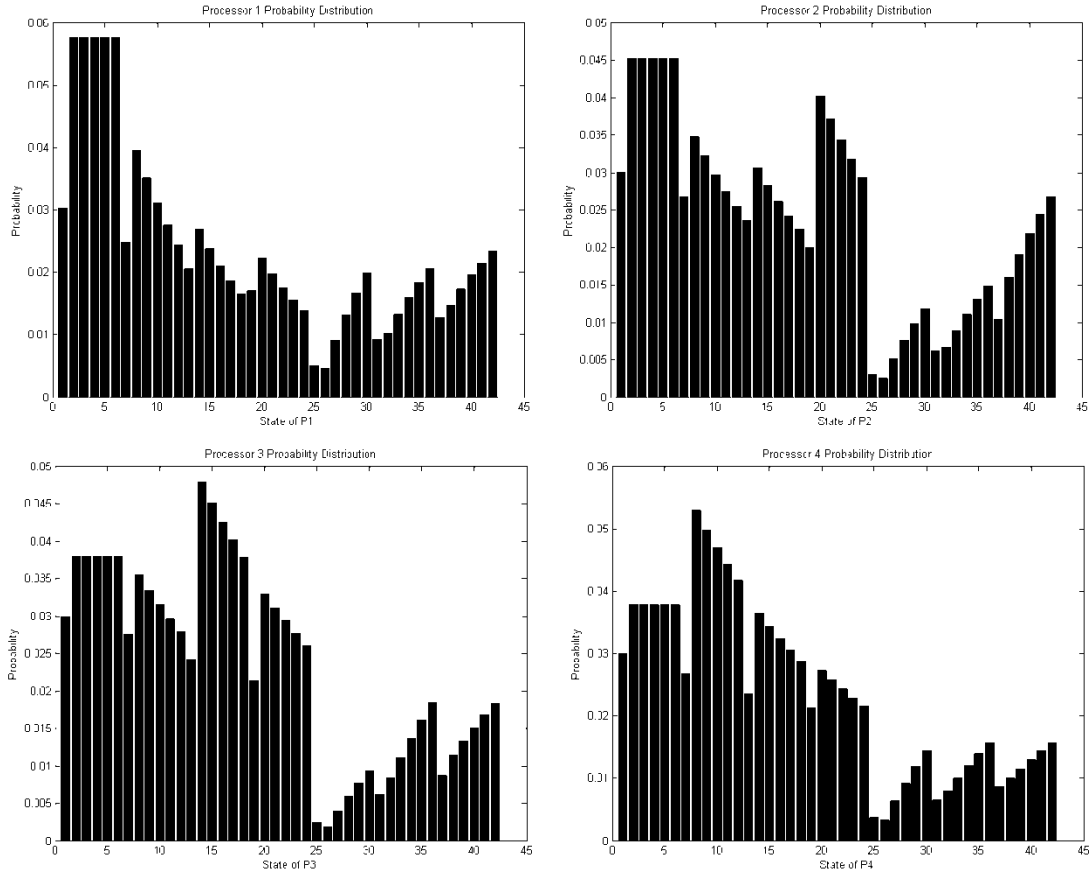


Fig. 4-8. Probability distributions of Markov chains representing memory access in first partition.

From these probability distributions, the probability of each processor having to wait to access the memory can be determined using (26). These probabilities are calculated to be: $\zeta_1=0.2627$, $\zeta_2=0.2178$, $\zeta_3=0.1876$, and $\zeta_4=0.1876$. The minimum probabilities of waiting is for processor 3 and processor 4, so $\zeta_{\min}=0.1876$. Using (27) the new time partition time can be calculated to be $t_{\text{partition_new}} = 616$, where the previous partition time was $t_{\text{partition_old}} = 500$. Using (28), the ratio of the task executed in the partition time to the total execution time of the task that is executed for each processor can be calculated as: $\theta_1 = 0.9060$, $\theta_2 = 0.4812$, $\theta_3 = 0.25$, and $\theta_4 = 1$. This means that after this partition is executed, 9.4% of task A is left to execute, 51.9% of task B is left, 75% of task C is left, and task E has finished being executed. After analysis of the first partition

the timing diagram of the system is adjusted to look as shown in Fig. 4-9. Also, the next partition that needs to be analyzed is shown.

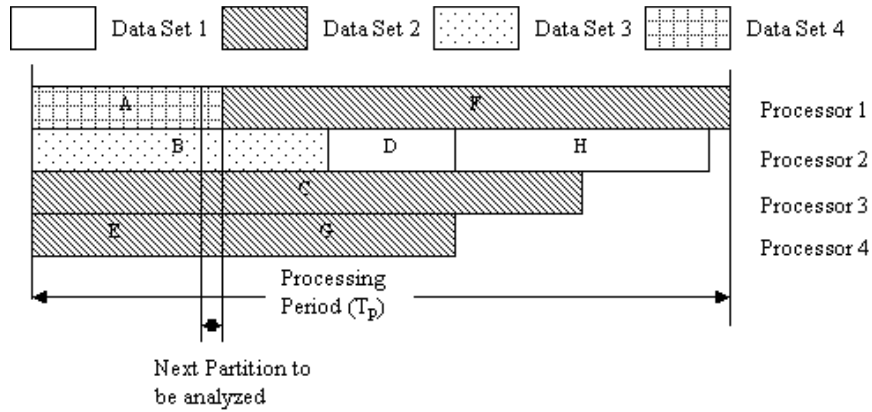


Fig. 4-9. Task arrangement in processing period after first partition analyzed.

The updated table of task information after analyzing the first partition is shown below.

Table 4-2. Task parameters after first partition is analyzed.

Task	Executing Processor	Task Execution Time	Probability of Memory Access Request (α_i)	Start Time in Processing Period	End Time in Processing Period
A	1	663	0.03125	0	663
B	2	1135	0.02	0	1135
C	3	2116	0.015	0	2116
D	2	500	0.025	1101	1601
E	4	616	0.015	0	616
F	1	2000	0.05	620	2620
G	4	1000	0.025	587	1587
H	2	1000	0.005	1601	2601

After the analysis of the first partition, the total processing period has increased from 2500 to 2663. The next partition that needs to be analyzed is from time 616 to time 663. In this window, processor 1 is executing task A, processor 2 is executing task B, processor 3 is executing task C, and processor 4 is executing task G.

This process is repeated until the entire processing window is processed. Each of the iterations will not be shown here, but there are six iterations in total needed to fully

analyze the example problem given. At the end of the analysis, the updated table of task parameters is as shown below.

Table 4-3. Task parameters after analyzing the entire processing period.

Task	Executing Processor	Task Execution Time	Probability of Memory Access Request (α_i)	Start Time in Processing Period	End Time in Processing Period
A	1	683	0.03125	0	683
B	2	1329	0.02	0	1329
C	3	2514	0.015	0	2514
D	2	733	0.025	1329	2062
E	4	616	0.015	0	616
F	1	2627	0.05	683	3310
G	4	1448	0.025	616	2064
H	2	1043	0.005	2062	3105

The processing period (t_p) at the end of the analysis is 3310, compared to the initial processing period of 2500. This same analysis can also be done using a different number of processors to evaluate the effect that the number of processors has on the execution time. The following graph shows the expected processing time using 1 through 6 processors to process the same DFG done in the example above.

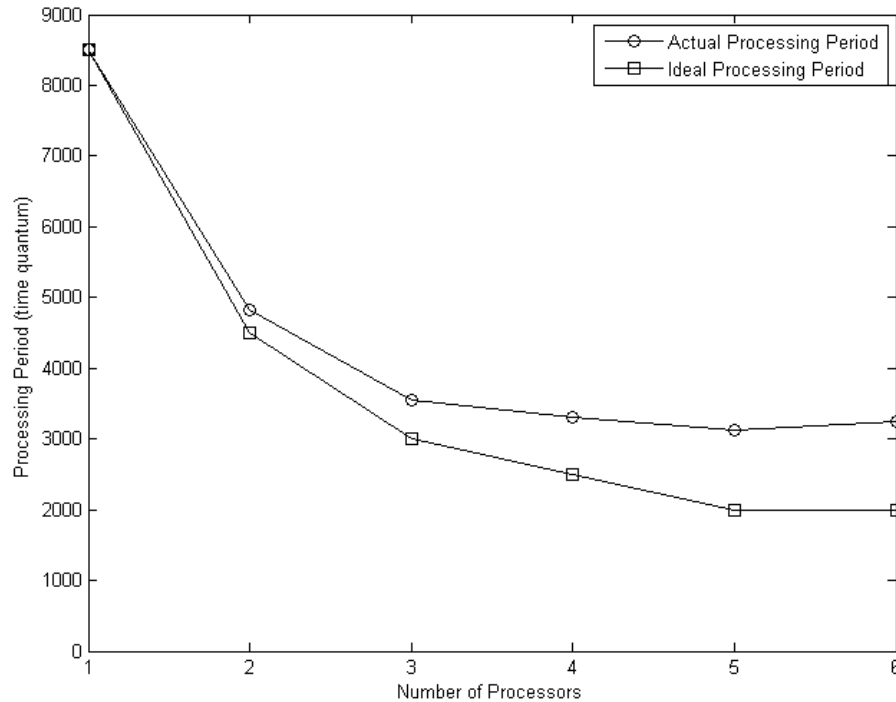


Fig. 4-10. The ideal processing period is compared with the actual processing period. The ideal processing period does not consider the effect of memory interference, where the actual processing period does consider memory interference.

Fig. 4-10 shows that as the number of processors increases, the difference between the ideal processing period and the actual processing period increases. This is an expected result, since more processors in the system means that there is a greater probability of memory contention. The above example demonstrates the importance of considering the effect that waiting for memory requests can have in shared memory multiprocessor systems. If the time that a processor waits for memory access when other processors are accessing the memory is ignored, the results can be significantly different than if this time is considered. Fig. 4-10 shows that the expected reduction in processing time may be perceived to be greater than the actual reduction in processing time that can be achieved if the effect of memory interference is not considered. The analysis method

proposed provides a method for determining the typical amount of time that each processor in an implementation of the MPSoC framework will wait for memory access because another processor is being served.

Chapter 5

TASK ALLOCATION OPTIMIZATION

5.1 Motivation for Task Allocation Optimization

In the previous chapter, a model of analysis of a particular task schedule was developed. This model can be used to determine the average processing period for a given set of tasks assigned and scheduled to a given set of processors, with consideration of the effects of global memory contention. This provides a method to evaluate potential MPSoC solutions, but it does not provide a method to determine an optimal solution. There has been much research in the area of scheduling of tasks in an MPSoC, and mapping tasks to processors within an MPSoC. These are technically two separate activities, where mapping of tasks to processors is the act of determining which task will be processed by which processor, and scheduling is the act of determining the order of the tasks to be processed on a given processor. In this thesis, the term “task allocation” will be used as the combination of these two activities, where mapping of the tasks to processors and scheduling of the tasks on each processor happens at the same time. Finding an optimal task allocation of tasks to a given set of processors is an NP (Nondeterministic Polynomial) -complete combinatorial optimization problem [BeRo08]. This means that an optimal solution cannot be found in polynomial time. There are many combinatorial optimization techniques that have been used to find solutions to NP-complete combinatorial optimization problems. Often the goal of a combinatorial optimization algorithm is to find a “good” solution in a reasonable amount of time, rather than spending a much longer period of time finding the global optimum.

5.2 Overview of Combinatorial Optimization Techniques Applied to Task Allocation Problems

There are many techniques that have been developed to find optimal solutions to task scheduling and mapping combinatorial optimization problems. Some of these include greedy algorithms [BaGa99], genetic algorithms (GA) [ZhSh07] [GuAK10], tabu search (TS) [ShPS09], particle swarm optimization (PSO) [VFMA09], quantum-inspired evolutionary algorithm (QEA) [YaHa09], and simulated annealing (SA) [LiHs90] [NaDS92] [HSSA10]. Each of the combinatorial optimization techniques mentioned above have strengths and weaknesses, and as a result are suitable for different types of applications. A benefit of the SA algorithm over other combinatorial algorithms is that it asymptotically approaches the global optimum. That is, given an infinite amount of time, SA is guaranteed to reach the global optimal solution. Practically this does not mean that SA will find the globally optimal solution faster than other algorithms, but it does mean that new solutions tend towards better solutions as the algorithm progresses. This is a very beneficial property for problems where a good solution is required, but not necessarily the global optimal solution. For this reason SA is a popular algorithm in routing problems (such as printed wiring board routing, or FPGA routing); these problems do not need the best solution possible, but rather a good solution in a reasonable amount of time.

Sometimes, human beings have an astonishing way of arriving at a solution for very difficult (NP-complete) problems. For example, given a set of tasks and resources on which those tasks can be performed (like the job a manager does by allocating tasks to employees) we can sometimes find a good solution in a very short amount of time. While

the actual process we use to arrive at a good solution is debatable, it seems reasonable that the process itself consists of first identifying a reasonable solution, then incrementally changing the solution and observing the behaviour of the system. If the system seems to move to a more reliable and stable solution, we keep that change and continue with the iterative process. If the system does not seem to move to a more stable solution, we inevitably need to keep the change, but try a different approach in the next iteration. Tweaking the solution this way in an iterative manner eventually leads to an acceptably good solution. This process is remarkably similar to the annealing of solids. After the initial formation of a solid, if heat is added and the solid is allowed to cool, the solid will either come to a more stable state or not. We can then modify the way heat is added to the solid to make the solid move to a more stable configuration. As such, this thesis tries to mimic the complex way humans use to solve combinatorial optimization problems by simulating the process of annealing in the task allocation algorithm.

Furthermore, in the task allocation problem we can observe that making a small change in the task allocation can bring about a large change in performance. For example, if two tasks are memory intensive, then allocating their execution on different processors at approximately the same time can result in higher global memory contention. If these tasks were spread out in as very little as one processing period, the performance of the system may dramatically improve. This observation motivates us to use chaos theory to drive the task allocation algorithm. As such, this thesis applies chaos theory to simulated annealing.

Simulated Annealing is a method for solving combinatorial optimization problems first proposed by Kirkpatrick et al [KiGV83]. SA is based on an analogy to the annealing of solids. The basic idea of SA is that an initial solution is chosen at random, and a temperature parameter starts at a high value. A new solution is chosen by making a small random perturbation of the solution. If the new solution is better, then it is kept, but if it is worse, it is kept with some probability related to the current temperature and the difference between the new solution and the previous solution. When the temperature is high, the probability of accepting the solution is high, and as the temperature decreases (according to some cooling schedule), the probability of accepting a solution worse than the current solution decreases. This method does not get stuck in local minima, because of the possibility of acceptance of worse solutions that can result in escaping from the local minima. It has been shown that the SA method can be guaranteed to find the global minimum with a sufficiently slow cooling schedule [Haja88]; however, this generally involves an unacceptably long solution convergence time, and is likely to take longer than a simple sequential search. Therefore, there has been much work in determining the parameters for particular problems that will result in a good solution in a reasonable amount of time, even if it is not the global optimum.

Simulated Annealing has been applied to task allocation and scheduling problems in multiprocessing systems in previous work. Van Laarhoven *et al.* [VaAL92] applied SA to a job shop scheduling problem where the goal is to find the optimal allocation of jobs for a set of machines. This problem is a well-known generalized scheduling problem. The paper, [VaAL92], demonstrated the benefits of SA over other heuristic methods for job scheduling. There are several papers [EPKD97] [FLPS10] [WiCL97] that have compared

SA to other optimization methods for problems that are similar to the task allocation problem presented in this thesis. These works demonstrate that SA has certain advantages and disadvantages over other optimization techniques, but remains as one of the leading methods for task scheduling and mapping optimization problems.

Chaotic Simulated Annealing (CSA) was proposed by Shaw and Kinsner [ShKi96] in the training of multilayer feedforward neural networks. This work explored several different chaotic attractors used to generate perturbations of weights of neural networks. Chen and Aihara [ChAi95] also proposed a variant of CSA as a method to find optimal solutions to combinatorial optimization problems using Hopfield neural networks (HNN) [Hopf82]. This work used transient chaos in the stabilization of the HNN and was demonstrated using the travelling salesman problem. There have been several papers that have expanded on this work [ChAi97] [MaAi02] [WaSm98] [WaTi00].

In all of the CSA algorithms applied to neural networks, the weights of the neural networks are continuous variables that are adjusted through variations of CSA to find optimal solutions to the applicable problems. While Chen and Aihara applied the neural networks in solving a combinatorial optimization problem, the variables perturbed at each iteration were continuous variables, and so a chaotic variable could be easily mapped to the continuous variable.

Another version of CSA was proposed by Mingjun and Huanwen [MiHu04]. The CSA proposed by Mingjun and Huanwen differs from the previous work in that it did not involve neural networks and it applied only to optimization of continuous functions, not to combinatorial optimizations problems. This method generated solutions at each

iteration of the SA algorithm by mapping values derived from a chaotic system to the solution space. As the temperature parameter decreased, the variation over the solution space also decreased, and the solution eventually would converge to a final optimal solution. This method depends on the system to be a continuous system where a chaotic variable could be easily mapped to a point in the solution space.

SA and a variation of the CSA algorithms are applied to the MPSoC task allocation problem in this thesis, and are compared to demonstrate the differences in the two algorithms for the MPSoC task allocation problem.

5.3 Application of Simulated Annealing to the Task Allocation Problem

The SA algorithm starts with an initial random solution, and then perturbs the solution on successive iterations to compare the solution to the previous solution. This means that it is essential to have a method to perturb the current task allocation to result in a new task allocation. The perturbation must be one that allows all possible solutions to be reached from any other solution, given enough of the perturbations. The perturbation should also result in a relatively small change in the solution, such that a local area could be explored to find a minimum.

There are many ways to generate different combinations of tasks that would result in perturbations that meet the above requirements, but the following task movement has been chosen for this thesis. For a single perturbation, a task is chosen at random and is moved in one of the following directions: up, down, left, or right. These directions are relative to a task arrangement in which each of the processors represents a row, and each

of the tasks allocated to a processor is shown in a row and ordered from left to right. This means that each task has a row index (which is the processor), and a column index (which is the order that the task is assigned to the processor). An up move is defined as decreasing the row index of the chosen task, but maintaining the column index. The affected tasks in the target row are shifted to the right to make room for the new task. A down move is defined as increasing the row index and maintaining the column index. The row index can be wrapped around if an up move is made by a task in the top row, or if a down move is made by a task in a bottom row. A left move is made by maintaining the row index and decreasing the column index. A right move is made by maintaining the row index and increasing the column index. In the case of a right or left move, the task position is swapped with the task next to the chosen task. Wrap-around of the column index is not permitted, so a left move is not possible if a task is the first task in a row, and a right move is not possible if a task is the last task in a row. An example of a down task movement by task D is shown in Fig. 5-1.

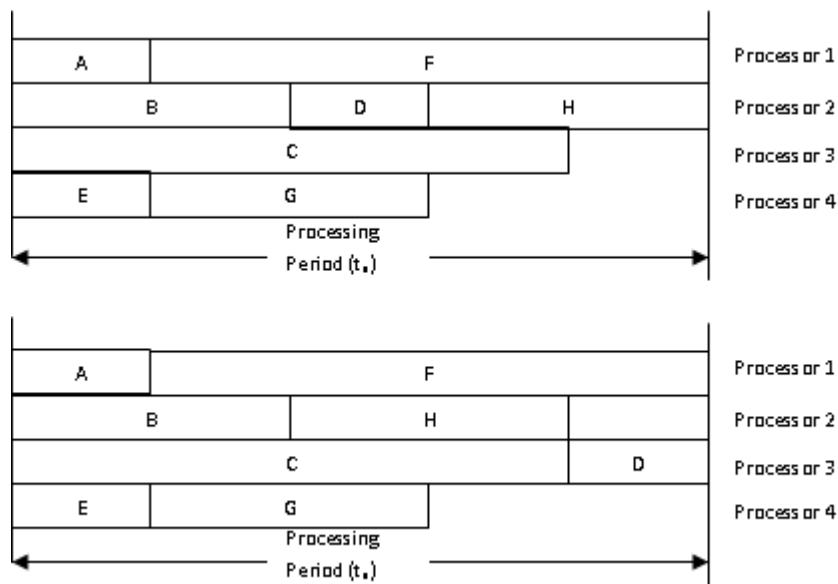


Fig. 5-1. Example of a down movement of task D.

The SA algorithm starts with an initial solution and a temperature parameter. The temperature parameter is decreased on each successive iteration until a minimum temperature value is reached. Also, at each iteration, a perturbation is made to the current solution and the cost function is calculated. If the new solution is better than the previous, then it is kept. If the new solution is worse, then it is kept with a probability associated with the temperature parameter and the difference between the new solution cost and the old solution cost. The SA procedure applied in the task allocation problem is shown below.

1. Choose an initial task arrangement ($G_{current}$)
2. Determine the processing period (t_p) of $G_{current}$
3. Initialize the temperature parameter ($T_1 \leftarrow T_{max}, n \leftarrow 1$)
4. Set the initial solution as the global minimum $G^* \leftarrow G_{current}, t_p^* \leftarrow t_p$
5. Perturb the current solution to generate a new solution (G_{new})
6. Determine the processing period (t_{pnew}) for the new solution G_{new}
7. If $t_{pnew} \leq t_p$ then $t_p \leftarrow t_{pnew}$ and $G_{current} \leftarrow G_{new}$, otherwise go to Step 9
8. If $t_{pnew} \leq t_p^*$ then $t_p^* \leftarrow t_{pnew}$ and $G^* \leftarrow G_{new}$, go to Step 10
9. If $t_{pnew} > t_p$ then accept the new solution with a probability of $e^{\left(\frac{-\Delta E}{k_b T_n}\right)}$,
where $\Delta E = t_{pnew} - t_p$, and k_b is a constant.
10. Reduce the temperature according to a cooling schedule $T_{n+1} = f(T_n)$
11. If $T_{n+1} \geq T_{min}$ then $n \leftarrow n + 1$ and go to Step 5, otherwise go to Step 12
12. Output best solution found, G^*

The parameters used for the task allocation problem are: $T_{max} = 400$, $T_{min} = 1$, $k_b = 2$, and the cooling schedule function is defined as $f(T_n) = T_n - 1$. A common variation of the SA algorithm is to allow for restarting of the search. Search restarts occur if a new solution is not accepted J times in a row, where J is some predetermined constant. After J rejections in a row, the current solution is set back to the global minimum, G^* . This is intended to prevent a solution from going too far off the path and searching areas of the solution space that will only have poor solutions. In the case of the task allocation problem, the search restart parameter was set to $J=5$, meaning that if 5 solutions in a row were rejected, then the current solution was set back to the current global minimum value.

5.4 Application of Chaotic Simulated Annealing to the Task Allocation Problem

Three variations of CSA are proposed and applied specifically to the task allocation problem in this thesis. The three variations of CSA are modified versions of the CSA proposed by Mingjun and Huanwen [MiHu04] for continuous variable optimization. First the Mingjun and Huanwen version of CSA is described, followed by a discussion of the difficulties of mapping a chaotic variable to a combinatorial optimization solution space, and finally, followed by a description of each of the three CSA methods proposed in this thesis.

5.4.1 Continuous Function Chaotic Simulated Annealing

The CSA algorithm proposed by Mingjun and Huanwen [MiHu04] is specific to continuous function optimization problems and does not address combinatorial

optimization problems. The algorithm is similar to the traditional SA problem (as described in section 5.3), but uses a chaotic system in the perturbation step to generate new solutions at each step (Step 5 of the SA procedure). Rather than making a small perturbation of the previous value, a new value is chosen by generating a value from a chaotic equation, and then mapping that value to the search space interval. For example, if the all values in the search space are in the interval of $[a, b]$, then a value from a chaotic system, z , is generated between 0 and 1, then the initial solution is calculated as $x = a + (b - a) \times z$. At each successive iteration a new solution is generated by the equation $y_m = x_m + \alpha_m \times (b - a) \times z_m$, where z_m is the chaotic variable at the m^{th} iteration, and α_m is a factor that reduces as the temperature decreases according to the equation $\alpha_m = \alpha_m \times e^{-\beta_c}$, and β_c is a constant.

Two different chaotic systems were used to generate the chaotic value z_m . The first equation was the well known logistic map:

$$z_{k+1} = \mu z_k (1 - z_k) \quad (30)$$

Where $\mu = 4$. The second equation, referred to as the *new chaotic map*, was:

$$z_{k+1} = \eta z_k - 2 \tanh(\gamma z_k) e^{-3z_k^2} \quad (31)$$

Where $\eta = 0.9$ and $\gamma = 5$.

Since this version of CSA is only applicable to continuous intervals, the new values generated at each iteration need to be able to be explicitly mapped to the interval. The chaotic systems were used instead of just generating random numbers in order to obtain a particular probability distribution, which could produce better results (i.e. faster convergence to the global optimum) than a Gaussian or uniform distribution. Mingjun

and Huawei show the benefit of the CSA over SA experimentally by showing the CSA algorithms converge to a global minimum of several continuous functions in less time than SA. Chen and Dong [ChDo08] expand on the work of Chen and Aihara to prove that the Markov Chain associated with the CSA algorithm is weakly ergodic, which guarantees that the asymptotic behaviour of the algorithm is independent of the initial states. A significant difference between the CSA algorithm and the SA algorithm is the probability distribution of the solutions generated at each iteration. If the functions where an optimal solution is to be found, have local minima that are far apart (and far from the initial solution), then the probability mass function resulting from the chaotic variables will be more likely to search a far reaching area, and therefore more likely to find the global minimum. However, this is dependent on the distribution of the optimal solutions, and the initial solution, and therefore there is no guarantee that CSA will produce a better solution than traditional SA. The histogram of the values produced by the logistic map and the new chaotic map are given in [MiHu04] and shown in Fig. 5-2.

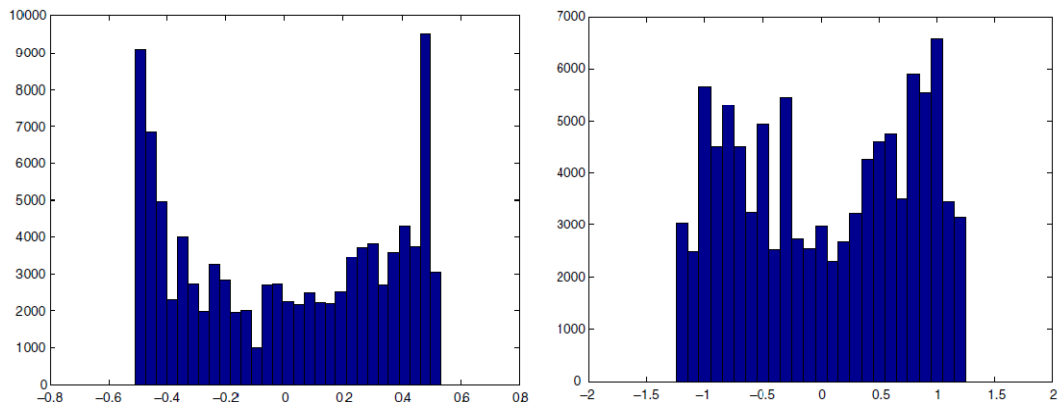


Fig. 5-2. Histogram of values generated by Logistic Map (left) and New Chaotic Map (right), from [MiHu04].

5.4.2 Mapping a Chaotic Variable in the Solution Space

In order to adapt the CSA method developed by Mingjun and Huanwen to the task allocation combinatorial optimization problem, a method to perturb the tasks in such a way that the solution space is searched according to the probability mass function derived from a chaotic variable is required. The mapping of the solution to a chaotic variable is done by mapping the chaotic variable to the number of perturbations made. While this is not guaranteed to search the solution space exactly according to the specified value (for example, since some perturbations may cancel out others), overall the more perturbations that are made, should result in solutions that are farther apart. Therefore, a maximum and minimum number of perturbations are chosen, and a generating function generates a value between 0 and 1. How the generating function generates the value between 0 and 1 differs for the different CSA variations, and will be described in the following sections. The number of perturbations is then mapped to the generated value from 0 to 1. This mapping also differs between the different variations of CSA, and it will be described in the following sections. In all cases, a maximum number of 30 perturbations and a minimum number of 1 perturbation are used for the experimental results.

5.4.3 CSA1 Method

The first variation of the CSA algorithm developed (referred to as CSA1 from this point forward) used the logistic map (given in (30), with $\mu = 4$) multiplied by an attenuation variable, α_n , to generate a chaotic variable, z_k , which was a value between 0 and 1. The initial attenuation variable is $\alpha_0 = 1$, and was decreased on each iteration of the CSA algorithm, according to the equation $\alpha_{n+1} = \beta_d \alpha_n$, where $\beta_d = 0.99$. The value

from the logistic map was determined by iterating the equation a random number of times, between 1 and 400, with an initial value of $z_0 = 0.65$. The number of perturbations made at each iteration in the algorithm was then determined by the mapping function:

$$num_{perturbs} = \alpha_n z_k (perturb_max - perturb_min) + perturb_min \quad (32)$$

CSA1 has the result of using the probability mass function of the logistic map with $\mu = 4$ but attenuated as the number of iterations increases. Therefore, initially there is a higher probability of a larger number of perturbations, and this number is decreased as the algorithm progresses. This has the effect of being able to search a large solution space initially, and then reducing the solution space to areas where the global minimum is more likely to exist. Fig. 5-3 shows a three dimensional histogram of the number of perturbations (y-axis) vs. the number of iterations into the logistic map (z-axis) vs. the number of iterations in the CSA1 algorithm (x-axis). The result of the CSA1 algorithm is that the number of perturbations is most likely to be either a large number, or a small number, but not likely to be in the middle for early iterations of the algorithm. As the algorithm progresses, the number of perturbations decreases, with the hope of narrowing in on the global optimum.

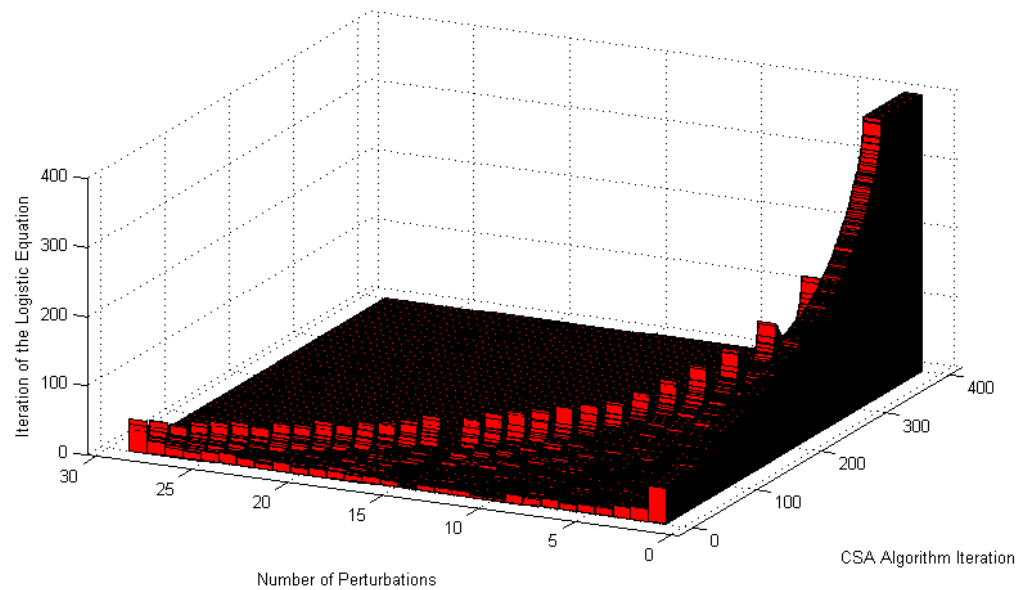


Fig. 5-3. Histogram of perturbations for CSA1.

5.4.4 CSA2 Method

The second variation of the CSA algorithm, (referred to as CSA2 from this point forward), was created based on the realization that the best probability distribution of the solutions was not known, and therefore it would be best to change the probability mass function of the chaotic variable as the algorithm progresses, with the hope of efficiently covering the solution space. Again the logistic map, (30), was used as the basis of the chaotic variable, but rather than using a fixed value of μ as was the case for CSA1, the value of μ is changed as the algorithm progresses to create different probability distributions at successive iterations. The mapping of the chaotic variable to the number of perturbations is done differently for CSA2 than it was done for CSA1. In this case, the chaotic variable is a value between 0 and 1, and the number of perturbations is a value that is proportional to the difference between the chaotic variable and the mean value for the distribution of the chaotic variable at its current CSA2 iteration. That is, the further

away the chaotic variable is from the mean value, the more perturbations will be made. Changing the value of μ means that the probability of the number of perturbations will have a distribution that follows the bifurcation diagram for the logistic map. At the beginning of the CSA2 algorithm the value of $\mu = 4$. The value of μ is changed linearly to a final value of $\mu = 2.8$ in the final CSA2 iteration. Fig. 5-4 shows the possible values of the chaotic variable at each iteration of the CSA2 algorithm. The red line through the diagram is the mean value, which the chaotic variable is compared to determine the mapping to the number of perturbations. This diagram is the bifurcation diagram for the logistic map. Fig. 5-5 shows a three dimensional histogram of the number of perturbations (y-axis) vs. the number of iterations into the chaotic map (z-axis) vs. the number of iterations in the CSA2 algorithm (x-axis). The histogram shows that there is rich variation in the probability distributions for the first 150 iterations or so. After that point, the periodic points in the bifurcation diagram are reached, and the number of perturbations is almost constant and continuously decreasing as the iterations in the CSA2 algorithm continue. The idea of this version is that, in the first 150 iterations there is a wide sweeping of the solution space, and as the algorithm progresses, the solution space is searched in a smaller and smaller area, with the hope of narrowing in on the global optimum.

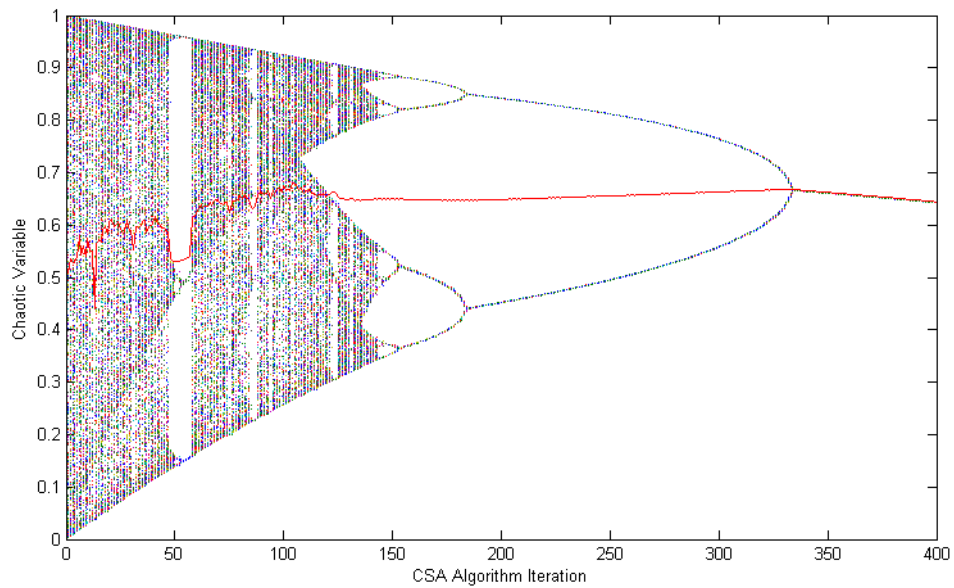


Fig. 5-4. Chaotic variable vs. algorithm iteration for CSA2.

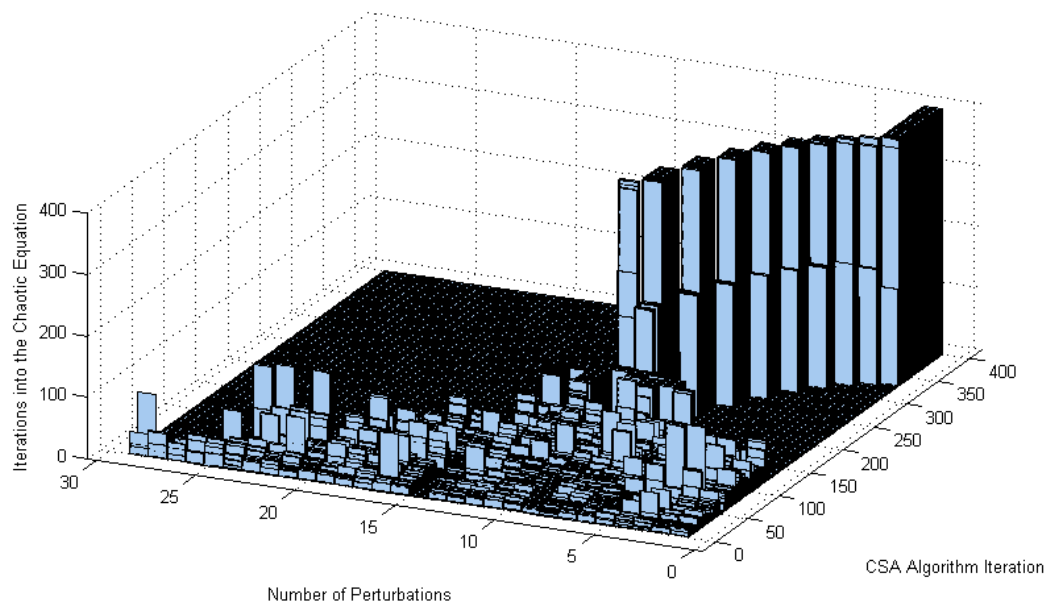


Fig. 5-5. Histogram of perturbations for CSA2.

5.4.5 CSA3 Method

The third variation of the CSA algorithm (referred to as CSA3 from this point forward) is almost exactly the same as CSA2, except that, instead of using the logistic

map to generate the chaotic variable, the equation referred to as the new chaotic map, (31), is used to generate the random variable. The γ of this equation is varied from a starting value of $\gamma = 5$ to a final value of $\gamma = 0.5$ over the iterations of the CSA3 algorithm. This chaotic map results in values outside of the range of 0 to 1, and therefore the chaotic variables are normalized to fall within the range of 0 to 1. Fig. 5-6 shows the possible values of the chaotic variable at each iteration of the CSA3 algorithm. The red line through the diagram is the mean value, to which the chaotic variable is compared to determine the mapping to the number of perturbations. This diagram has similar features to the bifurcation diagram of the logistic map, but there are different variations, which result in different probability mass functions at each iteration. Fig. 5-7 shows a three dimensional histogram of the number of perturbations (y-axis) vs. the number of iterations into the chaotic map (z-axis) vs. the number of iterations in the CSA3 algorithm (x-axis). The histogram shows that there is rich variation in the probability distributions for the first 275 iterations or so. After that point, the periodic points in the diagram are reached, and the number of perturbations is almost constant and continuously decreasing as the iterations in the CSA3 algorithm continue. The idea of this version is that, in the first 275 iterations, there is a wide sweeping of the solution space, and as the algorithm progresses, the solution space is searched in a smaller and smaller area, with the hope of narrowing in on the global optimum.

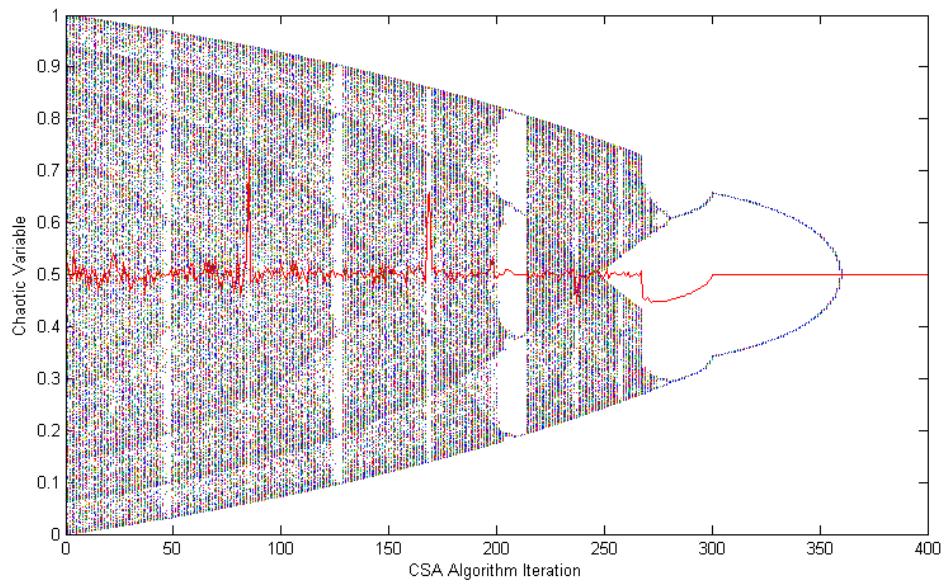


Fig. 5-6. Chaotic variable vs. algorithm iteration for CSA.

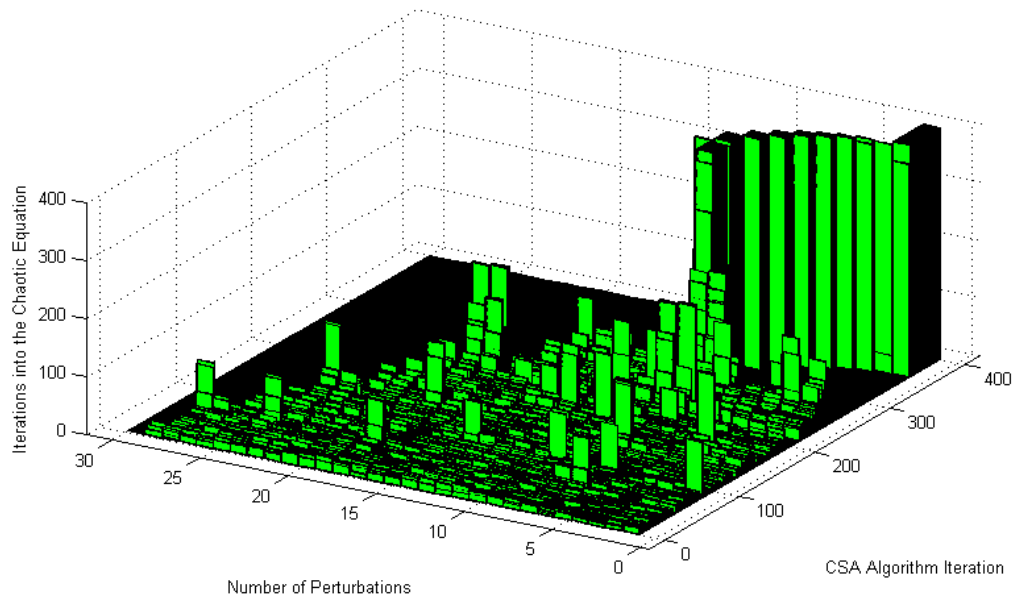


Fig. 5-7. Histogram of perturbations for CSA3.

5.5 Comparison of SA and CSA Results

The three variations of the CSA algorithms and the traditional SA algorithm were implemented on a task allocation problem where 16 tasks were to be allocated on 4

processors for two experiments, and then on 3 processors for another two experiments. Description of the method of implementation for these algorithms is described in appendix B. In both cases the systems are assumed to be homogeneous multiprocessing systems, where the tasks have the same initial processing time on all processors in the system. The method proposed is also valid for heterogeneous multiprocessing systems. The only difference in that case is that the initial task times and probability of memory access request for each of the tasks must differ depending on the processor that the task has been assigned to. This means that a unique table of initial tasks times and probability of memory access requests are required for each processor in the system (or each processor type, if there are more than one processor of the same type). The characteristics of the tasks for the first two experiments are shown in Table 5-1, and characteristics of the tasks for the second two experiments are shown in Table 5-2.

Table 5-1. Task characteristics for experiments 1 and 2.

Task	Experiment #1 and #2 Characteristics	
	Initial Task Time (ns x 10)	Probability of Memory Access Request
0	1484	0.054
1	1484	0.054
2	1484	0.054
3	1484	0.054
4	1484	0.054
5	1484	0.054
6	968	0.0525
7	1268	0.0549
8	968	0.0525
9	968	0.0525
10	1268	0.0549
11	968	0.0525
12	2630	0.0681
13	1268	0.0549
14	1368	0.0459
15	1368	0.0459

Table 5-2. Task characteristics for experiments 3 and 4.

Task	Experiment #3 and #4 Characteristics	
	Initial Task Time (ns x 10)	Probability of Memory Access Request
0	500	0.25
1	800	0.05
2	1500	0.01
3	700	0.025
4	250	0.35
5	350	0.05
6	3000	0.04
7	450	0.15
8	600	0.075
9	1800	0.025
10	100	0.1
11	1100	0.08
12	900	0.095
13	950	0.16
14	850	0.09
15	2100	0.06

In order to first get an idea of the size of the solution space for this problem, an estimate of the solution space can be made. This estimate can be made by considering a method of assigning the tasks to processors. If there are 4 processors, determining which processor a particular task is assigned requires 2 bits of information. Consider a method of assigning the tasks in order to each processor, where as a task is assigned it is given the left most available slot in the processing period. This means that all possible solutions could be arrived at by the permutation of task assignment, where each task could be assigned to any processor. Since two bits are required to specify the processor that a task is assigned to, and there is no restriction as to which a processor can be assigned, for a given permutation of task assignments, there are 2^{2N} , possible assignments, where N is the number of tasks. In the experiment case where N=16, this means that for every permutation of task assignments, there are $2^{32} = 4.29 \times 10^9$ possible arrangements. This number needs to be multiplied by the number of possible permutations of assignment of the tasks. The number of possible permutations that the tasks can be assigned in is given

by $N! = N \cdot N - 1 \cdot N - 2 \cdots 1$, for $N=16$; this means there are approximately 2.09×10^{13} , which means that there are $(4.29 \times 10^9)(2.09 \times 10^{13}) = 8.98 \times 10^{22}$ possible ways to assign the tasks to processors. Many of these ways to assign the tasks to processors will end up with the same solution, so this is an upper limit to the number of possible solutions, not the actual number of possible solutions, but it is obvious that the number of actual solutions must be very large.

Four different experiments were done in total. Experiments #1 and #2 were done with four processors and with task characteristics that did not vary significantly in execution time and probability of memory requests. Because the task characteristics do not vary significantly, this means that a very large number of the solutions will be fairly close to the global minimum. That is, there will be many “good” solutions. Experiments #3 and #4 were done with three processors and with task characteristics that varied more significantly. In this case, there will not be as many “good” solutions as there would be in the case where the task characteristics are more similar.

In Experiment #1 and Experiment #3 the initial solution was chosen to be an obviously bad solution (and therefore likely to be far away from the global optimum). In these cases, all of the tasks were assigned to a single processor. In Experiment #2 and Experiment #4, the initial solution was chosen to be a more reasonable solution, where the tasks were evenly divided among the processors. The initial task assignment for Experiment #1 is shown in Fig. 5-8, and the initial task assignment for Experiment #2 is shown in Fig. 5-9.

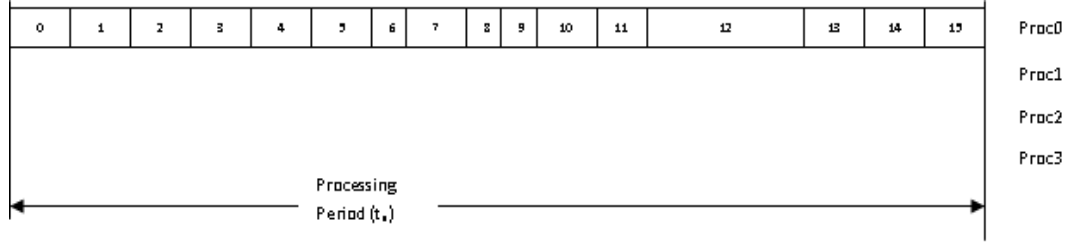


Fig. 5-8. Initial task solution for experiment #1.

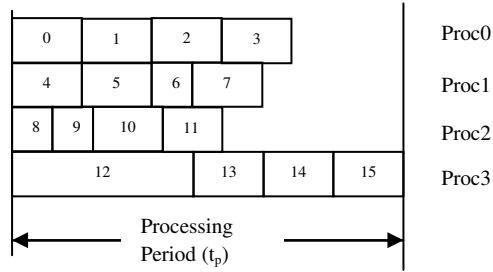


Fig. 5-9. Initial task solution for experiment #2.

5.5.1 Time Series Analysis

The length of the processing periods calculated at each iteration of the algorithms in Experiment #1 are shown in Fig. 5-10.

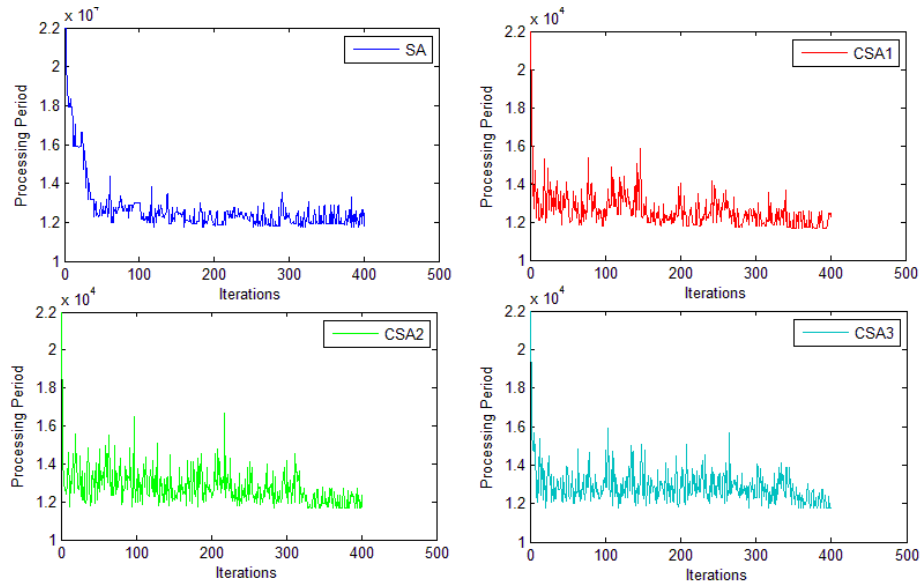


Fig. 5-10. Processing period calculated at each iteration for experiment #1.

These show that the traditional SA algorithm takes longer to find a solution close to its final solution, where the other algorithms move down to a pretty good solution in

only a few iterations. After the SA algorithm first reaches a solution that is close to the final solution, it does not have wide variations, showing that it is likely exploring local minima in a relatively small area of the solution space. While the three CSA algorithms achieve a solution close to the final solution relatively quickly, there is much more variation on further iterations, showing that a wider area of the solution space is being explored. Fig. 5-11 shows the number of perturbations at each iteration for each of the algorithms for Experiment #1. Similarities between the number of perturbations and the three dimensional histograms in Fig. 5-3, Fig. 5-5, and Fig. 5-7 can be seen, as is expected.

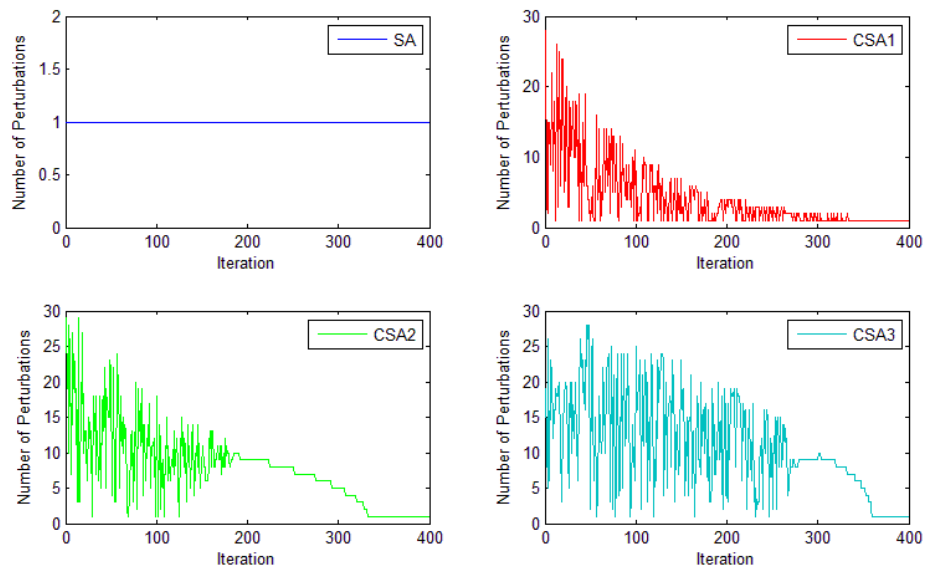


Fig. 5-11. Number of perturbations at each iteration for experiment #1.

Fig. 5-12 shows the processing periods at each iteration for Experiment #3.

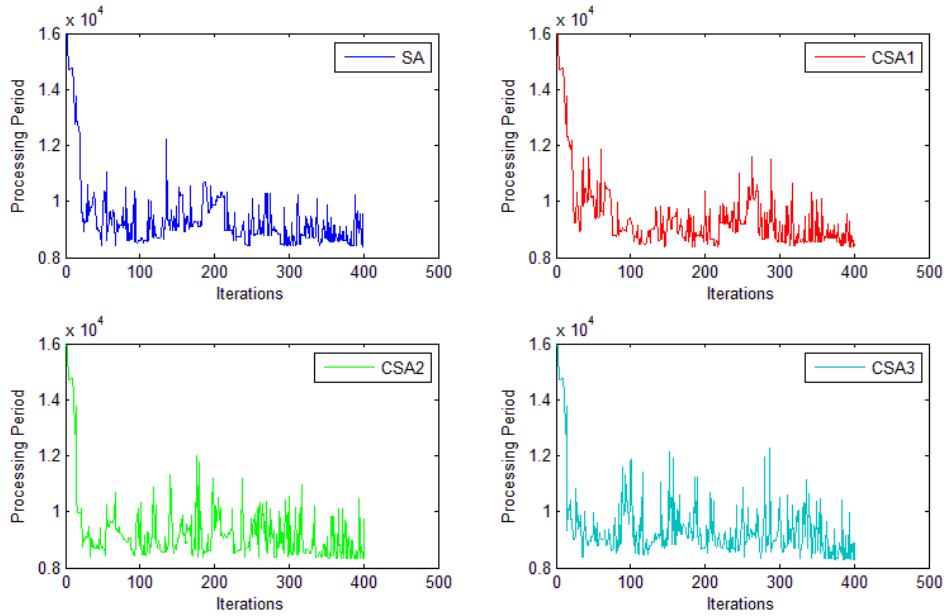


Fig. 5-12. Period calculated at each iteration for experiment #3.

Similar to Experiment #1, this experiment started with a poor solution, and so there is a steep descent to an area where better solutions are explored. However, because the characteristics of the tasks vary more significantly for Experiment #3 than for Experiment #1, there are more poor solutions, and so it is more likely that local minima are farther apart. This can be seen in the results of SA algorithm, where after the first descent to a low solution, there is a slow movement to a higher solution (at about the 200th iteration). The SA algorithm escapes the local minimum to search other areas, but it is a slow progression from one local minimum to another. A similar result can be seen for CSA1 at about the 250th iteration, but CSA2 and CSA3 seem to have more wide swings in the results until near the end of the algorithm as they settle into a solution.

Experiments #2 and #4 started with better initial solutions than that of Experiments #1 and #3. The lengths of the processing periods calculated at each iteration of each of the algorithms in Experiment #2 are shown in Fig. 5-13.

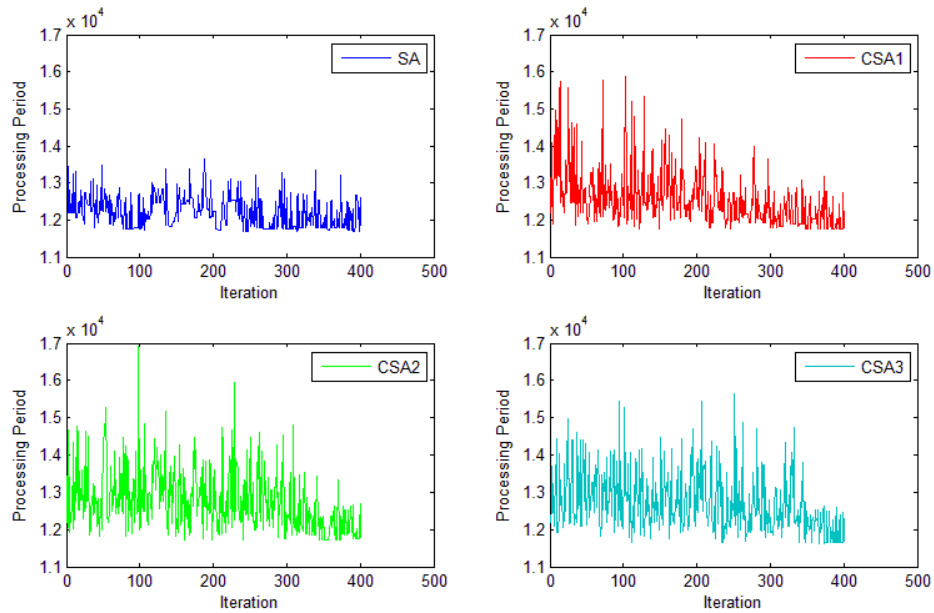


Fig. 5-13. Period calculated at each iteration for experiment #2.

These graphs show that the processing periods from the solutions in the traditional SA algorithm does not vary significantly compared to that of the CSA algorithms. Again this shows that the SA explores a relatively small area of the solutions space. In the case where the initial solution is close to a good solution, it may be beneficial only to explore a small area because the optimal solution is likely close to the initial solution. The solutions of the CSA algorithms vary quite a bit in the beginning iterations, and start to settle in to exploring a smaller area of the solution space near the end. In this case, there is no great benefit to searching a larger solution space, since the initial solution is already close to the optimal solution.

The lengths of the processing periods calculated at each iteration of each of the algorithms in Experiment #4 are shown in Fig. 5-14.

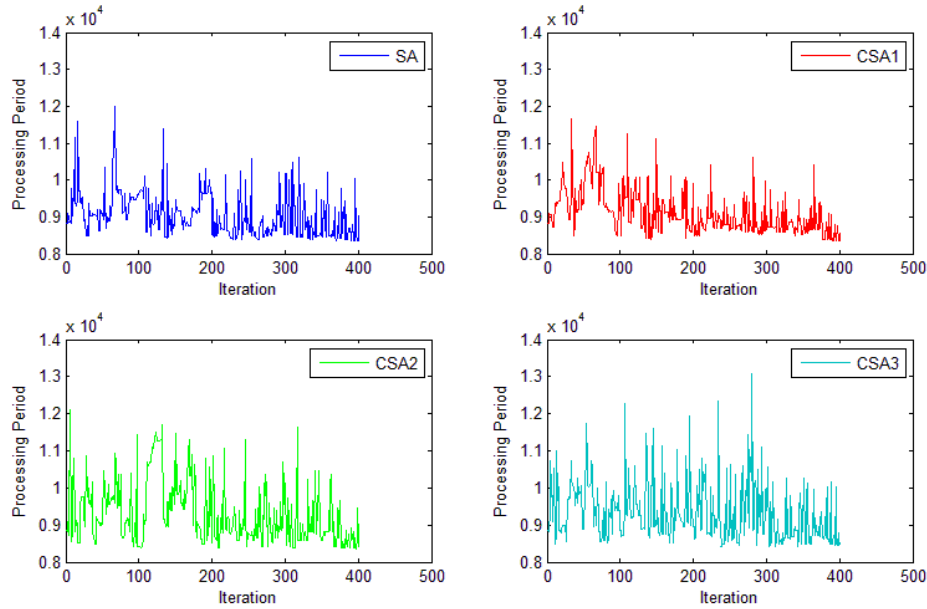


Fig. 5-14. Processing period calculated at each iteration for experiment #4.

Again the SA results do not vary as significantly as the CSA results, but in this case, there is not as large of a difference than in the case of Experiment #1. This is because, although the initial solution was relatively good, there are more poor solutions (because of the task characteristics).

The final solutions for each of the algorithms in each of the experiments are shown in Table 5-3. These results do not show any concrete conclusions about which method is likely to provide better results consistently. In Experiment #1 and #3, all of the CSA algorithms perform better than the SA algorithm. This is expected; since the initial solutions in these cases are far away from the global optimum, the SA algorithm takes longer to reach good solutions, and therefore spends more time searching poor solutions. In the cases where the initial solutions are relatively good (Experiments #2 and #4), there is no obvious trend in the results. In this case, the SA algorithm can produce good results because it searches in a small area close to the optimal solution, while the CSA algorithm

may wander away from the optimal solutions because the CSA algorithms can have wider swings in the search area of the solution space compared to the SA algorithm.

Table 5-3. Experimental final processing periods.

	SA Final Processing Period	CSA1 Final Processing Period	CSA2 Final Processing Period	CSA3 Final Processing Period
Exp. #1	11757	11677	11696	11663
Exp. #2	11678	11752	11704	11637
Exp. #3	8406	8362	8330	8298
Exp. #4	8353	8334	8367	8411

5.5.2 Power Density Spectrum

The power spectrum in the frequency domain can be determined by first performing a Fast Fourier Transform (FFT) on the processing period results that were calculated at each step of the algorithms, and then squaring the magnitude of each of the values in the Fourier series to obtain the power. The power density spectrum can then be analyzed by graphing the power vs. the frequency in a log-log plot. These graphs are shown in Fig. 5-15 for each of the SA, CSA1, CSA2, and CSA3 algorithms from the results of Experiment #3. These graphs show that the power spectrum is wide spread without peaks at specific frequencies, and that each of the power spectra have average slopes that are proportional to $\frac{1}{f^{\beta_s}}$, where $\beta_s \cong 1.8$. This means that the results are fractal in nature (similar characteristics at many scales), and have fractional Brownian characteristics. The fractal spectral dimension can be calculated by the following equation [Kins10], where $E = 1$ for a self-affine time series with a single independent variable:

$$D_\beta = E + \frac{3-\beta_s}{2} \quad (33)$$

In this case, since $\beta_s \cong 1.8$, the spectral dimension can be calculated to be

$$D_\beta = 1.6.$$

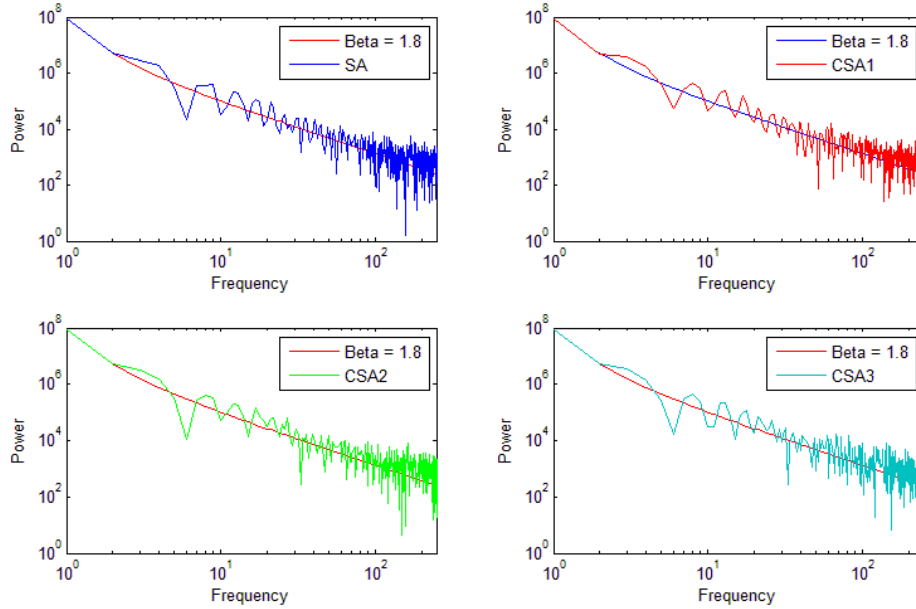


Fig. 5-15. Power density spectrum of experiment #3 results.

Power density spectrum analysis can be used to compare the rate of convergence of different combinatorial optimization techniques. This analysis shows the correlation between successive iterations of the combinatorial optimization algorithm. Highly correlated successive iterations will result in a fast rate of convergence to an optimal solution, whereas totally uncorrelated iterations will result in no convergence to an optimal solution. The more correlated the successive iterations, the greater the value of β_s . Totally uncorrelated iterations (white noise) result in a power density spectrum with a slope proportional to a β_s value in the range of $-1 < \beta_s < 1$. A fractional Brownian motion process results in a power density spectrum that has a slope proportional to a β_s value in the range of $1 < \beta_s < 3$, where $\beta_s=2$ is traditional Brownian process [Kins10]. Brownian motion occurs in geometrical space when an object moves in a random

direction, but limited to a distance from its last position. Therefore, the overall walk of the object is a random walk where each position is related to the last position. Similarly, a Brownian process can move randomly, but it is related to its last position. A Brownian type process is critical to SA, since a perturbation should move the solution to a solution nearby, to ensure that local minima can be properly explored. If the perturbation moves the solution to an area of the solution space that is totally unrelated to the previous solution, then there cannot be any progression of the search; rather, the search involves just randomly selecting different areas of the solution space. Since the power density spectrum of each of the results obtained from the experiments show that all of the algorithms are fractional Brownian processes, this indicates that the search to a global solution will progress with subsequent steps in the algorithm, and is not just randomly searching different areas of the solution space with no signs of progression towards a global minimum. A combinatorial optimization algorithm that resulted in a Black process (defined as process where $\beta_s > 3$) would be more desirable than a Brown process; however, this would be very difficult to achieve without getting stuck in local minima within the solution space.

The power density spectrum analysis in this case does not show any advantage of one algorithm over the others, since the correlation of successive iterations produce the same spectral dimension for each algorithm. However, the analysis does validate the perturbation chosen for generating new solution. If the perturbation changed the solution so much, at each iteration, that the new solution was not related to the previous solution in any way, it would be expected that the results would have the characteristics of white noise, because there is no correlation between one iteration and the next. The fact that

there are no significant differences in the power density spectrum for the CSA algorithms compared to the SA algorithm indicates that the action of searching a wide area of the solution space initially and then narrowing down to a smaller area of solution space as the algorithm proceeds (as is the case for the CSA algorithms), still maintains the desired fractional Brownian process characteristics evident in the SA algorithm.

5.6 Task Allocation Optimization Conclusions

The three CSA methods developed are common in that they all tend to search a relatively large solution space in the early iterations of the algorithm, and then settle into searching smaller areas of the solution space as the algorithm progresses. This is beneficial when the initial solution is far away from the global optimum, because convergence to the global optimum can be faster. However, in cases where the initial solution is close to the global optimum, searching a large solution space may result in moving away from the global optimum, and therefore, searching a smaller area more thoroughly can be beneficial (as is the case for the traditional SA algorithm). The power density spectrum analysis can be used to compare the expected rate of convergence of combinatorial optimization methods by determining how correlated the results are between iterations of the algorithm. The power density spectrum analysis in this case showed that the choice of perturbations result in a desired Brownian fractional process for the SA algorithm. This analysis also showed that the CSA algorithms have the desired fractional Brownian process characteristics evident in the SA algorithm. This means that the searching of a wide solution space performed by the CSA algorithms does not prevent

the solutions from progressing towards a global minimum, as might be the case for a totally random search of the solution space.

Each of the CSA algorithms proposed have slightly different characteristics from each other, due to the differences in the probability of distribution of the number of perturbations that can be made throughout the algorithms. The CSA2 and CSA3 algorithms are an attempt to find a universal tool for solving combinatorial optimization problems in that they walk through several probability distributions for determining the number of perturbations that should be made at each iteration of the algorithms, and are not necessarily only suited for problems that have particular characteristics of where the local minima in the solution space are located. Ultimately, no conclusion can be made that the CSA algorithms proposed are better than other SA algorithms for all types of combinatorial optimization problems. Rather, the “best” algorithm for any particular combinatorial optimization problem is specific to the unique characteristics of the problem that is to be solved. That is, there is no algorithm to solve NP-complete combinatorial optimization problems in the most efficient way that can be applied blindly to any problem without good understanding of the characteristics of the problem.

Chapter 6

EXPERIMENTAL RESULTS

This section describes the experimental implementation created for the MPSoC framework. The experimental results obtained from the MPSoC implementation, along with the experimental results from the analytical model experiments (Section 4.7) and the experimental results from the SA and CSA algorithms (Section 5.5) are used to validate the MPSoC framework, the analytical model, and the optimization method proposed in this thesis.

6.1 Experimental Application

In order to demonstrate the strengths and weaknesses of the MPSoC framework and to validate the analytical method developed in this thesis, an example application was developed for implementation using the MPSoC framework. The example application performs the function of a green screen video system, which is often used to superimpose one video feed onto the background of another video feed. There are typically two video sources; the primary video feed has a green background and a non-green foreground. Every green pixel in the primary video feed is replaced with a pixel in the same location from the second video feed. This is a common technique used for television weather forecasts to show the weather map behind the meteorologist. The example application takes two images that are in YUV colour format, converts them both to RGB colour format, and then replaces all of the green pixels in the primary image with a corresponding pixel from the secondary image. The combined image is then converted back to the YUV colour format. For simplicity in the example application, each pixel in

the original two images was generated by a random number generator, rather than actually having been input from a video source. This was done because the actual source or content of the image is irrelevant for the purposes of evaluating the MPSoC framework. The example application was divided up into 16 tasks, which have a DFG as shown in the following figure:

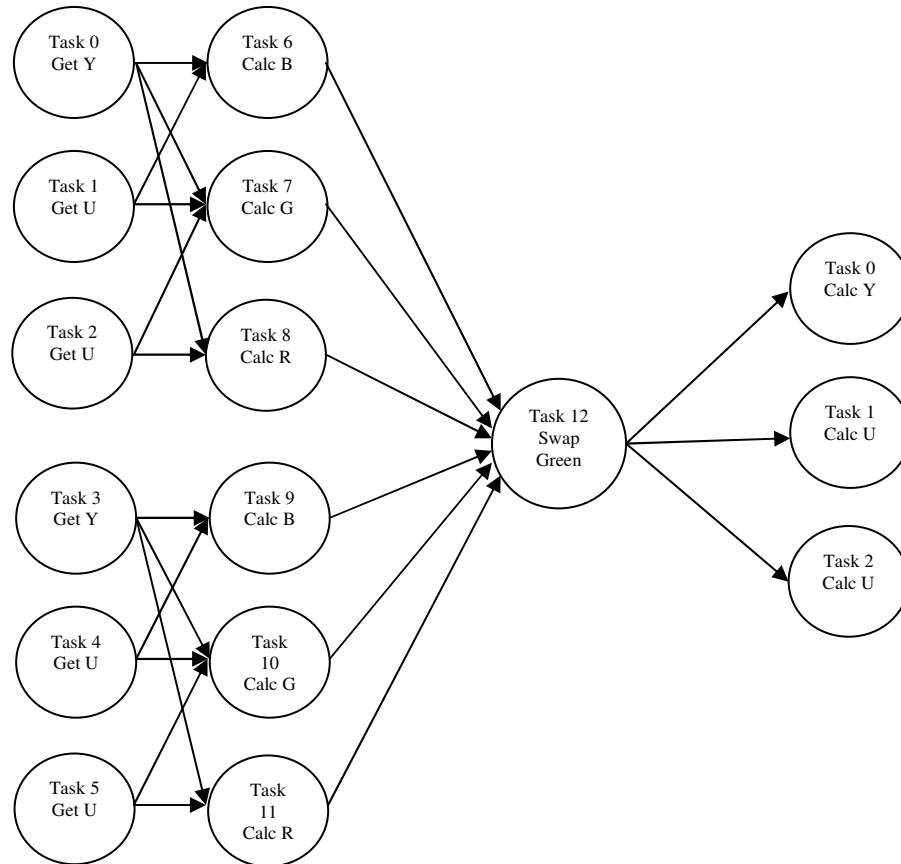


Fig. 6-1. A dataflow graph representing the green screen application.

Tasks 0, 1, and 2 generate the Y, U, and V portions of the primary image. Tasks 3, 4, and 5 generate the Y, U, and V portions of the secondary image. Tasks 6, 7, and 8 calculate the R, G, and B portions of the primary image. Tasks 9, 10, and 11 calculate the R, G, and B portions of the secondary image. Task 12 replaces the green pixels from the

primary image with the corresponding pixels in the secondary image. Tasks 13, 14, and 15 calculate the Y, U, and V components of the combined image.

The input and output for each task was read from, and written to, global memory, with the exception of the original YUV images, which were generated by a random number generator. Each image was 500 pixels (each pixel being 3 bytes). This is a very small size for a real-world image, but the system was limited by the amount of global memory available. However, the size of the application was sufficient for the purpose of evaluation of the MPSoC framework. Two versions of the application were developed, one in which minimal processing is done within each task in the DFG compared to the amount of time spent accessing global memory, and the other in which much more time is spent performing computations compared to the time accessing global memory. These two applications demonstrate the extreme spectrum of real-world applications, so the strengths and weaknesses of the MPSoC and analytical model can be demonstrated for each case. The two applications will be referred to, respectively, as the memory intensive application, and computationally intensive application. To simulate the computationally intensive application, the memory intensive application was modified by inserting a busy-wait loop that counted from 0 to 1000 for each piece of data that was operated on. This loop served no purpose in the application, but simulated the situation where a significant amount of calculations would need to be done for each task. Both the memory intensive application and the computational intensive application were run through a number of experiments in order to compare the results.

6.2 Experimental Implementation and Results

The MPSoC demonstrating the framework was implemented in a Xilinx Virtex-II Pro FPGA, on the Xilinx XUPV2P development platform [Xili08a]. Xilinx MicroBlaze V6.0 soft processors were used for the MPSoC implementation, each processor running with a clock speed of 100 MHz. Each processor had 16 kB of local memory to be used for instruction and data memory. The head processor used a serial port peripheral to output the performance metrics read from the snoopy block to a PC, in order to evaluate the system. The global memory size used for the example application was 16 kB of RAM internal to the FPGA. Details of the test setup are provided in Appendix A. Experiments were conducted using 1, 2, 3, and 4 processors with the example applications.

The first step in implementing the system was to run the memory intensive application on a system with a single processor. This is required to determine the serial execution time of each of the tasks in the system and the probability of a global memory access request. This information is required as input into the analytical tool. The snoopy block (as described in section 3.2.7) provided the method by which the parameters of the tasks could be measured. The total task time was determined to be the average time over all 500 task executions. Determining the probability of global memory access time is not necessarily a trivial exercise. Initially, it was determined by calculating the total number of accesses to global memory divided by the number of instructions executed during a single task execution. However, due to complexities in the operation of a processor (such as instruction pipelining, variable length instructions, etc.) this determined value may not be accurate. The determined probability must then be used within the analytical model, and compared against measured results of the two processor case to validate the

determined probability. The experimentally determined serial task execution times and probability of global memory access requests are shown in Table 6-1.

Table 6-1. Experimentally determined task parameters of the memory intensive application.

Task	Initial Task Time (ns x 10)	Probability of Memory Access Request (α_i)
0	1484	0.054
1	1484	0.054
2	1484	0.054
3	1484	0.054
4	1484	0.054
5	1484	0.054
6	968	0.525
7	1268	0.549
8	968	0.525
9	968	0.525
10	1268	0.549
11	968	0.525
12	2630	0.681
13	1268	0.549
14	1368	0.459
15	1368	0.459

In order to validate the analytical model and to demonstrate its benefits, the tasks were allocated according to a greedy heuristic algorithm, proposed in [BaGa99]. This resulted in task allocations as shown in Table 6-2.

Table 6-2. Task allocations for 2, 3, and 4 processors.

Number of Processors	Processor	1 st Task	2 nd Task	3 rd Task	4 th Task	5 th Task	6 th Task	7 th Task	8 th Task
2	P0	0	4	5	10	8	11	14	15
	P1	2	1	3	7	6	9	12	13
3	P0	0	1	10	11	15			
	P1	2	3	6	8	12			
	P2	4	5	7	9	14	13		
4	P0	0	5	11	13				
	P1	2	7	6	12				
	P2	4	10	8	14				
	P3	1	3	9	15				

The analytical method, as described in Chapter 4, was applied for a system with two, three, and four processors and compared to the experimentally measured processing period times. Also, the simpler analysis method proposed in [SLOW07] was applied; this

calculates the processing period by assuming that the maximum bandwidth of the global memory has been reached. Therefore, the processing period time is calculated by subtracting the portion of each task that is due to memory accesses, then calculating the processing period of the parallel tasks, then adding the sum of the time for each memory access (which was previously subtracted). This method essentially assumes that, every time a memory access was requested, the global memory was already being serviced by another processor, so the processing time can be parallelized, but all of the memory accesses are executed serially. The graph in Fig. 6-2 shows the comparison of the ideal processing period, (without considering the effects of memory contention), the actual experimental measurement, the value calculated by the analytical method proposed in this thesis, and the value calculated by a simple calculation, which assumes saturated bandwidth.

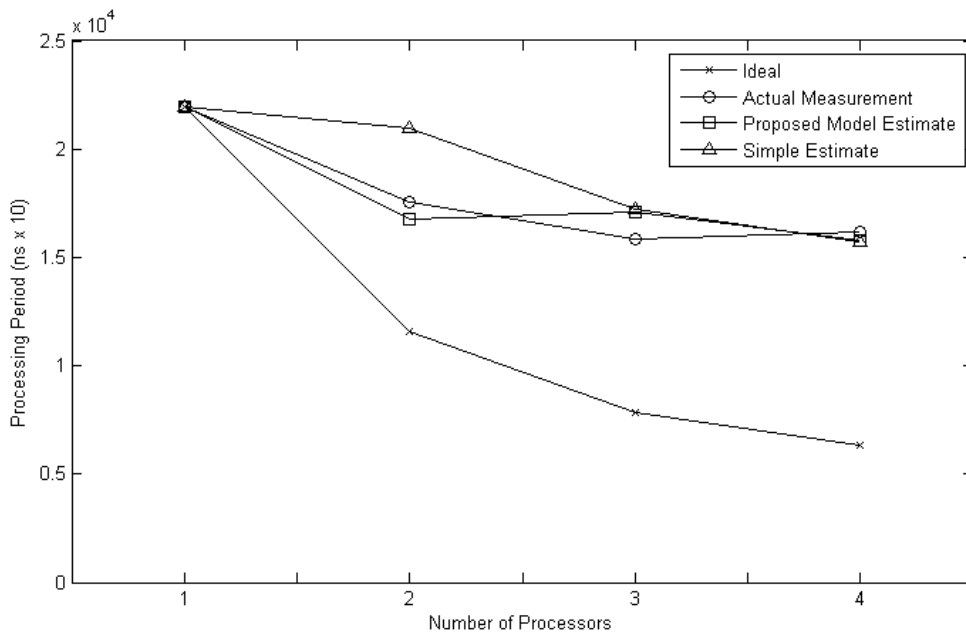


Fig. 6-2. Processing period results for the ideal value, the experimental measurement, the calculation by the proposed analytical method, and the calculation by the simple method (saturated bandwidth from [SLOW07]).

These experimental results show that the proposed analysis method results in estimations that are quite close to the actual measured task periods. The simple analysis method works well for the situation where three or four processors are used, but is not accurate in the case of two processors. This is because the global memory bandwidth becomes saturated in the case of three and four processors, but it is not saturated in the case of two processors. The simple analysis method really gives an upper limit to the expected processing period time, but only a relatively accurate estimation when the memory bandwidth is saturated. The analysis method proposed in this thesis is superior to the simple analysis method because it does not depend on memory bandwidth saturation. While Fig. 6-2 shows the benefit of the analytical method, it also shows that the MPSoC framework is not ideal for applications that required a large number of accesses to global memory. In this case, there is not much benefit in implementing more than two processors in the MPSoC, because the global memory contention limits the performance that can be achieved within the system.

Speedup is a measure that is used often in parallel processing research as a measure of improvement for an algorithm as the number of processors increase in a parallel processing system. It is defined as the time for the serial execution of the algorithm divided by the execution time with more than one processor. The ideal speedup for any parallel processing system is equivalent to the number of processors in the system. Fig. 6-3 shows the speedup for the memory intensive application. It can be seen from this graph that the speedup for the two processor case is not particularly impressive, and there is a slight improvement for three processors, followed by a reduced performance for four processors. Since only one processor can access global memory at a

time, adding more processors essentially reduces much of the program execution to serial execution, regardless of how many processors are added to the system.

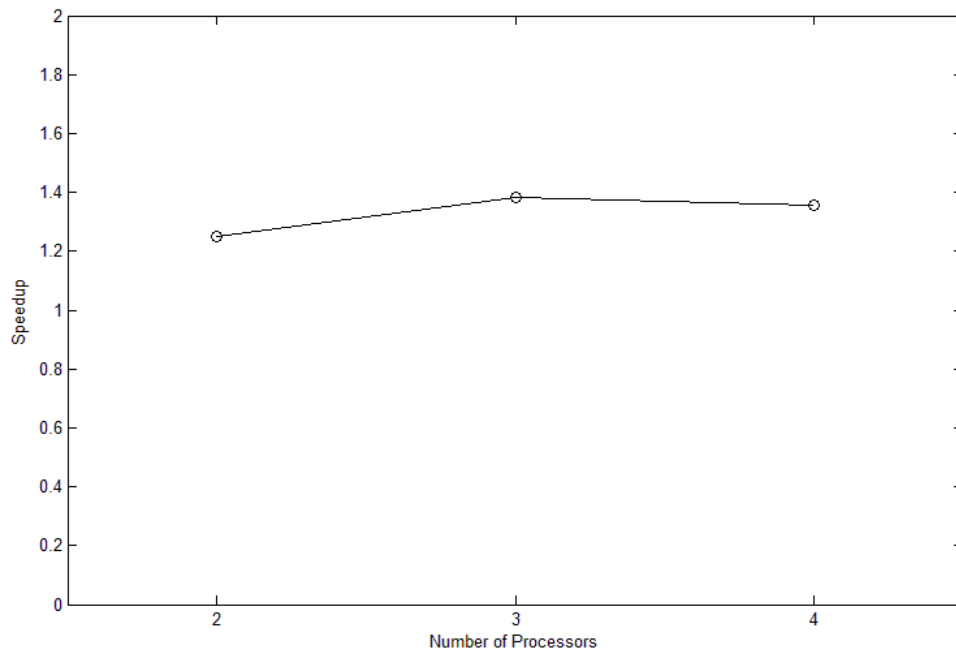


Fig. 6-3. Speedup of the memory intensive application for two, three, and four processors.

The following graph shows the execution times of each of the tasks with a different number of processors in the system.

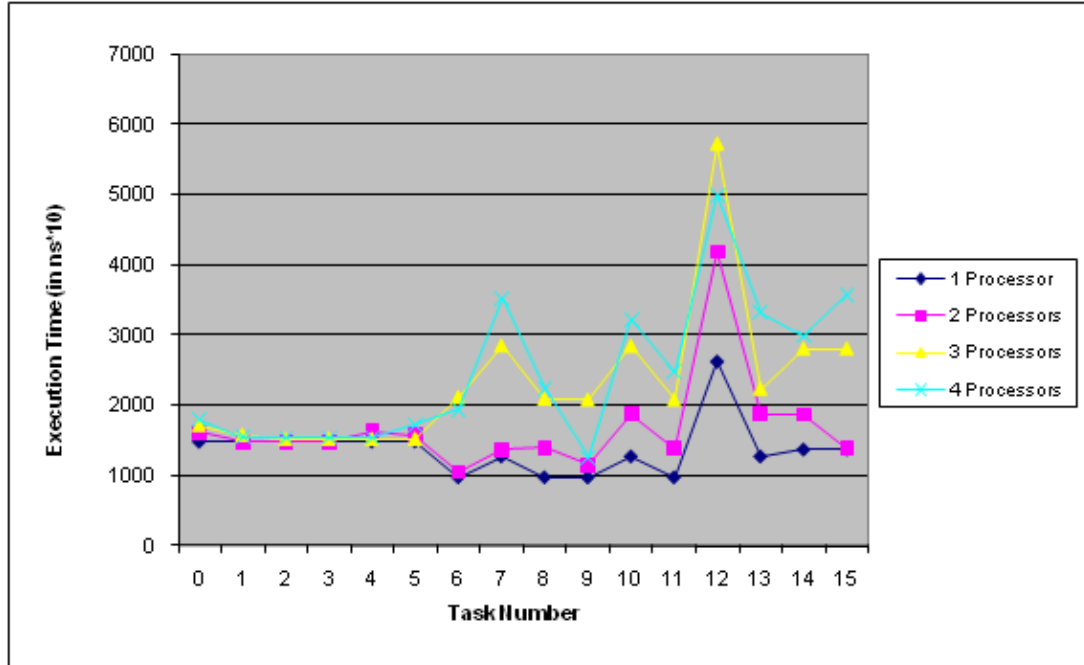


Fig. 6-4. Individual measured task execution times for the memory intensive application.

In general, Fig. 6-4 shows that, as the number of processors increases, the amount of time it takes to execute a task also increases; however, this is not always the case. The only reason that the execution time for a task would significantly increase would be if more time was spent waiting to access global memory. In the case of the single processor system, the task can always access global memory when needed because the processor does not have to compete with other processors for the memory. As the number of the processors increase, there is a greater chance that the tasks will have to wait for memory access. An interesting observation that can be made from Fig. 6-4 is that task 12 takes more time to execute with three processors than it does with four processors. This is counterintuitive because more processors should mean that it is more likely that there is contention for global memory resulting in a longer task execution time. This data shows that it is not always that straightforward because it depends on what other tasks are being executed at the same time, and the probability that those tasks are accessing memory.

Another interesting observation that can be made from Fig. 6-4 is that tasks 0 to 5 do not seem to have a significant increase in execution time as the number of processors increase. This is because tasks 0 to 5 are the tasks that generate the YUV values with the use of a random number generator. As a result, these are the most computationally intensive tasks, and have the least amount of memory accesses. Therefore, as the number of processors increase, these tasks are not affected significantly by increased probability of global memory contention.

Since the bottleneck in the system is the contention for global memory access, it seems that this system should perform much better with tasks that are computationally intensive, compared to tasks that access memory often. In order to demonstrate the global memory bottleneck, the computationally intensive application was created. This version, as described earlier, is the same application as the original, except a large number of computations that do not access global memory, were added to simulate computationally intensive tasks. The graph in Fig. 6-5 shows the execution time vs. the number of processors in the system for the computationally intensive application.

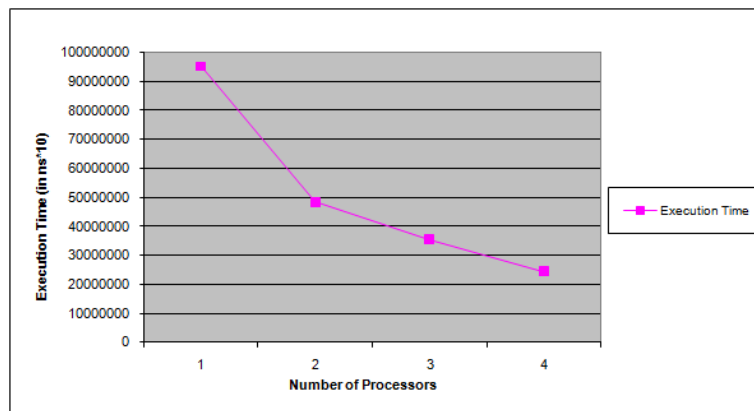


Fig. 6-5. Execution time vs. the number of processors for the computationally intensive application.

This next graph shows the speedup achieved for the multiprocessor systems over the single processor system with the computationally intensive application.

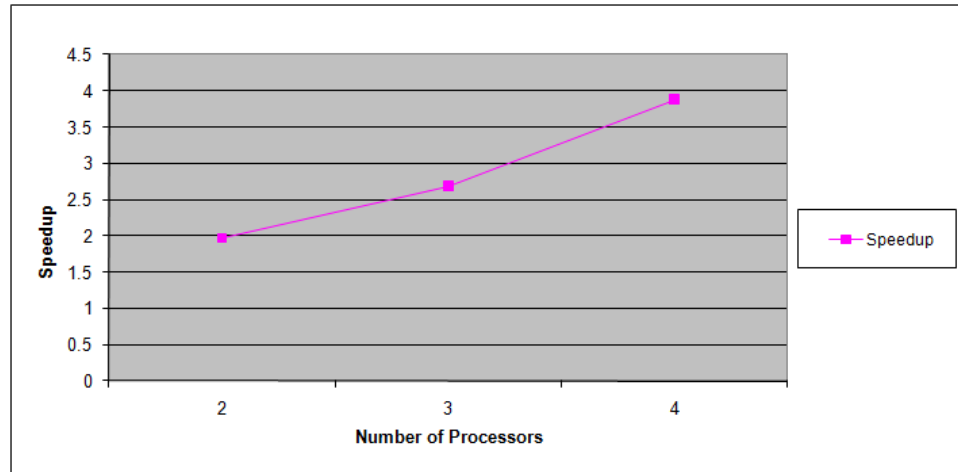


Fig. 6-6. Speedup vs. the number of processors for the computationally intensive application.

It is evident from Fig. 6-5 and Fig. 6-6 that the overall processing time is significantly reduced as more processors are added to the system. The speedup is very close to the ideal speedup for two and four processors. The efficiency (defined as the speedup divided by the number of processors [Quin04]) is almost ideal (ideal efficiency is 1) in the case of two and four processors (0.96 and 0.93, respectively), and is very good for three processors (0.9). The reason that the efficiency is slightly smaller for three processors is because the processor loading is not quite as well balanced as it is in the case of two and four processors. The processor loading imbalance can be seen by the following graph which shows the processor idle time for each of the processors in the system for the one, two, three, and four processor systems.

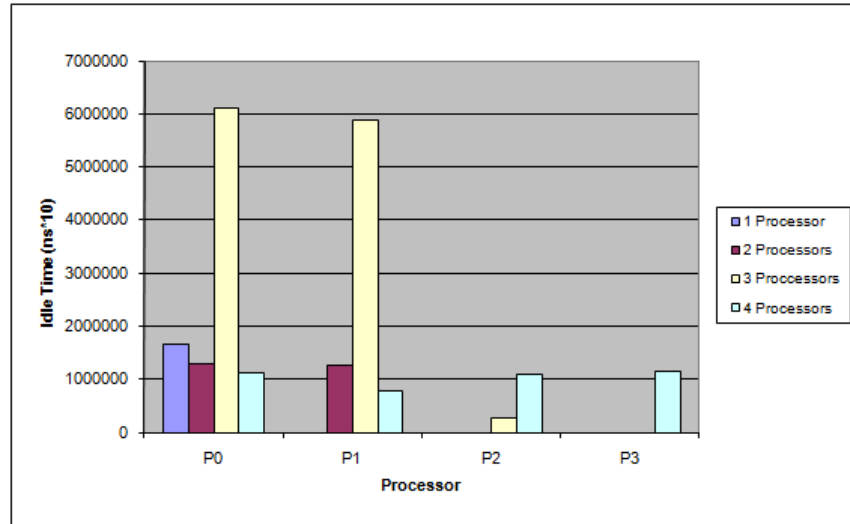


Fig. 6-7. Processor idle time of each processor in the one, two, three, and four processor cases.

There is some idle time for the single processor case because this is the task switching overhead. The idle time for the two processor system and the four processor system is fairly well balanced, showing that all the processors must have finished all of their tasks in close succession. The three processor system shows that P2 had much less idle time than P0 and P1. This must mean that P0 and P1 finished all of their tasks much earlier than P2 had, so the time that P0 and P1 were doing nothing while waiting for P2 to finish is considered idle time. The reason for this is demonstrated in the task allocation table (Table 6-2), where it is shown that P2 has six tasks to execute and P0 and P1 only have five tasks to execute. Because of the added busy-wait loops in the busy-wait application, each of the tasks has almost the same serial execution time. This means that executing six tasks will take approximately 20% longer than executing five tasks.

Chapter 7

CONCLUSIONS AND FUTURE WORK

7.1 Thesis Conclusions

This thesis has presented an MPSoC framework that is specific to a synchronous dataflow computing model intended for stream-oriented applications. Chapter Three specifies the architectural features of the MPSoC framework that are fixed, and the features of the MPSoC that are implementation specific. In Chapter Four, an analytical model was developed to model the MPSoC framework with the purpose of determining the processing pipeline period considering the effects of global memory contention for a particular allocation of tasks to a particular number of processors. The analytical model was then used as part of a cost function that can be used for optimization algorithms developed in Chapter Five, to determine a good solution to the MPSoC framework for a particular application. Three variations of Chaotic Simulated Annealing algorithms were proposed as the combinatorial optimization technique to be used for finding a good solution, and the characteristics of these algorithms were compared. Finally, Chapter Six describes the experimental implementation of the MPSoC framework that was developed for the purposes of verifying the analytical method, and to examine the characteristics of the MPSoC framework.

The experiments of Chapter Six demonstrate characteristics of the MPSoC framework architecture. The experimental results demonstrate that the computationally intensive program performs better than the memory intensive program due to reduced global memory contention. Amdahl's law specifies that, as the number of processors

increases in a parallel processing system, the total execution time asymptotically approaches the time it takes to execute the portion of the program that is not parallelizable [Amda67]. The experimental results show that using the synchronous dataflow model of computing results in the global memory access time being portion of the program that is not parallelizable, and therefore, as the number of processors increase, the execution time will asymptotically approach the cumulative global memory access time of the processors. However, even in the case of the memory intensive program, the experimental results show that the task parallelization achieved through pipelining results in performance improvements for multiple processors. In general, most real-world applications will have a mixture of computationally intensive tasks and memory intensive tasks. The pipelining of tasks achieved with the MPSoC for stream-oriented applications structured in a synchronous dataflow graph will result in parallelization of computations in both types of tasks.

The experimental results in Chapter Six also demonstrate the benefits of the analytical method proposed in Chapter Four over a model that does not consider the memory contention at all (the ideal case), and a model that considers only saturation of the communication channel by comparing the predicted results with the measured results taken from the MPSoC experimental implementation. This can be seen most clearly in Fig. 6-2. Since the analytical model is validated by comparison to measured results, there can be confidence in using the analytical model as a cost function to explore many different possible MPSoC implementations through combinatorial optimization techniques. Simulated annealing and variations of chaotic simulated annealing combinatorial optimization techniques are applied to the task allocation problem and

compared to each other in Chapter Five. The experiments in this chapter demonstrate the differences in the combinatorial optimization techniques. Analysis in the time domain (section 5.5.1) demonstrates that the chaotic simulated annealing algorithms jump to initial good solutions faster than the simulated annealing algorithm, but then have a larger variance in the solutions compared to the simulated annealing algorithm due to the broader search space. Analysis in the frequency domain (section 5.5.2) demonstrates that both the simulated annealing and chaotic simulated annealing algorithms are fractional Brownian processes, meaning that the solutions move towards improving solutions as more iterations are applied.

The main benefit of the proposed MPSoC framework is that the design space for multiprocessing systems targeting stream-oriented applications is significantly reduced. This means that the complexity of developing MPSoC can be reduced, resulting in faster development times, and ultimately faster time-to-market for MPSoCs developed using the proposed framework. This is achieved through the defined hardware and software architectural features that are beneficial to all applications within the stream-oriented application class, so that the design time can be spent on features that are application specific. The proposed chaotic simulated annealing optimization techniques, using the MPSoC analytical model as a cost function, add to the benefits of the fixed architectural features to automate task mapping and scheduling, which further reduces the number of design decisions that need to be made to develop an application specific MPSoC for stream oriented applications.

7.2 Recommendations for Future Work

There are many potential future research topics that could expand on the work in this thesis. A few of the areas for potential future work are identified in this section with the intention of highlighting the areas that would benefit from continued research.

An area that would benefit from future research is the memory arbitration method of the memory controller in the MPSoC framework. The memory controller developed in this thesis uses a polling method to cycle between each of the processors as described in section 3.1.1.4. The analytical model also assumes the polling method for memory arbitration. A possible improvement to the polling memory arbitration scheme is to use an interrupt based memory arbitration. This would require changes to the framework and the analytical model, but could result in reduced idle time for processors in the system. Additionally, differing priorities could also be associated with tasks accessing global memory, with access given to higher priority tasks over lower priority tasks. Prioritizing tasks would require memory controller and task controller coordination, which would add additional complexity to the MPSoC framework and analytical model, but could result in better overall performance.

Another area of research expanding on this thesis is to consider dynamic scheduling of tasks, rather than the hybrid scheduling discussed in section 3.1.3. A dynamic scheduler would require more processing power to determine the tasks to be executed on each processor during runtime; however, this would have the benefit of being able to better adapt to varying task execution times. In an ideal case, a dynamic scheduler could make an assessment of the system at run time through optimization

similar to the combinatorial optimization proposed in this thesis. However, the difficulties of running this optimization in real-time is that the analytical model and optimization techniques would have to be simplified such that a result could be obtained in a very short period of time. The optimization technique and the analytical model proposed in this thesis are complex and as a result require a significant amount of processing. The complexity has the benefit of accuracy in the model, but at the cost of greater computation times, such that it is beneficial to run at design time, but not practical for run-time optimization. Therefore, a dynamic task scheduler would have to use a simplified model of the system and optimization method in order to properly assess the system in real-time.

Areas of research related to the analytical model described in Chapter Four could be to consider variable task execution times for each of the tasks, rather than assuming the task variability is minimal. Task variability could occur due to executions of different branches within each of the tasks, or due to the use of local cache memories. The task variability would add some complexity to the analytical model, but would result in more accurate modeling of tasks that have significant execution time variability. Additionally, research into how the analytical model could consider non-uniform distribution of global memory accesses within each task, could be useful in modeling tasks where the memory accesses are distributed unevenly.

Research into alternative combinatorial optimization algorithms applied to the task allocation problem could be useful to compare the chaotic simulated annealing algorithms proposed in Chapter Five to alternate techniques, such as particle swarm

optimization, genetic optimization, as well as other combinatorial optimization techniques. These could potentially result in combinatorial optimization techniques that can find a good solution to the task allocation problem in a shorter period of time.

The areas of future work mentioned above are just a few of the many possible research topics that could spin off from the work in this thesis to further expand on the research into MPSoC frameworks. In general, this is an important area of research at the current time because computing technology has become more easily adaptable in both hardware and software, and the number of embedded applications has expanded at an exponential rate over recent years. This means that the potential MPSoC applications, and the technology to implement these applications, are now at a point where general purpose microprocessing systems are no longer competitive for many embedded systems, and application specific MPSoCs will likely become more and more prevalent. Frameworks for these MPSoCs then become important to reduce the development time and cost associated with the customization of hardware and software specific to applications.

7.3 Thesis Contributions

This thesis makes several contributions to new knowledge in the area of multiprocessing systems-on-chip. Below is a list of these contributions.

1. An MPSoC framework was proposed that was defined by four major parts: (1) the targeted class of applications, (2) the hardware and software architectural features, (3) a detailed model for analysis of the MPSoC, and (4) a method for design of the MPSoC (in this case automated optimization of the system using CSA). The definition of what constitutes an MPSoC framework is in itself a

contribution to the MPSoC field of study. Other proposed MPSoC, or MPSoC frameworks (as described in Section 2.4) only address one or two of these four parts, but they are never addressed as a whole, when in reality they are all very closely related to each other. The first two parts are related to the performance of the MPSoC system, and the last two parts are related to the design method of the framework. A good MPSoC framework should address both the performance of the proposed system, and the feasibility of the design. If a proposed MPSoC framework has good performance, but the development effort for an individual application is too large, then it will not likely to be practical for use. Similarly, if an MPSoC framework can be implemented quickly for a particular application, but it does not have good performance, then it is also not likely to be useful in practice. By demonstrating how all of these four parts fit together, and the demonstrating the importance of considering all of these parts as a whole, a template for comparison of MPSoC frameworks has been developed. That is, other MPSoC frameworks developed in the future could be defined and compared by the class of applications targeted, the architectural features of the MPSoC framework, analytical models used to estimate system performance, and a proposed design methodology.

2. Architectural features of MPSoC framework were developed to specifically target stream-oriented applications. These features were implemented in VHDL modules that could be used as the starting point for all implementations of the MPSoC framework. Many of these modules could also

be used as components for other MPSoC architectures that do not strictly fit into the MPSoC defined in this thesis.

3. An analytical method based on discrete time Markov chains was developed to analyze potential implementations of the MPSoC framework. This allows the system designer to compare different configurations of the MPSoC to determine the optimal configuration of the MPSoC before implementation. The analytical model could also be used as the starting point for models for other MPSoC architectures, by modifying features of the model to match the features of the processing architecture. The description of the analytical model in Chapter 4 provides the explanation of the features of the model, which allows for the model to be adapted for either slight changes in the MPSoC architecture, or major differences, because the features of the model are linked to features of the MPSoC architecture. The analytical model was implemented in Matlab scripts and in the C programming language. These implementations of the analytical method can easily be modified for other implementations of the MPSoC framework, by changing only the input parameters, or can be used as the base platform for implementations of analytical models for other MPSoC architectures.
4. A novel hybrid pipeline scheduling technique was developed and described in Section 3.1.3. This pipeline scheduling depends on design-time (static) scheduling of tasks, but then allows the number of pipeline stages to vary according to the tasks that are available for execution, to minimize the idle time of each individual processor. This hybrid pipelining method is specific to

the synchronous data flow model of computing and the global shared data memory architecture of the proposed MPSoC framework.

5. Application specific features of the MPSoC were developed for the experimental application. This includes application specific VHDL modules, implementation of the MicroBlaze processors, software written in C code for the processors within the MPSoC. While this work is specific to the experimental implementation of the MPSoC framework, it can be used as a model on how to implement the MPSoC framework. Some of the features of the MPSoC implementation that are not defined in the MPSoC, but could be useful for reuse are the peripheral interfaces between the MicroBlaze processors and the VHDL modules (task controller and memory controller). The implementation of the peripheral interfaces includes hardware modules and software drivers that could easily be modified to be interfaces from the MicroBlaze processors to other types of hardware modules. Another module that could be useful in other MPSoC systems is the Snoopy module (described in 3.2.7). This module was used to measure system performance of the experimental MPSoC implementation, and would be useful for other MPSoC implementations to obtain similar system performance metrics.
6. Algorithms were developed in Chapter 5 for the purpose of automated optimization of task allocation. A SA and three CSA variations were developed and implemented in the C programming language. These implementations of the algorithms can be modified with little effort for task allocation problems for different implementations of the MPSoC framework

by changing only input parameters. Alternatively, these programs can contribute to research using SA or CSA for entirely different applications by serving as the base code, requiring only changes to the functions that are specific to the perturbations for the task allocation problem.

7. This thesis demonstrated the benefits of using power density spectrum analysis to evaluate the perturbation used in SA and CSA algorithms in section 5.5.2. SA and CSA algorithms depend on the use of a perturbation that makes only a small change between the current solution and the previous solution. The chosen perturbation for a problem is specific to the nature of the problem itself. This is especially significant for combinatorial optimization problems, where it may not be obvious on how significant a change in the solution is made by a particular perturbation. Analysis of the results of a SA or CSA combinatorial optimization algorithms in the frequency domain can measure the degree of correlation between successive solutions in the algorithms by measuring the slope of the power density spectrum, which can be used as a measure to determine the effectiveness of a chosen perturbation for a problem. This can be used for many different types of SA and CSA applications to evaluate and compare different types of perturbations for a given problem.

REFERENCES

- [Amda67] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities" *AFIPS Conf. Proc.*, vol. 30, pp. 483-485, Apr 1967.
- [BaGa99] S. Bakshi and D. Gajski, "Partitioning and Pipelining for Performance-Constrained Hardware/Software Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 4, pp. 419-432, Dec 1999.
- [BaGo05] J. Barros and L. Gomes, *Models of Computation for Embedded Systems*. The Industrial Information Technology Handbook, Chapter 83, 2005.
- [BeBB08] S. Ben Othman, A.K. Ben Salem, and S. Ben Saoud, "MPSoC design of RT Control Applications based on FPGA SoftCore Processors", *15th IEEE International Conference on Electronics, Circuits, and Systems*, pp.404-409, Sept. 2008.
- [BeBe91] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, pp. 1270–1282, 1991.
- [BeRo08] A. Benoit and Y. Robert, "Mapping pipeline skeletons onto heterogeneous platforms," *J. Parallel Distrib. Comput.*, 68(6), pp.790–808, 2008.
- [BeTs93] D. Bertsimas and J. Tsitsiklis, "Simulated Annealing," *Statistical Science*, vol. 8, no. 1, pp. 10-15, 1993.
- [Bhan75] D. P. Bhandarkar, "Analysis of memory interference in multiprocessors," *IEEE Trans. Comp*, vol. C-24, no. 9, Sept 1975.
- [ChAi95] L. Chen and K. Aihara, "Chaotic Simulated Annealing by a Neural Network Model with Transient Chaos," *Neural Networks*, vol. 8, no. 6, pp. 915-930, 1995.
- [ChAi97] L. Chen and K. Aihara, "Combinatorial Optimization by Chaotic Dynamics," *IEEE Conference on Computational Cybernetics and Simulations*, vol. 3, pp. 2921-2926, 1997.
- [ChDo08] G. Chen and Z.Y. Dong, "On the Ergodicity of the Markov Chain Associated with a Chaotic Simulated Annealing Algorithm," *IEEE Congress on Evolutionary Computation*, pp. 1124-1127, 2008.
- [ChVe02] K.S. Chatha and R. Vemuri, "Hardware-software partitioning and pipelined scheduling of transformative applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. vol. 10, no. 3, pp. 193-208, Jun. 2002.

-
- [CoFA10] D. Cook, K. Ferens, and A.S. Alfa, "A Model for Analysis of Memory Contention in a Stream-Oriented Shared Memory Multiprocessor System," Unpublished.
- [CoFW10] D. Cook, K. Ferens, and W. Kinsner, "Application of Chaotic Simulated Annealing in the Optimization of Task Allocation in a Multiprocessing System," Unpublished.
- [CoHJ07] J. Cong, G. Han, and W. Jiang, "Synthesis of an Application-Specific Soft Multiprocessor System," *Proc. 2007 ACM/SIGDA 15th International Symposium of FPGA*, pp.99-107, 2007.
- [DaSe96] S.K. Das and S.K. Sen, "Analysis of memory Interference in Buffered Multiprocessor Systems in Presence of Hot Spots and Favorite Memories," *Proceedings of the 10th International Parallel Processing Symposium*, pp.281-285, Apr 1996.
- [DAMF06] P.G. Del Valle, D. Atienza, I. Magan, J.G. Flores, E.A. Perez, J.M. Mendias, L. Benini, G. De Micheli, "A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework," *2006 IFIP International Conference on Very Large Scale Integration*, pp.140-145, 2006.
- [EPKD97] P. Eles, Z. Peng, K. Kuchcinski, and A. Dobboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Design Automat. Embedded Syst.*, vol. 2, no. 1, pp. 5-32, Jan. 1997.
- [FLPS10] F. Ferrandi, P.L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 6, pp. 911-924, Jun. 2010.
- [Fly72] M. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.*, vol C-21, no.9, pp. 948-960, Sept 1972.
- [Fost95] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Reading, MA: Addison-Wesley, 1995.
- [GeSz92] L. Georgiadis and W. Szpankowski, "Stability of token passing rings," *Queueing Syst. Theory Appl*, vol. 11, no. 1-2, pp. 7-33, Jul 1992.
- [GuAK10] S. Gupta, G. Agarwal, and V. Kumar, "Task Scheduling in Multiprocessor System using Genetic Algorithm," *International Conference on Machine Learning and Computing*, pp.267-271, 2010.
- [Haja88] B. Hajak, "Cooling Schedules for Optimal Annealing," *Math. Oper. Res.*, vol. 13, pp. 311-329, 1988.
-

-
- [HaLe97] S. Ha and E.A. Lee, "Compile-time Scheduling of Dynamic Constructs in Dataflow Program Graphs," *IEEE Transactions on Computers*. Vol. 46, Issue 7, pp. 768-778, July 1997.
- [Hare87] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231-274, 1987.
- [Hopf82] J.J. Hopfield, "Neurons with Graded Response have Collected Computational Properties like those of Two-State Neurons," *Proc. Nat. Academy Sci. USA*, vol. 79, pp. 2554-2558, Apr. 1982.
- [HSSA10] M. Houshmand, E. Soleymanpour, H. Salami, M. Amerian, and H. Deldari, "Efficient Scheduling of Task Graphs to Multiprocessors Using a Combination of Modified Simulated Annealing and List Based Scheduling," *3rd International Symposium on Intelligent Information Technology and Security Informatics*, pp.350-354, 2010.
- [Hu61] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol.9, no.6, pp.841-848, 1961.
- [IsMa76] D.L. Isaacson and R.W. Madsen, *Markov Chains Theory and Applications*, N.Y.: Wiley, 1976.
- [JaPa08] H. Javad and S. Parameswaran, "Synthesis of Heterogeneous Pipelined Multiprocessor Systems using ILP: JPEG Case Study," *Proc. of 6th IEEE/ACM/IFIP International Conf. on Hardware/Software Codesign and System Synthesis*, pp.1-6, 2008.
- [JoLi96] L.K. John and Y. Liu, "Performance Model for a Prioritized Multiple-Bus Multiprocessor System," *IEEE Transactions on Computers*, vol. 45, no. 5, pp.580-588, May 1996.
- [KaLi01] I.H. Kazi and D.J. Lilja, "Coarse-Grained Thread Pipelining: A Speculative Parallel Execution Model for Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, Issue 9, pp. 952-966, Sept 2001.
- [KEGS09] V.V. Kindratenko, J.J. Enos, S. Guochun, M.T. Showerman, G.W. Arnold, J.E. Stone, J.C. Philips, and H. Wen-mei, "GPU clusters for high-performance computing," *International Conference on Cluster Computing and Workshops*, pp.1-8, 2009.
- [KFHM08] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, "Multiprocessor Systems Synthesis for Multiple Use-Cases of Multiple Applications on FPGA" *ACM Transactions on Design Automation of Electronic Systems*, vol.13, no.3, article 40, July 2008.

-
- [KiGV83] S. Kirkpatrick, C.D. Gelat, and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, 1983.
- [Kins10] W. Kinsner, *Fractal and Chaos Engineering*. Lecture Notes. Winnipeg, MB: Dept. Electrical and Computer Eng., Univ. Manitoba, 2010.
- [Klei75] L. Kleinrock, *Queueing Systems. Volume 1: Theory*, N.Y.: Wiley, 1975.
- [Lee88a] E.A. Lee, "Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages," *VLSI Signal Processing III*, IEEE Press, 1988.
- [Lee88b] E.A. Lee, "Programmable DSP architectures. I," *IEEE ASSP Magazine*, Vol.5, Issue 4, pp.4-19, 1988.
- [Lee89] E.A. Lee, "Programmable DSP architectures. II," *IEEE ASSP Magazine*, Vol.6, Issue 1, pp.4-14, 1989.
- [Lee91] E.A. Lee, "Consistency in dataflow graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, issue 2, pp. 223-235, April 1991.
- [LeMe87] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*. 75(9):1235-1245, September 1987.
- [LePa95] E.A. Lee and T.M. Parks, "Dataflow Process Networks," *Proc. of the IEEE*. Vol.83, Issue 5, pp.773-779, May 1995.
- [LeSa98] E.A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. vol. 17, issue 12, pp. 1217-1229, Dec. 1998.
- [LeSe92] D. Lee and B. Sengupta, "An approximate analysis of a cyclic server queue with limited service and reservations," *Queueing Syst. Theory Appl*, vol. 11, no. 1-2, pp. 153-178, Jul 1992.
- [LiHs90] F. Lin and C. Hsu, "Task assignment scheduling by simulated annealing," *IEEE Region 10 Conference on Computer and Communication System*, vol.1, pp.279-283, 1990.
- [LOCX05] T. Lv, I.B. Ozer, S.T. Chakradhar, J. Xu, W. Wolf, and J. Henkel, "A Methodology for Architectural Design of Multimedia Multiprocessor SoCs," *IEEE Design and Test of Computers*, vol.22, issue 1, pp.18-26, 2005.
- [LuDM09] G. Luo-feng, Z. Duo-li, and G. Mung-Lun, "Performance Evaluation of Cluster-Based Homogeneous Multiprocessor System-on-Chip using FPGA Device," *4th International Conference on Embedded and Multimedia Computing*, pp.1-4, 2009.

-
- [MBCG82] M.A. Marsan, G. Balbo, G. Conte, and F. Gregoretti, "Modeling Bus Contention and Memory Interference in a Multiprocessor System," *IEEE Transactions on Computers*, vol. C-32, no. 1, pp. 60-72, Jan 1983.
- [MaAi02] K. Masuda and E. Aiyoshi, "Solution to Combinatorial Problems by Using Chaotic Global Optimization Method on a Simplex," *Proc. of the 41st SICE Annual Conference*, vol. 2, pp. 1313-1318, 2002.
- [Meal55] G.H.A. Mealy, "Method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, pp. 1045-1079, 1955.
- [MiHu04] J. Mingjun and T. Huanwen, "Application of Chaos in Simulated Annealing," *Chaos, Solutions and Fractals*, vol. 21, no. 4, pp. 931-944, Aug. 2004.
- [Moor56] E.F. Moore, *Gedanken-Experiments on Sequential Machines*, Automata Studies, Princeton University Press, New Jersey, 1956.
- [MuAM87] T.N. Mudge, H.B. Al-Sadoun, and B.A. Makrucki, "Memory-Interference Model for Multiprocessors based on Semi-Markov Processes," *IEEE Proceedings on Computers and Digital Techniques*, vol. 134, issue 4, pp. 203-214, July 1987.
- [Nade88] M. Naderi, "Modelling and performance evaluation of multiprocessors, organizations with multi-memory units," *SIGARCH Comput. Archit. News*, vol. 16, no. 5, pp. 35-51, Dec 1988.
- [NaDS92] A.K. Nanda, D. DeGroot, and D.L. Stenger, "Scheduling directed task graphs on multiprocessors using simulated annealing," *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp.20-27, 1992.
- [Nels95] R. Nelson, *Probability, Stochastic Processes, and Queueing Theory*, N.Y.: Springer-Verlag, 1995.
- [PaMi98] J.M. Paul and M.H. Mickle, "Multiprocessor Shared Memory Access and Rewards," *Journal of the Franklin Institute*, vol. 335, Issue 4, pp. 629-641, May 1998.
- [Quin04] M. Quinn, *Parallel Programming in C with MPI and OpenMP*. 1st Ed., NY: McGraw-Hill, 2004.
- [Reis85] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, New York, 1985.
- [ReVa99] G.L. Reijns and J.C. van Gemund, "Analysis of a Shared-Memory Multiprocessor via a Novel Queueing Model," *J. Syst. Archit.(Netherlands)*, vol. 45, issue 14, pp.1189-1193, July 1999.

-
- [RGBM08] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini, "A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC Platforms with communication awareness," *International Journal of Parallel Programming*, vol. 36, issue 1, pp. 3-36, 2008.
- [Rose71] M. Rosenblatt, *Markov Processes. Structure and Asymptotic Behavior*, N.Y.: Springer-Verlag, 1971.
- [RSJK05] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer, "An FPGA-based soft multiprocessor system for IPV4 packet forwarding", *International Conference on Field Programmable Logic and Applications*, pp.487-492, Aug. 2005.
- [SeDe79] A.S. Sethi and N. Deo, "Interference in Multiprocessor Systems with Localized Memory Access Probabilities," *IEEE Transactions on Computers*, vol. C-28, no. 2, pp. 157-163, Feb 1979.
- [ShKi96] D. Shaw and W. Kinsner, "Chaotic Simulated Annealing in Multilayer Feedforward Networks," *Canadian Conference on Electrical and Computer Engineering*, vol. 1, pp. 265-269, 1996.
- [ShPS09] R. Shanmugapriya, S. Padmavathi, and S.M. Shalinie, "Contention Awareness in Task Scheduling Using Tabu Search," *IEEE International Advance Computing Conference*, pp.272-277, 2009.
- [SLOW07] J. Schlessman, M. Lodato, B. Ozer, and W. Wolf, "Heterogeneous MPSoC Architectures for Embedded Computer Vision," *2007 IEEE Conference on Multimedia and Expo*, pp.1870-1873, 2007.
- [Taka88] H. Takagi, "Queuing analysis of polling models," *ACM Comput. Surv.*, vol. 20, no. 1, pp. 5-28, Mar 1988.
- [TaMu86] H. Takagi and M. Murata, "Queueing analysis of nonpreemptive reservation priority discipline," *ACM SIGMETRICS '86/PERFORMANCE '86*, vol. 14, no. 1, pp. 237-244, May 1986.
- [TBCP09] A. Tumeo, M. Branca, L. Camerini, C. Pilato, P.L. Lanzi, F. Ferrandi, and D. Sciuto, "Mapping Pipelined Applications onto Heterogeneous Embedded Systems: A Bayesian Optimization Algorithm Based Approach," *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp.443-452, 2009.
- [VaAa87] P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications*. Eindhoven, The Netherlands: D. Reidel Publishing Company, 1987.

-
- [VaAL92] P.J.M. Van Laarhoven, E.H.L. Aarts, and J.K. Lenstra, "Job Shop Scheduling by Simulated Annealing," *Operations Research*, vol. 40, no.1, pp. 113-125, 1992.
- [VeKi83] M.P. Vecchi, and S. Kirkpatrick, "Global Wiring by Simulated Annealing," *IEEE Transactions on Computer-Aided Design*, vol. 2, no. 4, Oct. 1983.
- [VFMA09] S. Vakili, S.M. Fakhraie, S. Mohammadi, and A. Ahmadi, "Particle Swarm Optimization for Run-Time Task Decomposition and Scheduling in Evolvable MPSoC," *International Conference on Computer Engineering and Technology*, vol.2, pp.28-32, 2009.
- [WaSm98] L. Wang and K. Smith, "On Chaotic Simulated Annealing," *IEEE Transactions on Neural Networks*, vol. 9, no. 4, pp. 716-718, July 1998.
- [WaTi00] L. Wang and F. Tian, "Noisy Chaotic Neural Networks for Solving Combinatorial Optimization Problems," *Proc. of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, vol. 4, pp. 37-40, 2000.
- [WaZi05] X. Wang and S.G. Ziavras, "A Framework for Dynamic Resource Assignment and Scheduling on Reconfigurable Mixed-Mode On-Chip Multiprocessors", *2005 IEEE International Conference on Field-Programmable Technology*, pp.51-58, 2005.
- [WiCL97] T. Wiantong, P. Cheung, and W. Luk, "Comparing three heuristic search methods for functional partitioning in hardware-software codesign," *Design Automat. Embedded Syst.*, vol. 2, no. 1, pp. 5-32, Jan. 1997.
- [WoJM08] W. Wolf, A.A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.27, no.10, Oct. 2008.
- [WuPe04] B. Wu and C. Peng, "System-on-Chip Design with Dataflow Architecture," *Proceeding of the 8th international conference on Computer Supported Cooperative Work in Design*, vol.2, pp.712-716, 2004.
- [Xili05] Xilinx Inc., *EDK Base System Builder (BSB) support for XUPV2P Board*, Xilinx University Program Presentation, 2005.
- [Xili06] Xilinx Inc., *On-Chip Peripheral Bus V2.0 with OPB Arbiter(V1.10C)*. DS401, Aug 2006.
- [Xili07] Xilinx Inc., *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. DS083 V4.7, Nov 2007.
- [Xili08a] Xilinx Inc., *Xilinx University Program Virtex-II Pro Development System – Hardware Reference Manual*. UG069 V1.1, Apr 2008.
-

- [Xili08b] Xilinx Inc., *MicroBlaze Processor Reference Guide*. UG081 V9.0, 2008.
- [Xili08c] Xilinx Inc., *ISE 10.1 Quick Start Tutorial*, 2008.
- [Xili08d] Xilinx Inc., *ISE Design Suite 10.1 Release Notes and Installation Guide*, 2008.
- [Xili08e] Xilinx Inc., *Embedded System Tools Reference Manual – Embedded Development Kit- EDK 10.1, Service Pack 3*, 2008.
- [YaHa09] H. Yang and S. Ha, “Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC,” *Design, Automation & Test in Europe Conference & Exhibition 2009*, pp.69-74, 2009.
- [ZhSh07] L. Zhou and S. Shi-Xin, “A Genetic Scheduling Algorithm Based on Knowledge for Multiprocessor System,” *International Conference on Communications, Circuits and Systems*, pp.900-904, 2007.

APPENDIX A

A brief overview of the development flow for the experimental implementation of the MPSoC framework using the Xilinx Virtex-II Pro FPGA, the XUPV2P development board, and the Xilinx development tools is given in this appendix. The associated design files can be found in the attached CD-ROM in the “MPSoC Experimental Implementation” directory.

A.1. Xilinx Tool Development Flow

Two main development tools were used for the development of the experimental implementation of the MPSoC system. These tools are the Xilinx ISE version 10.1.03 [Xili08c] [Xili08d], and the Xilinx EDK version 10.1.03 [Xili08e]. The Xilinx ISE is the main development tool for Xilinx FPGA development. In this tool, the target Xilinx device is chosen, and the hardware design is described using VHDL. This tool synthesizes the hardware design, and then implements the design with translation, mapping, and place and routing. A programming bitstream file is then generated, and the FPGA can then be programmed through a JTAG connection. VHDL module is created for each hardware component within the design, except for the processors, which are created using the Xilinx EDK tool. The hardware modules in this design are:

1. CLK_DIVIDE_I – This is a hardware module to divide the input frequency for the crystal oscillator on the XUPV2P board from 100 MHz down to 10 MHz, for the global memory controller. The Microblaze processors used the 100 MHz clock, and the global memory controller uses the 10 MHz clock.

2. MEM_CNTL_I – This is the memory controller hardware module. This is the interface between the global memory and the processors as described in section 3.2.4.
3. GLB_MEM_I – This is the global memory. For the experimental implementation the global memory was made up of internal block RAMs within the Xilinx Virtex-II Pro FPGA. The size of the global memory used is 16 kB as described in section 3.2.6.
4. TASK_CONTROL_I – This is the task controller module, which is responsible for the scheduling of the tasks for each of the processors. This implementation is described in section 3.2.2.
5. SNOOPY_I – This is the snoop hardware module that monitors the operation time of each task and collects data for the purposes of analysis. This module is described in section 3.2.7.
6. Inst_Multiproc – This is the hardware block that implements all of the processors within the MPSoC. This block is generated using the Xilinx EDK tool, as described later in this appendix.

Two screenshots of the Xilinx ISE tool are shown below in Fig. A - 1 and Fig. A - 2. The first screenshot shows the ISE tool with the design summary page open, and the second screenshot shows the ISE tool with one of the VHDL modules open. These screenshots are shown to give a general idea of the development environment of the Xilinx ISE tool.

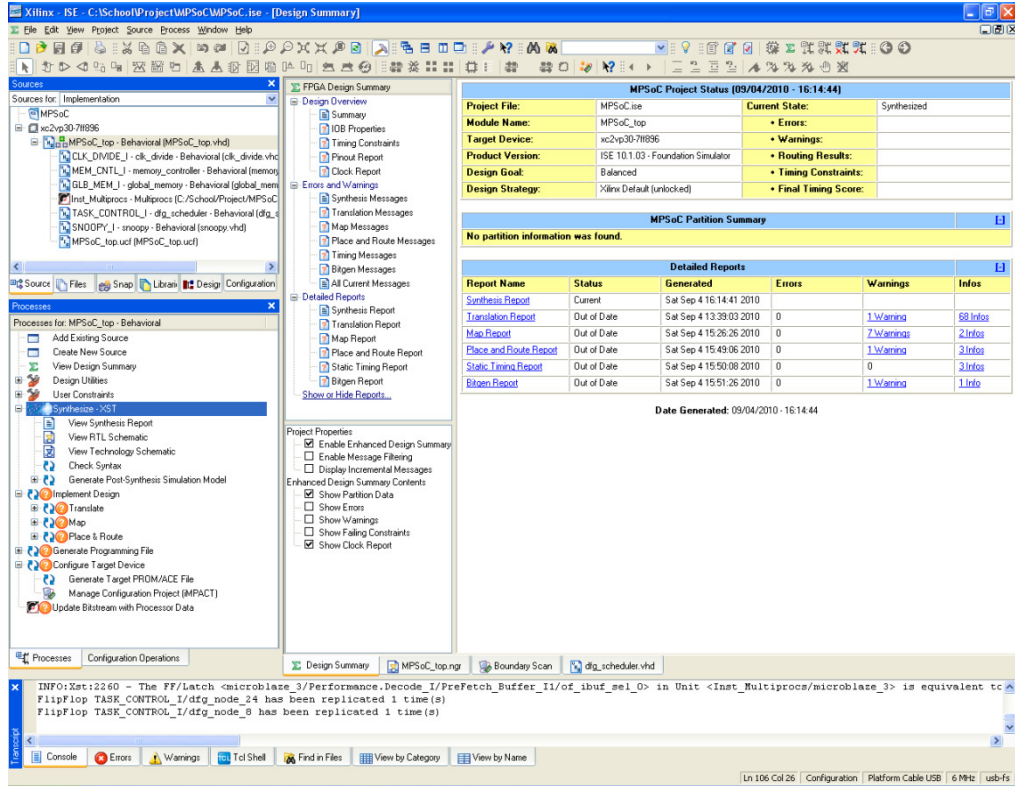


Fig. A - 1. A Screenshot of the design summary page within the Xilinx ISE 10.1.03 tool

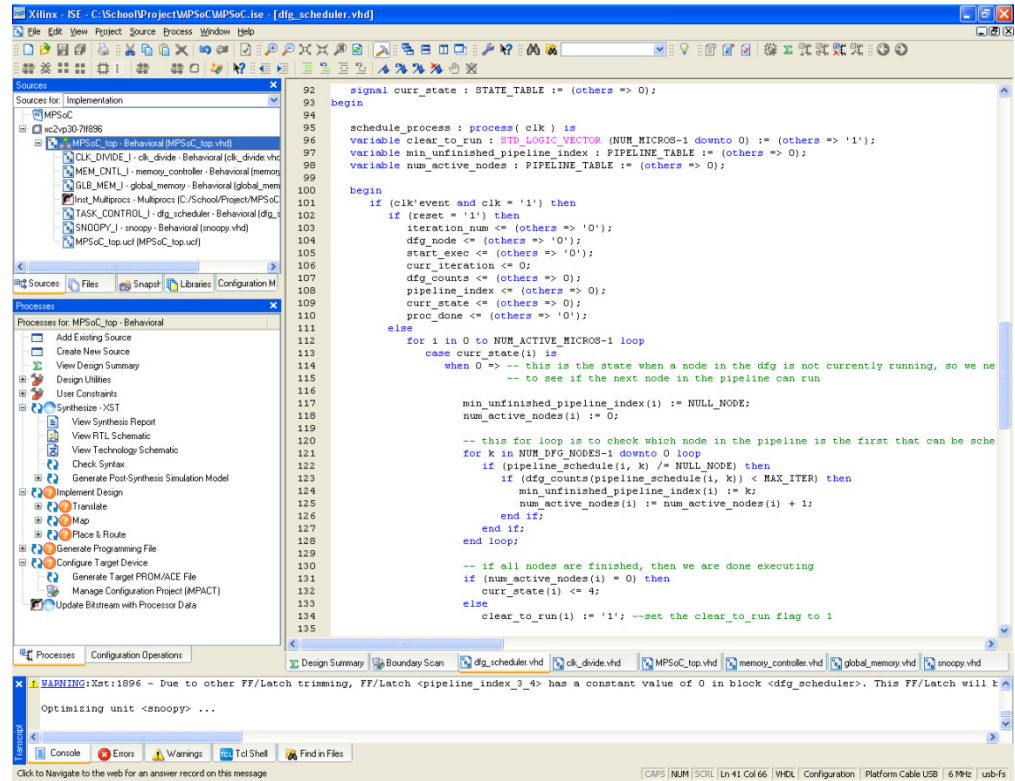


Fig. A - 2. A screenshot of a VHDL module Open within the Xilinx ISE 10.1.03 tool

The processors within the MPSoC experimental implementation are Microblaze processors (version 6.0). These are implemented using the Xilinx EDK (Embedded Development Kit) tool version 10.1.03. This development tool allows for Xilinx supported IP (intellectual property) blocks to be implemented into a single hardware module that can then be interfaced to other hardware modules in the ISE tool. The Xilinx EDK allows for multiple processors to be implemented within a single hardware block, and allows each processor to be customized through the addition of peripherals and/or processor features (such as pipelined instructions, floating point unit, etc.). Experiments were conducted with 1, 2, 3, and 4 processors within the EDK generated hardware block. Processor 1 was the head processor, and it had additional peripherals to allow for a serial port connection. This allowed data to be written to a serial port, which was connected to a RS-232 driver chip on the XUPV2P board, which was then interfaced to a PC to display the output results. Several custom peripherals were created to interface with the task controller (as described in 3.2.3), memory controller (as described in 3.2.5), and snoopy hardware block (as described in 3.2.8). The connection of peripherals to processors, and the routing of internal peripheral buses are done through the EDK tool GUI. Three screenshots of the EDK tool are shown below. Fig. A - 3 shows the GUI used to connect peripherals to particular internal processor buses. Fig. A - 4 shows the automatically generated block diagram within the EDK for the four processor case. This shows the processors, their internal memory, and peripherals, and how they are connected. Fig. A - 5 shows the EDK with a software application file open. These screenshots are shown to give a general idea of the development environment for the EDK tool.

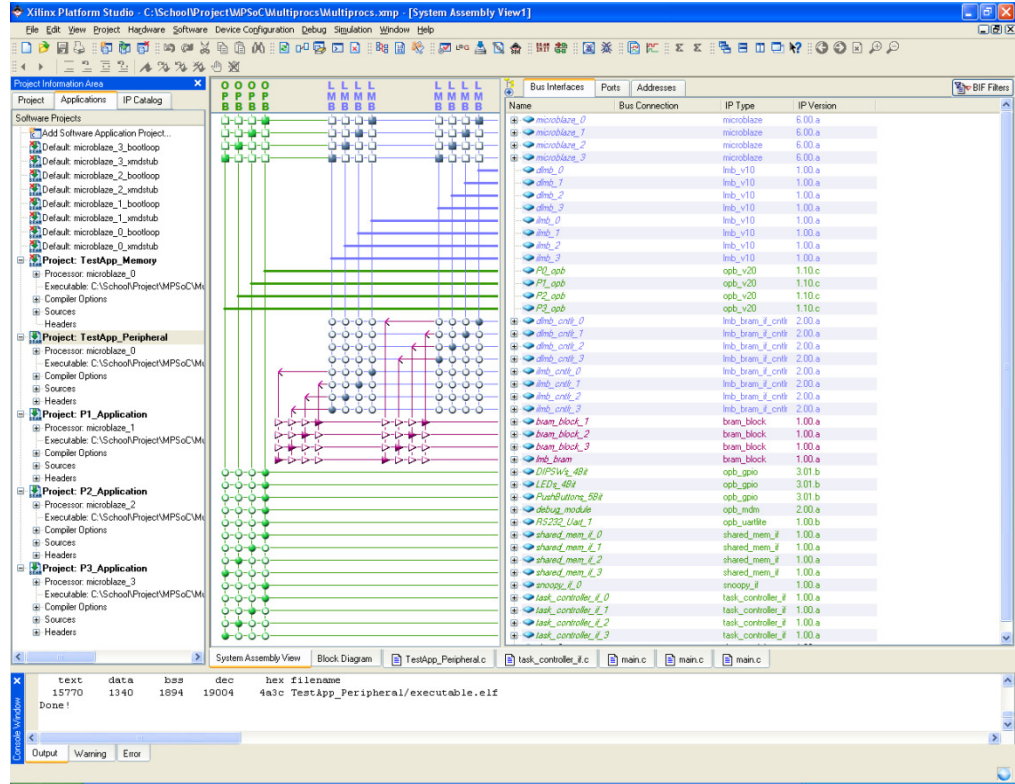


Fig. A - 3. A screenshot of the peripheral interface GUI within the Xilinx EDK tool.

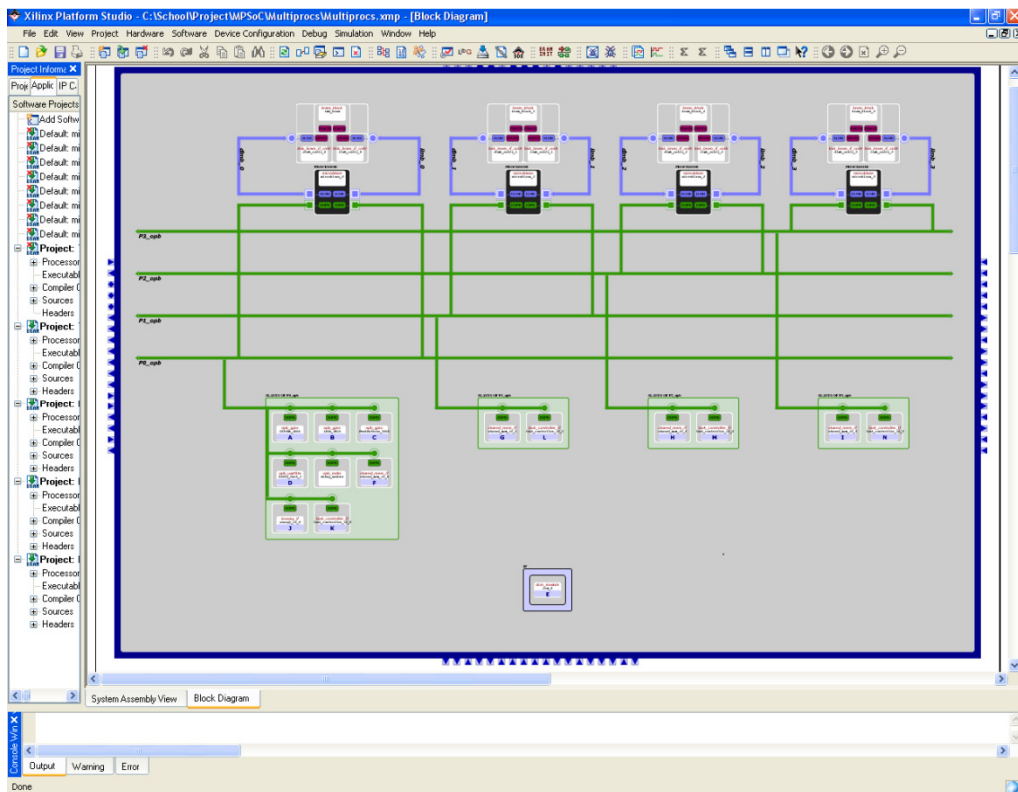


Fig. A - 4. A screenshot of the four Microblaze processor block diagram generated by the Xilinx EDK tool.

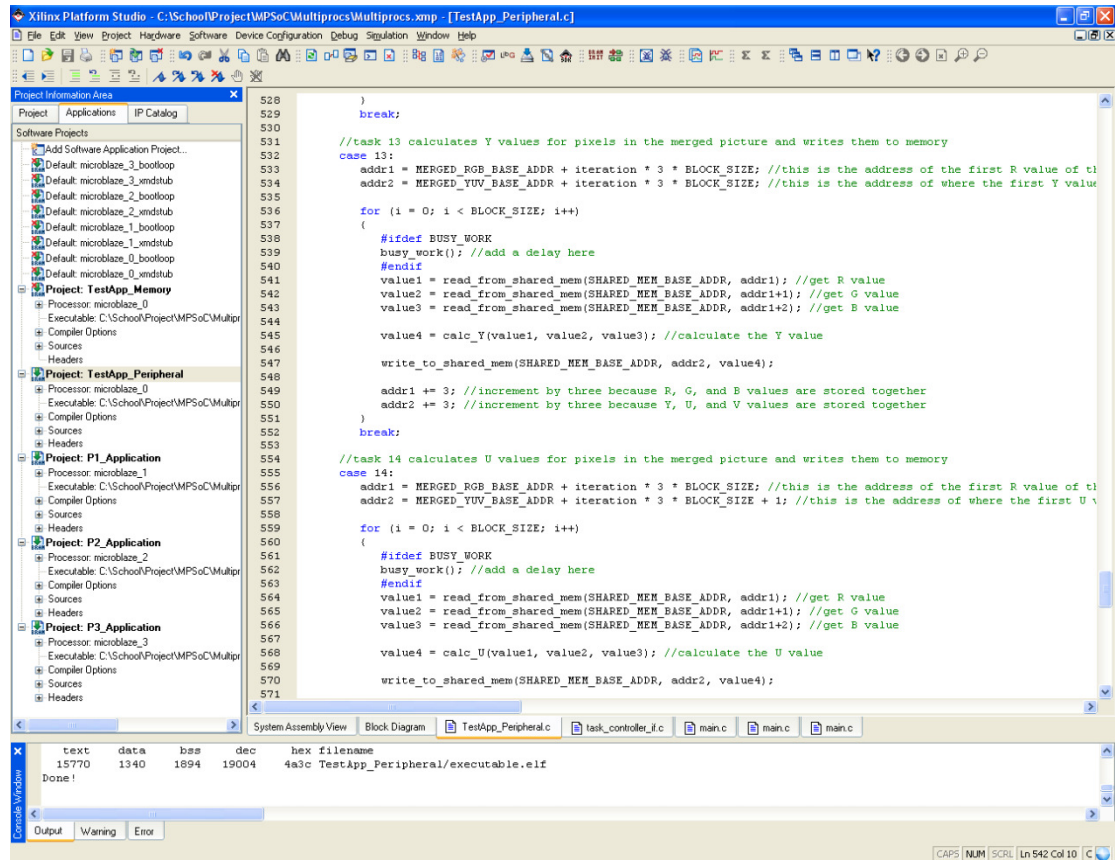


Fig. A - 5. A screenshot of the Xilinx EDK tool with a software application for one of the processors open.

The hardware and software build files are created by the EDK tool and then the ISE tool uses these to generate the overall bitstream to be loaded into the Virtex-II Pro FPGA. The application software for each of the processors is written in the CD programming language. In addition to the application code, peripheral drivers for the custom peripherals are also written in C.

A.2. Simulation of the MPSoC Framework Implementation

The MPSoC framework implementation is a complex digital design, which would be very difficult to implement successfully without the possibility of individual component simulation. The Xilinx ISE tool has a built-in simulation tool that allows test bench files to be generated that act as stimulus to each of the hardware modules. This

allows the functionality of each of the hardware modules to be verified through timing diagram simulations. Components can then be interfaced to each other and simulated at higher levels. Without the ability to simulate at a low level, the design would be too complex to effectively debug any issues within the design. Fig. A - 6 shows a test bench file for the task scheduler module. This is organized in a timing diagram view that is editable to force simulation inputs as desired for simulation.

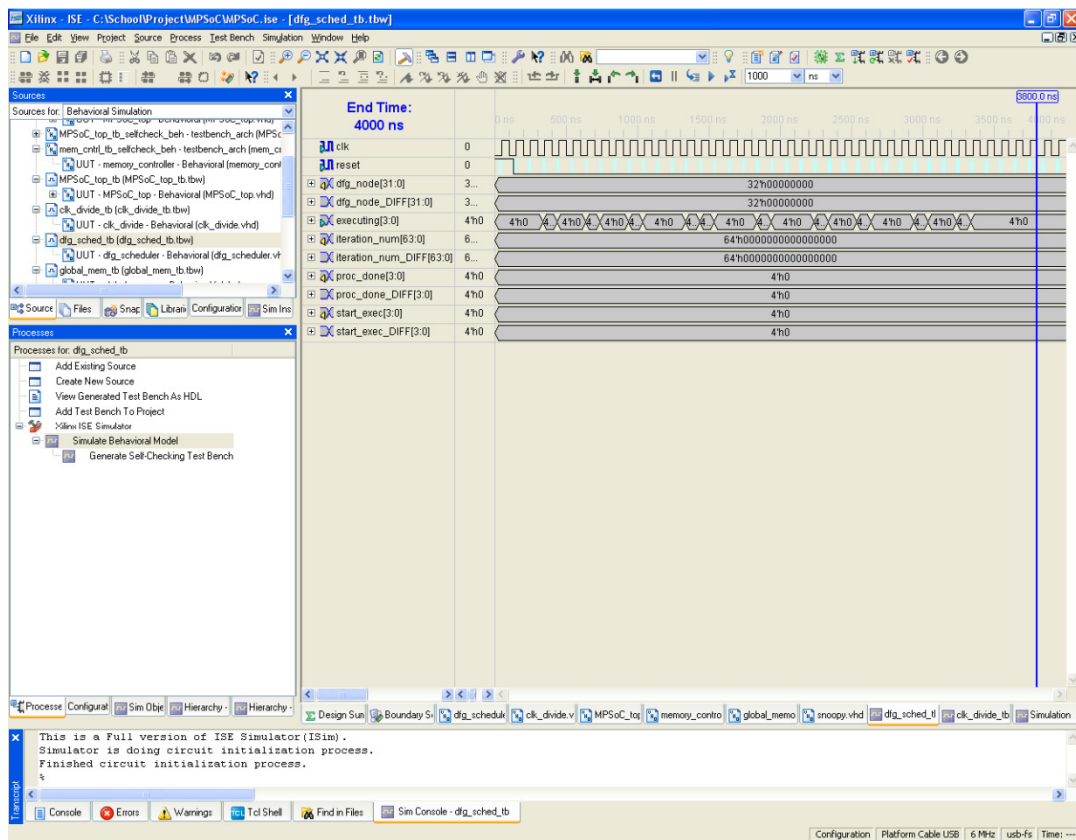


Fig. A - 6. A screenshot of a simulation test bench for the task scheduler module.

Fig. A - 7 shows a simulation run with the test bench that shows the output values given the stimulus defined in the test bench file. This simulation can be used to view output signals as well as signals internal to the hardware module, to allow for effective

debugging. The simulated output can then be compared to expected values to determine if the VHDL module is behaving as expected.

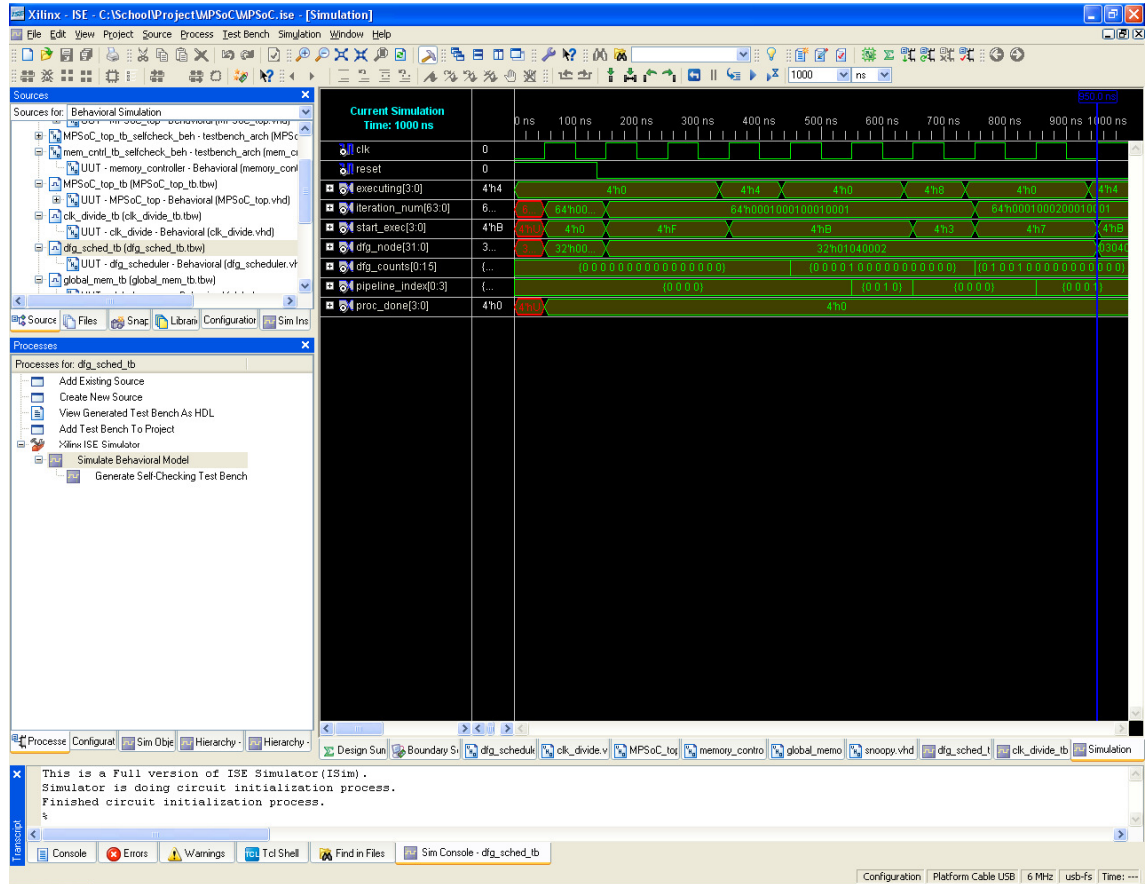


Fig. A - 7. A screenshot of a simulation of the task scheduler hardware module.

A.3. Xilinx University Program Virtex-II Pro Development Board

The Xilinx University Program Virtex-II Pro (XUPV2P) development board was used to implement the MPSoC experimental implementation. Fig. A - 8 shows a picture of the XUPV2P development board taken from [Xili05]. This picture labels many of the features available on the XUPV2P development board. For the MPSoC experimental implementation, the key features used were the Virtex-II Pro FPGA, the USB connection for loading the FPGA configuration via JTAG, and the RS-232 serial port, which was used to output data to a PC for viewing program metrics.

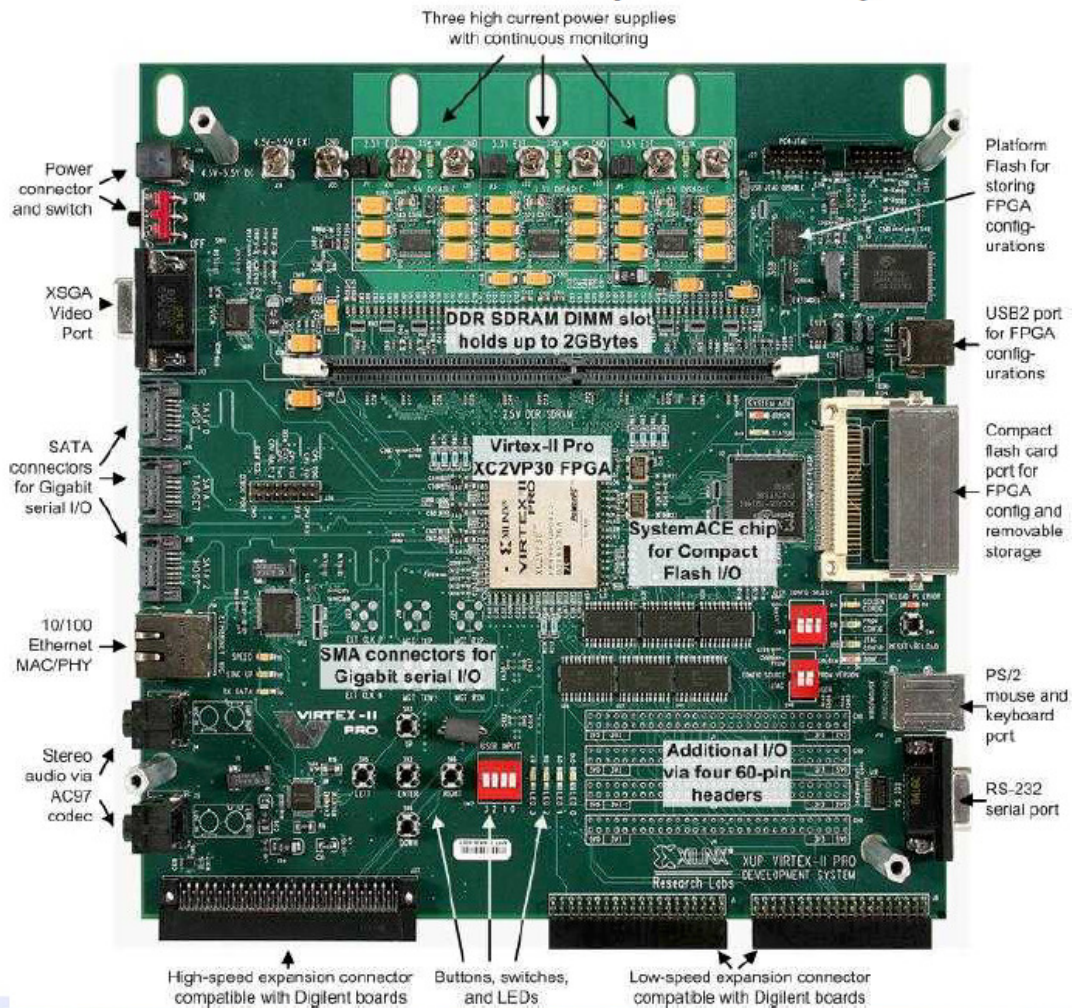


Fig. A - 8. Xilinx University Program Virtex-II Pro Development Board from [Xili05]

A.4. Experimental Setup

Development of the MPSoC experimental implementation was as described in the previous sections of this appendix. The test setup for running the experiments is as described below. Configuration bitstreams developed for the FPGA were downloaded using the Xilinx ISE tool through a USB connection. The reset signal for the FPGA was mapped to one of the switches on the XUPV2P board, so a hardware reset could be done at any time to restart the running of the application. The head processor was used to communicate results. This was done through communication over the RS-232 serial port

at a baud rate of 9600 bps, which then connected to a PC. The output was viewed using the Microsoft HyperTerminal program (version 5.1). Fig. A - 9 shows a high level block diagram of the test setup connections. Fig. A - 10 shows a screenshot of the output sent by the head processor within the FPGA and captured by the HyperTerminal program over the RS-232 serial connection.

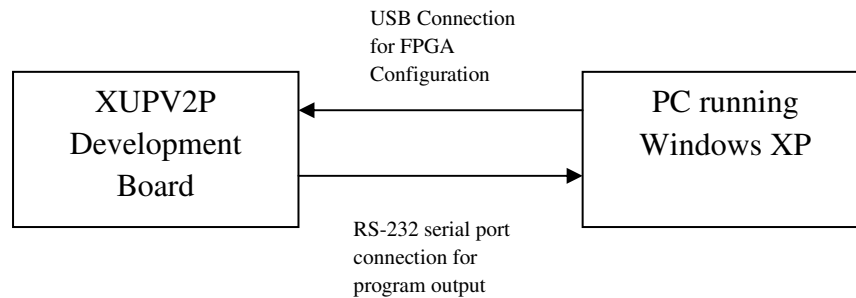


Fig. A - 9. Block diagram of MPSoC experimental test setup.

```

test2 - HyperTerminal
File Edit View Call Transfer Help
Starting processing...
Finished all processors tasks
Max iteration: 500
Total execution time upper word: 0 lower word: 3120070
Task 0: mem access: 500 exec time upper word: 0 lower word: 396128
Task 1: mem access: 500 exec time upper word: 0 lower word: 352800
Task 2: mem access: 500 exec time upper word: 0 lower word: 340542
Task 3: mem access: 500 exec time upper word: 0 lower word: 348612
Task 4: mem access: 500 exec time upper word: 0 lower word: 341462
Task 5: mem access: 500 exec time upper word: 0 lower word: 378312
Task 6: mem access: 1500 exec time upper word: 0 lower word: 415109
Task 7: mem access: 2000 exec time upper word: 0 lower word: 669928
Task 8: mem access: 1500 exec time upper word: 0 lower word: 484903
Task 9: mem access: 1500 exec time upper word: 0 lower word: 305644
Task 10: mem access: 2000 exec time upper word: 0 lower word: 641777
Task 11: mem access: 1500 exec time upper word: 0 lower word: 523668
Task 12: mem access: 4500 exec time upper word: 0 lower word: 962422
Task 13: mem access: 2000 exec time upper word: 0 lower word: 644316
Task 14: mem access: 2000 exec time upper word: 0 lower word: 458768
Task 15: mem access: 2000 exec time upper word: 0 lower word: 743540
Processor 0 idle time: 1177647 memory wait time: 1105044
Processor 1 idle time: 732069 memory wait time: 1639581
Processor 2 idle time: 1193161 memory wait time: 1242844
Processor 3 idle time: 1369475 memory wait time: 894181
  
```

Fig. A - 10. Screenshot of serial output received by HyperTerminal running on the PC, with data sent from the XUPV2P board.

APPENDIX B

A brief description of the implementation of the MPSoC framework analytical model and optimization algorithm experiments is given in this appendix.

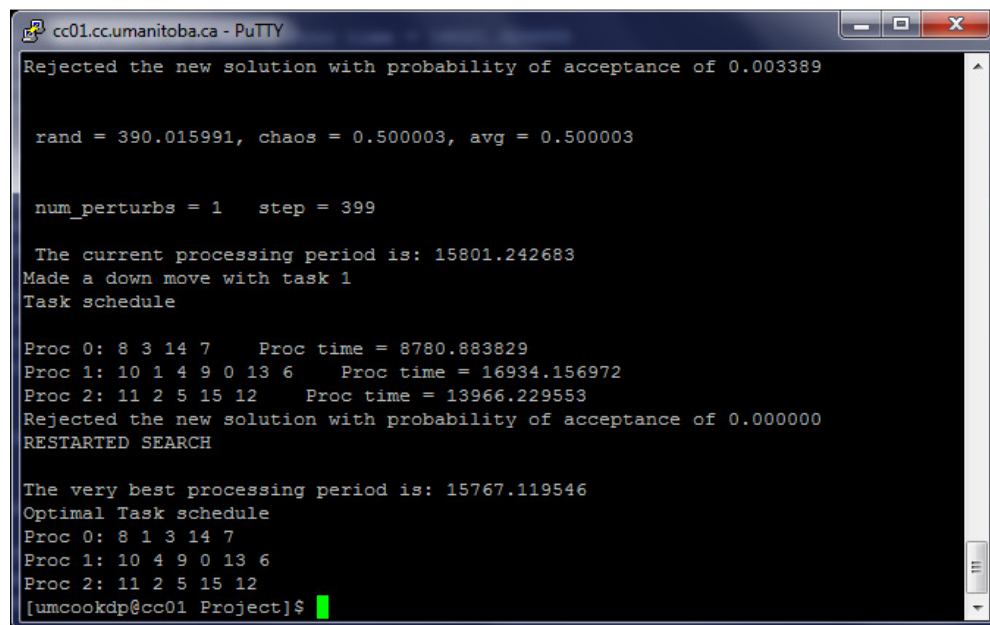
B.1. Matlab Implementation of the Analytical Model

The analytical model for the MPSoC framework described in Chapter 4 was first implemented as a script to be run in Matlab. The benefit of using Matlab is that many of the matrix operations required for the analytical model are built-in operations in Matlab. The Matlab script ran for a single window within a pipelined processing period (such as the example shown in Fig. 4-4). While the Matlab had the benefit of many built-in functions to allow for fast implementation, because it is a scripting language, it is inherently slower than a compiled program. Running a single window within a pipeline period could take several minutes, and then this would need to be repeated for each window in a pipeline period to properly analyze a given MPSoC framework implementation. Since the simulated annealing optimization algorithms (as described in Chapter 5) require many MPSoC framework implementations to be analyzed, the Matlab script would take days to run the optimization algorithms. The associated analytical model Matlab files can be found in the attached CD-ROM in the “Analytical Model” directory.

B.2. Implementation of the Analytical Model and Optimization Algorithm in the C Programming Language

The simulated annealing and chaotic simulated annealing algorithms described in Chapter 5 require analysis of many MPSoC framework implementations using the

analytical method proposed in Chapter 4. The Matlab scripts described in the previous section are too slow to run the optimization algorithms in a reasonable amount of time. Therefore, the analytical method was implemented in the C programming language. This required writing code for many of the matrix operations that are inherently built into Matlab, but had the benefit of much faster running time. A program was written for each of the simulated annealing and chaotic simulated annealing programs in the C programming language, and then compiled, using the publically available GNU C compiler. The programs were compiled and run remotely on the University of Manitoba IST machines (cc01.cc.umanitoba.ca, cc02.cc.umanitoba.ca, cc03.cc.umanitoba.ca, or cc04.cc.umanitoba.ca). Fig. B - 1 shows a screenshot of the output from one of the chaotic simulated annealing algorithms (in this case run with 3 processors).



```
cc01.cc.umanitoba.ca - PuTTY
Rejected the new solution with probability of acceptance of 0.003389

rand = 390.015991, chaos = 0.500003, avg = 0.500003

num_perturbs = 1 step = 399

The current processing period is: 15801.242683
Made a down move with task 1
Task schedule

Proc 0: 8 3 14 7 Proc time = 8780.883829
Proc 1: 10 1 4 9 0 13 6 Proc time = 16934.156972
Proc 2: 11 2 5 15 12 Proc time = 13966.229553
Rejected the new solution with probability of acceptance of 0.000000
RESTARTED SEARCH

The very best processing period is: 15767.119546
Optimal Task schedule
Proc 0: 8 1 3 14 7
Proc 1: 10 4 9 0 13 6
Proc 2: 11 2 5 15 12
[umcookdp@cc01 Project]$
```

Fig. B - 1. Screenshot of output from chaotic simulated annealing program.

The associated analytical model Matlab and C code files can be found in the attached CD-ROM in the “Chaotic Simulated Annealing” directory.