Scheduling in DSP Multiprocessor Systems

by

Suryani Chao

A thesis submitted to the Faculty of Graduate
Studies and Research in partial fulfilment of the
requirements for the degree of
Master of Applied Science
Department of Systems and Computer
Engineering

Carleton University

Ottawa, Ontario

September 27, 2002

©copyright

2002, Suryani Chao

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79695-7

Canada

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of this thesis

Scheduling in DSP Multiprocessor Systems

Submitted by Suryani Chao
in partial fulfilment of
the requirements for the degree of
Master of Applied Science

Thesis Co-supervisor

Thesis Co-supervisor

Chair, Department of Systems and Computer Engineering

Carleton University

September 27, 2002

Abstract

To provide the flexibility of updating the system parameters in Voice over Internet Protocol (VoIP) gateways, a programmable *Digital Signal Processing* (DSP) multiprocessor system is used. The pattern of assignment of service requests to the processors directly affects the number of service requests that can be processed. This has a major impact on the DSP multiprocessor system performance in VoIP. This thesis describes research towards optimal assignment of service requests to processors, a difficult NP-Complete problem. A number of heuristic algorithms to perform *service-request-to-processor assignment* (SRTPA) are described. These SRTPA algorithms are tested on simulated traffic representing current and expected future conditions in VoIP.

Acknowledgements

I would like to thank Professor John Chinneck and Professor Rafik Goubran, who gave me guidance and support throughout the entire thesis process, without which I would never have succeeded. I would also like to thank the many people whose comments and assistance have been so helpful in this work.

I must also thank the Communications and Information Technology Ontario (CITO) for its funding.

And a very special thank you to my family and friends, for their support, love and prayers.

TABLE OF CONTENTS

List of Tables	VIII
List of Figures	IX
List of Abbreviations	XI
Chapter 1 Introduction	1
1.1 An Overview of Scheduling in DSP Multiprocessor Systems	1
1.2 Thesis Outline	3
Chapter 2 Background Information	4
2.1 A Review of Bin Packing	4
2.1.1 Classical Off-line Algorithms	6
2.1.2 The Harmonic Algorithm and its Variants	10
2.1.3 Online Matching Algorithm	12
2.2 Review of DSP Multiprocessor Systems	15

Chapter 3	State of the Art in SRTPA Algorithm	18
Chapter 4	The SRTPA Problem	20
4.1	Thesis Problem Statement.....	20
4.2	Analysis of the State of the Art In SRTPA	22
4.3	Importance of SRTPA.....	23
Chapter 5	Algorithms for the SRTPA problem.....	24
5.1	Classical/Reasonable Algorithms.....	27
5.1.1	First-Fit Algorithm (FFA).....	28
5.1.2	Best-Fit Algorithm (BFA).....	30
5.1.3	Worst-Fit Algorithm (WFA).....	32
5.1.4	Modified-First-Fit Algorithm, Modified-Best-Fit Algorithm and Modified-Worst-Fit Algorithm (MFFA, MBFA and MWFA)	34
5.2	Modified Harmonic Algorithm and Online Matching Algorithm (MHA and OMA)	36
5.2.1	Preprocessing	36
5.2.1.1	MHA Preprocessing	37
5.2.1.2	OMA Preprocessing	44
5.2.2	Processing Incoming Service Requests.....	48
5.2.3	Processing the Termination of Service Instance	53
5.3	Best Possible Fit Algorithm	53
5.3.1	Preprocessing	54
5.3.2	Processing Incoming Service Requests.....	59
Chapter 6	Empirical Results	64
6.1	Simulation Setup	65

6.1.1	Traffic Model	67
6.1.2	Multiprocessor Model	69
6.1.3	Baseline Comparison.....	69
6.1.4	Matlab Framework	70
6.2	Results	72
6.2.1	Results for Current Traffic Stream.....	72
6.2.2	Results for Future Traffic Stream.....	75
6.2.3	Results for Low Bit Rate Current Traffic Stream	79
6.2.4	Results for Low Bit Rate Future Traffic Stream	82
6.3	Discussion of Results	85
Chapter 7	Conclusions and Summary of Results	90
7.1	Conclusions	90
7.2	Summary of Contributions	91
7.3	Future Work	92

LIST of TABLES

6.1 List of DSP Algorithms Used.....	65
6.2 Distributions of Service Types in Traffic Streams.....	67
6.3 Blockage Rate for $TS_{\text{current_all}}$ using SRTPA Algorithms	71
6.4 Blockage Rate for $TS_{\text{future_all}}$ using SRTPA Algorithms	75
6.5 Blockage Rate for $TS_{\text{current_lowbitrate}}$ using SRTPA Algorithms	78
6.6 Blockage Rate for $TS_{\text{future_lowbitrate}}$ using SRTPA Algorithms	81
6.7 Measure of Loading of Each Traffic Stream.....	86
6.8 Rules for Switching Algorithms.....	88

LIST of FIGURES

2.1	An Example of the First Fit Algorithm	7
2.2	An Example of the Best Fit Algorithm	8
2.3	An Example of the Worst Fit Algorithm.....	9
2.4	An Example of the Harmonic H_3 Algorithm.....	11
2.5	An Example of the ONLINEMATCH OLM_4 Algorithm	14
5.1	List of Service Types in EX1	26
5.2	Illustration of FFA Processing Service Requests Using EX1	29
5.3	Illustration of BFA Processing Service Requests Using EX1.....	31
5.4	Illustration of WFA Processing Service Requests Using EX1	33
5.5	Illustration of MBFA Processing Service Requests Using EX1	35
5.6	Illustration of MHA Preprocessing for EX1	42
5.7	Illustration of OMA Preprocessing for EX1	48
5.8	Illustration of MHA Processing Service Requests for EX1	49
5.9	Illustration of OMA Processing Service Requests for EX1	51
5.10	List of Service Types in EX2	55
5.11	Illustration of BPFA Preprocessing for EX2.....	59
5.12	Illustration of BPFA Processing Service Requests for EX1	61
6.1	Graphical Interface for DSP Multiprocessor Simulation	70
6.2	Performance for $TS_{current_all}$	73
6.3	Performance for $TS_{current_all}$ When More DSP Processors Are Used.....	74
6.4	Performance for TS_{future_all} When Fewer DSP Processors Are Used.....	76
6.5	Performance for TS_{future_all} When More DSP Processors Are Used.....	77
6.6	Performance for $TS_{current_lowbitrate}$ When Fewer DSP Processors Are Used ..	79

6.7	Performance for $TS_{\text{current_lowbitrate}}$ When More DSP Processors Are Used ...	80
6.8	Performance for $TS_{\text{future_lowbitrate}}$ When Fewer DSP Processors Are Used ...	82
6.9	Performance for $TS_{\text{future_lowbitrate}}$ When More DSP Processors Are Used.....	83

LIST of ABBREVIATIONS

BF – Best-Fit
BFA – Best-Fit Algorithm
BPFA – Best Possible Fit Algorithm
DSP – Digital Signal Processing
FF – First-Fit
FFA – First-Fit Algorithm
FSMC – Fixed Service Multiple Channel
FSSC – Fixed Service Single Channel
Kbps – Kilo bytes per second
MBFA – Modified Best-Fit Algorithm
MFFA – Modified First-Fit Algorithm
MHA – Modified Harmonic Algorithm
MIPS – Million Instructions Per Second
MWFA – Modified Worst-Fit Algorithm
OMA – Online Matching Algorithm
RAM – Random Access Memory
SRTPA – Service-Request-To-Processor-Assignment
VoIP – Voice Over Internet Protocol
WF – Worst-Fit
WFA – Worst-Fit Algorithm

CHAPTER 1

Introduction

Using a programmable Digital Signal Processing (DSP) multiprocessor system in Voice over Internet Protocol (VoIP) gateways provides the flexibility of updating the system parameters, which cannot be achieved using conventional hardwired systems. It also provides the possibility of changing some of the call parameters while the call is active (e.g. changing the vocoder type or echo cancellation nonlinear processor parameters). For this reason, there is great interest in scheduling and service requests allocation in DSP multiprocessor systems used in VoIP gateways. This thesis reports research on a first step towards optimal service-request-to-processor assignment (SRTPA) algorithms.

1.1 An Overview of Scheduling in DSP Multiprocessor Systems

The service-request-to-processor-assignment problem is stated as follows: given a multiprocessor DSP architecture with n DSP processors and a stream of service

requests, assign incoming service requests to DSP processors while satisfying a set of constraints imposed by the identical DSP cores in the architecture. The total MIPS requirement for all of the service instances assigned to a particular DSP core should never exceed the MIPS processing capability for the DSP core. There is a limit on the total program memory and a separate limit on the total data memory.

Scheduling and service requests allocation in DSP multiprocessor systems used in VoIP gateways nowadays are handled in an inefficient way by very simple algorithms. The conventional algorithms handle allocation such that each DSP processor handles only one kind of DSP algorithm. However, the more recent generation of DSP processors that are used in VoIP gateways can support more than one DSP algorithm.

The service-request-to-processor-assignment problem previously considered in the literature deals with a fixed number of dimensions. When assigning a service request to a DSP processor, all three processing requirements of a service type (program memory, data memory and MIPS) have to be allocated to a DSP processor. This problem is identical to the three-dimensional bin-packing problem so well-known in operations research (e.g. [Li and Cheng, 1992]). In contrast, the SRTPA problem considered here deals with allocating only one copy of program memory if another instance of the same service type exists in a specific DSP processor. This problem has not been dealt with previously.

This thesis presents an improved method that will provide better SRTPA results. The SRTPA algorithms reported here use ideas from classical bin-packing algorithms and the bin-packing literature. New heuristic algorithms have also been

derived. All of the above algorithms are compared to conventional algorithms used in current VoIP gateways.

The newly developed SRTPA algorithms reduce blockage rates significantly compared to currently used conventional algorithms. This study has also identified the best algorithm to use for solving the SRTPA problem under different conditions.

1.2 Thesis Outline

In the next few chapters a complete description and analysis of SRTPA, and methods to provide solutions to SRTPA problems will be provided. An introduction to bin-packing and various algorithms to solve bin-packing problems, as well as an overview of DSP multiprocessor systems will be covered in Chapter 2. Chapter 3 reviews the state of the art in SRTPA. Chapter 4 gives a formal statement of the problem considered here and discusses the importance of SRTPA. In Chapter 5, which is the main chapter of this thesis, a thorough description of the algorithms for solving SRTPA problems is presented. Chapter 6 summarizes test results and compares all the SRTPA algorithms. A summary of the research and a discussion of future directions is covered in the last chapter.

CHAPTER 2

Background Information

In this chapter a brief introduction to bin-packing and its various algorithms is given. Bin-packing and its various algorithms are defined and discussed in Section 2.1. In Section 2.2, the DSP multiprocessor system is outlined.

2.1 A Review of Bin-packing

In classical one-dimensional bin-packing, we are given a list of items, $L_n = \{X_1, X_2, \dots, X_n\}$, each X_i with a size $s(X_i) \in (0, 1]$, and a list of unit-capacity bins, and are asked to assign as many items as possible into the given bins.

There are several variants of the bin-packing problem, namely, off-line, on-line, one-dimensional, multidimensional and dynamic. Bin-packing algorithms that have full knowledge of all items before assigning any items to a bin are

called *off-line*. It is an *on-line* algorithm if it assigns every item based only on knowledge of the sizes of the current item and all previous items, i.e., without any information on the subsequent items. It is possible to sort all items in off-line bin-packing before assigning any items to a bin, but not possible in on-line bin-packing.

The main difference between one-dimensional and multidimensional problems is the number of item and bin attributes. In a *multidimensional* problem, instead of an item and a bin having a single attribute, there is a vector of attributes. Each item is a d -dimensional vector $X_i = (s_1(X_i), s_2(X_i), \dots, s_d(X_i))$, where $s_j(X_i) \in (0, 1]$ holds for $1 \leq j \leq d$. As mentioned by J. Csirik and G. G. Woeginger [1998], there are different types of multidimensional bin-packing problems, namely, vector packing, box packing and strip packing. In *vector packing*, one has to assign items subject to the constraint that in every bin the sum of all vectors is at most one in every coordinate. This problem arises as a crucial subproblem in scheduling with resource constraints [Garey et al. 1976]. Similarly to the vector packing problem, each item X_i is a d -dimensional vector in *box packing*. However in this version of the problem these vectors determine a d -dimensional box, a d -dimensional solid geometric object. Furthermore the bin is now a d -dimensional unit cube. J. Csirik and G. G. Woeginger [1998] also mentioned *strip packing* which we will not discuss here.

In *dynamic* bin-packing, the service request only stays in the system for a certain time period. Hence, there are two possible types of information provided: (1) the arrival of some new service request or (2) some existing service request leaves

the system. This problem models memory allocation problems in paged computer systems, for example, [Coffman, 1983].

The bin-packing problem is a well known to be NP-hard [Garey and Johnson, 1979], and thus research has concentrated on the study and development of approximation algorithms. Of course the algorithms to solve these variants are not mutually exclusive, as each may use techniques from other areas.

This section surveys the literature on approximation algorithms for bin-packing and outlines the ideas behind the different techniques. It is assumed that all items have independent, identically distributed sizes.

2.1.1 Classical Off-line Algorithms

FIRST FIT (FF) algorithm: when assigning some new item, FF assigns it into the lowest indexed bin in which it can be accommodated. BEST FIT (BF) assigns the new item to the bin where it will fit with the smallest unused space left over. WORST FIT puts a new service request into the non-empty bin with the largest unused space left over, starting a new bin only if it will not fit into a partially-filled bin. Kou and Markowsky [1977] call an approximation algorithm for multidimensional vector packing “reasonable” if it never yields a packing in which the contents of two non-empty bins could be combined into a single bin. Note that the above three algorithms are all reasonable.

Figure 2.1, Figure 2.2 and Figure 2.3 show examples for the First Fit algorithm, Best Fit algorithm and Worst Fit algorithm respectively. X_i represents the size of the i^{th} item. Here $L_{10}=(X_1, X_2, \dots, X_{10})=(0.2, 0.9, 0.7, 0.4, 0.3, 0.1, 0.8, 0.5, 0.2, 0.4)$ and five bins are given.

2.1.2 The Harmonic Algorithm and its Variants

Lee and Lee [1985] introduced the HARMONIC (H_k) algorithm which is based on a partitioning of the interval $(0,1]$ into k subintervals, where the partitioning points are $1/2, 1/3, \dots, 1/k$. H_k partitions the items into subintervals $(1/2, 1], (1/3, 1/2], \dots, (1/k, 1/(k-1)], (0, 1/k]$. To each of these subintervals there always corresponds a set of bins, and only items belonging to a subinterval are packed into the corresponding bin. If a new item arrives that does not fit into its corresponding active bin, a new bin is activated. Note that when a bin that corresponds to subinterval $(0, 1/k]$ is full, it has at least k items. Any full bins that correspond to subinterval $(1/(k+1), 1/k]$ have exactly K items.

Figure 2.4 shows an example of the Harmonic H_3 algorithm. X_i represents the size of the i^{th} item. Here $L_9 = (X_1, X_2, \dots, X_9) = (0.2, 0.9, 0.7, 0.4, 0.3, 0.1, 0.8, 0.5, 0.2)$ and five bins are given. It partitions L_9 into $(1/2, 1], (1/3, 1/2]$ and $(0, 1/3]$. X_1, X_5, X_6 and X_9 belong to the subinterval $(0, 1/3]$. Therefore, they are packed together in the same bin. Similarly, X_4 and X_8 are packed together into B_4 . X_2, X_3 and X_7 belong to the subinterval $(1/2, 1]$ and are packed alone in B_2, B_3 and B_5 respectively. Ramanan et al. [1989], developed the MODIFIED HARMONIC algorithms, which added the possibility of packing still smaller items of size in $(1/2, 1/2+y]$, and a consequently more complicated algorithmic structure.

Richey's HARMONIC+1 algorithm [Richey, 1991] is substantially more complicated than its predecessors, with a significantly more complex interval structure involving over 70 intervals and with a packing rule that itself depends on an almost full-page table of constants.

The SIMPLIFIED HARMONIC algorithm, which is a variant of the original HARMONIC scheme introduced by [Woeginger, 1993], relies on a more complicated interval structure.

The first online harmonic algorithm for multidimensional bin-packing is due to Coppersmith and Raghavan [1989]. The main idea of this algorithm is to first

round up the largest dimension of a newly arrived item to some number in the set $\{2^{-k}/3, 2^{-k} \mid k \geq 0\}$ and then to pack it according to a classification scheme, in a similar way as the HARMONIC algorithm. As a next result, Csirik and van Vliet [1993] combined a rounding procedure that is more complex than that of Coppersmith and Raghavan with a clever generalization of the harmonic algorithm to the multidimensional case. Li and Cheng [1992] proposed a class of approximation algorithms called level-strip algorithms. All of the above online multidimensional harmonic algorithms are used to solve box packing in particular.

2.1.3 Online Matching Algorithm

There is another approach developed by Hoffman [1982]. As in [Coffman et al., 1997], we illustrate Hoffman's approach using a variant studied by Coffman and Lueker [1991]. The approach actually involves yet another indexed sequence of algorithms, called ONLINEMATCH (OLM_k) algorithms and which are closely related to the INTERVAL FIRST FIT algorithms of Csirik et al. [1986]. OLM_k partitions items into subintervals $(^{(k-1)}/_k, 1], (^{(k-2)}/_k, ^{(k-1)}/_k], \dots, (^{1}/_k, ^{2}/_k], (0, ^{1}/_k]$. Call s and s' *companions* if $size(s) + size(s') \leq 1$. Items belonging to subinterval $(^{K}/_k, ^{K+1}/_k]$ and $(^{K-2}/_k, ^{K-1}/_k]$, $1 < K < k-1$, are companions if they can fit together in a single bin. Companions are not defined for an item belonging to subinterval $(^{(k-1)}/_k, 1]$ and any two items belonging to subinterval $(^{1}/_2, ^{1}/_2 + ^{1}/_k]$, are companions. Algorithm OLM_k maintains two initially empty sets of bins; one set is *partial match* bins, and the

other is a set of bins that contains only items of size $\leq \frac{1}{2}$. The algorithm packs the next item s according to the following two-part rule:

1. If $\text{size}(s) \geq \frac{1}{2}$, assign s to an empty bin and add the bin to the partial match set.
2. If $\text{size}(s) < \frac{1}{2}$ and there exists a bin B in the partial match set that contains only a companion of s , assign s in B along with its companion. If no such bin exists, assign s into the other set of bins, i.e., set of bins that contain only items of size $\leq \frac{1}{2}$.

Figure 2.5 shows an example of the ONLINEMATCH OLM_4 algorithm. X_i represents the size of the i^{th} item. Here $L_8 = (X_1, X_2, \dots, X_8) = (0.2, 0.9, 0.7, 0.4, 0.3, 0.1, 0.8, 0.2)$ and five bins are given. The subintervals are $(\frac{3}{4}, 1]$, $(\frac{2}{4}, \frac{3}{4}]$, $(\frac{1}{4}, \frac{2}{4}]$, and $(0, \frac{1}{4}]$. Items belonging to subinterval $(\frac{3}{4}, 1]$ are packed alone in a bin. Items belonging to subinterval $(\frac{2}{4}, \frac{3}{4}]$ and $(0, \frac{1}{4}]$ are companions and are put together in the same bin. Items belonging to subinterval $(\frac{1}{4}, \frac{2}{4}]$ are packed into the same bin.

X_1 and X_3 are companions and are assigned into the same bin B_1 . Similarly, X_4 and X_5 belong to the subinterval $(\frac{1}{4}, \frac{2}{4}]$ and hence they are companions and are packed together in B_3 . There are no bins in the partial match set that contain a companion of X_6 only. Therefore, it is packed into empty bin B_4 and B_4 is added to set of bins that contain only items of size $\leq \frac{1}{2}$. X_8 is packed into B_4 , since there are no bins in the partial match set that contain companions of X_8 alone. Both X_2 and X_7 are packed into empty bins, since they belong to the subinterval $(\frac{3}{4}, 1]$.

Chang et al. [1993] defined a HashPacking (H_k) algorithm for d -dimensional box packing, which assumes that the parameters of each item are uniformly distributed in $(0,1]^d$. It uses an idea very similar to Hoffman's. Items are classified into two types, namely, "big" and "small": "Big" if it is greater than $1/2$ and "small", otherwise. Furthermore, a function h depending on m is defined as $h_m(x) = \lfloor 2yk \rfloor$,

where x is an item's size and $y=(1-x)$ if $x>1/2$ and $y=x$ otherwise. By defining $h_m(x)$, H_k partitions the interval $(0,1]$ into $2k$ subintervals instead of k subintervals as in OLM_k , where the partitioning points are $^{2k-1}/_{2k}, ^{2k-2}/_{2k}, \dots, ^1/_ {2k}$. Note that for a big item t_B and a small item t_S , if $h_m(t_B) = h_m(t_S)$, then they can be assigned in the same bin.

2.2 Review of DSP Multiprocessor Systems

Many researchers have looked at various aspects of DSP multiprocessor scheduling (e.g. Parhi and Messerschmitt [1991], Lee and Messerschmitt [1987]). We address a simpler subset of this problem dealing with the use of DSP multiprocessor system in VoIP gateways. Typical processors are from the Texas Instruments TMS320C54X generation. For instance, the TMS320C541 processing capacity is 64 Kwords for data memory, 64 Kwords for program memory and 40 MIPS processing capability.

A scheduler is responsible for allocating a service request to one of the given DSP processors. A service request is a DSP algorithm. The classical DSP algorithm requirements for a VoIP gateway DSP engine are:

- Vcoders: G.711, G.723.1, G.729a, G.726/727, G.722.
- Fax: T.38.

G.711, G.723.1, G.729a, G.726/727 and G.722 are international telecommunication union (ITU) standards for speech codecs (coder/decoders). The G.711 format has been the standard for digitizing voice by the telephone companies

starting in the 1960s. It operates at a bit rate of 64 Kbps (Kilo bytes per second). However, the newer algorithms, G.723.1, G.729a and G.726/727, have lowered the bit rate considerably, and respectable quality can be obtained at 16 Kbps and well below that, depending on the quality of all the components in the system. G.723.1 provides toll quality audio at 5.3 or 6.3 Kbps. The higher rate requires less processing power. G.729.A provides toll quality audio at 8 Kbps and is designed for simultaneous voice and data. G.727 operates at 40, 32, 24 or 16 Kbps depending on available channel bandwidth. G.722 is a wideband vocoder operating at 64, 56 or 48 Kbps. All of the above DSP algorithm bit rates are obtained from Minoli and Minoli [1998].

T.38 is an ITU standard for real-time fax over IP. It is crucial to build the new world of real-time fax over IP. Without such a standard, fax machines from different vendors would not be able to talk to each other.

Each DSP algorithm requires specific processing resources, namely, data memory, program memory and processor speed in MIPS. Different DSP algorithms will have different specific processing requirements. For instance, Virata's vocoder DSP algorithm processor resource requirements are as follow [GlobespanVirata, 2001]:

- G.711 requires 7.2 Kwords (Kilo words) program memory, 116 words data memory and 0.2 MIPS.
- G.726 requires 3.6 Kwords program memory, 58 words data memory and 4.5 MIPS.

- G.729A requires 10.1 Kwords program memory, 571 words data memory and 8.9 MIPS.
- G.723.1 requires 18 Kwords program memory, 400 words data memory and 17 MIPS.

CHAPTER 3

State of the Art in SRTPA Algorithms

DSP multiprocessor systems used in VoIP gateways nowadays are configured such that each DSP processor can accommodate multiple service instances of the same DSP algorithm. Multiple service instances of different DSP algorithms cannot be assigned to the same DSP processor. For instance, consider TMS320C541 processor and Virata's vocoder DSP algorithms that are discussed in section 2.2. The maximum number of instances that can be allocated in a TMS320C541 processor are 200, 8, 4 and 2 for G.711, G.726, G.729A and G.723.1 respectively.

There are two common scheduling algorithms that are used currently, namely, "Fixed Service Single Channel" (FSSC) and "Fixed Service Multiple Channel" (FSMC). The FSSC algorithm is based on dedicated hardware. However, the FSMC algorithm is the first algorithm for programmable hardware.

In FSSC, each DSP processor is pre-assigned to support one specific DSP algorithm only. For instance, consider a DSP multiprocessor system that has ten TMS320C541 processors, $P_1, P_2, \dots, P_{10}$. P_1 and P_2 are assigned to support G.711. P_3, P_4 and P_5 are assigned to support G.726. P_6 and P_7 are assigned to support

G.729A. Finally, P_8 , P_9 and P_{10} are assigned to support G.723.1. Each incoming service instance of type d , has to be assigned to one of its designated processors. For example, a service instance of type G.729A has to be assigned to either P_6 or P_7 . If both of P_6 and P_7 can no longer accommodate G.729A, then the next incoming service instance of type G.729A will be discarded, in other words, blockage occurs.

Unlike FSSC, a DSP processor is not pre-determined in FSMC. A DSP processor can support any DSP algorithm as long as all instances within one processor are of the same type. If the first service instance of a type d is allocated in DSP processor P , then P cannot accommodate any other service type other than d . However, once all service instances of type d in P have terminated (i.e. P becomes empty), any DSP algorithm can be assigned to it. Each incoming service instance of type d is assigned to a DSP processor that has a service instance of type d in it. If no such processor exists, then it is assigned to an empty DSP processor. A service instance of type d is discarded if the processors handling type d are full and there are no empty processors.

CHAPTER 4

The SRTPA Problem

The following three sections give a clear and concise statement of the service-request-to-processor assignment (SRTPA) problem, further analyze the currently available approaches to this problem and close the chapter with a discussion of the importance of this type of scheduling problem.

4.1 Thesis Problem Statement

The goal of this thesis is to develop a service-request-to-processor assignment algorithm which minimizes the blockage rate. The problem is stated as follows:

Given the following:

1. *n identical DSP programmable processors with specific data memory capacity, program memory capacity and MIPS (Million Instructions Per Second) processing capability;*
2. *a set of service types with specific processing resource requirements, namely, data memory requirements, program memory requirements,*

and MIPS processing requirements. Each instance of a service type has identical processing requirements.

Derive an online algorithm that assigns each incoming service request to a DSP processor so that the blockage rate is minimized.

Each new incoming call represents a service request to the scheduler, which must allocate the request to one of the n DSP processors in the multiprocessor system. The incoming service requests are limited to a given set of service types. Once the call is terminated, the scheduler is also responsible for de-allocating the task from the DSP processor.

Any service instance can be allocated to any DSP processor as long as it satisfies constraints imposed by each of the identical DSP processors in the architecture: the total MIPS requirement for all of the service instances assigned to any DSP processor cannot exceed the MIPS processing capability for the DSP processor, and there is a limit on the total program memory and a separate limit on the total data memory. Again, the sum of the program memory for all the services assigned to any DSP processor must not exceed the total program memory available for the DSP processor. A similar condition applies to the data memory. When multiple instances of the same service type are allocated to the same DSP processor then a single program memory allocation is sufficient.

4.2 Analysis of the State of the Art In SRTPA

The problem that is outlined in Section 4.1 is not handled in an efficient way by the FFSC and FSMC algorithms that are discussed in chapter 3. Both algorithms handle allocation such that each DSP processor handles only one kind of DSP algorithm. FSSC assigns each incoming service instance of type d to one of the DSP processors that is designated for type d . FSMC assigns each incoming service instance of type d to a DSP processor that has a service instance of type d in it or to an empty processor. If all DSP processors that handle a particular service type are full, the next incoming service request of the same type has to be discarded even if other DSP processors that handle other service types are not full.

Because the current state of the art only does not handle service allocation of the problem outlined in an efficient way, we saw a need to develop an improved method that will provide better SRTPA results under the conditions outlined in section 4.1.

Furthermore as Garey and Johnson [1979] pointed out, the bin-packing problem is NP-hard. Thus, the true optimum cannot be found in practice, so research must concentrate on the study and development of approximation algorithms. Because we can allocate a single program memory to a specific DSP processor for multiple instances of the same service type, the SRTPA problem is even more difficult than the standard bin-packing problem.

4.3 Importance of SRTPA

Each DSP processor that is used in current VoIP gateways supports one specific DSP algorithm. However, the more recent generation of DSP processor that is used in VoIP gateways is more powerful and can support more than one DSP algorithm. While more powerful DSP processors have been incorporated into internet voice gateways, no similar systematic approach has been developed to facilitate the analysis of scheduling and service request allocation in DSP multiprocessor systems.

If the program memory for each of the service instances assigned to a specific DSP processor has to be allocated, then it is identical to the 3-dimensional bin-packing problem. The ability to allocate only a single copy of the program memory to a specific DSP processor for multiple instances of the same service type makes the SRTPA problem different from the standard bin-packing problem and of greater theoretical interest.

Telecommunication companies generate more revenue by handling a greater volume of service requests. By using an efficient method to assign service requests to DSP processors, the system is able to handle more service requests without increasing the number of DSP processors. In other words, revenue will be increased without expanding the existing system. Therefore, there is great interest in the telecommunication industry in research on improved SRTPA algorithms.

Hence, developing methods to assign a service requests to processors in an efficient way is a worthwhile undertaking.

CHAPTER 5

Algorithms for the SRTPA Problem

It is useful to classify processors into one of three categories: empty, active and closed. An *empty* processor is one into which no service requests have been placed. After a service request has been assigned into a processor, we consider it *active* if it is still available for assigning of additional service requests, and *closed* if it is not.

An example of possible service types that can be handled by the system is given in Figure 5.1. The algorithms assume that the system has the attributes indicated below:

- *ServiceList*: A list of the service types that can be handled in the system
- *TotalServiceTypes*: Total number of service types in *ServiceList*
- *TotalProcessors*: Total number of processors
- Service Attributes: All requirements discussed below are in percent of maximum capacity.
 - *Req(i)*: A vector of the processing requirements of service type i . The three elements are *pmem*, the program memory requirement, *dmem*, the data memory requirement, and *mips*, the MIPS requirement.
 - *DataMipsReq(i)*: A vector of the data memory requirement and MIPS requirement of service type i . It has two elements only, namely, *dmem*,

the data memory requirement, and *mips*, the MIPS requirement of service type *i*.

- *AssignedTo(i)*: A list of processors that have service type *i* in them.
- Processor Attributes: All requirements discussed below are in percent of maximum capacity.
- *Load(p)*: A vector of the total processing requirements of all service instances assigned to processor *p*. The three elements are *pmem*, the total program memory requirement, *dmem*, the total data memory requirement, and *mips*, the total MIPS requirement.
- *DataMipsLoad(p)*: A vector of the total data memory requirement and MIPS requirement of all service instances assigned to processor *p*. It contains two elements, namely, *dmem*, the total data memory requirement and *mips*, the total MIPS requirement.
- *ProcServCapacity(p,i)*: The total processing requirement of all service instances in *p* along with a single instance of type *i*. It has three elements, namely *pmem*, the total program memory requirement, *dmem*, the total data memory requirement, and *mips*, the total MIPS requirement. It is equal to $\text{Load}(p) + \text{Req}(i)$, if *p* does not exist in *AssignedTo(i)*. Otherwise the *pmem*, *dmem* and *mips* elements in *ProcServCapacity(p,i)* are equal to $\text{Load}(p).\text{pmem}$, $\text{Load}(p).\text{dmem} + \text{Req}(i).\text{dmem}$ and $\text{Load}(p).\text{mips} + \text{Req}(i).\text{mips}$ respectively.

We refer to an individual element of a vector by using the dot notation, e.g. the MIPS requirement element of the *Req(i)* vector is shown as *Req(i).mips*. Vector

norm bars indicate the largest element in the vector, e.g. $|\text{Req}(i)| = 85$ where $\text{Req}(i) = [60, 25, 85]$.

Requirement	Type 1	Type 2	Type 3	Type 4	Type 5
pmem	45	70	50	79	20
dmem	25	6	31	18	33
mips	35	45	36	25	5

pmem : program memory requirement
dmem : data memory requirement
mips : mips processing requirement
(in % of processor resources)

Figure 5.1: List of Service Types in EX1

The symbols $\leq$ or $\geq$ indicate that all elements in the vector are less than or equal to, or greater than or equal to, e.g. $\text{Req}(i) + \text{Req}(j) \leq 100$, indicates $\text{Req}(i).\text{mips} + \text{Req}(j).\text{mips} \leq 100$, $\text{Req}(i).\text{dmem} + \text{Req}(j).\text{dmem} \leq 100$ and $\text{Req}(i).\text{pmem} + \text{Req}(j).\text{pmem} \leq 100$.

When a service instance s of type i terminates in processor p or s is assigned to p , $\text{AssignedTo}(i)$ and $\text{Load}(p)$ will be updated. If s leaves the system and s is the only service instance of type i in processor p , then p is removed from $\text{AssignedTo}(i)$ list. However, p remains in $\text{AssignedTo}(i)$ list if another service of type i remains in p after s terminates in p . As discussed in Section 4.1, one copy of program memory is allocated to a processor for multiple instances of the same service type in a

specific processor. If s is the only service instance of type i in p , then each element in $\text{Load}(p)$ will be reduced by the corresponding elements in $\text{Req}(i)$ after s terminates its process in p or it will be incremented after s is allocated to p . On the other hand, if p has a service instance of type i other than s in it, then $\text{DataMipsLoad}(p)$ will be reduced or incremented by the corresponding elements in $\text{DataMipsReq}(i)$.

Because allocating a single copy of program memory for multiple instances of the same service type in a specific processor is sufficient, we observe that if the sum of the program memory requirements for all service types in ServiceList is less than or equal to 100, then the SRTPA problem is exactly a two-dimensional bin-packing problem. Before allocating any service request s to a processor p , the program memory requirements of all service instances in p does not need to be considered, since the sum of the program memory requirements for all service instances in p will always be less than or equal to 100. Before the system processes any service request s , it checks whether the program memory requirement for s can be neglected.

5.1 Classical/Reasonable Algorithms

The simplest methods for solving the SRTPA problem are the classical First-Fit, Best-Fit and Worst-Fit bin-packing algorithms. Section 5.1.1, Section 5.1.2 and Section 5.1.3 give details on how slight variations of all three algorithms which can handle the shared program memory are used to solve the SRTPA problem.

Modified-First-Fit, Modified-Best-Fit and Modified-Worst-Fit which are similar to First-Fit, Best-Fit and Worst-Fit respectively are discussed in Section 5.1.4.

5.1.1 FIRST-FIT Algorithm (FFA)

The FFA assigns each incoming service request s of type i in the following manner:

Find the lowest p which can accommodate s . If p exists in $\text{AssignedTo}(i)$ (i.e. there is a service instance of type i in p) then check whether $\text{DataMipsLoad}(p) + \text{DataMipsReq}(i) \leq 100$, and otherwise check $\text{Load}(p) + \text{Req}(i) \leq 100$. If no p can accommodate s then blockage occurs, so discard s . Algorithm details follow.

GIVEN: service request s of type i

1. For each processor p
 - 1.1 If $\text{ProcServCapacity}(p, i) \leq 100$
 - 1.1.1 Assign s into processor p and exit
2. Blockage occurs and exit .

This algorithm is illustrated in Figure 5.2 by application to the EX1 list of service types shown in Figure 5.1. The first service instance s_{31} is assigned to the first processor P_1 . $\text{Load}(P_1)=[50,31,36]$ after s_{31} is assigned. P_1 is the first processor that can accommodate s_{52} , therefore it is allocated to P_1 . $\text{Load}(P_1)$ is now equal to $[70,64,41]$. s_{43} is assigned to empty processor P_2 which is the first processor that can accommodate a service instance of type 4. $\text{Load}(P_2)$ is now updated as $[79,18,25]$. Neither of the non-empty processors P_1 nor P_2 can accommodate service type 2;

therefore, s_{24} has to be allocated to empty processor P_3 . $\text{Load}(P_3)$ is now equal to $[70,6,45]$. All of the processors, P_1 , P_2 and P_3 , can accommodate s_{55} . However, by the FFA rules, it will be allocated to the first active processor, which is P_1 . $\text{Load}(P_1)$ is updated to $[70,97,46]$. $\text{Load}(P_1).\text{pgm}$ is not updated, since one copy of program memory for service type 5 is sufficient. None of the processors can accommodate s_{16} . Hence blockage occurs and it has to be discarded.

The following assumptions are made:

- Incoming service requests: $(s_{31}, s_{52}, s_{43}, s_{24}, s_{55}, s_{16})$ where s_{ij} is j^{th} service request, and it is of type i .
- Total number of DSP processors = 3. We will have (P_1, P_2, P_3) .
- None of the above service instances expire during the allocation

Allocation for each service request to the given three DSP processors:

$P_1 : s_{31} s_{52} s_{55}$

$P_2 : s_{43}$

$P_3 : s_{24}$

s_{16} is blocked.

Figure 5.2: Illustration of FFA Processing Service Requests Using EX1

5.1.2 BEST-FIT Algorithm (BFA)

BFA assigns each incoming service request s of type i in the following manner:

Find a processor p , such that by assigning s into it, the smallest unused space will be left over. If a processor is found, assign s to it and exit. Otherwise blockage occurs and s is discarded.

We define *CurrentMaxUsed* as a vector of the maximum of the largest elements in the processing requirement among all processing requirements for all of the service instances assigned to each processor in the system. The three elements are *mips*, the total MIPS requirement, *dmem*, the total data memory requirement, and *pmem*, the total program memory requirement. The current incoming service request is assigned to *AssignedToProcessor*. Algorithm details follow.

GIVEN: service request s of type i

1. *CurrentMaxUsed.mips*=0, *CurrentMaxUsed.dmem*=0,
CurrentMaxUsed.pmem=0; *AssignedToProcessor*=0.
2. For each active processor p
 - 2.1 If $\text{ProcServCapacity}(p,i) \leq 100$ and
 $|\text{ProcServCapacity}(p,i)| > |\text{CurrentMaxUsed}|$ then
 - 2.1.1 *CurrentMaxUsed*= $\text{ProcServCapacity}(p,i)$
 - 2.1.2 *AssignedToProcessor*= p
3. If *AssignedToProcessor*>0 then assign s to *AssignedToProcessor* and exit.
4. If there is an empty processor p then assign s to p and exit.
5. Blockage occurs and exit.

Figure 5.3 illustrates the BFA by application to the EX1 list of service types of Figure 5.1. The first service instance s_{31} is assigned to the first processor P_1 . After s_{31} is assigned, $\text{Load}(P_1)=[50,31,36]$. Obviously, by assigning s_{52} to P_1 , it will have

the smallest unused space left over. $\text{Load}(P_1)$ is now updated to $[70,64,41]$. The first processor that can accommodate s_{43} is an empty processor, P_2 . Therefore s_{43} is assigned to it. $\text{Load}(P_2)$ is now equal to $[79,18,25]$. s_{24} is assigned to an empty processor P_3 , since none of the active processors, P_1 and P_2 , can accommodate s_{24} . $\text{Load}(P_3)$ is now updated to $[70,6,45]$. All of the processors, P_1 , P_2 and P_3 , can accommodate s_{55} . However, P_2 will have the smallest unused space left over after assigning s_{55} into it. In other words, after assigning s_{55} to P_1 , P_2 or P_3 , $|\text{Load}(P_2)|$ would have the largest value compared to $|\text{Load}(P_1)|$ and $|\text{Load}(P_3)|$. Therefore, s_{55} is assigned to P_2 . None of the processors can accommodate s_{16} . Hence blockage occurs.

The following assumptions are made:

- Incoming service requests: $(s_{31}, s_{52}, s_{43}, s_{24}, s_{55}, s_{16})$ where s_{ij} is j^{th} service request, and it is of type i .
- Total number of DSP processors = 3. We will have (P_1, P_2, P_3) .
- None of the above service instances expire during the allocation

Allocation for each service request to the given three DSP processors:

$P_1 : s_{31} s_{52}$

$P_2 : s_{43} s_{55}$

$P_3 : s_{24}$

s_{16} is blocked.

Figure 5.3: Illustration of BFA Processing Service Requests Using EX1

5.1.3 WORST-FIT Algorithm (WFA)

WFA assigns each incoming service request s of type i in the following manner:

1. Find an active processor p , such that assigning s into it will leave the largest unused space left over. If p exists, assign s to it and exit.
2. If there is an empty processor p , then assign s into p and exit. Otherwise, blockage occurs.

We define *CurrentMinUsed* as a vector of the minimum of the largest elements in the processing requirements among all processing requirement for all of the service instances assigned to each processor in the system. The three elements are *mips*, the total MIPS requirement, *dmem*, the total data memory requirement, and *pmem*, the total program memory requirement. The current incoming service request is assigned to *AssignedToProcessor*. Algorithm details follow.

GIVEN: service request s of type i

1. *CurrentMinUsed.mips*=101, *CurrentMinUsed.dmem*=101, *CurrentMinUsed.pmem*=101.
2. For each active processor p
 - 2.1 If $\text{ProcServCapacity}(p,i) \leq 100$ and $|\text{ProcServCapacity}(p,i)| < |\text{CurrentMinUsed}|$ then
 - 2.1.1 *CurrentMinUsed*=*ProcServCapacity*(p,i)
 - 2.1.2 *AssignedToProcessor*= p
3. If *AssignedToProcessor* > 0 then assign s to *AssignedToProcessor* and exit.
4. If there is an empty processor p then assign s to p , exit.
5. Blockage occurs, exit.

Figure 5.4 illustrates WFA by application to the EX1 list of service types of Figure 5.1. The first service instance s_{31} is assigned to the first processor P_1 and

The following assumptions are made:

- Incoming service requests: $(s_{31}, s_{52}, s_{43}, s_{24}, s_{55}, s_{16})$ where s_{ij} is j^{th} service request, and it is of type i .
- None of the above service instances expire during the allocation
- Total number of DSP processors = 3. We will have (P_1, P_2, P_3) .

Allocation for each service request to the given three DSP processors:

$P_1 : s_{31} s_{52}$

$P_2 : s_{43}$

$P_3 : s_{24} s_{55}$

s_{16} is blocked.

Figure 5.4: Illustration of WFA Processing Service Requests Using EX1

Load(P_1) is updated to [50,31,36]. P_1 is the only active processor that can accommodate s_{52} . Hence it is allocated to P_1 . Load(P_1) is now updated to [70,64,41]. The only active processor P_1 cannot accommodate s_{43} . Therefore, it is assigned to an empty processor P_2 . Load(P_2) is now equal to [79,18,25]. s_{24} is assigned to an empty processor P_3 , since none of the active processors, P_1 and P_2 , can accommodate s_{24} . Load(P_3) is now updated to [70,6,45]. All of the active processors, P_1 , P_2 and P_3 , can accommodate s_{55} . Checking the assignment of s_{55} into each active processor, P_1 , P_2 and P_3 , the Load variable for each processor would be updated as follows: Load(P_1)=[70,97,46], Load(P_2)=[99,51,30], Load(P_3)=[90,39,50]. | Load(P_1) |,

$|\text{Load}(P_2)|$ and $|\text{Load}(P_3)|$ are equal to 97, 99 and 90 respectively. Since $|\text{Load}(P_3)|$ is the smallest value (i.e. P_3 has the largest unused space left over), s_{55} is assigned to P_3 . None of the active processors can accommodate s_{16} and there is no empty processor. Hence blockage occurs.

5.1.4 Modified-First-Fit, Modified-Best-Fit and Modified-Worst-Fit Algorithm (MFFA, MBFA and MWFA)

MFFA, MBFA and MWFA are similar to FFA, BFA and WFA respectively. They have been modified so that they assign the same service types together to the same processor, before using the corresponding basic method. For instance, if there is a processor p that can accommodate a service request s of type i and p has a service instance of type i in it, then MFFA/MBFA/MWFA will assign s to p . Otherwise, FFA/BFA/WFA will be used to allocate s to a processor.

MFFA assigns each incoming service request s of type i in the following manner:

1. Find the lowest p in $\text{AssignedTo}(i)$ such that $\text{Load}(p) + \text{Req}(i) \leq 100$. If such p exists, then assign s into p and exit.
2. Use FFA to allocate s to a processor.

MBFA and MWFA assign each incoming service request s of type i in a manner similar to MFFA except MBFA assigns s to a processor p that leaves the least space

leftover and MWFA such that there is the most space leftover. BFA or WFA is used if the first procedure fails.

The following assumptions are made:

- Incoming service requests: $(s_{31}, s_{52}, s_{43}, s_{24}, s_{55}, s_{16})$ where s_{ij} is j^{th} service request, and it is of type i .
- None of the above service instances expire during the allocation
- Total number of DSP processors = 3. We will have (P_1, P_2, P_3) .

Allocation for each service request to the given three DSP processors:

$P_1 : s_{31} s_{52} s_{55}$

$P_2 : s_{43}$

$P_3 : s_{24}$

s_{16} is blocked.

Figure 5.5: Illustration of MBFA Processing Service Requests Using EX1

Figure 5.5 illustrates MBFA by application to the EX1 list of service types of Figure 5.1. The first four service instances, s_{31} , s_{52} , s_{43} and s_{24} , are assigned the same way as BFA. The Load variables for P_1 , P_2 and P_3 are updated as $[70,64,41]$, $[79,18,25]$ and $[70,6,45]$ respectively. AssignedTo variables for each service type are also updated as follows: $\text{AssignedTo}(2)=\{P_3\}$, $\text{AssignedTo}(3)=\{P_1\}$, $\text{AssignedTo}(4)=\{P_2\}$ and $\text{AssignedTo}(5)=\{P_1\}$. There is a processor P_1 that belongs

to AssignedTo(5) and can accommodate a new service instance of type 5, hence, s_{55} is assigned to it. AssignedTo(1) is empty. Therefore, s_{16} has to be allocated using BFA. However none of the processors can accommodate service instance of type 1. Thus, blockage occurs.

5.2 Modified Harmonic Algorithm (MHA) and Online Matching Algorithm (OMA)

The new MHA and OMA algorithms will be described in this section. There are three main parts in both of the algorithms: (1) preprocessing, (2) processing each incoming service request, and (3) handling termination of each service instance. In preprocessing, service types will be divided into a number of classes using an idea similar to the classical harmonic and online matching algorithms.

5.2.1 Preprocessing

The two methods developed to classify service types into a number of classes will be described in Section 5.2.1.1 and Section 5.2.1.2. Each section gives details on how to partition service types into a number of classes using ideas similar to the classical one-dimensional harmonic algorithm by Lee and Lee [1985] and HashPacking by Chang et al. [1993], respectively.

The ServiceList is sorted based on the program memory requirements, data memory requirements and MIPS processing requirements to yield *pmemLists*, *dmemLists* and *mipLists* respectively. $mipLists(j)=k$ is the j^{th} element in *mipLists* and it is of service type k . A similar definition holds for *pmemLists(j)* and *dmemLists(j)*.

We will assume that the program memory requirement cannot be neglected from now on. We could easily modify the preprocessing procedure below by replacing *Load(p)* and *Req(i)* with *DataMipsLoad(p)* and *DataMipsReq(i)* respectively, if the program memory requirement could be neglected.

5.2.1.1 MHA Preprocessing

As mentioned in section 2.1.2 with regard to Lee and Lee's [1985], Ramanan et al.'s [1989], Richey's [1991] and Woeginger's [1993] approaches, the main idea is to divide items into a number of intervals and to assign items that belong to the same subinterval to the same bin. All of the surveys mentioned above assume that all items have independent, identically distributed sizes. Any item is assumed to be of size between zero and one (not including zero) and bins are assumed to be of unit capacity. Furthermore, the number of bins is unlimited. MHA has been modified so that it can handle service instance processing requirements when they are drawn from a finite set instead of having independent identically distributed values.

If k is equal to four in the H_k algorithm by Lee and Lee [1985], then items are classified into four categories, namely items belonging to the intervals $(0, \frac{1}{4}]$, $(\frac{1}{4}, \frac{1}{3}]$, $(\frac{1}{3}, \frac{1}{2}]$ and $(\frac{1}{2}, 1]$. Consider the generalization of the one-dimensional case to three-dimensional, then items are classified into sixty-four (4^3) categories. The processing requirement of a service request in the SRTPA problem is chosen from a finite set rather than having an independent identically distributed size. Therefore, we can classify service types into a number of classes before handling any service request. Service instances of the same class will be assigned together in the same processor. There is only a limited number of different processing requirements, since incoming service requests are limited to a given set of service types. Therefore, it is more than likely that each class has just one member service type only. Different service types are unlikely to belong to the same class. In other words, service instances assigned together in the same processor are only of the same type. The number of processors is limited. Hence, for each service request s of type i , if none of the processors has service type i in it, then s will be allocated to any processor that can accommodate it. This method of allocation yields the same result as MFFA, MBFA, MWFA, depending on which algorithm is used after the first procedure fails.

The service requirements for each service type mentioned in Section 1.1 are pre-determined. Before allocating any service request, the system already knows the processing requirement for each service type. Therefore we can use the information above to classify the service types. The partitioning point can be arbitrary instead of incrementing the denominator value by 1 (i.e the usual partitioning point are $\frac{1}{2}$, $\frac{1}{3}$,

..., $1/k$) as in H_k . The partitioning point here can be determined by the processing requirements for the service types. Consider classification of service types based on pmemList . The first partitioning point will be $\text{Req}(\text{pmemList}(n_1))$, such that a single service instance of all service types of the first n_1^{th} elements in pmemList can be assigned together in the same processor, but adding service type $\text{pmemList}(n_1+1)$ to the processor will violate the processor capacity constraint. The first class contains $\text{pmemList}(1), \dots, \text{pmemList}(n_1)$. The next partitioning point, $\text{Req}(\text{pmemList}(n_2))$, is determined by finding n_2 where n_2 is the largest integer such that all of the service types $\text{pmemList}(n_1+1), \text{pmemList}(n_1+2), \dots, \text{pmemList}(n_2)$ can fit in one processor. $\text{pmemList}(n_1+1), \text{pmemList}(n_1+2), \dots, \text{pmemList}(n_2)$ belong to the second class. The classification of the rest of the service types continues in this manner, until the last member of pmemList is reached. To each of these classes c there always corresponds a set of processors $Bclass(c)$, and only service types belonging to class c are assigned to one of the processors in $Bclass(c)$.

The above service types classification could have a single class covering all service types, if a single service instance of all service types can be assigned to the same processor. In other words, all active processors would be in $Bclass(c)$. Therefore to allocate a service request s of type i , the first processor among all active processors that can accommodate s is used. This is exactly the FFA that is discussed in Section 5.1.1. In the above classification, service types are assigned to the same class if the total processing requirement is less than or equal to 100. We define *MaxCapacityInClass* as the maximum value of all processing requirement elements of all service types in ServiceList , i.e.

$\max\{ \lfloor \text{Req}(1) \rfloor, \lfloor \text{Req}(2) \rfloor, \dots, \lfloor \text{Req}(\text{TotalServiceTypes}) \rfloor \}$. If a single processor can accommodate a single instance of all service types, then *MaxCapacityInClass* will be the capacity limit within a class, instead of 100.

Based on each *mipsList*, *dmemList* and *pmemList*, divide the service types into a number of class so that a single service instance of all service types in the same class can be assigned to the same processor without exceeding *MaxCapacityInClass*, i.e. for any class j , $\left\lfloor \sum_{\text{all } k} \text{Req}(k) \right\rfloor \leq \text{MaxCapacityInClass}$, where $k \in$ service type in class j . If the program memory requirement can be neglected, then *DataMipsReq*(k) will be used instead of *Req*(k).

Partition each of the *MLists*, where $Me\{mips, dmem, pmem\}$ into the following classes:

- $(0, \text{Req}(\text{MLists}(j)).M]$, where j is the largest integer such that $\left\lfloor \sum_{x=1}^j \text{Req}(\text{MLists}(x)) \right\rfloor \leq \text{MaxCapacityInClass}$.
- $(\text{Req}(\text{MLists}(j)).M, \text{Req}(\text{MLists}(k)).M]$, where k is the largest integer such that $\left\lfloor \sum_{x=j+1}^k \text{Req}(\text{MLists}(x)) \right\rfloor \leq \text{MaxCapacityInClass}$.
- Continue in this manner until the end of *MLists* is reached
- The last interval is as follows: $(\text{Req}(\text{MLists}(m)).M, \text{Req}(\text{MLists}(n)).M]$, where n is the total number service types in *ServiceList* such that $\left\lfloor \sum_{x=m+1}^n \text{Req}(\text{MLists}(x)) \right\rfloor \leq \text{MaxCapacityInClass}$.

Any service requests of the same class are assigned to the same processor, if possible. The details of assigning each incoming service request to a processor are discussed in Section 5.2.2. Preliminary experiments showed that the system achieves

a lower blockage rate using the partitioning with a smaller total number of classes. Under the same set of incoming service requests, fewer service requests are processed before the first blockage occurs when the partitioning with the larger total number of classes is used, in other words the system has a higher blockage rate. Therefore, MHA uses the service types classification based on whichever of mipsList, dmemList or pmemList has the minimum total number of classes. Algorithm details follow.

GIVEN: $max_num_class=0$; mipsList, dmemList and pmemList

1. For each MLists, where $M \in \{mips, dmem, pmem\}$:
 - 1.1 Initialize $index=1$, $current_class=1$ and $current_class_member = \{MLists(1)\}$, $MClass=\emptyset$, $MnumClass=0$.
 - 1.2 If $index < TotalServiceTypes$ then
 - 1.2.1 If $\left| Req(MLists(index+1)) \right| + \left| \sum_{all\ i} Req(i) \right| \leq$
 $MaxCapacityInClass$, where $i \in current_class_member$ then
 - 1.2.1.1 Add $MLists(index+1)$ to $current_class_member$
 - 1.2.1.2 $index=index+1$
 - 1.2.1.3 Go to step 1.2
 - 1.2.2 Else
 - 1.2.2.1 $MClass(current_class)=current_class_member$
 - 1.2.2.2 $current_class=current_class+1$
 - 1.2.2.3 $current_class_member=\{MLists(index+1)\}$
 - 1.2.2.4 $index=index+1$
 - 1.2.2.5 Go to step 1.2
 - 1.3 Else
 - 1.3.1 $MNumClass$ is equal to the number of classes in $MClass$.

RESULT: *ClassUsed*, the classification of service types in MHA, where *ClassUsed* is equal to whichever of mipsClass, dmemClass or pmemClass has the smallest total

number of classes among the three, i.e. $\min\{\text{mipsNumClass}, \text{dmemNumClass}, \text{pmemNumClass}\}$. If mipsClass, dmemClass and pmemClass have the same total number of classes then choose arbitrarily.

Figure 5.6 illustrates MHA preprocessing by application to the EX1 list of service types in Figure 5.1. The number of classes in pmemClass, dmemClass and mipsClass is four, four and three classes respectively. Hence, ClassUsed is equal to dmemClass, since it has the smallest number of classes in it.

Processing *pmemList*={5, 1, 3, 2, 4}

- Step 1.2.1.1-1.2.1.3 for *current_class*=1, *current_class_member*={5, 1}
- Step 1.2.2.1: *pmemClass*(1)={5,1}
- Step 1.2.1.1-1.2.1.3 for *current_class*=2, *current_class_member*={3}
- Step 1.2.2.1: *pmemClass*(2)={3}
- Step 1.2.1.1-1.2.1.3 for *current_class*=3, *current_class_member*={2}
- Step 1.2.2.1: *pmemClass*(3)={2}
- Step 1.2.1.1-1.2.1.3 for *current_class*=4, *current_class_member*={4}
- Step 1.2.2.1: *pmemClass*(4)={4}
- Step 1.3: *pmemNumClass*=4
- *pmemClass*=[{5, 1}; {3}; {2}; {4}]

Processing *dmemList*={2, 4, 1, 3, 5}

- Step 1.2.1.1-1.2.1.3 for *current_class*=1, *current_class_member*={2}
- Step 1.2.2.1: *dmemClass*(1)={2}
- Step 1.2.1.1-1.2.1.3 for *current_class*=2, *current_class_member*={4}
- Step 1.2.2.1: *dmemClass*(2)={4}
- Step 1.2.1.1-1.2.1.3 for *current_class*=3, *current_class_member*={1,3}
- Step 1.2.2.1: *dmemClass*(3)={1,3}
- Step 1.2.1.1-1.2.1.3 for *current_class*=4, *current_class_member*={5}
- Step 1.2.2.1: *dmemClass*(4)={5}
- Step 1.3: *dmemNumClass*=4
- *dmemClass*=[{2}; {4}; {1,3}; {5}]

Processing *mipsList*={5, 4, 1, 3, 2}

- Step 1.2.1.1-1.2.1.3 for *current class*=1, *current_class_member*={5, 4}
- Step 1.2.2.1: *mipsClass*(1)={5,4}
- Step 1.2.1.1-1.2.1.3 for *current_class*=2, *current_class_member*={1, 3}
- Step 1.2.2.1: *mipsClass*(2)={1,3}
- Step 1.2.1.1-1.2.1.3 for *current_class*=3, *current_class_member*={2}
- Step 1.2.2.1: *mipsClass*(3)={2}
- Step 1.3: *mipsNumClass*=3
- *mipsClass*=[{5,4}; {1,3}; {2}]

Result: *ClassUsed*=*mipsClass*

Figure 5.6: Illustration of MHA Preprocessing for EX1

5.2.1.2 OMA Preprocessing

OMA has been modified from the classical Online-Matching algorithms [Chang et al., 1993] [Coffman and Lueker, 1991] so that it can handle service instance processing requirements when they are drawn from a finite set instead of having independent identically distributed values.

The main idea of Online-Matching algorithms [Chang et al., 1993] [Coffman and Lueker, 1991] surveyed in Section 2.1.3 is to classify items into *big* and *small* types. An item is *big* if it is greater than $\frac{1}{2}$ and *small*, otherwise. Items of type *big* and *small* are assigned together in the same bin. In other words, items that are of *big* and *small* types will be put in the same class. Moreover, Chang et al. [1993] divide the $(0,1]$ interval into a number of subintervals by using a hashing function h_m . Items of big and small types have the same value of h_m in the same class in Chang et al. [1993]. As mentioned in section 2.1.3, a *big* item's companion is an item of a *small* type and vice versa. Both of them can be assigned together into a bin.

The algorithm of Chang et al. [1993] can be generalized to the multi-dimensional case. Each item is classified into eight (2^3) types instead of two types as in the one-dimensional case. Items are assumed to be uniformly distributed on $(0,1]^d$, where d is the number of dimensions.

Consider applying Chang et al.'s [1993] classification method to the service types in the SRTPA problem. The service requirements for each service type in the SRTPA problem are pre-determined. In other words, we are dealing with item sizes that are from a finite set. As discussed in the previous section, it is more than likely

that each service type belongs to a different class. By having a limited number of processors in the system, if none of the processors belonging to the same class can accommodate a service request s , then s is assigned to any processor that can accommodate it. This method of allocation is exactly the same as MFFA, MBFA, MWFA, depending on which algorithm is used after the first procedure fails.

The *big* and *small* classes discussed above are still used, but $1/2$ is no longer used as the boundary between the two. It can be an arbitrary number, depending on the processing requirements of the service types.

Two service types, s_i and s_j , are called *companions* if $\text{Req}(s_i) + \text{Req}(s_j) \leq 100$. $\text{Comp}(\text{pmemList}(i))$, $\text{Comp}(\text{dmemList}(i))$ and $\text{Comp}(\text{mipsList}(i))$ are service types that are companions of the $\text{pmemList}(i)$ service type, $\text{dmemList}(i)$ service type and $\text{mipsList}(i)$ service type respectively.

Classifying service types based on pmemList will be examined below. The first class members are $\text{pmemList}(1)$ and the last service type in pmemList such that a single service instance of both service types can be assigned to the same processor. The second class members are $\text{pmemList}(2)$ and the last index before the $\text{pmemList}(1)$'s companion in pmemList such that a single service instance of both service types can be handled in a processor. For instance, assume that a single instance of service type $\text{pmemList}(2)$ and $\text{pmemList}(\text{TotalServiceTypes}-1)$ can be assigned together in the same processor. If $\text{pmemList}(1)$'s companion is the last service type in pmemList , then $\text{pmemList}(2)$'s companion is the second last service type in pmemList . For any $\text{pmemList}(i)$ that does not have a companion, it will be put alone in a class. The rest of the service types will be classified in this manner,

until service type $\text{pmemList}(i+1)$ has been assigned to a class. Continue applying the above procedure to the service types in pmemList that have not been assigned to any class. This procedure will eventually assign all service types to a class, since every $\text{pmemList}(i)$ that does not have a companion will be assigned to a class alone.

Partition each of the $MList$, where $M \in \{mips, dmem, pmem\}$ into the following classes:

- find $\text{Comp}(MList(1))$, i.e. find a companion of $MList(1)$, where $\text{Comp}(MList(1))$ is the largest integer k such that $1 < k \leq \text{TotalServiceTypes}$ and $|\text{Req}(MList(1))| + |\text{Req}(\text{Comp}(MList(1)))| \leq 100$. Put $MList(1)$ and $\text{Comp}(MList(1))$ together in the first class. If $\text{Comp}(MList(1))$ is not found, then $MList(1)$ will be assigned to the first class.
- for each i , $1 < i < \text{Comp}(MList(i-1))$, find the companion of $MList(i)$, $\text{Comp}(MList(i))$, where $\text{Comp}(MList(i))$ is the largest integer k such that $i < k < \text{Comp}(MList(i-1))$ and $|\text{Req}(MList(i))| + |\text{Req}(\text{Comp}(MList(i)))| \leq 100$. Put $MList(i)$ and $\text{Comp}(MList(i))$ together in the same class. If $\text{Comp}(MList(i))$ is not found, then $MList(i)$ will be assigned to the current class alone.

For the same reason discussed in the previous section, OMA uses the service types classification based on whichever of mipsList , dmemList or pmemList has the minimum total number of classes.

Algorithm details follow.

GIVEN: $\text{max_num_class}=0$; mipsList , dmemList and pmemList

1. For each $MList$, where $M \in \{mips, dmem, pmem\}$:
 - 1.1 Initialize: $\text{current_class}=1$, $\text{TmpList}=\emptyset$, $\text{MClass}=\emptyset$, $\text{MNumClass}=0$.

- 1.2 $st_index=1$, $comp_index$ =number of elements in *MList*,
 $last_comp_index=comp_index$, $current_class_member=\{MList(1)\}$.
- 1.3 If ($|Req(MList(st_index))| + |Req(MList(comp_index))| \leq 100$) and
 $(st_index < comp_index)$ then
 - 1.3.1 Add *MList(comp_index)* to *current_class_member*
 - 1.3.2 $MClass(current_class)=current_class_member$
 - 1.3.3 $st_index=st_index+1$; $last_comp_index=comp_index-1$;
 $comp_index=comp_index-1$;
 $current_class_member=current_class_member+1$;
 - 1.3.4 If $st_index \geq comp_index$ then
 - 1.3.4.1 Copy all elements in *MList* that have not been
assigned to a class to *TmpList*
 - 1.3.4.2 If the number of elements in *TmpList* is greater than
one then
 - 1.3.4.2.1 $MList=TmpList$
 - 1.3.4.2.2 Go to step 1.2
 - 1.3.4.3 Else
 - 1.3.4.3.1 $MNumClass$ is equal to the number of
classes in *MClass* and exit
 - 1.3.5 Go to step 1.2
- 1.4 Else
 - 1.4.1 $comp_index=comp_index-1$;
 - 1.4.2 If $st_index > comp_index$ then
 - 1.4.2.1 $current_class_member=\{MList(st_index)\}$;
 - 1.4.2.2 If all members in *MList* has been assigned to a class
then $MNumClass$ is equal to the number of classes in
MClass and exit.
 - 1.4.2.3 $st_index=st_index+1$; $comp_index=last_comp_index$;
 - 1.4.3 Go to step 1.2

RESULT: *ClassUsed*, the classification of service types that will be used by OMA,
where *ClassUsed* is equal to one of *mipsClass*, *dmemClass* or *pmemClass*,

whichever has the smallest total number of classes among the three (i.e. $\min\{\text{mipsNumClass}, \text{dmemNumClass}, \text{pmemNumClass}\}$). If `mipsClass`, `dmemClass` and `pmemClass` have the same total number of classes then choose arbitrarily.

Figure 5.7 illustrates preprocessing of MHA by application to the EX1 list of service types in Figure 5.1. The number of classes in each of `pmemClass`, `dmemClass` and `mipsClass` is three. Therefore, one of `pmemClass`, `dmemClass` or `mipsClass` can be chosen arbitrarily for `ClassUsed`, since all of them have the same number of classes.

5.2.2 Processing Incoming Service Requests

We define the following data structure:

- $Bclass(c)$, for each class c , is a list of processors that have service types of class c in them.

MHA assigns each incoming service request s of type i of class c in the following manner:

1. If there is a processor, p , in $Bclass(c)$, such that $\text{ProcServCapacity}(p, i) \leq 100$, then assign s into p and exit.

Processing *pmemList*={5, 1, 3, 2, 4}

- Step 1.3.1-1.3.2: for *current_class*=1, *current_class_member*={5,4},
pmemClass(1)={5,4}
- Step 1.3.1-1.3.2: for *current_class*=2, *current_class_member*={1,3},
pmemClass(2)={1,3}
- Step 1.4.2.1-1.4.2.2: for *current_class*=3, *current_class_member*={2}
pmemClass(3)={2}
- *pmemClass*=[{5,4}; {1,3}; {2}], *pmemNumClass*=3

Processing *dmemList*={2, 4, 1, 3, 5}

- Step 1.3.1-1.3.2: for *current_class*=1, *current_class_member*={2, 5},
dmemClass(1)={2,5}
- Step 1.4.2.1-1.4.2.2: for *current_class*=2, *current_class_member*={4},
dmemClass(2)={4}
- Step 1.3.1-1.3.2: for *current_class*=3, *current_class_member*={1,3}
dmemClass(3)={1,3}
- *dmemClass*=[{2,5}; {4}; {1,3}], *dmemNumClass*=3

Processing *mipsList*={5, 4, 1, 3, 2}

- Step 1.3.1-1.3.2: for *current_class*=1, *current_class_member*={5,2},
MipsClass(1)={5,2}
- Step 1.4.2.1-1.4.2.2: for *current_class*=2, *current_class_member*={4},
MipsClass(2)={4}
- Step 1.3.1-1.3.2: for *current_class*=3, *current_class_member*={1,3}
MipsClass(3)={1,3}
- *mipsClass*=[{5,2}; {4}; {1,3}], *mipsNumClass*=3

Result: *ClassUsed*=*pmemClass*/*dmemClass*/*mipsClass*

Figure 5.7: Illustration of OMA Preprocessing for EX1

2. If there is a processor, p , in $\text{AssignedTo}(i)$, such that $\text{ProcServCapacity}(p,i) \leq 100$, then assign s into p and exit.
3. If there is an empty processor p , then assign s to p and exit.
4. Use FFA to allocate s .

The following assumptions are made:

- Incoming service requests: $(s_{31}, s_{52}, s_{43}, s_{24}, s_{55}, s_{16})$ where s_{ij} is j^{th} service request, and it is of type i .
- None of the above service instances expire during the allocation.
- Total number of DSP processors = 3. We will have (P_1, P_2, P_3) .

Allocation for each service request to the given four DSP processors:

$$P_1 : s_{31} \ s_{16}$$

$$P_2 : s_{52} \ s_{43} \ s_{55}$$

$$P_3 : s_{24}$$

None of service requests is blocked.

Figure 5.8: Illustration of MHA Processing Service Requests for EX1

Figure 5.8 illustrates MHA processing service requests by application to the EX1 list of service types in Figure 5.1. mipsClass classification of service types will

be used. The first two service requests, s_{31} and s_{52} , are assigned to empty processors, P_1 and P_2 respectively, since $BClass(2)$, $BClass(1)$, $AssignedTo(3)$ and $AssignedTo(5)$ are all empty. After assigning the first two service requests, $AssignedTo$ and $BClass$ are updated as follows:

- $AssignedTo(1)=\{P_1\}$, $Assigned(5)=\{P_2\}$.
- $BClass(2)=\{P_1\}$, $BClass(1)=\{P_2\}$.

s_{43} is of class 1 and there is an active processor P_2 in $BClass(1)$. Therefore, s_{43} is assigned to P_2 and P_2 is added to $AssignedTo(4)$ list. Both of $AssignedTo(2)$ and $BClass(3)$ are empty. Hence s_{24} is assigned in an empty processor P_3 and $AssignedTo$ and $BClass$ are updated as follows:

- $AssignedTo(2)=\{P_3\}$.
- $BClass(3)=\{P_3\}$.

$BClass(1)$ has P_2 in it and P_2 can accommodate s_{55} . Hence s_{55} is assigned to P_2 . The last service request s_{16} is assigned to P_1 , since P_1 is in $BClass(2)$ and it can accommodate service type 1.

OMA processing service requests by application to the EX1 list of service types in Figure 5.1 is illustrated in Figure 5.9. $dmemClass$ classification of service types will be used. The first three service requests, s_{31} , s_{52} , s_{43} are assigned to empty processors, P_1 , P_2 and P_3 respectively, since $AssignedTo(3)$, $AssignedTo(5)$, $AssignedTo(4)$, $BClass(2)$, $BClass(1)$ and $BClass(3)$ are all empty. After assigning the first three service requests, $AssignedTo$ and $BClass$ are updated as follows:

- $AssignedTo(1)=\{P_1\}$, $Assigned(5)=\{P_2\}$, $AssignedTo(4)=\{P_3\}$.
- $BClass(3)=\{P_1\}$, $BClass(1)=\{P_2\}$, $BClass(2)=\{P_3\}$.

BClass(1) has P_2 in it and P_2 can accomodate s_{24} . Therefore, s_{24} is assigned to P_2 .

BClass(1) has P_2 in it and P_2 can accommodate s_{55} . Hence s_{55} is assigned to P_2 .

OMA assigns the last service request s_{16} to P_1 , since P_1 is in BClass(3) and it can accommodate service type 1.

The following assumptions are made:

- Incoming service requests: $(s_{31}, s_{52}, s_{43}, s_{24}, s_{55}, s_{16})$ where s_{ij} is j^{th} service request, and it is of type i .
- None of the above service instances expire during the allocation.
- Total number of DSP processors = 3. We will have (P_1, P_2, P_3) .

Allocation for each service request to the given four DSP processors:

$$P_1 : s_{31} \ s_{16}$$

$$P_2 : s_{52} \ s_{24} \ s_{55}$$

$$P_3 : s_{43}$$

None of service requests is blocked.

Figure 5.9: Illustration of OMA Processing Service Requests for EX1

5.2.3 Processing the Termination of a Service Instance

As each service instance s of type i of class c terminates its service in processor p , update $BClass(c)$ and $AssignedTo(i)$ accordingly. Algorithm details follow.

Given: service instance s of type i of class c is terminating from processor p .

1. deallocate s from processor p .
2. if there are no other service instances in processor p of type i , then delete p from $Assigned(i)$.
3. if there are no other service instance in processor p of class c , then delete p from $BClass(c)$.

5.3 BEST POSSIBLE FIT Algorithm (BPFA)

The BPFA has two major parts, namely, preprocessing and processing of each service request.

The main idea behind the BPFA algorithm is to generate a rank-ordered list of service instance combinations during preprocessing, then the assignment for each incoming service request to a processor is based on that rank-ordered list. Assignments higher in the rank-ordered list are preferred. For instance, all processors p_x , p_y and p_z can accommodate a service request s of type i . However, the combination of all service instances in p_y along with s is higher in the list compared

to the combination of all service instances in p_x or p_z along with s . Therefore, the BPFA assigns s to p_y .

5.3.1 Preprocessing

The notation in MHA and OMA will be used here.

Before going any further, let us briefly review the relevant probability theory. Consider picking n objects from a set of m distinct objects. After selecting an object and noting its identity, the object is placed back in the set before the next choice is made and we record the result without regard to order. The number of different ways of picking n objects from a set of m distinct object using the above rule is given by [A. Garcia, 1994]

$$\frac{(m-1+n)!}{n! (m-1)!}$$

A feasible combination of service instances is a combination of service instances that can be assigned to the same DSP processor. n objects and m distinct objects in the above discussion are equivalent to n service instances in a single processor and the number of distinct service types, TotalServiceTypes respectively in this case. The maximum number of service instance combinations for n service instances in a single processor, NumServCombFor $_n$, is given by

$$\frac{(\text{TotalServiceTypes}-1+n)!}{n! (\text{TotalServiceTypes}-1)!}$$

Therefore, if the maximum number of service instances that can be assigned within the same processor is m , then the total number of combinations that need to be

examined for their feasibility are $\text{NumServCombFor}_m + \dots + \text{NumServCombFor}_2$, where NumServCombFor_x is the number of service instance combinations for x service instances in a single processor. In other words, there are

$$\frac{(\text{TotalServiceTypes}-1+m)!}{m! (\text{TotalServiceTypes}-1)!} + \frac{(\text{TotalServiceTypes}-2+m)!}{(m-1)! (\text{TotalServiceTypes}-1)!} + \dots + \frac{(\text{TotalServiceTypes}+1)!}{2! (\text{TotalServiceTypes}-1)!}$$

combinations that need to be examined for their feasibility. A processor can always handle a single service instance, therefore the feasibility test does not have to be done for those combinations.

$\{a_1, a_2, \dots, a_n\}$ implies the combination of n service instances of type $a_1, a_2, \dots, a_n$ in a processor. AllCombo_n is a list of all service instance combinations for n service instances in a processor. The first combination of AllCombo_n (i.e. $\text{AllCombo}_n(1)$) is $\{1, 1, \dots, 1\}$. $\text{AllCombo}_n(2)$ can be obtained by incrementing a_n by one; in other words, $\text{AllCombo}_n(2) = \{1, 1, \dots, 1, 2\}$. The next combination of any $\text{AllCombo}_n(k)$, for $k \leq \text{TotalServiceTypes}$, can be obtained by incrementing the last member of $\text{AllCombo}_n(k-1)$ by one. Note that the $(\text{TotalServiceTypes}+1)^{\text{th}}$ combination of AllCombo_n cannot be obtained by incrementing the last member of the previous combination. However, the $(\text{TotalServiceTypes}+1)^{\text{th}}$ member of AllCombo_n is obtained by incrementing the second last member of the previous combination by one and setting the last member to have the same value as the second last member. Hence, $\text{AllCombo}_n(\text{TotalServiceTypes}+1) = \{1, 1, \dots, 2, 2\}$. Given the k^{th} combination of AllCombo_n , $\text{AllCombo}_n(k) = \{a_1, a_2, \dots, a_n\}$, the above procedure for obtaining the $(k+1)^{\text{th}}$ combination can be summarized as follows: starting from the last member of $\text{AllCombo}_n(k)$ find a_i such that $a_i \leq \text{TotalServiceTypes}$, then increment a_i by one and set $a_j, j=i+1, \dots, n$ to have the same

value as a_i . If no such a_i exists, then stop the above procedure, since all service instance combinations for n service instances in a processor have been generated.

$TotalProcessingComb(\{a_1, a_2, \dots, a_n\})$ is a vector of the total processing requirements of service type $a_1, a_2, \dots$, and a_n . The three elements are *pmem*, the program memory requirement, *dmem*, the data memory requirement, and *mips*, the MIPS requirement. As discussed in section 4.1, only one copy of program memory is sufficient for multiple service instances of a particular service type. For instance, using the list of service types in EX2 (Figure 5.10), $TotalProcessingComb(\{s_2, s_2, s_2\})=[50,90,90]$.

Requirement	Type 1	Type 2
pmem	45	50
dmem	35	30
mips	8	30

pmem : program memory requirement
dmem : data memory requirement
mips : mips processing requirement
(in % of processor resources)

Figure 5.10: List of Service Types in EX2

A small list of service types to illustrate preprocessing in BPFA is given in Figure 5.10. The system has two service types, s_1 and s_2 , and the maximum number of service instances that can be put in a processor is three. Then we have the

following possible service instance combinations for three service instances in a processor: $(s_1 s_1 s_1)$, $(s_1 s_1 s_2)$, $(s_1 s_2 s_2)$ and $(s_2 s_2 s_2)$.

Find all feasible combinations of service instances as follows:

- find *MaxServPerProcessor* which is the maximum number of service instances that can be assigned within a single DSP processor
- for $NumServiceInstInProcessor=1,2,\dots, MaxServPerProcessor$
 - find all combinations of *NumServiceInstInProcessor* service instances that can be assigned within the same DSP processor p
 - sort the above service instance combinations based on $|Load(p)|$ in descending order

We make the following definitions:

- *CurrentCombination* is the current service instance combination being examined for feasibility. *CurrentCombination(i)* is the i^{th} member of this combination.
- *IncrementMember* is an integer, representing the *IncrementMemberth* element in *CurrentCombination*.
- *FeasibleCombination* is a list of all feasible combinations of service instances that can be assigned to a single processor.
- *FeasibleCombCap* is a list that corresponds to *FeasibleCombination*. The i^{th} element of *FeasibleCombCap* is the largest element of the total processing requirements of all service instances of *FeasibleCombination(i)* (i.e. $|TotalProcessingComb(FeasibleCombination(i))|$).

Algorithm details follow.

GIVEN: ServiceList, TotalServiceTypes

1. Initialization:
 - $MinProcReq = \min\{\max(Req(1).dmem, Req(1).mips), \max(Req(2).dmem, Req(2).mips), \dots, \max(Req(TotalServiceTypes).dmem, Req(TotalServiceTypes).mips)\}$
 - $MaxServPerProcessor = \lfloor 100 / MinProcReq \rfloor$
 - $FeasibleCombination = \emptyset; FeasibleCombCap = \emptyset$
2. If $MaxServPerProcessor = 1$ then go to step 4
3. For $i = 2$ to $MaxServPerProcessor$
 - 3.1 For $j = 1$ to i
 - 3.1.1 $CurrentCombination(j) = 1$
 - 3.2 $IncrementMember = i$;
 - 3.3 While $IncrementMember \neq 0$
 - 3.3.1 If $TotalProcessingComb(CurrentCombination) \leq 100$
 - 3.3.1.1 Append $CurrentCombination$ to $FeasibleCombination$
 - 3.3.1.2 Append $\lfloor TotalProcessingComb(CurrentCombination) \rfloor$ to $FeasibleCombCap$
 - 3.3.2 $IncrementMember = i$
 - 3.3.3 While $CurrentCombination(IncrementMember) = TotalServiceTypes$
 - 3.3.3.1 $IncrementMember = IncrementMember - 1$
 - 3.3.4 If $IncrementMember > 0$
 - 3.3.4.1 $CurrentCombination(IncrementMember) = CurrentCombination(IncrementMember) + 1$
 - 3.3.4.2 For $j = (IncrementMember + 1)$ to i
 - 3.3.4.2.1 $CurrentCombination(j) = CurrentCombination(IncrementMember)$
4. Add a single service instance combination to $FeasibleCombination$ and record its Req to $FeasibleCombCap$

5. Sort *FeasibleCombination* based on *FeasibleCombCap* (i.e. sort *feasible_combination* based on the total processing requirements of all service instances in each combination in descending order)

RESULT: *FeasibleCombination* is a list of all possible combinations of service instances that can be assigned within the same DSP processor, and it will be used by BPFA.

Figure 5.11 illustrates preprocessing of BPFA by application to the EX2 list of service types of Figure 5.10. *FeasibleCombination* is equal to $\{(1,1,2),(1,2),(1,2,2),(2,2,2),(1,1),(2,2),(2),(1)\}$.

5.3.2 Processing Incoming Service Requests

total_feasible_comb is the total number of combinations in the *FeasibleCombination* list. BPFA assigns each incoming service request, s , in the following manner:

1. find the smallest j such that s along with the service instances in processor p are members of *FeasibleCombination*(j). If such j exists and
 - if there are a number of processors p such that s along with the service instances in processor p are member of *FeasibleCombination*(j), then assign s to the processor p that has the most number of service instances in it and exit.
 - if there is only one processor p then assign s to the processor p and exit.

ServiceList={1, 2, 3}, *TotalServiceTypes*=3

- Step 1: *MinProcReq*=30, *MaxServPerProcessor*=3.
- Step 3: *i*=2
- Step 3.1.1: *CurrentCombination*=(1,1), *CurrentCombCapacity*=(45,70,16)
- Step 3.3.1.1-3.3.1.2: *FeasibleCombination*={(1,1)}, *FeasibleCombCap*=(70)
- Step 3.3.4.1: *CurrentCombination*=(1,2), *CurrentCombCapacity*=(95,75,38)
- Step 3.3.1.1-3.3.1.2: *FeasibleCombination*={(1,1),(1,2)}, *FeasibleCombCap*=(70,95)
- Step 3.3.4.1-3.3.4.2: *CurrentCombination*=(2,2), *CurrentCombCapacity*=(50,60,60)
- Step 3.3.1.1-3.3.1.2: *FeasibleCombination*={(1,1),(1,2),(2,2)},
FeasibleCombCap=(70,95,60)
- Step 3: *i*=3
- Step 3.1.1: *CurrentCombination*=(1,1,1), *CurrentCombCapacity*=(45,105,24)
- *CurrentCombination is not a feasible service instances combination*
- Step 3.3.4.1: *CurrentCombination*=(1,1,2), *CurrentCombCapacity*=(95,100,46)
- Step 3.3.1.1-3.3.1.2: *FeasibleCombination*={(1,1),(1,2),(2,2),(1,1,2)},
FeasibleCombCap=(70,95,60,100)
- Step 3.3.4.1-3.3.4.2: *CurrentCombination*=(1,2,2), *CurrentCombCapacity*=(95,95,68)
- Step 3.3.1.1-3.3.1.2: *FeasibleCombination*={(1,1),(1,2),(2,2),(1,1,2),(1,2,2)},
FeasibleCombCap=(70,95,60,100,95)
- Step 3.3.4.1-3.3.4.2: *CurrentCombination*=(2,2,2), *CurrentCombCapacity*=(50,90,90)
- Step 3.3.1.1-3.3.1.2: *FeasibleCombination*={(1,1),(1,2),(2,2),(1,1,2),(1,2,2),(2,2,2)},
FeasibleCombCap=(70,95,60,100,95,90)
- Step 4: *FeasibleCombination*={(1,1),(1,2),(2,2),(1,1,2),(1,2,2),(2,2,2),(1),(2)},
FeasibleCombCap=(70,95,60,100,95,90,45,50)
- Step 5: *FeasibleCombination*={(1,1,2),(1,2),(1,2,2),(2,2,2),(1,1)(2,2),(2),(1)}.

Figure 5.11: Illustration of BPFA Preprocessing for EX2

2. if there is an empty processor p , then assign s to p and exit.
3. blockage occurs, s is discarded and exit.

We define *max_service_match* as the maximum number of service instances in a processor p such that service request s along with the service instances in p are members of one of combinations in FeasibleCombination. Algorithm details follow.

Given: service request s of type i

1. if s is the first service request then
 - 1.1 assign s into the first processor p_1 and exit.
2. for $i = 1$ to total_feasible_comb
 - 2.1 *max_service_match*=0; *cand_proc*= $\emptyset$.
 - 2.2 for each active processor p that can accommodate s
 - 2.2.1 if s along with the service instances in p are members of FeasibleCombination(i) then
 - 2.2.1.1 if *max_service_match* < 1+(number of service instances in p) then
 - 2.2.1.1.1 *max_service_match*= 1+(number of service instances in p)
 - 2.2.1.1.2 *cand_proc*= p
 - 2.3 if *cand_proc* $\neq \emptyset$ then
 - 2.3.1 assign s to *cand_proc* and exit
3. if there is an empty processor p then assign s to p and exit.
4. blockage occurs and exit.

After running the preprocessing for EX1, *FeasibleCombination*={(4,4,4,4),(4,4,4,5),(4,4,5),(4,5,5),(5,5,5),(4,5),(3,5,5),(3,3,5),(1,3),(2,5),(1,5,5),(1,1,5),(4,4,4),(4,4),(3,3),(1,1),(3,5),(5,5),(1,5),(4),(2),(3),(1),(5)}.

The following assumptions are made:

- Incoming service requests: $(s_{31}, s_{52}, s_{43}, s_{24}, s_{55}, s_{16})$ where s_{ij} is j^{th} service request, and it is of type i .
- None of the above service instances expire during the allocation.
- Total number of DSP processors = 3. We will have (P_1, P_2, P_3) .

Allocation for each service request to the given four DSP processors:

$P_1 : s_{31} s_{52}$

$P_2 : s_{43} s_{55}$

$P_3 : s_{24}$

s_{16} is blocked.

Figure 5.12: Illustration of BPFA Processing Service Requests for EX1

Figure 5.12 illustrates BPFA processing service requests by application to the EX1 list of service types of Figure 5.1. The first service request s_{31} is assigned to an empty processor P_1 . s_{52} is assigned together with s_{31} in P_1 , since there is a (3,5,5) combination in *FeasibleCombination* list. The only active processor P_1 cannot handle s_{43} . Therefore, it is assigned to an empty processor P_2 . Similarly, s_{24} is assigned to an empty processor P_3 , since none of the active processors can accommodate s_{24} . Both active processors P_1 and P_2 can accommodate s_{55} . However, the feasible combination that has service types 4 and 5 is ahead of the feasible combination that has service types 3 and 5 in *FeasibleCombination* list.

Consequently, s_{55} is assigned to P_2 instead of P_1 . s_{16} cannot be assigned to any of the active processors and there are no empty processors left. Hence, s_{16} is discarded and blockage occurs.

CHAPTER 6

Empirical Results

A “Traffic” has three elements, namely, service type (one of the DSP algorithms), duration of the service and arrival time for the next service. A list of traffic that is generated by a random traffic generator is a “traffic stream”.

Throughout this chapter, the SRTPA algorithms are designated as follow:

FF – First-Fit algorithm (Section 5.1.1)

MFF – Modified-First-Fit algorithm (Section 5.1.4)

BF – Best-Fit algorithm (Section 5.1.2)

MBF – Modified-Best-Fit algorithm (Section 5.1.4)

WF – Worst-Fit algorithm (Section 5.1.3)

MWF – Modified-Worst-Fit algorithm (Section 5.1.4)

MH – Modified-Harmonic algorithm (Section 5.2)

OM – Online-Matching algorithm (Section 5.2)

BPF – Best-Possible-Fit algorithm (Section 5.3)

FFMC – Fixed-Service-Multiple-Channel algorithm (Chapter 3).

The various versions of SRTPA algorithms have been tested on four different sets of traffic streams based on actual traffic streams in the current VoIP applications and also predicted future VoIP applications. Each traffic stream is solved using various SRTPA algorithms and different numbers of DSP processors. The average blockage rate for each set of traffic streams using different numbers of DSP processors is used to compare the various SRTPA algorithms.

All of the above algorithms were implemented in Matlab and test results were obtained from simulation on a 2.4 GHz Intel Pentium 4 station with 128 Mbytes of Random Access Memory (RAM) under Windows NT operating system.

6.1 Simulation Setup

The services used in the simulations are the DSP algorithms listed in Table 6.1. T.38 is a DSP algorithm for fax, and all of the others are DSP algorithm for vocoders. These are all DSP algorithms that are commonly used in VoIP applications.

The call duration for each incoming service instance is generated using the Uncentral F-Distribution (5,2,5). Call duration for T.38 is generated in this way so that it centralizes around two minutes, and three minutes for all vocoder DSP algorithms. The maximum duration for all incoming service instances is 120

minutes. The duration between any two service-instances is based on a discrete uniform distribution $U(0,0.3]$. These choices are based on information that we gathered from the industry.

DSP Algorithm Processor Resource Requirements			
DSP Algorithm	Program Memory	Data Memory	Processor Speed
G.711	7.2 Kwords	116 words	0.2 MIPS
G.726	3.6 Kwords	58 words	4.5 MIPS
G.729.A	10.1 Kwords	571 words	8.9 MIPS
G.723.1	18 Kwords	400 words	17 MIPS
T.38	25.1 Kwords	2086 words	4.1 MIPS
Table 6.1: List of DSP Algorithms Used			

Random traffic is generated for a simulated 120 minutes during which approximately 700 service instances are generated. The total number of simultaneous active calls can never exceed the input link capacity. Typical input trunks are T1 and E1, and their capacities are 24 and 32 respectively. The system can have multiple T1 or E1 trunks. In this study, the system has two T1 input trunks. Therefore, the system can handle up to 48 service instances simultaneously.

The random traffic stream generating procedure accepts two inputs, a list of service types (DSP algorithms) and the distribution for each service.

6.1.1 Traffic Model

Preliminary experiments showed that different distributions of the service types affect the performance of an SRTPA algorithm. Thus test results accumulated in this study are classified according to the distribution of each service type in the traffic stream. For each set of service type distributions in the traffic stream, a random traffic stream generator was implemented and fifty traffic streams were generated, which were subsequently solved using the various SRTPA algorithms.

In this study, traffic streams are classified into four categories according to the distribution of each service type in the traffic stream. The four different distributions of service types, namely, $TS_{current_all}$, TS_{future_all} , $TS_{current_lowbitrate}$ and $TS_{future_lowbitrate}$ are summarized in Table 6.2.

The DSP algorithms applied in current VoIP gateways operate at a higher bit rate. As discussed in section 2.3, G.711 and G.726 have considerably lower bit rates compared to G.729.A and G.723.1 and they both require less processing power than later ones. $TS_{current_all}$, which reflects existing traffic stream conditions contains more frequent G.711 algorithms that have smaller processing requirements. However, DSP algorithms with lower bit rates will be used more frequently in future VoIP gateways. TS_{future_all} reflects the conditions expected in future traffic streams. It has more frequent G.726 and G.729.A algorithms that operate at lower bit rates, but have more processing requirements. In some systems, it is not possible to have a DSP algorithm that operates at a high bit rate, so G.711 cannot be supported. Both $TS_{current_lowbitrate}$ and $TS_{future_lowbitrate}$ traffic stream sets exclude G.711. $TS_{current_lowbitrate}$

has an equal frequency for each service type and $TS_{future_lowbitrate}$ exemplifies likely conditions in future traffic streams.

DSP Algorithms	G.711	G.726	G.729.A	G.723.1	T.38
$TS_{current_all}$	50 %	15 %	15 %	15 %	5 %
TS_{future_all}	10 %	15 %	35 %	35 %	5 %
$TS_{current_lowbitrate}$	0 %	25 %	25 %	25 %	25 %
$TS_{future_lowbitrate}$	0 %	15 %	40 %	40 %	5 %
Table 6.2: Distributions of Service Types in Traffic Streams					

Consider the BPF algorithm that is discussed in section 5.3. With the existence of service type G.711, the maximum number of service instances that can be assigned within one DSP processor is 200. Therefore, there are

$$\frac{204!}{200! 4!} + \frac{203!}{199! 4!} + \frac{202!}{198! 4!} + \dots + \frac{6!}{2! 4!} = 2.8724 \times 10^9$$

service combinations that can possibly be assigned within one DSP processor. Not all of these combinations are feasible, but the solutions must be generated before feasibility can be tested. The solution space to be searched is extremely large. Therefore, $TS_{current_all}$ and TS_{future_all} will not be run for the BPF algorithm in the current study, since finding all feasible combinations would require excessive computation time. However, G.711 is excluded from $TS_{current_lowbitrate}$ and $TS_{future_lowbitrate}$. Thus both $TS_{current_lowbitrate}$ and $TS_{future_lowbitrate}$ can be run for all algorithms.

6.1.2 Multiprocessor Model

For each of the $TS_{\text{current_all}}$, $TS_{\text{future_all}}$, $TS_{\text{current_lowbitrate}}$ and $TS_{\text{future_lowbitrate}}$ traffic stream distributions, different numbers of DSP processors are used to handle each traffic stream. Each DSP processor is a Texas Instruments TMS320C541 processor. Its capacity is 64 Kwords for both data and program memory and 40 MIPS processing capability. The Texas Instruments TMS320C541 processor is a typical DSP processor that is widely used in current VoIP application.

For a given traffic stream, the more DSP processors the system has, the lower the blockage rate is going to be. Starting with four DSP processors, the number of DSP processors used by the system is incremented until the blockage rate is zero for all SRTPA algorithms that are discussed in chapter 5.

6.1.3 Baseline Comparison

As discussed in Chapter 3, Fixed Service Single Channel (FSSC) and Fixed Service Multiple Channel (FSMC) are two common scheduling algorithms that are used in current VoIP gateways. Therefore, in this study we will use these two algorithms as baseline comparisons against the various SRTPA algorithms discussed in Chapter 5.

FSSC is based on dedicated hardware, so each DSP processor is limited to supporting one specific service type only. The more recent generation of DSP

processors that is used in VoIP gateways is programmable. Therefore, FSSC is not a good comparison.

On the other hand, FSMS is based on programmable hardware. There is no pre-assignment of DSP processors to a service type in FSMC. An empty DSP processor can accommodate any service type. Thus, FSMC will be used as a baseline comparison against all SRTPA algorithms discussed in chapter 5.

6.1.4 Matlab Framework

David Lo has implemented a graphical interface (Figure 6.1) for evaluating the performance of scheduling algorithms. The graphical interface permits the specification of a DSP multiprocessor system.

It accepts as input:

1. Service Definition: list of DSP algorithms (e.g. G.711, G.726, etc.) and the parameters (data and program memory requirements, and processor speed) associated with each algorithm.
2. Architecture Definition: type of DSP processor along with its parameters (data and program memory capacities and MIPS processing capability) and the number of processors that are used by the system.
3. Allocation Definition: list of scheduling algorithms that are available for use.

The allocation of each incoming service instance to a DSP processor and termination of a service instance is shown in the “Load Monitor” section. After the simulation

terminates, statistical data on the blockage rate is plotted in the “Call Blockages” section.

Figure 6.1: Graphical Interface for DSP Multiprocessor Simulation

The results are given in Table 6.3, which shows the average blockage rate over fifty traffic streams. Note that BO is always the lowest blockage rate in an individual run among all SRTPA algorithms. The italic bold letters in each row indicate the best average performance over all SRTPA algorithms.

When four and five DSP processors are used, the best average performance is provided by the Online-Matching and Modified-Harmonic algorithms respectively. The best performance among all SRTPA algorithms is the Best-Fit algorithm when there are more than five DSP processors in the system. The histogram of the average blockage rates for all algorithms is shown in Figure 6.2. The blockage rate for Fixed-Service-Multiple-Channel (FSMC) algorithm is significantly higher than the rest.

Let us take a closer look at the performance of the SRTPA algorithms under $TS_{current_all}$ when more DSP processors are used (Figure 6.3). We eliminated FSMC from the graph, since its blockage rate is much higher than the others. We observe that the Worst-Fit algorithm has significantly higher average blockage rate than the other ones, while the Best-Fit algorithm has the best performance. First-Fit, Online-Matching, Modified-Harmonic and Modified-First-Fit have comparable performances. Modified-Worst-Fit has slightly worse performance than the four algorithms mentioned above. Furthermore, the Modified-Best-Fit algorithm performs better than any other algorithm except the Best-Fit algorithm.

Figure 6.2: Performance for TS_{current_all}

6.2.2 Results for Future Traffic Stream

The performances of each SRTPA algorithm have been compared under the future traffic stream patterns, TS_{future_all} . Table 6.4 lists the average blockage rates over the TS_{future_all} traffic stream set.

Number of DSP Processors vs SRTPA Algorithm										
	FSMC	FF	BF	WF	MH	OM	MFF	MBF	MWF	BO
4	84.30	71.33	72.32	71.28	71.15	71.03	71.15	71.56	71.00	69.9
5	80.35	67.31	68.16	67.39	66.94	67.13	67.13	67.60	66.71	65.78
6	76.44	62.46	63.42	62.36	63.29	62.36	62.54	63.07	62.02	60.76
7	73.17	56.29	57.72	56.08	56.41	56.28	56.20	56.62	56.03	54.38
8	71.08	49.05	50.10	49.31	49.16	48.61	49.17	49.58	48.66	47.08
9	68.22	41.63	41.23	42.19	41.51	41.10	41.79	41.63	41.49	39.59
10	64.81	34.40	32.65	34.04	34.26	33.90	34.73	34.29	33.78	31.43
11	60.73	27.67	24.55	26.64	27.13	26.98	27.59	26.84	27.27	24.08
12	54.13	21.06	16.94	20.71	20.48	20.73	21.28	19.89	21.23	16.83
13	46.32	14.21	10.26	15.36	14.20	14.32	14.65	13.29	14.71	10.22
14	37.67	8.58	5.12	10.89	8.50	8.33	8.35	7.46	9.07	5.12
15	28.19	4.00	2.01	6.70	4.00	3.84	3.96	3.27	4.75	2.00
16	18.87	1.34	0.49	3.69	1.22	1.16	1.20	0.92	1.65	0.46
17	20.84	0.19	0.03	1.60	0.27	0.16	0.16	0.12	0.35	0.03
18	5.93	0.01	0	0.43	0.03	0.01	0.01	0.01	0.02	0
19	2.48	0	0	0.06	0	0	0	0	0	0
20	0.76	0	0	0	0	0	0	0	0	0
Table 6.4: Blockage Rate for TS _{future_all} using SRTPA Algorithms										

The best average performance when fewer than eight DSP processors are used is provided by the Modified-Worst-Fit algorithm. The Online-Matching algorithm outperforms all other algorithms when eight and nine DSP processors are used. When there are more than ten DSP processors the Best-Fit algorithm has the lowest blockage rate. Figure 6.4 shows the average blockage rate for all algorithms when few DSP processors are used. When fewer DSP processors are used and the blockage rate is therefore higher, the FSMC algorithm blockage rate is not significantly higher than the rest. However, the blockage rate for the FSMC algorithm is much higher than the other algorithms when more processors are used.

Future Traffic Stream ---- (G711, G726, G729.A, G723.1, T38) - (10%, 15%, 35%, 35%, 5%)

Figure 6.4: Performance for TS_{future_all} When Fewer DSP Processors Are Used

Figure 6.5 shows the histogram of the average blockage rate for all algorithms (except for FSMC algorithm), when many DSP processors are used. The average blockage rate for the worst performer, Worst-Fit algorithm, is more than twice that of the best performer, Best-Fit algorithm. Modified-Best-Fit algorithm is the second best performer after Best-Fit algorithm. The blockage rate for the following four algorithms: First-Fit, Online-Matching, Modified-Harmonic and Modified-First-Fit are not much different. The Modified-Worst-Fit is the second worst performer after the Worst-Fit algorithm.

Figure 6.6: Performance for $T_{\text{current_lowbitrate}}$ When Fewer DSP Processors Are Used

Figure 6.7 shows the average blockage rate for each DSP algorithm when more DSP processors are used. The average blockage rate of the Worst-Fit algorithm is more than five times higher than that of the Best-Fit algorithm. The performance of the following five algorithms: First-Fit, Online-Matching, Modified-Harmonic, Modified-First-Fit and Best-Possible-Fit are comparable.

The Modified-Best-Fit and Modified-Worst-Fit are the second best and the second worst performers respectively.

6.2.4 Results for Low Bit Rate Future Traffic Stream

Table 6.6 lists the performance of each SRTPA algorithm under the future traffic stream which does not support high bit rate DSP algorithms. The Worst-Fit algorithm is the best performer when the system is using fewer than ten DSP

Number of DSP Processors vs SRTPA Algorithm											
	FSMC	FF	BF	WF	MH	OM	MFF	MBF	MWF	BBF	BO
4	94.48	80.72	80.99	80.30	80.68	80.53	80.49	80.83	80.56	80.56	79.15
5	90.78	76.88	77.10	75.75	76.63	76.53	76.59	76.86	76.47	76.60	75
6	87.94	73.06	73.34	72.08	72.82	72.68	72.98	73.22	72.42	73.06	71.02
7	85.34	69.46	69.79	68.81	69.57	69.47	69.60	69.79	69.10	69.40	67.87
8	83.33	66.13	66.51	65.38	65.80	65.85	66.23	66.69	65.69	66.13	64.32
9	80.58	60.93	62.19	60.39	61.47	60.85	61.32	61.80	61.10	61.19	58.89
10	78.94	53.86	54.67	53.99	53.64	54.30	54.11	53.92	53.85	54.11	51.47
11	76.18	45.02	44.71	45.03	45.56	45.37	45.29	45.12	45.14	46.39	42.30
12	73.91	36.96	35.06	36.13	36.98	37.63	37.66	37.83	37.13	37.78	33.40
13	72.61	29.64	25.38	28.10	29.01	29.38	29.45	28.36	29.53	28.88	24.79
14	70.29	21.32	16.95	22.43	21.50	21.51	21.95	20.84	22.52	21.91	16.86
15	68.99	14.79	9.73	16.73	14.52	14.50	14.62	13.39	16.15	14.71	9.682
16	59.41	7.84	4.49	11.86	8.05	7.73	7.91	6.93	9.23	8.18	4.50
17	59.13	3.07	1.47	7.46	3.13	3.01	3.17	2.46	4.28	3.60	1.43
18	26.55	0.73	0.24	3.82	0.72	0.67	0.67	0.44	1.16	0.97	0.23
19	22.87	0.08	0.02	1.39	0.10	0.06	0.07	0.02	0.15	0.11	0.01
20	6.76	0.01	0	0.34	0.01	0	0	0	0.01	0.01	0
21	0.53	0	0	0.06	0	0	0	0	0	0	0
22	0.07	0	0	0	0	0	0	0	0	0	0

Table 6.6: Blockage Rate for TS_{future_lowbitrate} using SRTPA Algorithms

Figure 6.8: Performance for TS_{future_lowbitrate} When Fewer DSP Processors Are Used

processors. If ten DSP processors are allowed in the system, then the Modified-Harmonic algorithm narrowly outperforms the other algorithms. The best performer among all SRPTA algorithms is the Best-Fit algorithm when the system has more than ten DSP processors. The histogram of Table 6.6 is shown in Figure 6.8.

The average blockage rate for each DSP algorithm when more DSP procesors are used is shown in figure 6.9. The blockage rate for the Worst-Fit

algorithm is significantly higher than the rest of the algorithms. All of the First-Fit, Online-Matching and Modified-First-Fit algorithms have comparable performances. The Best-Possible-Fit algorithm has slightly worse performance than the four algorithms mentioned above. The Modified-Best-Fit and Modified-Worst-Fit algorithms are the second best and the second worst performers respectively.

6.3 Discussion of Results

We define the *MeasureOfLoading* vector for measuring how loaded the system is. The two elements are *dload*, the data memory loading of the system, and *mload*, the MIPS loading of the system. All loadings of the system are in percent of maximum capacity. The program memory loading of the system is not included in *MeasureOfLoading*, since multiple instances of the same service type in a specific DSP processor share the program memory. *AvgSimultReq* is the average number of simultaneous service requests in the system over a certain period. *ServiceDistribution(i)* is the frequency for service type *i* expected by the system expressed as a percentage. The average total number of instances of service type *i* in the system at any time, *NumInst(i)*, is given by $AvgSimultReq * ServiceDistribution(i) / 100$. The same notation, *Req(i)*, *TotalProcessors* and *TotalServiceTypes* defined in Chapter 5 will be used here. Therefore, *MeasureOfLoading.dload* is given by

$$\begin{aligned} & ((NumInst(1)*Req(1).dmem) + (NumInst(2)*Req(2).dmem) + \dots + \\ & (NumInst(TotalServiceTypes)*Req(TotalServiceTypes).dmem)) / \\ & TotalProcessors. \end{aligned}$$

MeasureOfLoading.mload is calculated by replacing *dmem* in the above equation with *mips*. If one of the elements in *MeasureOfLoading* is greater than 100%, this indicates that the system is overloaded by

$$(\max\{MeasureOfLoading.dload, MeasureOfLoading.mload\} - 100)\%.$$

In the simulation, the Best-Fit algorithm is the best performer after the number of DSP processors used in the system is greater than a certain number. We define *BorderNumProcessors* as the number of processors at which the best algorithm changes. Under $TS_{current_all}$, TS_{future_all} , $TS_{current_lowbitrate}$ and $TS_{future_lowbitrate}$ traffic streams, the Best-Fit algorithm is the best performer if the system has more than five, nine, eight and ten DSP processors respectively; and these numbers are denoted as *BorderNumProcessors*. In this study, *AvgSimultReq*, the average number of simultaneous service requests in the system over 120 minutes for $TS_{current_all}$, TS_{future_all} and $TS_{future_lowbitrate}$ traffic stream is 46.905 and 47.512 for $TS_{current_lowbitrate}$ traffic stream. $TS_{current_all}$, TS_{future_all} and $TS_{future_lowbitrate}$ have the same *AvgSimultReq*, since service requests for fax (T.38) and vocoders (G.711, G.726, G.729.A and G.723.1) have the same frequency in these three traffic streams.

Table 6.7 displays the *MeasureOfLoading* under each traffic stream when *BorderNumProcessors* and (*BorderNumProcessors*+1) DSP processors are used

in the system. All entries in Table 6.7 are in percentage. Bold entries indicate the largest elements in $MeasureOfLoading$, $|MeasureOfLoading|$. All $|MeasureOfLoading|$ values are greater than 120% in the $BorderNumProcessor$ column except under $TS_{current_all}$, and they are less than 120% in the $(BorderNumProcessor+1)$ column. A value of $|MeasureOfLoading|$ less than 120% indicates that the system is under 20% overloaded.

Measure of Loading of Each Traffic Stream (<i>dload</i> , <i>mload</i>)		
# of Processor	(<i>BorderNumProcessor</i> +1)	<i>BorderNumProcessor</i>
$TS_{current_all}$	(3.86, 95.08)	(4.63, 114.10)
TS_{future_all}	(3.40, 116.85)	(3.77, 129.84)
$TS_{current_lowbitrate}$	(6.41, 113.83)	(7.21, 128.06)
$TS_{future_lowbitrate}$	(3.33, 119.83)	(3.67, 131.80)
Table 6.7: Measure of Loading of Each Traffic Stream (in %)		

In the $TS_{current_all}$, when five DSP processors are used, the blockage rate for the Best-Fit and the best performer, Modified-Harmonic algorithm are 12.96% and 12.90% respectively. The difference is negligible, so the Best-Fit algorithm can also be considered to be a good performer. Therefore, the value of $BorderNumProcessors$ in $TS_{current_all}$ could be modified from five to four, then the values in the first row in Table 6.7 become (4.63,114.10) for $(BorderNumProcessors+1)$ and (5.79,142.62) for $BorderNumProcessors$. Thus, we draw the following conclusion from the figures

in Table 6.7 with the above modification: in general when the system is under 20% overloaded, the Best-Fit algorithm should be used; otherwise, the Worst-Fit and Modified-Worst-Fit algorithm should be used.

The system expects more service requests during peak hours; in other words, the number of simultaneous active calls in the system is going to be higher than during regular hours. *AvgSimultReq* used in the above *MeasureOfLoading* formula is the average number of simultaneous service requests over certain period. However, the number of simultaneous service requests varies from time to time and the number of DSP processors is fixed in a real VoIP application. The above formula for *MeasureOfLoading* can be modified to reflect the real world application by replacing *AvgSimultReq* with the number of simultaneous service requests expected in the system during a certain period of time. Therefore, we can explicitly develop a multi-algorithm system: during periods when the system is under 20% overloaded, the Best-Fit algorithm is applied; otherwise the Worst-Fit or Modified-Worst-Fit algorithm is applied.

If the frequency of service requests pattern of the traffic stream matches one of the simulated traffic distributions, $TS_{current_all}$, TS_{future_all} , $TS_{current_lowbitrate}$ or $TS_{future_lowbitrate}$, discussed in Section 6.1.1., then we suggest the blockage rate within a time window is a good signal as to when to switch algorithms. The rules as to when to switch algorithms for each traffic stream are summarized in Table 6.8. Each entry indicates when to use a specific algorithm within a certain blockage rate range. The $[x\%,y\%)$ indicates a blockage rate between $x\%$ and $y\%$, not including $y\%$. In

$TS_{current_all}$, Best-Fit, Modified-Harmonic and Online-Matching algorithm are applied if $0\% < \text{blockage rate} \leq 8\%$, $8\% < \text{blockage rate} < 13\%$ and $13\% \leq \text{blockage rate} < 100\%$ respectively. In other words, 8% and 13% are the critical blockage rates for switching algorithms. If the blockage rate is higher and lower than 8%, the system uses Best-Fit and Modified-Harmonic algorithm respectively. Furthermore, the system uses Modified-Harmonic and Online-Matching algorithm when the blockage rate is lower and higher than 13% respectively. Similar interpretations apply to the TS_{future_all} , $TS_{current_lowbitrate}$ and $TS_{future_lowbitrate}$ traffic streams.

Traffic Streams	
$TS_{current_all}$	[0,8%] use BF, (8%,13%) use MH, [13%,100%) use OM
TS_{future_all}	[0,33%) use BF, [33%,49%) use OM, [49%,100%) use MWF
$TS_{current_lowbitrate}$	[0,24%) use BF, [24%,100%) use WF
$TS_{future_lowbitrate}$	[0,45%) use BF, [45%,54%) use MH, [54%,100%) use WF
Table 6.8: Rules for Switching Algorithms	

CHAPTER 7

Conclusions and Summary of Results

The main objective of this thesis has been to develop a good SRTPA for the assignment of service requests to processors. We develop a method that deals with the more recent generation of DSP processors used in VoIP gateways, which made the problem more interesting and difficult to solve.

7.1 Conclusions

- The problem stated in Chapter 4 has been solved in an efficient way as shown in Chapter 5. The blockage rates for all of the SRTPA algorithms discussed in Chapter 5 are significantly lower than the FSMC algorithm used in current VoIP systems.
- When the system is under 20% overloaded, the Best-Fit algorithm is the best performer and the Worst-Fit algorithm is the worst performer.

- When the system is above 20% overloaded, the Modified-Worst-Fit and Worst-Fit algorithms perform the best and the Best-Fit algorithm performs poorly.
- During peak hours when the system is above 20% overloaded, the system should use the Modified-Worst-Fit or Worst-Fit algorithms. Other than peak hours, the system should use the Best-Fit algorithm instead.
- If the frequency of service requests of the traffic stream matches one of the simulated traffic distributions discussed in Table 6.2, the system could use the rules for switching algorithms described in Table 6.8.

There is confidence that the proposed SRTPA algorithms will be of real use in scheduling and service request allocations in DSP multiprocessor systems used in modern VoIP gateways.

7.2 Summary of Contributions

The work reported in this thesis contributed in many ways to the current research on SRTPA:

- Showed that the newly developed SRTPA algorithms reduce blockage rates significantly compared to currently used conventional algorithms.
- Enhanced an existing classical bin-packing algorithm (as discussed in Section 5.1) to solve the SRTPA problem. The existing classical bin-packing algorithm was generalized to multidimensional problems. This method was further

generalized to handle multidimensional problems that are not fixed in the number of dimensions, since one copy of program memory is sufficient for multiple instances of the same service type in a processor.

- Used the classical harmonic algorithm and online matching algorithm ideas to solve the SRTPA problem when the service instance's processing requirement is drawn from a finite set instead of from independent identically distributed values.
- Developed the new Best-Possible-Fit algorithm (as discussed in Section 5.3) to solve the SRTPA problem.
- Identified the best algorithm to use for solving the SRTPA problem under different conditions.

7.3 Future Work

Although the main objective of this thesis has been attained, future research is needed to explore the following:

- Given n DSP programmable processors of different processing capabilities, how should the system allocate service requests to processors?
- In this thesis, we use blockage rate to measure the quality of different algorithms. Future research is needed to evaluate SRTPA algorithms using different measurements, for instance, the time when the first blockage occurs or processor utilization.

- In this SRTPA problem, once a service instance is allocated to a processor, it is irrevocable. Future research might improve the performance of the algorithms presented in Chapter 5 by allowing modification of assignments. In other words, any service instance can be moved from one processor to another as long as both processes run simultaneously for some time.

References

- Chang E. E., W. Wang and M. S. Kankanhalli. Multidimensional on-line bin packing: An algorithm and its average-case analysis. *Inform. Process. Lett.*, 48:121-125, 1993.
- Coffman E.G., M.R. Garey, and D.S. Johnson. Approximation algorithms for bin packing a survey. In D. Hochbaum, editor, *Approximation algorithms for NP-hard Problems*, pages 46-93. PWS Publishing Company, 1997.
- Coffman E.G.. An introduction to combinatorial models of dynamic storage allocation. *SIAM Rev.*, 25:311-325, 1983.
- Coffman. E. G. Jr. and G. S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms*. Wiley, New York, 1991.
- Coppersmith D. and P. Raghavan. Multidimensional on-line bin packing: Algorithms and worst case analysis. *Oper. Res. Lett.*, 8:17-20, 1989.
- Csirik J. and A. Van Vliet. An on-line algorithm for multidimensional bin packing. *Oper. Res. Lett.*, 13:149-158, 1993.
- Csirik J. and G. G. Woeginger. On-line Packing and Covering Problems. In A. Fiat and G. Woeginger, editor, *Online Algorithms The State of the Art*, pages 147-171. Springer, 1998.
- Csirik J., J. B. G. Frenk, A. Frieze, G. Galambos, and A. H. G. Rinnooy Kan. A probabilistic analysis of the next fit decreasing bin packing heuristic. *Oper. Res. Lett.*, 5:233-236, 1986.

Garcia A.. Probability and Random Processes for Electrical Engineering. Addison Wesley, 1994.

Garey M. R. and D. S. Johnson. Computers and Intractability (A Guide to the theory of NP-Completeness. W. H. Freeman and Company, San Francisco, 1979.

Garey M.R., R.L. Graham, D.S. Johnson, and A.C.C. Yao. Resource constrained scheduling as generalized bin packing. J. Comb. Th. Ser. A. 21:257-298, 1976.

GlobespanVirata, Inc., "DSP algorithm library", September 2000,
<http://www.virata.com/products/dspsoftware.html> (accessed 11 January 2002).

Hoffman U.. A class of simple stochastic online bin packing algorithms. Computing, 29:227-239, 1982.

Kou L.T. and G. Markowsky. Multidimensional bin packing algorithms. IBM J. Research and Development, 21:443-448, 1977.

Lee C. C. and D. T. Lee. A simple on-line bin packing algorithm. J. Assoc. Comput. Mach., 32:562-572, 1985.

Lee E.A. and D.G. Messerschmitt. Static Scheduling of Data Flow Programs for Synchronous Digital Signal Processing. IEEE Transactions on Computers, January 1987.

Li K. and K. H. Cheng. Heuristic Algorithms for On-line Packing in Three Dimensions. J. of Algorithms, 13:589-605, 1992.

Minoli D. and E. Minoli. Delivering Voice over IP Networks. Wiley, New York, 1998.

Parhi K. and D.G.Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via. Optimum Unfolding. IEEE Transactions on Computers, Feb. 1991.

Ramanan P., D. J. Brown, C. C. Lee, and D. T. Lee. On-line bin packing in linear time. *J. Algorithms*, 10:305-326, 1989.

Richey M. B.. Improved bounds for harmonic-based bin packing algorithms. *Disc. Appl. Math.*, 34:203-227, 1991.

Woeginger G.. Improved space for bounded-space, on-line bin-packing. *SIAM J. Disc. Math.*, 6:575-581,1993.