EXPLORING REGISTER FILE AND MEMORY ORGANIZATION IN ASIP SYNTHESIS

by

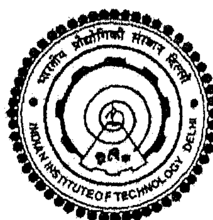
MANOJ KUMAR JAIN

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Submitted

in fulfillment of the requirement of the degree of Doctor of Philosophy

to the

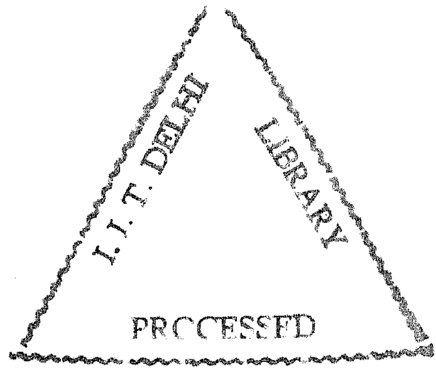


Indian Institute of Technology Delhi

September 2003

Operating systems; Computer design; ASIP operating systems; Computer file organisation

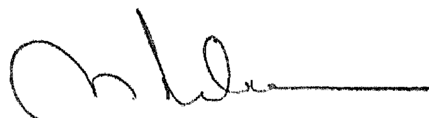
I. I. T. DELHI.
LIBRARY
Acc. No. TH-3014



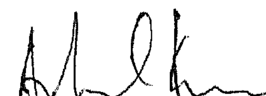
TH
681.326.3
JAI-E

Certificate

This is to certify that the project titled "**EXPLORING REGISTER FILE AND MEMORY ORGANIZATION IN ASIP SYNTHESIS**", being submitted by **Manoj Kumar Jain**, in partial fulfillment for the award of **Doctor of Philosophy**, is a record of the bonafide work carried out by him under our guidance and supervision at the Department of Computer Science and Engineering, **Indian Institute of Technology Delhi**. Also, the work presented in this project has not been submitted elsewhere, either in part or in full, for the award of any other degree or diploma.



Prof. M. Balakrishnan



Prof. Anshul Kumar

Department of Computer Science & Engineering,

Indian Institute Technology Delhi

New Delhi, India

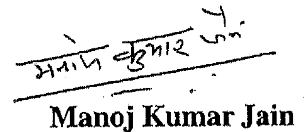
Acknowledgments

I am grateful to my supervisors **Prof. M. Balakrishnan** and **Prof. Anshul Kumar** for their all time, guidance and support. I cannot express my regardful thanks to them in words. Next, I want to thank other research scholars of the **Embedded Systems Group**, namely **Anup Gangwar**, **Basant Dwivedi** and **Rajeshwari Banakar** for their consistent co-operation.

I would like to thank project sponsors, namely Navel Research Board, Government of India and Department of Science & Technology, Government of India. I have worked in a cooperation project between our group and embedded systems group at University of Dortmund, Germany. I am grateful to **Prof. Peter Marwedel** for their valuable guidance. I want to thank research scholars, namely **Lars Wehmeyer** and **Stefan Steinke** of that group for their cooperation during my Dortmund visit and afterwards. It was a nice opportunity for me to work with that group and I enjoyed the same.

I am thankful to all other faculty members and staff members of the department of computer science and engineering at IIT Delhi, for their help throughout my research work. I am also thankful to M. L. Sukhadia University, Udaipur (Rajasthan), India for providing me study leave to persue PhD degree from I.I.T. Delhi.

Last but not least, I would like to mention that blessings of my father **Sh. Naresh Chandra Jain** was one of the most important secret of my success. I would like to devote my thesis to my late mother **Dhapu Bai** as it was her dream which had come true now. Patience of my wife **Sunita** and my sons **Ronak** and **Yash** played a key role to enable me for devoting sufficient time for research.



Manoj Kumar Jain

Abstract

An Application Specific Instruction Set Processor (ASIP) is a processor designed for one particular application or for a set of specific applications. An ASIP exploits special characteristics of the given application(s) to meet the desired performance, cost and power requirements. A typical ASIP design flow includes key steps such as application analysis, design space exploration, instruction set generation, code generation for software and hardware synthesis [1]. Performance estimation which drives the design space exploration is usually done by simulation. Such technique needs a retargetable compiler to generate code for different processor configurations to be explored and then simulating the generated code. This process is generally very slow. Further, there is a well known trade off between retargetability and code quality in terms of performance and code size when compared to the hand optimized code. This is because when the design space is large, all possible target specific optimizations can not be performed in that case. Therefore, in our opinion, compiler-simulator based approach is not suitable for an early design space exploration.

In the domain of Application Specific Instruction set Processors (ASIP), this problem can be solved by scheduler based approaches, which are much faster. However, existing scheduler based approaches do not help in exploring storage organization. We demonstrated the importance of including storage organization in design space exploration through a study using a retargetable code generator encc and a standard simulator. We studied the impact of register file size on performance, code size, power and energy consumption [2] for selected benchmarks on ARM7TDMI processor. Results indicated that choice of an appropriate number of registers has a significant impact on performance and energy in many cases.

The major contribution of this thesis is a scheduler based approach to explore register file size, number of register windows and cache memory configurations in an integrated manner. Proposed technique estimates the cycle count for application execution on chosen processor and memory configuration. We consider a parameterized model for processor as well as memory. Performance for different register file sizes are estimated by predicting the number of memory spills and its delay. The technique developed does not require explicit register assignment. We estimate register needs

on unscheduled code using the concept of reuse chains with significant extensions [3]. Cost due to register spills are considered while merging the reuse chains. Proposed technique also considers the global register needs.

For an architecture with register windows, the number of context switches leading to spills are estimated for evaluating the time penalty due to a limited number of register windows. A sequence of function calls and returns during the execution of the application is generated by instrumenting the input application. Number of window spills and restores required for a specific number of windows is computed by stack based analysis of generated trace of function calls and returns. We observe that usually the number of memory locations required to store spilled scalar variables and register windows is small compared to the total number of cache locations. Therefore, the spilling overhead is insensitive to the cache organization. This observation allows us to estimate the two independently. We use sim-cache simulator of simplescalar tool set to know cache misses statistics. Once we know the number of memory misses for a particular cache, based on the block size and delay information we compute the additional schedule overhead due to cache misses.

Performance estimates for a range of register file sizes, register windows and cache memory configurations are generated for selected benchmarks for different processors. The processors include ARM7TDMI, LEON [4] and Trimedia (TM-1000). Experiments showed that our estimates were within 9.6%, 9.7% and 3.3% for these processors respectively, compared to the actual performance results produced by standard tool sets. Further, this technique was nearly 77 times faster compared to simulator based approach.

We have shown utilization of our approach by a case study on LEON processor. We used the addresses of the registers which are decided to be 'spare' by our technique in addressing coprocessor registers. This simplified the coprocessor interface to the LEON processor and saved a number of loads and stores. Work presented here is integrated in an overall ASIP Synthesis Methodology named ASSIST being developed at IIT Delhi.

Contents

List of Figures v

List of Tables vii

1 Introduction and Objectives 1

1.1 Application Specific Instruction Set Processor 1

1.2 Steps in ASIP Synthesis 2

1.2.1 Application Analysis 3

1.2.2 Architectural Design Space Exploration 4

1.2.3 Instruction Set Generation 5

1.2.4 Code Synthesis 5

1.2.5 Micro-architecture Design and Hardware Synthesis 5

1.3 Variations in the Design Flow 5

1.4 Objectives 6

1.5 Overall Approach 6

1.6 Organization of Thesis 7

2 Related Work 9

2.1 Application Analysis 9

2.2 Architectural Exploration 10

2.2.1 Broad Architectural Features 11

2.2.2 Performance Estimation 12

2.2.3 Search Control 15

2.3 Instruction Set Generation 17

2.3.1 Instruction Set Synthesis 17

2.3.2 Instruction Selection 18

2.3.3 Instruction Set Extension 21

2.4 Code Synthesis 22

2.4.1 Retargetable Code Generator 22

2.4.2 Compiler Generator 25

2.5 Micro-architecture Design and Hardware Synthesis 27

2.6 Conclusion 27

3 Impact of Register File Size on Performance and Power Metrics **29**

3.1 Experimental Setup 29

3.1.1 The *ARM7TDMI* Processor 30

3.1.2 Benchmark Suite 30

3.1.3 The ***encc*** Compiler 31

3.1.4 Power Model 32

3.1.5 Experimentation with Register File Size 33

3.2 Results 34

3.2.1 Number of Executed Instructions 34

3.2.2 Number of Cycles 35

3.2.3 Ratio of Spill Instructions to Total Static Code Size 36

3.2.4 Average Power Consumption 37

3.2.5 Energy Consumption 39

3.2.6 Analysis of Results 41

3.3 Conclusion 45

4 Overview of Our Methodology **47**

4.1 ASSIST : ASIP Design Methodology 47

4.2 Storage Space Exploration 49
4.3 Execution Time Estimation with Limited Registers 51
4.4 Our Approach and Compilers 54
4.5 Conclusion 55

5 Execution Time Estimation at the Basic Block Level **57**
5.1 Register Allocation using Register Reuse Chains (RRCs) 58
5.2 Generating Initial Register Reuse Chains 59
5.3 Converting Initial Chains into Dependence Conservative Chains 61
5.4 Computing Merging Costs 64
5.5 Chain Merging and Performance Estimation 66
5.6 Complexity Analysis 67
5.7 Conclusion 68

6 Global Performance Estimation and Validation **69**
6.1 Addressing Global Register Needs 69
6.2 Illustrative Example 73
6.3 Experimental Setup 74
6.3.1 ARM7TDMI 74
6.3.2 TM-1000 74
6.3.3 Benchmark Suite 74
6.3.4 Processor description 75
6.4 Results of performance Estimations 75
6.5 Validation 76
6.5.1 Validation for ARM7TDMI 77
6.5.2 Validation for *TM-1000* 78
6.6 Limitations 79
6.7 Conclusion 80

7 Register Windows and Cache Memory Exploration **81**

7.1 Estimating Register Window Spills 81

7.1.1 Results 83

7.2 Cache Miss Overheads 86

7.2.1 Results 88

7.3 Execution Time Validation 88

7.4 Conclusion 91

8 Illustrative Case Studies and Applications of Our Approach **93**

8.1 ADPCM Encoder and Decoder Storage Exploration 93

8.2 Collision Detection Execution Time Validation 94

8.3 Applications 96

8.3.1 Reducing Bits in Instruction 97

8.3.2 Alternate use of Spare Registers 97

8.4 Hardwiring some Constants to the Spare Registers 98

8.5 Utilizing Spare Register Addresses to Interface Co-processors 99

8.6 Conclusion 101

9 Conclusions and Future Work **103**

9.1 Summary of Our Contributions 103

9.2 Limitations and Future Work 104

Bibliography **105**