An intergrated soft-and hard- programmable multithreaded architecture

Shi Zhong

Doctor of Philosophy

Institute for Computing Systems Architecture

School of Informatics

University of Edinburgh

2007

Abstract

This thesis investigates an asynchronous, miconet-based design model for multithreaded architecture with reprogrammable hardware elements, targeted at energy-conscious, high performance application in the mobile wireless sector.

We envisage that a synergistic combination of these features will bring benefits to microprocessor architecture design. A novel architecture named Micronet-based Asynchronous Processor System plus Reconfigurable Function Units (MAPS+) is described in detail in this thesis, which combines hard-programmability, in the form of field programmable logic, and soft-programmability in a multithreaded instruction set architecture.

Compiler techniques for extracting speculative thread-level parallelism and hardware-software partitioning were investigated in the thesis. The simulation results for a subset of the MiBench benchmarks on an event-driven instruction set simulator of the MAPS+ architecture demonstrates the performance improvement of different combination of architecture design techniques and the trade-offs between performance and power consumption.

Acknowledgements

I would like to thank my supervisor, D.K.Arvind, for his guidance, support, and encouragement throughout my doctoral study. Arvind has provided me with lots of opportunities to learn. I am also thankful to Arvind for providing good facilities and financial support throughout my research. My work was sponsored by studentship from Xilinx and an Overseas Research Student Scholarship. Thanks to Hamish Fallside for his support.

I would like to thank the colleagues in the Institute for Computing Systems Architecture, in particular, Irwin Kennedy for inspiring conversations regarding the FPGA architectures, and Jae Hosell for providing the memory module for the SPAMSIM2 simulator.

I would also like to thank my friends Andrew, Michelle, Jeff, Ross, Gary Clemo, Tim Farnham, and Tim Lewis at Bristol for their support and advice.

Most of all, I would like to thank Mum, Dad, and my wife Mei, for their continual support and patience. Also thanks to my sister and brother in law for looking after our parents in China.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Shi Zhong)

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Problem Description

According to Moore's law, the number of transistors per square inch of integrated circuits had doubled every year since the integrated circuit was invented [101][102][103], and the circuit density has doubled approximately every 18 months. The scaling of process geometries enables more transistors to be placed in a single chip, with the integration of different types of devices and functions. As a result, modern high performance microprocessors, e.g. Intel's Pentium IV [104], have more than 100 million transistors on-die, and reached clock frequencies of more than 3.4 GHz. This trend will continue for in the foreseeable future, leading to chips with a billion gates per square centimetre.

It is expected that silicon-based bulk Complementary Metal Oxide Semiconductor (CMOS) technology will continue to be the mainstay of the microelectronics industry for the next decade. The International Technology Roadmap for Semiconductors (ITRS) report (2005 edition) [105] has indicated improvements in technology scaling and processor performance. The forecast of decreasing half-pitch die sizes of processors, Random Access Memories (RAMs) and flash memories in the next 15 years is shown in Figure 1-1. It is expected that by the year 2020, CMOS technology will have reached 14nm node and the on-die transistor count will cross 1.1 billion by the year 2012, and reach up to 7.2 billion by 2020. However, performing more and more computations in less space at faster speeds presents enormous technical challenges.

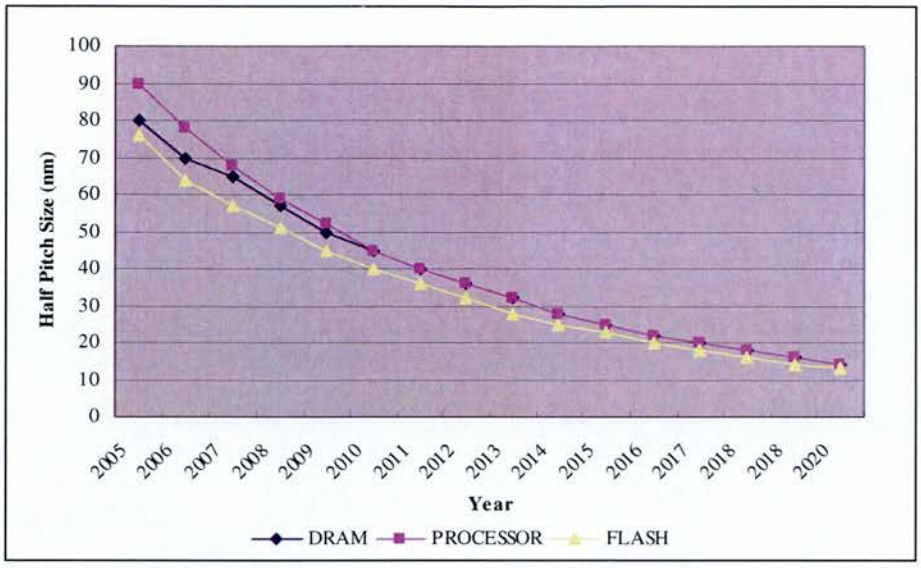


Figure 1-1: International Technology Roadmap for Semiconductors report, 2005 edition [105].

The trends described above are accompanied by high clock frequency domain implementation. Local clock frequencies are predicted to rise above 10 GHz by the year of 2011, to reach 88 GHz by 2020 [105], which will make it difficult to maintain global synchronisation. At present multiple clock frequencies are usually only required to support off-chip accesses. In the future the number of individual clock domains and the associated synchronisation problems are likely to increase considerably. Even with a move towards multiple clock domains, the generation and distribution of clocks with cycle times of much less than a nanosecond will be problematic. Significant resources are already required to analyse and construct clock distribution networks. Traditionally, the clock has been viewed as a simplifying assumption providing opportunities to explore predictable behaviour and minimise critical paths dominated by gate delays. The key requirement in applying a synchronous approach efficiently is the ability to accurately predict on-chip delays, as synchronous clock frequencies must be set by considering worst-case delays. Difficulties in predicting delays and an increase in data-dependent delays will mean that, on average, the amount of useful work performed in a clock cycle will drop.

Increases in clock frequencies, and greater transistor counts have also resulted in a sharp growth in power consumption per chip. Higher power consumption means

higher power costs and shorter battery lifetime of mobile applications. Despite the fact that the supply voltage is decreasing for each chip generation, the total power consumption is rising. For example, the average power consumption for high-end microprocessors has already reached 150 W, and may exceed 300 W by the year 2018 [105]. This increase in power will be largely due to rise in leakage current, higher on-die clock frequency and higher levels of integration. It is expected that the power supply voltage will reach about 0.7 V by 2018, resulting in an increase in current from the present levels of about 130 A, to more than 400 A by 2018. Techniques for reducing and managing power have become important factors in the design of all high-performance Very Large Scale Integration (VLSI) systems due to the problems of both supplying power and the costs associated with cooling the devices. Many of the current low-power techniques have focussed on reducing the power dissipated when transistors switch, which include: clock and signal gating, the use of multiple on-chip supply voltages, the dynamic scaling of supply voltage and more recently, dynamic reconfiguration of components to minimise load capacitances. Additional techniques will also be required to handle the predicted increase in power due to leakage currents, which is increasing by around a factor of five per process generation.

As the complexity of VLSI circuit grows, system-wide synchronisation becomes infeasible due to the lack of robustness in the fact of manufacturing variability, as the CMOS transistor is subject to ever larger statistical variability in its behaviour. Furthermore, it is difficult to try all combinations of inputs systematically and to verify that the resulting behaviour is correct, and to analyse them accurately and exhaustively for precise timing characteristics. When device sizes decrease to a point some defects are unavoidable, such as minor imperfection in the manufacturing process, random thermal fluctuations, or even cosmic rays can invalidate parts of a circuit. Although some of the modern circuits do have error detection and correction capabilities, these solutions are still not universally used in all parts of a processor design.

Complex Electronic Design Automation (EDA) tools are required for synthesising

circuits described in a Hardware Description Language (HDL). However the complexities of designing circuits in traditional HDL prolongs the time-to-market of a product. Languages such as System C [189], Handel-C [190], and JHDL [191] raise the level of abstraction for specifying system-on-chip designs, with the ability to synthesise hardware.

1.2 Aim of the thesis

The aim of the thesis is to tackle some of problems described in previous section. It investigates the impact of combining asynchronous design techniques, multithreaded executions and runtime reconfigurations in microprocessor architectures, which could lead to some solutions for existing problems in VLSI system design. A novel architecture named MAPS+ is explored in detailed, which combines hard-programmability, in the form of field programmable logic, and soft-programmability in a multithreaded instruction set architecture. Compiler techniques for extracting instruction-level parallelism (ILP) and thread-level parallelism (TLP) have also been investigated in the thesis. Based on the simulation results, the MAPS+ architecture demonstrates performance improvement for different combination of architecture design techniques and power consumption trade-offs.

1.3 Thesis outline

The remaining chapters in the thesis are summarised as follows:

Chapter 2. This chapter introduces ideas from three background areas of research which overlap in this thesis: asynchronous design, multithreaded architectures and reconfigurable computing. Firstly, the role of asynchronous control in architectural design is introduced, along with the advantages and disadvantages of such an approach. An overview of previous work in this area is also presented. Secondly, the multithreaded architecture is described, which includes multithreaded execution, comparison between two multithreaded architectures, thread extraction mechanisms

and existing multithreaded processor designs. Finally reconfigurable computing concepts are also reviewed, and related issues are introduced, such as FPGA architectures with different coupling mechanisms, and compilation and evaluation techniques.

Chapter 3. The MAPS+ architecture is described in detail. The MAPS+ architecture combines hard-programmability, in the form of field programmable logic, and soft-programmability in a multithreaded instruction set architecture. Techniques enabling thread synchronisation, data value prediction and runtime reconfiguration are described.

Chapter 4. The compilation techniques for generating multiple-threaded and hardware-software partitioning codes are described in this chapter. Conventional loop threading compilation and speculative thread partitioning techniques and transformations have been investigated. Finally, automatic compilation techniques of generating RFU operations for grouping arithmetic and logic operations are also investigated.

Chapter 5. The chapter presents a simulation framework for the MAPS+ architecture and its compilation framework. The SPAMSIM2 simulator is introduced, which provides an integrated environment to model the functionalities of MAPS+, and to evaluate its performance. The MAPS+ compiler is based on the SUIF2 and Machine SUIF compilers. The Software Hardware Partitioner (SHP) and the Thread Analyser (TA) compiler passes are also described, which implements the algorithms for hardware-software and thread-level partitioning.

Chapter 6. The chapter summarises the benchmark programs that have been chosen for evaluating the MAPS+ architecture. The configurations for the synchronous baseline and the asynchronous MAPS+ architecture are introduced. With the SPAMSIM2 simulator, performance, power and energy consumption results are

evaluated. Four sets of MAPS+ experiments were executed: asynchronous MAPS against synchronous MIPS baseline; asynchronous MAPS+ RFU against synchronous MIPS; asynchronous multithreaded MAPS against synchronous MIPS; asynchronous multithreaded MAPS+ RFU against synchronous MIPS.

Chapter 7. This chapter presents conclusions for the thesis and proposes future work to enhance the MAPS+ compiler for threading partitioning and hardware software partitioning.

Chapter 2
Background

2.1 Introduction

This chapter reviews three background areas of research for the MAPS+ architecture design: asynchronous design, multithreaded architectures and reconfigurable computing.

Firstly, the role of asynchronous control in architectural design is introduced, along with the advantages and disadvantages of such an approach. An overview of previous work in this area is also presented.

Then the principles of multithreaded architecture are described. Two design mechanisms dominate current multithreaded architectures: Simultaneous Multithreading (SMT) and Chip Multiprocessors (CMP). The features of existing multithreaded architectures are also summarised, both academic and commercial ones.

Finally concepts in reconfigurable computing concepts are reviewed, drawing comprises between tightly-coupled and loosely-coupled reconfigurable architectures. And the compilation and evaluation techniques for these architectures are also introduced.

2.2 Asynchronous Design

Although asynchronous circuit design was first investigated in the 1950's by Huffman and Muller, commercial circuit design has been dominated by synchronous technology.

Synchronous design is characterised by certain fundamental assumptions: all

signals are binary and time is discrete. The clock signal in a synchronous circuit serves as a global timing reference for communicating data among the different units in a system. The data communication is via pipelined latches, which use the clock as an enabler signal.

The asynchronous design provides an alternative approach to circuit design. In an asynchronous system there is no notion of a clock and all communication is explicit using a channel that follows a handshaking protocol: the sender, after making the data available, sends a request signal notifying the receiver, which processes (or stores) the data and sends an acknowledgement back when it has finished. Advantages of asynchronous design are summarised as follows:

- **No clock skew**: Asynchronous designs avoid problems due to clock skew in the absence of a clock.
- **Modularity**: The asynchronous designs can be improved by modifying part of the circuit without disturbing the rest of the circuit. In theory, the new blocks only need to obey the communication protocol of the interface.
- **Robustness**: Delays in circuits can vary across different fabrication processes and operating conditions. Asynchronous circuits are more tolerant to variants in physical parameters, such as temperature, and power supply. The asynchronous circuits do not have critical timing requirements, and can run as fast as the operating conditions allow. Furthermore, they are able to guarantee the correctness of their functionality regardless of the operating conditions.
- **Low peak electromagnetic emissions**: Electromagnetic emissions of circuits are sources of side-channel leakage, which bring security threats to allow secret information stored in cryptographic devices to be retrieved [117][118]. Asynchronous design reduces peak electromagnetic emissions and provides the possibility to resist side-channel attacks [118].

However, asynchronous designs do have a number of disadvantages:

- **Design complexity**: The lack of global timing in the design of asynchronous

circuits does introduce several problems, such as handling signal sequentially, avoiding hazards and non-determinism, in order to ensure the correct behaviour. As a result, different styles of asynchronous circuits have emerged, e.g. DI [114][116] circuits, SI [116] circuits and QDI [115] circuits.

- **Completion detection**: By removing the global timing reference, the asynchronous approach shifts the problem to generating a completion signal of any operation. Extra hardware is required to maintain synchronisation among blocks, which increases the complexity of the design process.
- **Lack of design tools:** Commercial Electronic Design Automation (EDA) tools for design, simulation, synthesis, routing and verification of synchronous circuits have been well developed. In contrast, tools and methodologies for asynchronous design have been available in the academic community, industrial strength tools are scarce.
- **Testing problem**: Testing asynchronous circuits is considered a difficult task. Asynchronous circuits make extensive use of handshaking and the presence of a fault is likely to cause the circuit to halt. Given the large scale space, exhaustively testing of circuits are impractical. The characteristics of state-holding elements together with the self-timed behaviour make it hard to test the feed-back circuitry.
- **Performance measurement**: For an asynchronous circuit, the time for completing a task will depend on hardware delays and on the input data, which means that the performance measurement is variable and the performance metric is based on the average measure.

Apart from the points summarised in the previous sections, it is still unclear that asynchronous circuits consume lower power than the synchronous counter part. Traditionally, lower power consumption has been the most cited benefit of asynchronous design. The absence of the global clock signal in asynchronous circuits causes power consumption to be more evenly distributed over time. Also, an asynchronous system activates only those parts of the circuit which are required and the rest parts of the circuit that is not being used does not dissipate any power.

However, it becomes debatable while clock gating power-saving [120] techniques and more advanced chip fabrication processes (e.g. 60nm, 45nm) are introduced to synchronous circuits. Clock gating techniques add additional logic to a circuit to prune the clock tree and disable portions of the circuit. Our simulation results in Chapter 6 show similar level of power consumptions of the asynchronous designs to their synchronous counterpart.

Also average case performance is another advantage of asynchronous designs used to be stated. Synchronous designs cannot avoid worst-case performance since all possible computations must complete before results can be latched. Thus, a margin of time is added to the clock speed to ensure that all blocks completed. By comparison, asynchronous designs are free from this dilemma since the circuitry is capable of sensing a computation's completion. However, with the improvement of fabrication technology, a long and more precise pipeline design of processors, performance of synchronous systems becomes very close to the asynchronous cases.

In the following sections, different asynchronous design methodologies are discussed.

2.2.1 Asynchronous circuits classification

In asynchronous circuits, an event is a transition in the logic level due to changes in the value on the wire. Due to varying delays through different logic paths, a particular wire in a block of combinational logic may perform a number of transitions before reaching a stable value. For an asynchronous circuit, it is important to ensure such effects do not cause the circuit to malfunction. As a result, asynchronous circuits need to make assumption about wire and gate delays. The models of asynchronous circuits with different levels of timing assumptions are listed below:

Delay-insensitive (DI) circuits

In DI [114][116] circuits all transitions on gates or wires must be acknowledged

before transitioning again. This condition stops unseen transitions from occurring, and any transition on an input to a gate must be seen on the output of the gate before a subsequent transition on that input is allowed to happen. This forces some input states or sequences to become illegal. For example OR gates must never go into the state where both inputs are one, as the entry and exit from this state will not be seen on the output of the gate. Though the model is robust, the heavy restrictions cause the DI circuits to be impractical.

Quasi delay-insensitive (QDI) circuits

QDI [115][116] circuits make no assumptions about the delays of any of the circuit's elements, except to assume that certain fanouts are isochronic forks. Isochronic fork allow signals to travel to two destinations and only receive an acknowledgement signal from one. Both ends of isochronic forks see the transitions, such as the acknowledging destination and the other end. Two kinds of isochronic forks have been proposed, such as asymmetric and symmetric types. The asymmetric types ensures that the signal will arrive at the acknowledging fork destination before or at the same time as it will at the other fork destination, but the symmetric type ensures that both fork destinations will be reached at the same time.

Speed-independent (SI) circuits

In SI [116] circuits, wire delays are assumed to be zero, or less than the minimum gate delay and the circuit exhibits correct operation regardless of the delays in any circuit elements. The assumption of zero wire delay is valid for small circuits.

2.2.2 Asynchronous communication and signalling

In the asynchronous system, components communicate with each other via a

handshaking mechanism. In this scheme the sender is responsible for initiating the transaction and the receiver responds when it is ready to receive.

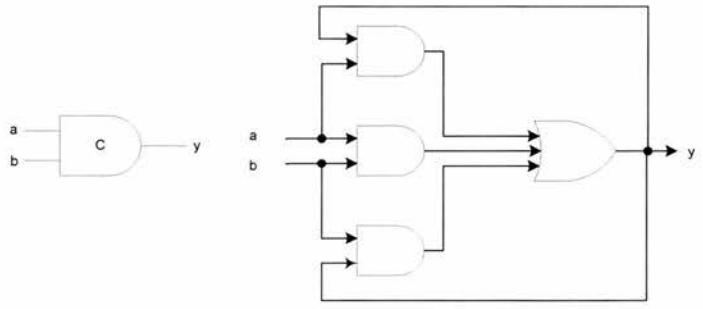
The most efficient signalling convention is two-phase handshaking protocol. Consecutive signals or events are indicated by alternating low-to-high and high-to-low voltage transitions. The term two-phase [106] stems from the fact that two events take place: the first phase is represented by the sender requesting transfer data (1), and the second phase by the actual transfer of the data (2), as depicted in Figure 2-1(a). The major advantages of two-phase handshaking, also known as transition signalling or Non Return-to-Zero (NRZ) signalling, are that it is as fast and as energy efficient as possible. However, in practice, additional logic and state information may be required in each element, since logic devices tend to be sensitive to voltage levels or only transitions in a particular direction.

The four-phase [107][108][109] signalling protocol uses the level of the signalling wires to indicate the validity of data and its acceptance by the receiver. When this signalling scheme is used to pass the request and acknowledge timing information on a channel, a Return-to-Zero (RZ) phase is necessary so that the channel signalling system ends up in the same state after a transfer as it was in before the transfer. As shown in Figure 2-1 (b), four events take place in the case of the four-phase: (1) the sender starts the transactions, (2) the receiver acknowledges, (3) the sender stops sending the data, and (4) the receiver finishes the handshake. This scheme thus uses twice as many signalling edges per transfer than its Two-phase counterpart. Another characteristic of four-phase handshakes scheme is that the second half of the handshake can be concurrent with the computation. This is advantageous considering that transactions spend most of the time in computation rather than communication. Four phase circuits can achieve higher performance and lower costs than two phase implementations using level-sensitive technologies such as CMOS.

Figure 2-1: (a) Two-phase of handshake and, (b) four-phase of handshake.

2.2.3 Muller C-element

The Muller C-element [48] applies logical operations on the inputs and has relatively simple gate logic design. Figure 2-2 shows the Muller C-element symbol, gate-level implementations, and the truth table. The Muller C-element acts as the AND element for events. As shown in the truth table, if both inputs are matched, its output reflects that state. If the two inputs differ, the output retains its previous state using its internal storage. When a transition is triggered, the Muller C-element will produce an event at its output port. Such a model can be extended to the asymmetric C-element and some inputs only affect the operation in one of the transitions.



a	b	y
0	0	0
0	1	No change
1	0	No change
1	1	1

Figure 2-2: (a) Muller C element, (b) Gate level implementation and (c) Truth table.

2.2.4 Data representation

The handshaking protocol described in Section 2.2.2 can also be used to transmit data. As the simultaneous arrival of both data and request signals cannot be guaranteed under a delay-insensitive model, a technique is required to detect when data is present. The arrival of a particular bit of data is only possible if it produces an event on a wire.

If the data was sent un-encoded this would mean that only those bits changed could be detected.

The dual-rail encoding scheme [110] uses two wires to represent each bit of information. Each transfer will involve activity on only one of the two wires for each bit, and a dual-rail circuit thus uses 2n signals to represent n bits of information. Timing information is also implicit in the code, in that it is possible to determine when the entire data word is valid by detecting a level (for 4-phase signalling) or an event (for 2-phase signalling) on one of the two rails for every bit in the word. A separate signalling wire to convey data readiness is therefore unnecessary. Four-phase dual-rail data encoding is popular for the QDI design style but, as with all dual-rail techniques, it carries a significant area overhead in both excess wiring and the large fan-in networks that it requires to detect an event on each pair of wires to determine when the word is complete and the next stage of processing can begin. In practice, the dual-rail encoding needs to detect when all of the bit-lines have returned to zero, for which the AND gates must be replaced by Muller C-elements as described in the previous section.

An alternative approach is to use the bundled-data [111] scheme to detect the presence of data and introduce a safety margin by delaying the request event. Timing information is passed on separate request and acknowledgment lines which allow the sender to indicate the availability of data and the receiver to indicate its readiness to accept more new data. Bundled-data encoding schemes contain inherent timing assumptions in that the delay in the signal line indicating data readiness must be no less than the delay in the corresponding data path.

Instead of using inherent timing assumptions for bundled-data scheme, a completion detection can be achieved using Current-Sensing Completion Detection (CSCD) [112][113]. But it is not typically implemented in bundled-data scheme. CSCD monitors the transient-current flow, inherent in CMOS logic functions during processing of input variables, to detect completion of a given operation. Single-rail design is popular, mainly because its area requirements are similar to those of synchronous design, as is the construction of any arithmetic components using this

scheme.

2.2.5 Micropipeline

Micropipeline [48] is an asynchronous implementation of pipelines. As shown in Figure 2-3, each stage has a bundled data (BD) interface, which is used to communicate with the previous and next stages. Data is supplied at the inputs to the system, and then a transition occurs on the R(in) wire. Then the local handshake signals of each BD interface determine the earliest time at which the next stage may receive the data. Logic blocks between different stages accomplish the computation tasks on input data. The explicit delay elements must match the worst-case logic block delay.

Micropipeline structure provides the benefit by direct implementation of the logic blocks used in the synchronous designs. Therefore, a micropipeline can be constructed from a synchronous pipeline by replacing the clocked level-sensitive latches with the micropipeline control structure. De-synchronisation [121][122] has been suggested, whereby the clock distribution tree of traditional synchronous circuit is replaced by standard handshaking circuits, allowing controllers to be implemented using a direct-mapped method and datapaths are implemented using conversional synthesis and matched delay elements.

Furthermore, because of the removal of lockstep with the global clock, the micropipelined design provides better elasticity, and arbitration timing of data sending and receiving can be guaranteed. The micropipeline also introduces some problems. As the bounded-delay models are used in micropipelines, the data-path suffers worst-case performance. Also testing difficulties are also one of the problems due to the delay assumption made for different pipeline stages.

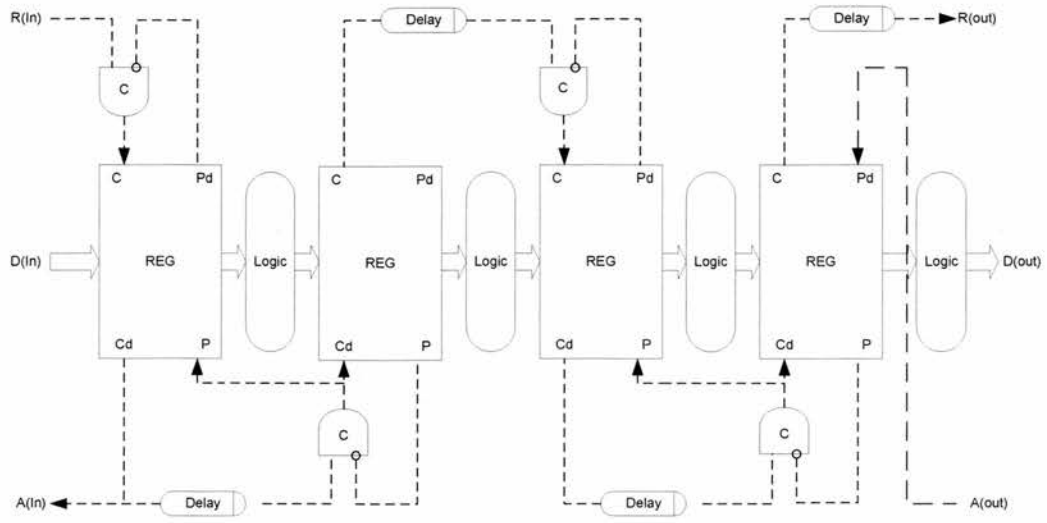


Figure 2-3: Micropipeline block diagram.

2.2.6 Asynchronous architectures

The following section presents an overview of contemporary asynchronous architectures, which includes academic and commercial asynchronous processors.

AMULET

The AMULET group at Manchester University has developed a number of asynchronous ARM-compatible processors. AMULET1 [123] is an asynchronous micropipelined versions of the ARM6 microprocessor. A two-phase, bundled-data communication protocol is implemented. It relied on a lock FIFO to ensure dependencies are respected. The width of the FIFO buffer is equal to the number of registers, whereas the size of the buffer defines the number of pending write operations. Each location consists of a bit that indicates which register is to be written. Control logic ensures that at most one column attempts to write to a register. The AMULET1 design incorporates a number of concurrent units which cooperate to give instruction level compatibility with the synchronous ARM architecture, which includes an Address Unit, a register file, execution unit, and a memory unit. From AMULET group's implementation, the AMULET1 demonstrated a 70% performance

enhancement over an ARM6 at 20MHz, with an overhead of 20% larger area size. It delivered a power efficiency of 80MIPS/W, which was less than ARM6's 120 MIPS/W. However, AMULET1 was a first generation of asynchronous implementation, but the ARM6 was a highly efficient commercial processor which has undergone several design iterations, so the performance was reasonable. The approach did prove the feasibility of a large-scale asynchronous architecture.

The AMULET2 [109] improved on the AMULET1 in several aspects. The AMULET2 used the four-phase handshake scheme, which is faster and more power efficient. AMULET2 provides a flexible pipeline design to allow unnecessary pipeline stage to be bypassed, e.g. the memory access pipeline. This behaviour is advantageous as it prevents the availability of independent ALU results depending on the completion of earlier memory operations. As a result, a separate lock FIFO is introduced for internally generated results and loads from memory. Write-After-Wrote (WAW) hazards may be avoided by stalling the issue of an instruction until its destination register is unlocked. Furthermore, a data forwarding mechanism reduced pressure on the register file and the Branch prediction mechanism reduced the percentage of prefetched instructions that were discarded when a branch was taken. The AMULET2 demonstrated the potential for power efficiency of 280MIPS/W, which was greater than ARM7.

The AMULET3 [124][125] offers similar performance and functionality as the ARM9TDMI microprocessor. A Thumb decoder was incorporated for full compatibility with the Thumb instruction set, and a reorder buffer was incorporated at the write-back stage. The reorder buffer replaced the register-lock FIFO buffer used in previous AMULET designs. It enables data forwarding to be more dynamic and flexible. The forwarding event can take place in parallel with the register write-back. The reorder buffer shortens the path for results normally written and read immediately via the register file, thus reducing the response time for results to be available. If the instruction results do not need to be forwarded, then they written back in order.

Micronet

Micronets [28][29][30] developed at the University of Edinburgh, are networks of entities that compute concurrently and communicate asynchronously. The entities can themselves be networks by a recursive fashion, leading to a practical model of system design, allow mixture of both self-timed and clocked implementation of the entities. As a generalisation of Sutherland's micropipelines, micronets explore both fine-grained instruction-level parallelism and the actual execution costs of instruction in the form by using fine-grain microagents. The micro agents in any stage can operate concurrently, and microagents in the different stages communicate with each other asynchronously. Program instructions only utilise the relevant microagents and for just as long as is necessary.

In normal micropipelined architecture, the number of active instructions is never greater than the number of pipeline stages, and at any time only a subset of the resources in each of the stages is normally utilised. In micronets, the number of instructions which may be active at any time is bounded by the number of microagents. An instruction which does not require any of the resources within a stage can skip it. Furthermore, the time spent by instructions in microagents may vary. Due to these reasons instructions may overtake. This feature will be explored to implement out-of-order instruction completion.

Because there are effectively a number of paths, different instructions need not necessarily complete in the order they were initiated. Also, the micronet is controlled at two levels: the data transfer between the microagents is controlled locally, whereas the choice of micro-operations within the microagent and the destinations of the results are controlled by the control unit or by other microagents. Communication between microagents may occur either across dedicated lines or via shared buses. The micro-operation control signals can also be used to prevent contention on shared buses.

MiniMIPS

The MiniMIPS [132][133] processor based on the MIPS R3000 instruction set, was designed and fabricated at Caltech between 1995 and 1998. The asynchronous circuits of MiniMIPS are QDI, which are less conservative than DI and robust to physical parameters variations. The robustness of QDI makes it possible to exchange energy and throughput against each other through voltage adjustments.

The processor consists of 32 32-bit general-purpose registers, a program counter, and two special-purpose registers for multiplication and division. Two 4KB caches are also included: an instruction cache, and a direct-mapped write-through data cache. In the MiniMIPS, all execution units are connected in parallel in the pipeline, and can execute concurrently. As result, a number of techniques are used to improve the performance, e.g. pipelined completion detection, pipelined caches, and the design of a low-latency adder. Furthermore, techniques to optimise the number of pipeline stages and buffering are also used, while guaranteeing correctness. Results are simply reordered by polling functional units in the order they were used. This technique is similar to the use of a result shift register, and is the simplest way in which precise interrupts may be supported. Data forwarding is supported from one instruction to its immediate successor. The cases when forwarding can take place are detected during decode by maintaining a record of the previous instruction's destination register.

This design was implemented in a full-custom hand layout. It was fabricated in HP's 0.6 μm CMOS process. The power consumption of the processor was reported at 4.2 Watt (3.3 V) at 180 MIPS.

SCALP

The Superscalar low-power processor (SCALP) architecture [131] does not use a global register bank but implements results forwarding schemes. Several instructions

are fetched from memory at a time. Each instruction has a small number of easily decoded bits that indicate which functional unit will execute it. The instruction issuer is responsible for distributing the instructions to the various functional units on the basis of these bits. A queue is associated with each operand required by each functional unit. Queues provide buffering for both instructions and results, reducing the need to stall the issue unit and the operation of functional units.

Though the design was aimed at reducing power consumption by increasing code density and decreasing the overall complexity of the processor, it brought some drawbacks to the architecture. The register-less scheme was hard to implement due to non-deterministic behaviour introduced by control hazards and asynchronous operation. Communication across braches is also problematic as the destination of the result cannot be determined a priori. In this case a register bank functional unit is used. Duplicate instructions are also introduced into the program to allow results to be distributed to more than one destination. Overall performance is said to be lower than expected, due to a combination of poor code density and the inability of the architecture to expose and explore instruction level parallelism.

Philips 80C51 Microcontroller

Philips' asynchronous 8-bit 80C51 [126][127] microcontroller was implemented by using the Tangram toolset, which was a high-level language developed by Philips Research Laboratories. The Tangram program can be compiled automatically into a gate-level netlist, using handshake circuits as intermediate stage. The single-rail bundled-data asynchronous implementation of the microcontroller was fabricated in a 0.5 μm CMOS process and showed a power advantage of a factor 4 compared to a synchronous implementation in the same technology at a cost of twice the silicon area.

The 80C51 microcontroller was built around the bus that acts as the communication channel between any two registers. ALU places its output onto the

bus, the Special Function Registers (SFRs) are the registers that handle the communication between CPU and peripherals; RAM and ROM modules are available as constant arrays, which are mapped onto a dedicated handshake component. These components consist of a handshake wrapper around standard RAM and ROM modules, to which a ready signal is added to provide completion detection of read or write accesses.

PCA-1

Plastic Cell Architecture (PCA) [134][135][136] is a FPGA-like device using asynchronous design proposed by NTT's Network Innovation Labs. As shown in Figure 2-4, PCA is composed of the Built-in Parts (BP) and the plastic parts (PP), and one plastic part has some basic cells. This hardware architecture consists of programmable logics and on-chip network, which is similar to a Field-Programmable Gate Array (FPGA) design.

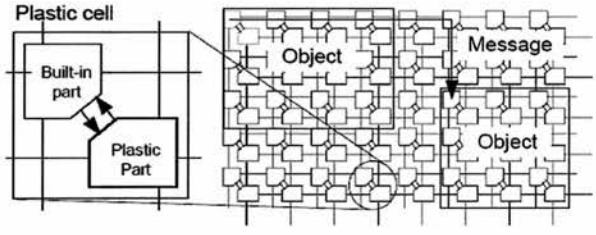


Figure 2-4: Block diagram of a PCA architecture (reproduced from [134])

The PP is a programmable logic block. On the PP, self-timed circuits with 4-phase bundled data signalling are implemented. It includes a 3-bit asynchronous finite state machine. This object implements the function of sending a series of commands to the basic part immediately after the basic part has established a connection with the plastic part. Array of basic cells are also integrated into the PP, which are composed of four, 4-input-and-1-output Look-Up-Tables (LUTs).

The BP acts as switch or router as used in network-on-chip architectures and provides configuration control for the LUT memory in PP side. The BP contains five

input ports and five output ports. Within each input port, asynchronous state machines are used to control the communication with neighbours. Asynchronous arbiters in the output port select the output messages. Wormhole routing algorithms are used to pass the messages around.

The benefit of the PCA asynchronous architecture is the ability to map the Communicating Sequential processes (CSP) [137] model onto it. Concurrency can be explored by dividing a problem into independent subtasks for parallel execution and these tasks are mapped onto the PP. Communication is mapped to the BP with the description of connections and control of programmable objects. The PCA is suitable for Digital Signal Processing (DSP) applications with data parallelisms, but for traditional embedded applications with complex control flows, it still lacks of the processing ability due to the limitation of the reconfigurable logic.

Power consumption is proportional to the number of working objects on the chip around 24 mW per PCA cell. PCA-1 was fabricated using a 0.35 μm CMOS standard cell process operating at 3.3 V. It contains a 6x6 array of PCA cells and with each PP containing 79 K transistors and BP containing 15 K transistors.

Sun UltraSPARC IIIi

Though UltraSPARC IIIi [128] processor was not a fully asynchronous design, it was Sun's first commercial product with elements of asynchronous logic technology incorporated in the synchronous SPARC V9 64-bit architecture. The processor's memory controller includes asynchronous logic-based FIFO circuits in the memory controller input/output section to absorb clock skew variations inherent in large chips containing tens-of-millions transistors.

The processor has a 14-stage nonstalling pipeline that allows the concurrent execution of four instructions per cycle in 6 execution units: 2 ALUs, 2 FPUs, 1 memory unit, and 1 branch unit. The translation look-aside buffer supports 8 KB, 64 KB, 512 KB, and 4 MB pages. The on-chip L1 caches include a 64 KB data cache, a 32 KB instruction cache, a 2 KB data prefetch cache, and a 2 KB write-cache. All

caches are 4-wayset associative. The instruction and data caches have parity protection.

The scalable design of UltraSPARC IIIi makes it possible to be extended to four-way multiprocessing system by using the fast JBUS interconnect between processors, which is a 200 MHz cache coherent SMP bus interface. The chip was fabricated on Texas Instruments' 0.13 μm technology with copper interconnect process, and achieved 1.28 GHz clock speed with average power consumption below 60 Watts.

ARM996HS

ARM996HS [129][130] is ARM's first commercially-available synthesizable 32-bit CPU using asynchronous design technology provided by Handshake Solutions. The ARM996HS is a 32-bit RISC processor core based on the ARMv5TE Instruction Set Architecture (ISA). As shown in Figure 2-5, its core is based on a 5-stage integer pipeline with fast 32-bit multiply-accumulate block. It has tightly coupled memories for both instruction and data, each of which can be up to 4MB. Dual AMBA and AHB-Lite synchronous buses provide the instruction and data interfaces. Specific security enhancements for the ARM996HS core include a memory protection unit (MPU) and the provision of non-maskable interrupts (NMI). A hardware divide co-processor is also provided. The pipeline within the ARM996HS core mirrors the normal ARMv5TE pipeline, with the exception that it is implemented as an asynchronous design. The pipeline handshakes with the system controller to fetch instructions and to load and store data.

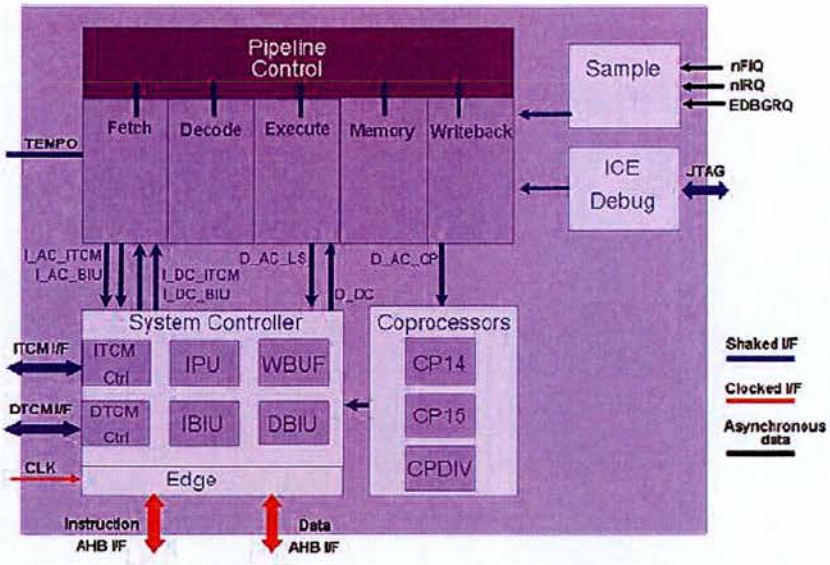


Figure 2-5: ARM996HS Core Block Diagram (reproduced from [129]).

From the ARM's benchmarks, the ARM996HS processor has been shown to consume 2.8 times less power than the clock-gated ARM968E-S core – a conventionally clocked processor with a similar specification to the ARM996HS.

2.3 Multithreaded architectures

Asynchronous architectures introduced in the previous section provide possible solutions for high clock frequency, high power consumption in billion transistor system design. In this section, alternative architectures that explore multiple threads of control are described. A multithreaded architecture is able to pursue two or more threads of control in parallel within the processor pipeline. The definition of a hardware thread is an abstraction that represents an instruction stream that is able to execute independent of another thread [138][139]. A multithreaded processor should provide independent program counter, stack pointer and frame for each thread. In addition to its private thread context, all threads share public resources such as heap storage, shared static memory, and shared variables. Two alternative micro architectures have been proposed to explore the thread level parallelism by researchers: Simultaneous Multithreading (SMT) and Chip Multiprocessor (CMP).

2.3.1 Thread level parallelism

Two parallelism techniques have been investigated in current architectural research: Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). These techniques aim to improve the throughput of a processor by identifying independent instructions that can execute in parallel and therefore can utilise parallel hardware. Instruction level parallelism is usually explored by a multiple-issue superscalar architecture using different hardware and software techniques, such as branch prediction, register renaming, speculative execution, and out-of-order execution. Thread level parallelism allows workload across different threads that can be issued to separate execution pipelines. Thread level parallelism can be explicitly identified by a programmer, or alternatively identified by a compiler or dedicated hardware components.

2.3.2 Simultaneous Multithreading Architecture

SMT [140][141][142] processors extend wide-issue superscalar processors with hardware that allows the processor to execute instructions from multiple threads of control concurrently when possible, dynamically selecting and executing instructions from many active threads simultaneously. SMT has the ability to use TLP and ILP interchangeably for executing parallel applications. For a single threaded program, all of the SMT processor's resources can be dedicated to that thread. Furthermore, when more TLP exists, this parallelism can compensate for a lack of per-thread ILP. The SMT is more like a conventional wide-issue superscalar without thread level parallelism. In order to improve the throughput and keep the SMT pipeline busy, advanced techniques in superscalar processors, such as branch prediction, register renaming, out-of-order instruction issue, and non blocking caches are used. As shown in Figure 2-6, latencies occurring in the execution of single threads are bridged by issuing operations of the remaining threads loaded on the processor.

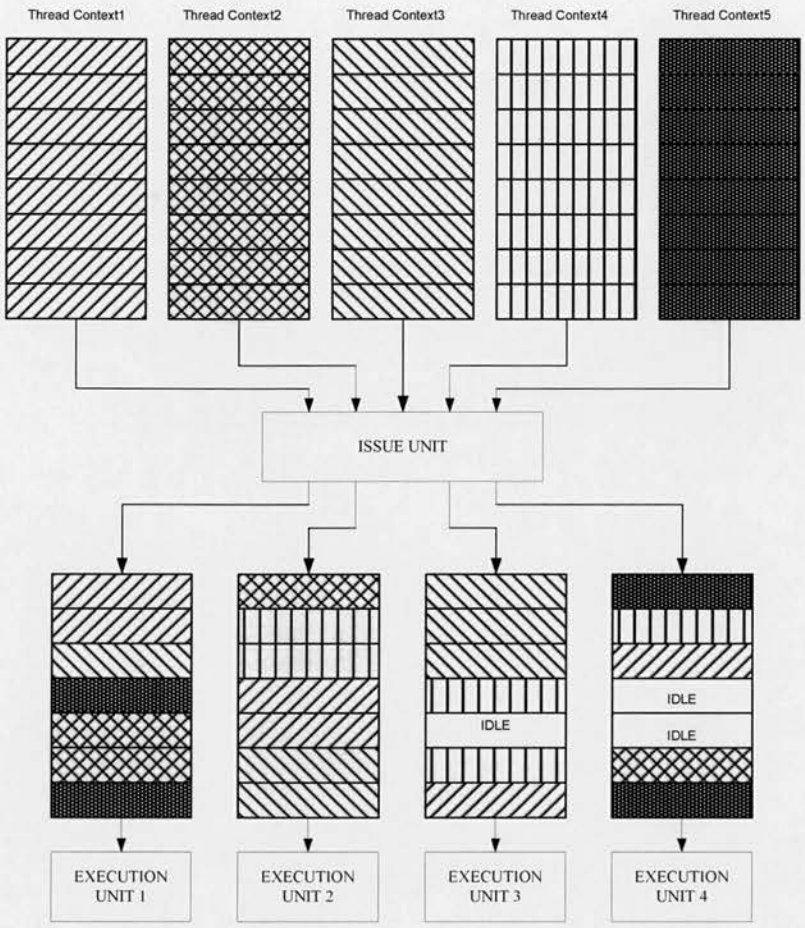


Figure 2-6: Simultaneous Multithreading Architecture.

SMT architectures have several benefits, which are summarised below:

- The major benefit of SMT architecture is that the full issue bandwidth can be utilised. Each thread context competes for SMT execution units to improve the processor throughput. The issue bandwidth can be shared among different thread contexts, and the issue units are able to fetch and issue instructions from several threads in each cycle. Also the issue unit chooses the most appropriate execution unit for execution; the prioritised issue scheme guarantees the load balances across different threads. As a result SMT most effectively utilises the available resources to achieve better throughput and improve program execution speed significantly.
- SMT architectures have less chip area overhead as logical processors are

increased. For example, a Pentium IV hyperthreading (SMT) version just requires 5% more extra transistors for one extra logical processor [146], which are used for multiple program counter, instruction pointer and register files. Obviously, the variation of extra chip area is implementation dependent, which is dependent on the processor architecture chosen and components that need to be replicated.

On the other hand, the SMT architectures have drawbacks due to the complexity of the design:

- The wide-issue stage design of SMT architecture requires a complex issue unit to explore ILP and TLP as much as it can.
- A longer cycle time is necessary for the following reasons. Long and high capacitance I/O wires span the large buffers, queues, and register files. Extensive use of multiplexers and crossbars to interconnect these units adds more capacitance. Delays associated with these wires will probably dominate the delay along the CPU's critical path. Complicated logic requires more clock cycles to keep clock period short and deeper pipeline increases the branch mispredict penalty.
- Multiple threads share the same level-1 cache, TLB and branch predictor unit, which causes contention, resulting in increase in cache misses.
- Finally, from an implementation point of view, the complexity increases design time and verification costs.

2.3.3 Chip Multiprocessor Architecture

An alternative approach to the SMT is the CMP [143][144]. CMP architecture integrates two or more complete processors on a single chip. Every unit of a processor is duplicated and used independently of its copies on the chip. As shown in Figure 2-7, within each core of the CMP, a relatively narrow issue processor is implemented. Each single thread processor provides moderate abilities to explore parallelism and it

is completely independent and tightly integrated with its own cache.

TLP is exploited in CMP by partitioning programs into separate thread contexts, which can run on separate processors. The CMP design has the following benefits:

- CMP increases layout efficiency, resulting in more functional units within the same silicon area plus faster clock rates.
- CMP architecture is much less sensitive to poor data layout and poor communication management, since the inter-processor communication latencies are lower and bandwidths are higher.
- As the interconnects delays are becoming much slower than transistor gate delays with current fabrication technology, it requires the processor architecture to be partitioned into small, localised thread processing units, which favours CMP. Also, due to the replication of thread processing units in the implementation, the design and verification costs are under control.

The drawbacks of a CMP design are:

- The hardware partition of on-chip processors restricts performance. The hardware partition results in smaller resources since the cache, TLBs, branch predictors, and functional units are divided among the multiple thread processing units. Hence single threaded programs cannot use resources from the other processor cores, and the smaller cache size per thread processing unit causes increased miss rate.
- A CMP chip is significantly larger than the size of a single-core chip and therefore more expensive to manufacture.
- The traditional CMP approach of statically partitioning the chip resources between threads may lead to wasted resources when one of the threads stalls due to hazards or when the application lacks threads.

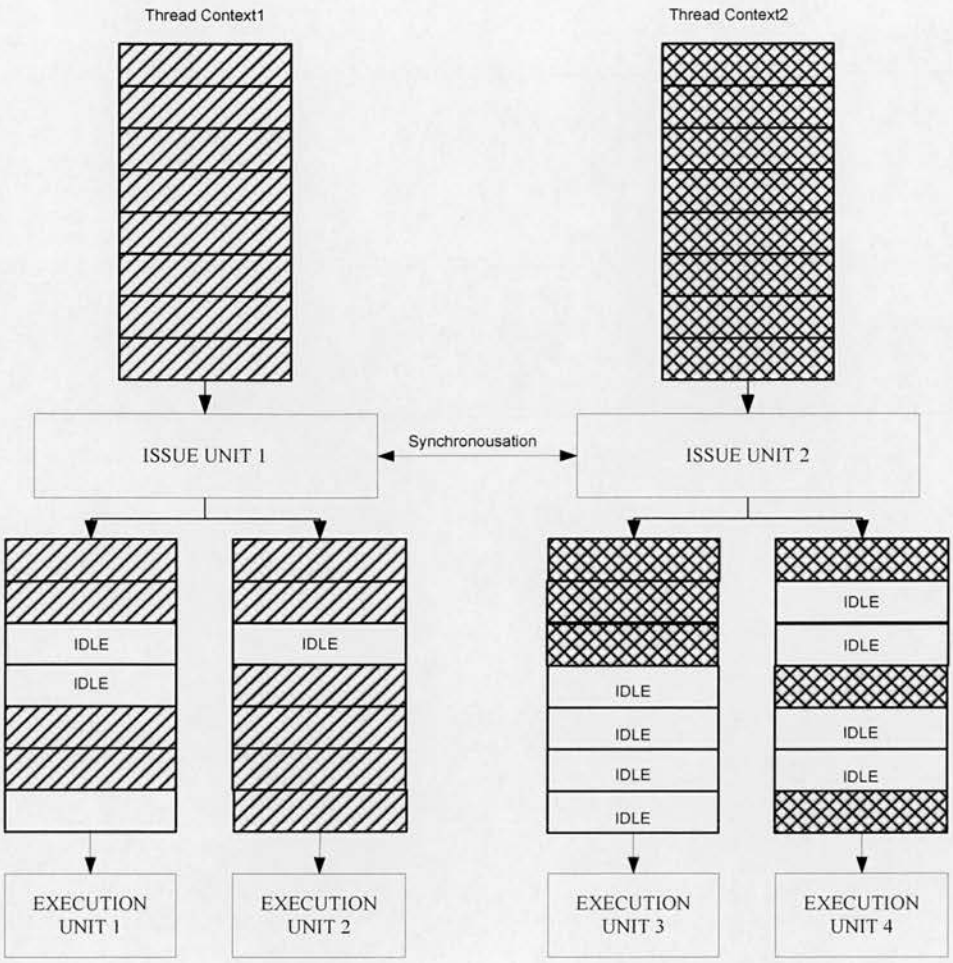


Figure 2-7: Chip Multiprocessor Architecture.

2.3.4 Thread Extraction Mechanisms

In order to utilise the resources in the multithreaded architectures efficiently, thread level parallelism of a program needs to be exploited. This can be done manually by using explicit threading program mechanisms, or implicit threading programs via automatic thread extraction methods.

Explicit multithreaded programs allow the programmer to express the thread parallelism in a natural and flexible way. This is supported by using programming languages, e.g. Java [151], or through run-time libraries, e.g. POSIX threads [152], MPI [154], or through multithreaded directive pragmas inserted into a program, e.g. OpenMp [153].

Techniques for automatic thread extractions can be performed either at compile time or at runtime. Dynamic or run-time approaches aim to use information that can be collected at run-time to improve its performance. These techniques include load balancing, dynamic process creation and dynamic scheduling. Static or compile-time approach techniques only rely on information directly available from the source code of the application. The following section concentrates on the automatic thread extraction methods.

Run-time approaches

In a dynamic thread extraction approach, threads are created dynamically by the hardware and executed speculatively. Dynamic multithreaded processor (DMT) [165] is an example of implementation of this approach. A new thread is spawned when the processor encounters a procedure call or a loop boundary. Each thread unit holds its own states, such as the trace buffer, the program counter, and the load and store queues. It shares with other threads the system resources such as the physical register file, the waiting instruction buffers, and the instruction and data caches. Instructions are fetched, dispatched, and executed out of order, and are completely re-ordered after execution so that the final results are committed in order. The advantage of these dynamic and out-of-order threads is that a look-ahead mechanism can be used to search for potential parallelism far away from the currently executing instruction.

Compile-time approaches

Compile-time parallelisation approaches focus on partitioning the overall problem into separate threads, allocating threads to different TPUs and synchronising the tasks to ensure dependencies are respected. Automatic compile time parallelisation enables a compiler to automatically exploit the parallelism inherent in the computation. A pure implicitly parallel language does not need special directives, operators or functions to

enable parallel execution. Two following two steps are used to identify parallelism.

At first, the compiler performs a data dependence analysis of the loop construct to determine whether iterations of the loop can be executed independently. The loop analyses include general loop restructuring, such as loop fusion, loop peeling, and invariant code motion. In loops with independent iterations, vector computations are carried out independently for every element. Loop iteration interleaving is used to assign iteration to threads in an interleaved manner. Data dependence can sometimes be dealt with, but it may incur additional overhead in the form of message passing, synchronisation of shared memory, or some other method of thread communication. The second step is to justify the parallelisation effort by comparing the theoretical execution time of the code after parallelization to the code's sequential execution time, as code does not always benefit from parallel execution. In practice, the extra overhead associated with using multiple processors can dilute the potential speedup of parallelised code.

Examples of automatic parallelising compiler include: McCAT [157], Panorama [158], Polaris [159], PTRAN [160], and SUIF [161], Intel's C/C++ Compiler 9.0 [162], and IBM's CELL Processor compiler [163]. Traditionally, automatic parallel programming is applied to problems that are inherently parallelisable, those without data dependencies.

Speculative multithreading

Speculative multithreading, also known as thread level speculation (TLS), is a dynamic parallelisation technique that depends on out-of-order execution to achieve speedup on multiprocessor CPU's. It enables threads to start execution before the conditions on which they are dependent are resolved. As a result, mutual exclusion and independence are not guaranteed, therefore any violations of independence assumptions must be detected and resolved by the hardware. Apart from the control speculation, data speculation is also employed to resolve data dependency among

threads.

The Superthreaded [164] architecture at the University of Minnesota and the Multiscalar [155][156] architecture at University of Wisconsin enable speculative multithreading at both architectural and compiler level. Multiscalar processors divide a single-threaded program into a collection of tasks. In the architecture, a program is represented as a Control Flow Graph (CFG), where basic blocks are nodes and arcs represent the flow of control from one basic block to another and tasks are collections of basic blocks. Processing units are organised in a circular queue, with a different task assigned to each unit. Hardware support is provided to squash tasks if the branches between tasks are incorrectly predicted. The global sequencer predicts the next task which is one of the possible successors indicated in the current task descriptor. Superthreaded architecture contains multiple thread processing units which are connected by a unidirectional ring, and a thread can only fork a thread on the successor processing unit. The architecture relies on a thread pipelining execution model to facilitate overlapping between threads and the enforcement of run-time inter-thread data dependences. Both hardware and software supports are needed for a superthreaded processor for data dependency checking, parallelisation and speculation.

The DMT processor enables the speculative multithreading without compiler support. An adaptive thread predictor is used to assign priority to threads based on collected runtime information, such as look-ahead distances and global history. Thread-level control mis-speculation might cause high overhead due to wasted execution time on squashed thread body. In order to keep the overhead low, many techniques to expose high-confident branches to the control speculation have been exploited based on compiler techniques or architecture designs.

For data value speculation, several value speculation techniques have been proposed: Last value predictor [74][75], stride predictor [72][73], context-based prediction [76][77]. These techniques are based on the history pattern seen by individual instruction operands.

2.3.5 Multithreaded processors

A number of architectural techniques are incorporated in multithreaded architecture. An SMT architectural model to evaluate the performance potential of simultaneous multi-thread instruction issue was presented by Tullsen et al. [140]; Hirata et al.[148] propose an architecture that dynamically packs instructions from different threads. A SMT analytic model and its simulator were proposed by Yamamoto et al.[149] On the other hand, CMP is also active research area: Hydra chip multiprocessor [144] is a research processor designed at Stanford University, which evaluates the shared-secondary-cache CMP; Venkata Krishnan[150] presented a CMP architecture for speculative execution of sequential binaries without source re-compilation. The following section provides a summary for existing commercial multithreading processors.

ARM 11 MPcore

ARM11 MPCore [147], Figure 2-8, is a newly developed CMP implementation of ARM11 micro architecture. The ARM11 micro architecture has a configurable number of processors, ranging from one to four. The ARM11 core contains eight pipeline stages, which includes two fetch stages, one decode stage, one issue stage, and four stages for integer execution pipeline.

An important component for control cache coherency across several processors is the Snoop Control Unit (SCU). In a CMP environment, each processor exposes logic and control onto the system bus to allow each master to maintain coherency by snooping into the other master's copy of memory. The SCU keeps communication local, at full core speed, and provides the logic and state to ensure the implementation of coherency is minimised. Because ARM11 MPCore is targeted at real time embedded systems, a fast-response interrupt system is also important. The separate interrupt system is closely coupled to the processor cores, which removes the bottleneck of shared system bus. In addition, each interrupt system contains a

per-CPU aliased memory map, permitting the same code to run on any CPU without it always needing to check its state before executing a command. Special broadcast mode to allow a CPU to send a software interrupt to all other CPUs, is commonly used to maintain some state between processors, a broadcast to all allows any CPU to action the request, and a broadcast to self allows code to defer some interrupt action back into the OS schedule queue. Another advanced feature of MPcore is the Intelligent Energy Management (IEM), which is a HW/SW solution to dynamically predict the required performance levels and scale the frequency and the voltage to the minimum in order to get the work done. ARM's benchmark indicates an energy saving for up to 50%, but the saving depends on the actual application.

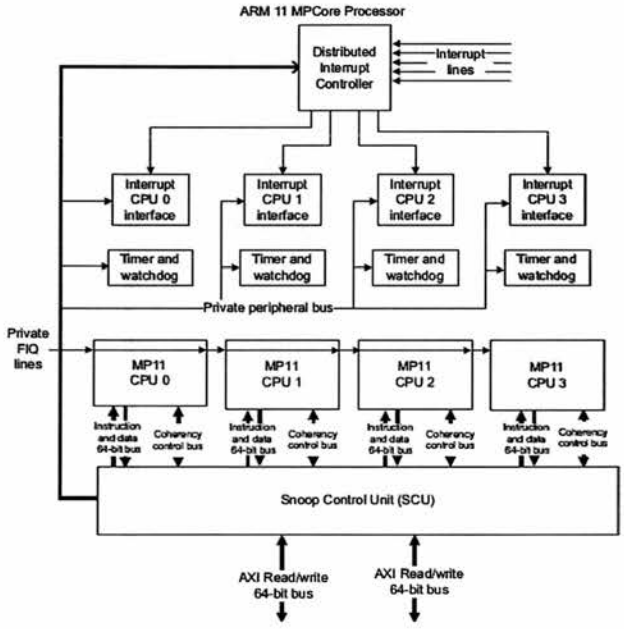


Figure 2-8: ARM MPCore processor block diagram (reproduced from [147]).

Intel Pentium IV with HT

Hyper-Threading Technology [146] is Intel's implementation of simultaneous multi-threading technology. As shown in Figure 2-9 , Hyper-Threading Technology makes a single physical processor appear as multiple logical processors, and each logical processor holds a copy of the architecture state and the physical execution

resources are shared by logical processor. Such a design enables the apparent simultaneous execution of instructions from two logical processors.

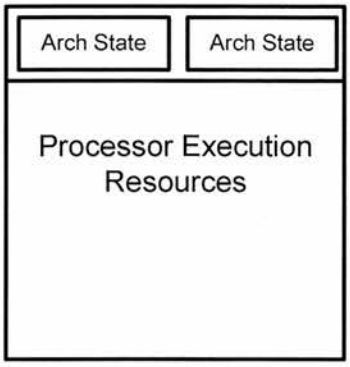


Figure 2-9: Processor with Hyper-Threading Technology (reproduced from [146]).

The architecture state is the state of thread held in registers, including general-purpose registers, the control registers, the advanced programmable interrupt controller registers, and some machine state registers. Because every logical processor holds its own interrupt controller, the interrupt requests will only be handled by that specific logical processor.

In the instruction fetch stage, instructions are fetched from the Execution Trace Cache (TC). Two sets of next-instruction-pointers independently track the progress of the two software threads executing and the two logical processors arbitrate access to the TC every clock cycle. The cache is 8-way set associative, and entries are replaced based on a Least-recently-used (LRU) algorithm. Apart from the TC, a shared microcode ROM is also used for storing the complex IA-32 instructions. Branch prediction tables and return stacks are also used to hide control latencies. Each logical processor has its own set of two 64-byte streaming buffers to hold instruction bytes in preparation for the instruction decode stage. The decode logic is more complex than that used in a RISC architecture due to inherent complexity of a CISC instruction set. Therefore, the decode logic is shared by the two logical processors, and a coarser level switching scheme is used. In order to improve the processor throughput, an out-of-order execution engine is also implemented, which enables register renaming, instruction re-ordering, tracing, and sequencing. The scheduler balances the

instruction execution flow for different execution units. Each scheduler has its own scheduler queue of eight to twelve entries from which it selects instructions to send, based on dependent inputs and availability of execution resources. After the execution stage, results are written back to the shared register pool directly, and the forwarding logic is used to forward results and the re-order buffer and retirement logic guarantees the architecture state is committed in program order.

As mentioned in the previous section, one benefit of the SMT (hyperthreading) design is limited chip area overhead. From Intel's own benchmark, its HT implementation used an additional 5% of the die area over a non-HT processor, yet yielded performance improvements of 15-30%.

IBM POWER5

IBM's POWER5 [145] processor was released in 2004, which combines SMT and CMP architecture design. The processor is single chip with dual-core POWER5, each core has 8-way superscalar pipeline stage and is SMT-enabled. As a result, the chips will look like four logical processors to the operating system. It is fabricated in a 130 nm process, and there is 24% area overhead per POWER5 core due to SMT implementation.

In the SMT mode, the Power5 uses two separate instruction fetch address registers to store the program counters for the two threads. Up to eight instructions from the instruction cache can be fetched every cycle. Two threads share the instruction cache and the instruction translation components, and only instructions from the same thread can be fetched every cycle. Branch prediction methods have been implemented in the POWER5 core by using three branch history tables. These tables are shared resources across two logical threads. Also subroutine returns can be predicted using a return stack, one for each thread. For instruction decoding, the processor selects instructions from the fetch queues based on thread priorities. The Power5 supports eight software-controller priority levels for each thread. The higher a

thread's priority relative to other threads, the more of the processor's resources can it monopolise. After all input operands are available, an instruction will be issued and there is no distinction between instructions from different threads. Then normal pipeline stages will proceed, including input registers reading, execution stage, and writing results back.

Two major features were implemented in the Power5 design to improve the SMT performance: Dynamic resources balancing and adjustable thread priority. The dynamic resource-balancing enables two threads to execute on the same processor seamlessly. Adjustable thread priority lets software determine when one thread should have a greater share of execution resources. For example, a thread in a spin loop waiting for a lock, or an idle thread can be given a lower priority and conversely real-time tasks can be assigned a higher priority.

Sun UltraSPARC T1

Similar to the multithreading techniques used in IBM POWER5, the SPARC UltraSPARC T1 [176] processor combines SMT and CMP architectural designs which features CoolThreads technology in its design. The process contains eight processing core with four threads per core, and as a result 32 simultaneous threads are available to the operating system, which is even higher than POWER5.

Four memory controllers are embedded in the chip, which route data between the processing cores and the memory and allows data to be transferred into the chip as fast as it can be processed. The on-chip level-2 cache hides the long latency of memory accesses. Also, the processor improves system security and delivers better throughput via dedicated hardware crypto accelerators. The chip is fabricated in a 90 nanometre process, and each core operates at 1.2 GHz.

2.4 Reconfigurable computing

Reconfigurable computing presents a flexible and upgradeable approach to system

design. As shown in Figure 2-10, currently available computing components can be classified into four categories: processors, DSP, ASIC and FPGA, and each have different levels of flexibility and energy efficiency [19].

The flexibility offered by processors allows a single machine to perform a multitude of functions and be deployed in applications unanticipated at the time the device was designed or manufactured. Processors have been the platform for general-purpose computing. Similarly, the Digital Signal Processors (DSPs) have been the work horse for telecommunication equipment. By changing the software instructions, the functionality of the system is altered without changing the hardware. The price for such flexibility is execution overhead. On the other hand, an Application Specific Integrated Circuit (ASIC) is designed specifically to perform a given computation efficiently in terms of speed and power. However, the circuit cannot be altered after fabrication. This forces a redesign and refabrication of the chip for a different application. In between it sits the Field Programmable Gate Arrays (FPGAs).

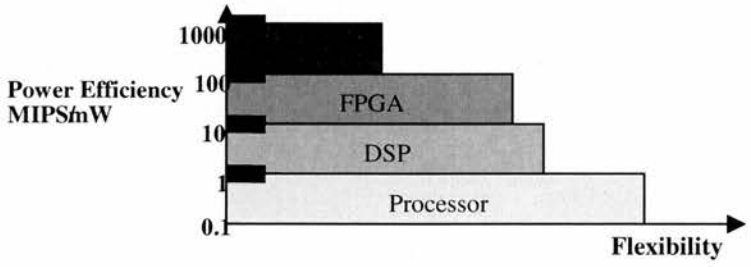


Figure 2-10: The Power-flexibility gap.

The enabling technology for building reconfigurable computing systems was the FPGA [20]. FPGAs consist of an array of logic blocks, routing channels to interconnect the logic blocks, and surrounding I/O blocks. Static RAM (SRAM) based FPGAs use SRAM cells to control the functionality of the logic, I/O blocks, and routing. They can be reprogrammed in-circuit arbitrarily often by downloading a bitstream of configuration data to the device. FPGAs are fine-grained architectures that operate on bit-wide data types and use Look-Up Tables (LUTs) as computing

elements. Today's devices feature millions of gates of programmable logic, dense enough to host complete computing systems, e.g. Xilinx provides the soft-core processor MicroBlaze [166] which can be implemented in a Virtex IV that runs at 180MHz clock frequency. In the last decade, a new class of hybrid reconfigurable computing devices has emerged, which promises to combine the flexibility of processors with the efficiency of FPGAs. These hybrid designs provide the benefits of on-demand functionality, on-demand accelerations and shorter time to market. In a hybrid system, the coupling between the CPU core and the reconfigurable logic array determines the type of applications that benefit most from the hybrid reconfigurable processor.

The following sections outline the state of the art in designing reconfigurable computing system. In the first section, the concepts for coupling reconfigurable logic arrays into a computing system are outlined. The second section discusses compilation, simulation and evaluation techniques for hybrid reconfigurable computing systems that have been discussed.

2.4.1 Loosely-Coupled Reconfigurable Architectures

In a loosely-coupled reconfigurable architecture, the reconfigurable or dynamic unit is usually implemented as a coprocessor. The unit is equipped with some extra hardware to perform control flow operations and for accessing the memory. Figure 2-11 illustrates the block diagram of the loosely-coupled reconfigurable architecture.

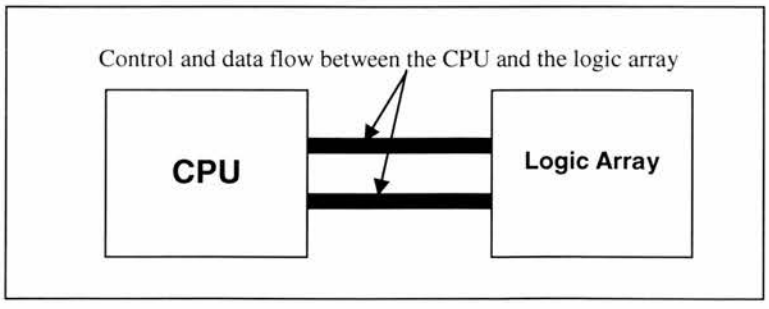


Figure 2-11: A Loosely-Coupled Reconfigurable Architecture

The PRISM [167] architecture was developed at Brown University. It links a Motorola 68010 microprocessor running at 10 MHz to a board-based FPGA array consisting of four Xilinx 3090 FPGAs. Speedup factors have been reported which range from 5 to 50.

The Berkeley Garp [94][95][96] architecture combines configurable logic with a standard MIPS processor, all on the same chip. The configurable array is composed of a matrix of logic blocks that are organized into 32 rows of 24 blocks. One block in each row is a control block, and the remaining are logic blocks which can implement 2-bit operations. Four memory buses run vertically through the rows for moving information into and out of the array. They can be used to transfer data and perform accesses to memory. A separate wire network provides interconnection between the logic blocks. The loading and execution of the configurations is under the control of the main processor. A transparent integrated configuration cache holds the equivalent of 128 rows of configurations (as 4 cached configurations for each row). Reconfigurations from this cache take 4 cycles, irrespective of the number of rows. The operation of the reconfigurable array is carried out by instructions which extend the MIPS instruction set. The reconfigurable array, however, can cache data or access memory independent of the MIPS core.

The NAPA C language [168] provides pragma directives so that the programmer can specify where data is to reside and where computation is to occur with statement-level granularity. Its compiler targets National Semiconductor's NAPA1000 chip, performs semantic analysis of the pragma-annotated program and co-synthesises a conventional program executable in combination with a configuration bit stream for the adaptive logic. Compiler optimisations include the synthesis of hardware pipelines for some loops.

2.4.2 Tightly-Coupled Reconfigurable Architectures

A tightly-coupled reconfigurable architecture (Figure 2-12), contains reconfigurable hardware in the host processor to provide dynamic functional units. This allows for a

traditional programming environment with the addition of custom instructions that may change over the duration of program execution. The reconfigurable units execute as dynamic functional units on the microprocessor datapath, with registers being used to hold the input and output operands.

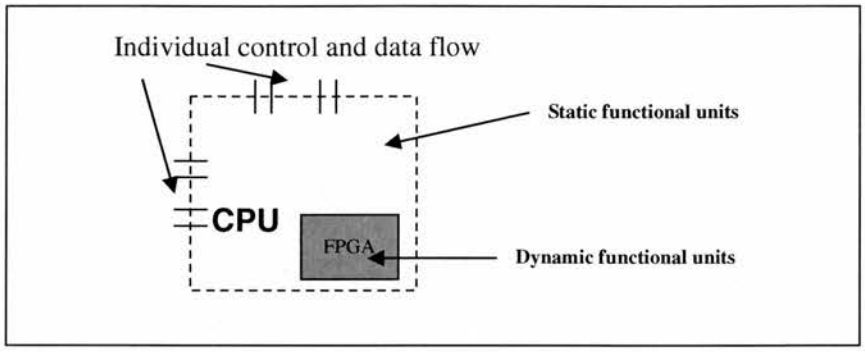


Figure 2-12: A Tightly-Coupled Reconfigurable Architecture

The PRISC is a proof-of-concept architecture [169], which is a 200-MHz RISC CPU with an augmented datapath. The reconfigurable logic is integrated into the microprocessor by adding one Programmable Functional Unit (PFU) in parallel with existing Functional Units (FU), such as the ALU. A PFU is implemented as a regular structure of interconnects and LUT. The complexity of functions that are implemented in a PFU is such that their latency does not exceed the cycle time of the microprocessor. Context switching amongst precompiled PFU images is supported and different hardware functions are addressed through extensions to the existing instruction set of the static CPU. The microprocessor-to-reconfigurable-logic interface enables PFUs to make use of the existing datapath functionality such as source and result operand handling as well as hazard detection and forwarding. Through this tight integration of the PFU within the CPU, data and control flow overhead are kept at a lower cost than in any of the reconfigurable computing systems discussed previously. The purely combinational nature of the PFU requires tools to automatically extract hardware functions from the application and generate hardware images with fine granularities. Nonetheless, from their simulation, the SPECint92 benchmarks showed

speedups ranging from 1.06 to 1.91 for the applications when executed on PRISC in comparison with a software-only execution on a RISC machine, without the extra PFU.

Northwestern University's Chimaera [63][170] architecture comprises the following components: the Reconfigurable Array (RA), the Shadow Register File (SRF), the Execution Control Unit (ECU), and the Configuration Control and Caching Unit (CCCU). The operations are executed in the RA; the ECU decodes the incoming instruction stream and directs execution. The ECU communicates with the control logic of the host processor for coordinating execution of reconfigurable functional unit (RFU) operations; the CCCU is responsible for loading and caching configuration data; finally, the SRF provides input data to the RA for manipulation. In the core of the RFU lies the RA, which is a collection of programmable logic blocks organized as interconnected rows. The logic blocks contain lookup tables and carry computation logic. Across a single row, all logic blocks share a fast-carry logic that is used to implement fast addition and subtraction operations. By using such an organisation, arithmetic operations such as addition, subtraction, comparison, and parity can be supported very efficiently. The routing structure of Chimaera is also optimised for such operations. During program execution, the RA may contain configurations for multiple RFU instructions. Managing the RA-resident set of RFU instructions is the responsibility of the ECU and the CCCU. The CCCU loads configurations in the RA, provides fast access to recently-evicted configurations through caching, and provides the interfaces necessary to communicate with the rest of the memory hierarchy. The ECU decodes the instruction stream. It detects RFU instructions and guides their execution through the RA and, if necessary notifies the CCCU of recently-unloaded configurations.

There are several problems needed to be solved when building a tightly-coupled computing architecture, the most important one being timing. The timing of the components must be compatible, which is very critical as over timing components will result in erroneous results. For example, adding a programmable functional unit to PRISC architecture is not without its difficulties, since the reconfigurable

component is usually 15 to 25 times slower than the processor. The second problem is the issue of placement. In order to exchange configurations on reconfigurable units, it requires ample spaces to place the new configurable elements, which incurs extra area overheads. Finally, routing is the third issues which arise in these architectures, as the functional units must have ports to connect to the reconfigurable logic.

2.4.3 Compilation and Evaluation techniques

The hybrid reconfigurable computing system relies on an efficient compiler to extract hardware and software codes to run on the reconfigurable logic and the processor respectively. For commercial programming environment, a manual partitioning still dominates the market. Processor code and configuration data for the reconfigurable logic arrays are hand-crafted and wrapped into library functions that are linked with the user code. But this presents problems to software programmers with little hardware knowledge. As a result, automatic HW/SW partitioning techniques have been investigated by several researchers [95][96][173][174], constituting a research field of its own within the reconfigurable computing domain. Common technique for performing automatic HW/SW partitioning is described next.

The compiler constructs a control flow graph from the source program. Execution times will be assigned to the basic blocks based on the profiling information. Finally the most executed block will be chosen to generate hardware codes for the reconfigurable logic. On the other hand, if the compiler lacks the run-time information provided by compiler, inner loops of programs are good candidates for reconfigurable fabrics. In this thesis, automatic HW/SW compilation techniques used for MAPS+ architecture are based on these techniques, but with some optimization procedures to increase basic block size, which is similar to those used in Very-Long Instruction Word (VLIW) architectures.

In the Onechip [171][172] architecture, the modified SimpleScalar[175] simulator is used, and black-box simulation module for the RFU is integrated. While a RFU operation is decoded by the simulator, it will be issued by the RFU black-box.

Then the input and output parameters and the execution latencies of different configurations are modelled. As a result, the overall performance can be evaluated by this approach. A similar black-box approach has been adopted by the Chimaera [63] simulation environment, but with a different latency model. Its latency models are based on counting the number of original instructions that are replaced by the RFU operation. A more complex model is also proposed, which is based on hand-mapping the RFU operations onto the reconfigurable array and measuring the number of transistor levels in the critical path. Also reconfiguration overhead is also considered as latency for executing a RFU operation.

In our simulations in Chapter 5, a similar black-box approach has been chosen. But the latency model is based on a more accurate data obtained from the actual FPGA synthesis tool. In addition to the latency model, a power consumption model has also been implemented.

2.5 Summary

Several problems exist in current synchronous architectures, e.g. clock skews, high power consumption, design and verification complexity and long time to market. Alternative system design approaches have been proposed to address these problems. In this chapter, a background survey of these techniques has been made, including asynchronous design, multithreading execution and reconfigurable computing.

Asynchronous designs have a number of advantages, e.g. no clock skew, modularity, robustness, lower peak electromagnetic emissions. But the asynchronous designs also suffer from such problems: design complexity, problem of completion and detection, lack of industry standard EDA tools, testing problems and performance measurement. Traditionally, lower power and average case performance are two of the benefits cited for by asynchronous designs. However, with progress in design and clock-gating for synchronous circuits, these advantages are unclear.

Current multithreaded architectures are discussed, e.g. SMT and CMP. The issues in static and the dynamic thread extraction mechanisms are summarised. For a

multithreaded architecture, thread level parallelism can be exploited to improve the throughput of a processor by identifying independent threads that can execute in parallel and can therefore utilise multiple thread processing units.

Finally, reconfigurable computing system and its compilation and simulation environment have been introduced. The benefit of reconfigurable computing is the ability to realise the flexibility of a software-based solution, while retaining the execution speed of a more traditional, hardware-based approach.

Obviously, a major challenge in taking advantage of the benefits is how to combine these design techniques, and how to implement a compilation flow to extract the potential concurrency. The following chapters develop and characterise a number of components and techniques used to integrate an asynchronous multithreaded architecture augmented with a reconfigurable logic unit.

Chapter 3
Asynchronous MAPS+ architecture

3.1 Introduction

This chapter introduces a hybrid architecture called MAPS+ "Micronet-based Asynchronous Processor System Plus Reconfigurable Function Units" , based on the MAPS architecture described earlier in [28][29][30]. MAPS+ is designed to provide thread-level as well as instruction-level parallelism, and combines hard-programmability, in the form of field programmable logic, and soft-programmability in a multithreaded instruction set architecture. First, the top level multithreaded architecture is described briefly, followed by detailed descriptions of functional units. Then components enabling thread synchronisation, data value prediction and runtime reconfiguration are described. Issues in asynchronous design, speculative thread and asynchronous RFU are discussed at the end of the chapter.

3.2 The MAPS+ architecture

As discussed earlier in Chapter 2, micronets are networks of entities that compute concurrently and communicate asynchronously. The entities are themselves networks in a recursive fashion, leading to a hierarchical approach to system design. In its purest form, the implementation of the micronet system will be free of any clocks, with the entities being implemented using self-timed design techniques. However, the micronet approach to system design does allow a mixture of both self-timed and

synchronous implementations of the entities. Micronet-based multithreaded architectures exploit concurrency at different levels: between threads, within threads and between instructions. A coarse-grained level parallelism can be investigated using loop analysis, and mapped to different threads. Fine-grained parallelism, such as instruction level parallelism (ILP) [31] is mapped to Reconfigurable Function Units (RFU). Within each Thread Processing Unit (TPU), there are functional units which are either fixed or which can be configured.

The superscalar principle relies primarily on exploiting spatial parallelism, which is achieved by running multiple operations concurrently on duplicated hardware. In contrast, pipelining relies on exploiting temporal parallelism by overlapping multiple operations on common hardware and operating with a faster clock. ILP is limited by data dependencies between instructions, structural dependencies and also control transfers in pipelined architectures. In MAPS+, the RFU accelerates applications by customising reconfigurable fabric for computation-intensive tasks and executing several operations in parallel. Thread level parallelism on MAPS+ architecture is supported by switching thread context among different TPUs at run time. Thread contexts are the machine state associated with the execution of a thread, including the values of the program counter, data registers and status registers. As a result, MAPS+ is able to pursue two or more threads of control in parallel within the pipelines.

The MAPS+ architecture, as shown in Figure 3-1, is modelled at different levels of abstraction: at the level of thread processing units with gross values for computation and communication latencies and power costs, down to the handshaking protocols of the micronet architecture which reflect more accurately the costs of speed and power consumption. Quantifiable changes in the speed and power consumption may be affected by program optimisations at the higher levels in the design framework and partitioning the program between the soft- and hard-programmable entities.

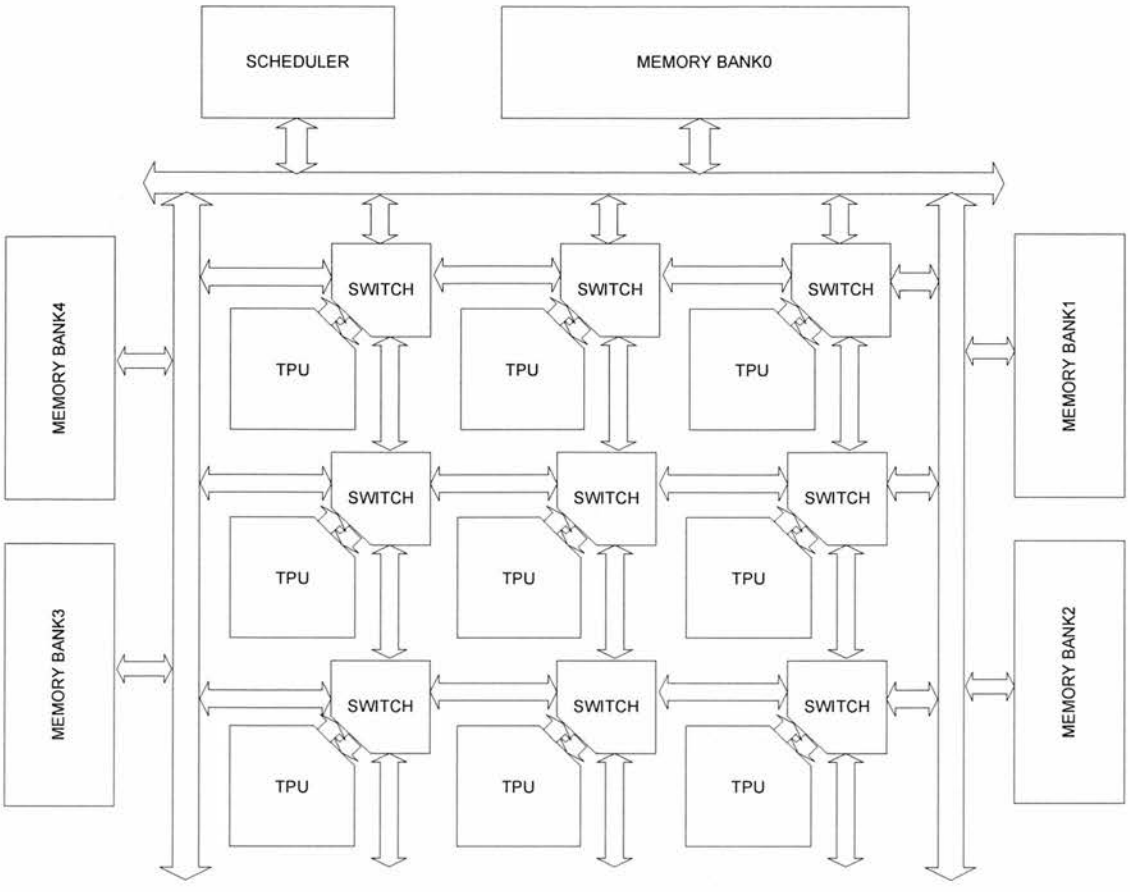


Figure 3-1: Block diagram of MAPS+ architecture

3.3 The Switch model and inter-thread communications

Communication between TPUs is often implemented via shared memory with a single on-chip bus [32][33][34][35]. This approach has it drawbacks. The shared bus suffers from poor contention and is a performance bottleneck when multiple TPUs are competing for the memory. Figure 3-1 illustrates a mesh topology for connecting the TPUs, which will alleviate the contentions in inter-TPU communications.

As shown in Figure 3-2, a switch block routes messages between source TPUs and destination TPUs and the memory bank, and is situated at the intersection of vertical and horizontal data channels. Each switch connects to four other neighbouring switches or shared memory blocks via four interconnect ports named N (North), S (South), E (East) and W (West). The switch is also linked to TPU via the L (Local) port, which serves as thread communication interface for the TPU. Each port has two

channels, one each for incoming and outgoing packets. Similar switch structures can be found in synchronous Network On Chip (NOC) architectures [36][37][38][39]. Self-timing logics are integrated into the switches and data communications between any two TPUs is strictly mediated by a full handshake protocol. The input channel of each port receives the data and the output channel is responsible for feeding outgoing data into other switches. All the ports act independently, and the module can send and receive data concurrently. Furthermore, in order to control data switching, an asynchronous switch controller is required, which provides selection logic for outgoing data and arbiter for incoming requests.

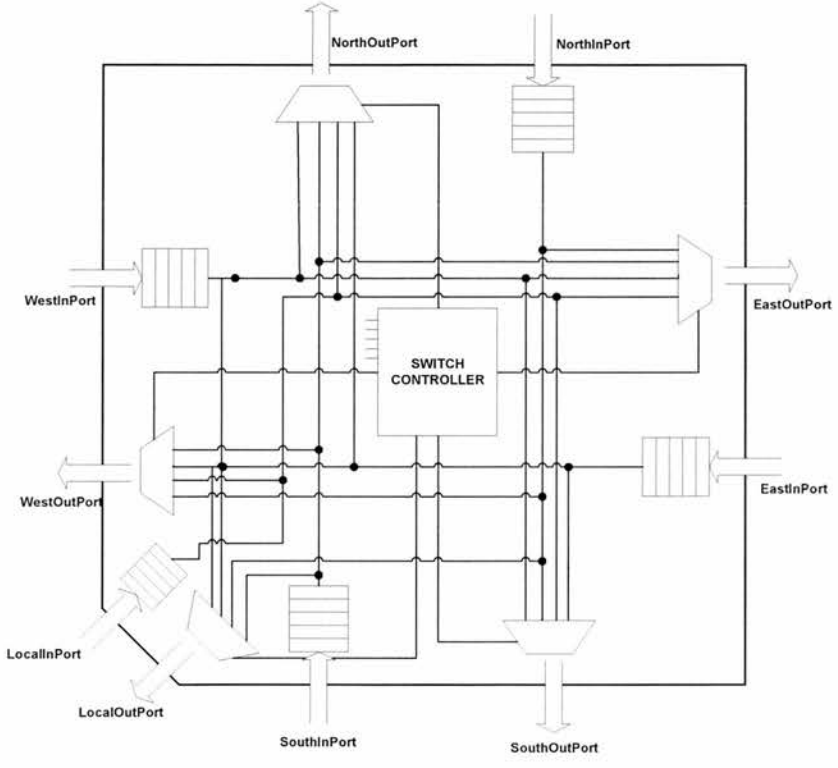


Figure 3-2: Block diagram of a switch

In order to fulfil the requirements of a reconfigurable MAPS+ architecture and provide flexibility for run-time reconfiguration, a table routing method for inter-TPU communications was chosen. A lookup table is placed in each switch, with entries corresponding to each TPU, each shared memory blocks and the scheduler. Figure 3-3 depicts a sample address partitioning method for the MAPS+ architecture. Each group

has a range of addresses associated with the destination block, and it starting with *START_ADDR* and ending with *END_ADDR*.

Figure 3-3: Address partitioning for destination blocks

Given a destination address, the corresponding entry in the table indicates which outgoing ports should be used to forward the data. A sample look-up table is illustrated in Table 3-1. The number of TPUs in the MAPS+ architecture has been limited to 16, which indicates the upper bound on the number of concurrent threads likely in embedded sequential applications.

Start Address	End Address	Type	Direction
INSTRUCTION_START_ADDR	INSTRUCTION_END_ADDR	INSTRUCTION	NORTH
DATA_START_ADDR	DATA_END_ADDR	DATA	NORTH
TPU_0_START_ADDR	TPU_0_END_ADDR	TPU0	LOCAL
TPU_1_START_ADDR	TPU_1_END_ADDR	TPU1	EAST
TPU_2_START_ADDR	TPU_2_END_ADDR	TPU2	EAST
TPU_3_START_ADDR	TPU_3_END_ADDR	TPU3	SOUTH
ROP_0_START_ADDR	ROP_0_END_ADDR	MEM_BANK3	WEST
ROP_1_START_ADDR	ROP_1_END_ADDR	MEM_BANK4	WEST
SHM_START_ADDR	SHM_END_ADDR	MEM_BANK0	NORTH

Table 3-1 : A sample look up table for local port.

The input channel transmits incoming data to the selected output channels using the lookup table. Address translator and selection logic blocks are attached to each port, which are parts of the switch controller. A local port and its selection logics are illustrated in Figure 3-4. For the incoming data, buffers operate as input queues for temporarily storing data. When the destined output channel is ready, data is forwarded to the destination via the crossbar. Wormhole routing [40] approach can be implemented to provide fast data communications among TPUs. Packets are segmented, with each switch required to store a fraction of the whole packet. Long packets are distributed over several consecutive switches and do not require extra buffer spaces. The selection logic passes channel requests to the output channels. The direction of such request is decoded by the address translator. For example, if *Direction_N* direction request signal gets high, then the C-element to the *North* port output channel issues a request. A dual-rail encoding method in the address translator is needed to guarantee that output channel request signal does not arrive before the direction signal has become stable.

Deadlocking occurs when network resources (e.g. link bandwidth or buffer space) are suspended waiting for each other to be released, i.e., where one path is blocked leading to the other being blocked in a cyclic fashion. There are several methods for solving deadlock or livelock problems, e.g. dimension-ordering routing [41][42] and virtual channels [43]. In dimension-ordered routing, the packets always route on one

dimension first, e.g., column first, upon reaching the destination row (or column), and then switch to the other dimension until reaching the destination. Dimension-ordered routing is deterministic: packets will always follow the same route for the same source-destination pair. In the virtual channel approach, one physical channel is split into several virtual channels. Virtual channels solve the deadlock problem while achieving high performance. Nevertheless, this scheme requires a large buffer space for the waiting queue of each virtual channel. For example, if one channel is split into four virtual channels, it will use four times as much buffer space as a single channel. The virtual channel arbitration also increases the complexity of circuit design. For the sake of simplifying the MAPS simulator design, the dimension-ordered routing approach was chosen. Therefore, contention cannot be avoided, and when it occurs, the packets have to wait for the channel to be free.

Livelock represent a state in which one or more messages could be forever denied of the resources they require to progress towards their destinations. Unlike deadlock, livelock does not stop a packet's movement, but rather its progress towards its final destination.

Routing can be minimal or nonminimal in terms of the path length [44][45]. Minimal routing algorithms guarantee shortest paths between source and target addresses. In nonminimal routing, the packet can follow any available path between source and target. Nonminimal routing offers great flexibility in terms of possible paths, but can lead to livelock situations and increase the latency to deliver the packet. In our simulation, a minimal routing was chosen, and livelock was avoided.

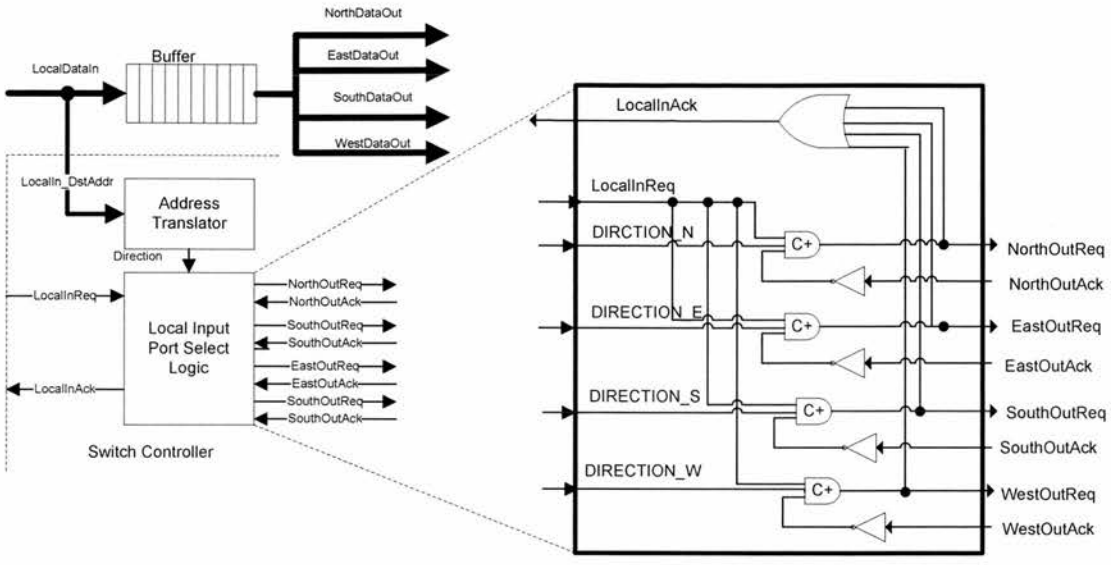


Figure 3-4: Local Port and its selection logic block diagram

Input and output channels are connected by a 5 × 5 crossbar, which is formed by multiplexers to aggregate every input to the output. A packet coming from an input port may have a choice of multiple output channels, e.g., data of a local port can be forwarded to north output, east output and so on. Similarly, multiple input ports may request simultaneously access to the same output channel. As a result, access to output port has to be arbitrated. On the output port, the arbiter decides which direction of the waiting channel can proceed. Figure 3-5 depicts an arbiter for the output channel built with tree arbiters and mutual exclusion elements. The tree arbiter, as shown in Figure 3-6 is a slightly modified version of that presented in [46][47]. In order to simplify the design, the blocks have been mapped to standard cells e.g. Mutual Exclusion Element. It guarantees that the acknowledge signal, *Ack1* or *Ack2* is not released until both the corresponding request signals, *Req1* and *Req2*, and the acknowledge input *Ack,* are lowered. The Muller C-elements in the feedback path grant compliance with a full handshaking protocol. With the tree arbiter, selection signal will then be forwarded to the output channel multiplexer and the appropriate data output will be chosen. The arbitration method is fair amongst the four inputs and results in a "first come first served" arbitration scheme. It is therefore guaranteed that the module can forward data into the corresponding output channel.

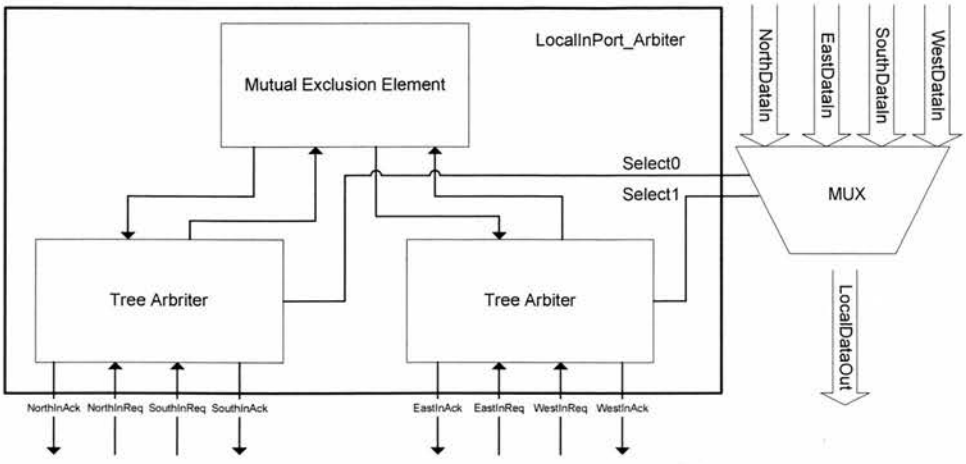


Figure 3-5: Local output port arbiter block diagram

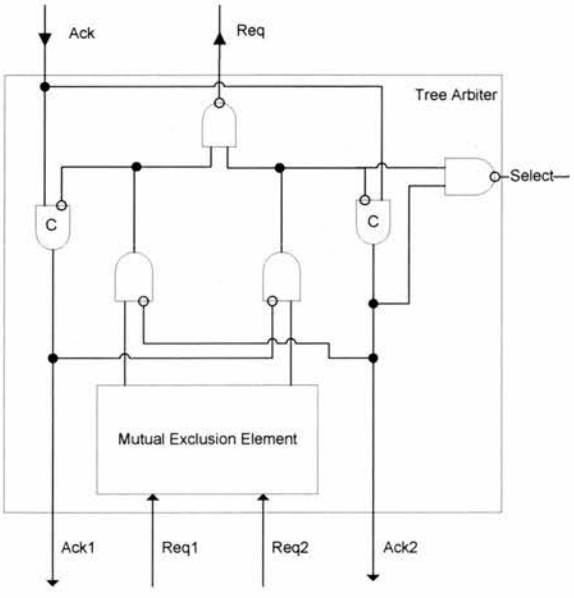


Figure 3-6: Tree arbiter with selection signal output block diagram

3.4 Thread Processing Unit

Multiple threads compete for shared computational resources. Therefore, hardware thread processing units for thread collaboration and competition are integrated in the MAPS+ architecture. Based on the micronet model, a TPU does not have a clock signal or centralised control for passing data between architectural components. The communicating microagents (CMs) effectively allow the functional units (FUs) to transfer control signals between themselves and their neighbours. In order to exploit data dependencies or variable delays, it is assumed that the CMs in the self-timed

design can be modified so that they generate completion signals. Each TPU, as shown in Figure 3-7, in turn, is a network of CMs and FUs, which includes instances of Arithmetic and Logic Units (ALUs), a Reconfigurable Functional Unit (RFU), a Floating-Point Unit (FPU), a Memory Unit (MU), a Fetch Unit, a Branch Unit (BU), a Thread Issue Unit (TIU), a Thread Control Unit (TCU), a thread speculation buffer, a data value predictor, and instruction and data caches which reads from, and writes to, the memory hierarchy via the switches.

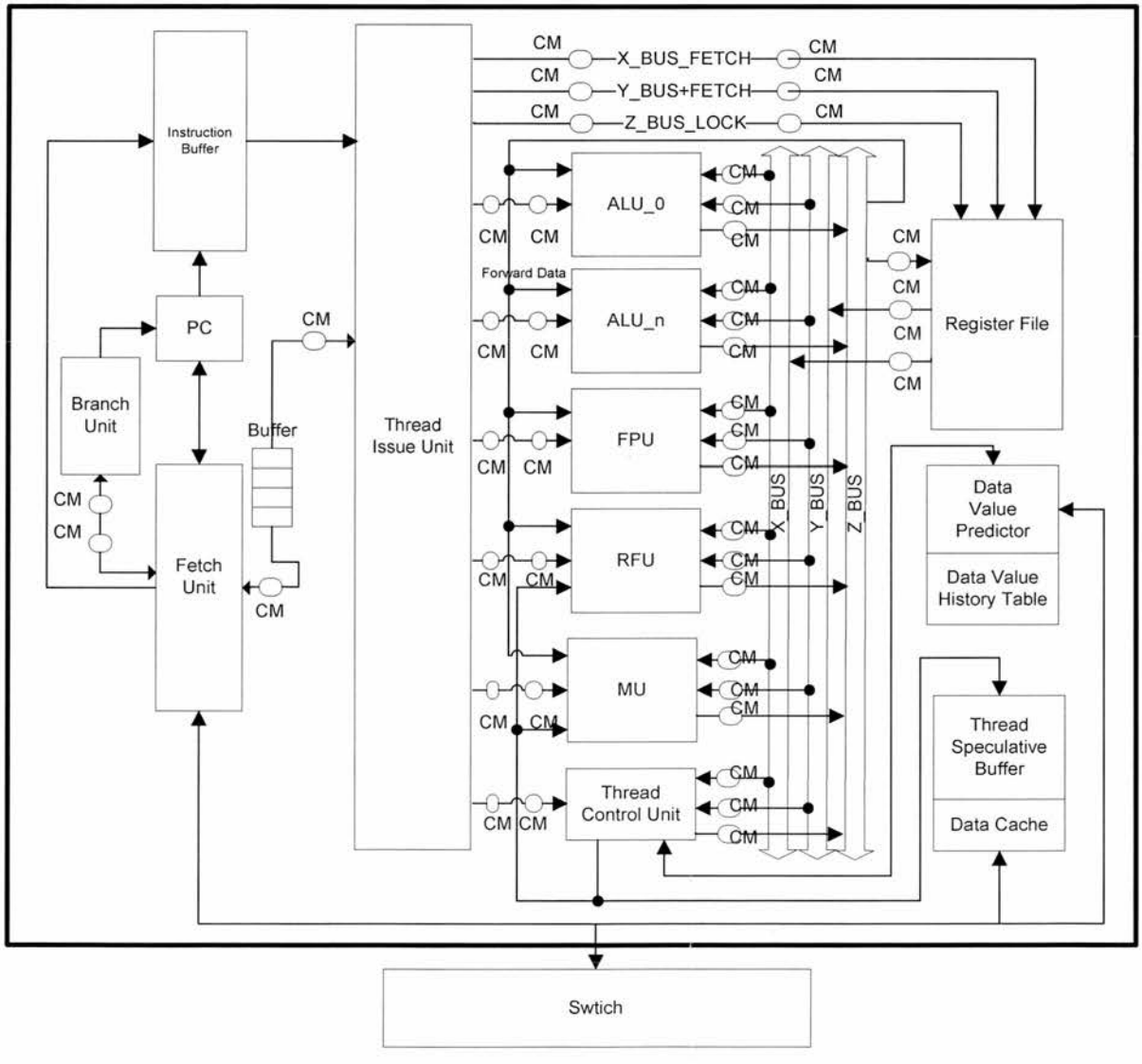


Figure 3-7: Thread Processing Unit block diagram.

3.4.1 Communicating Microagent (CM)

CMs are responsible for receiving their FUs' micro-operation control signals from the Control Unit (CU), returning the corresponding acknowledgement signal, obtaining the operand data for that operation and presenting these to the FU, and if necessary, returning the result of a micro-operation to the correct destination. Incoming signals and outgoing signals of CM are displayed in Figure 3-8. In the absence of a clock, the data transmissions have to be encoded to enable the receiver to recognise valid information. A four-phase handshaking protocol was adopted for CM. This allows for a simpler design through the use of various types of Muller C-elements and conventional logic gates. In the case of control signals, although four-phase protocol would be considered twice as expensive compared to a two-phase one, the same efficiency is obtained as two back-to-back, two-phase handshakes by representing two events in each cycle. CM can be implemented with two different approaches, which mainly depends on how completion is determined: bundled data [48][49][51] or completion detection [50][51].

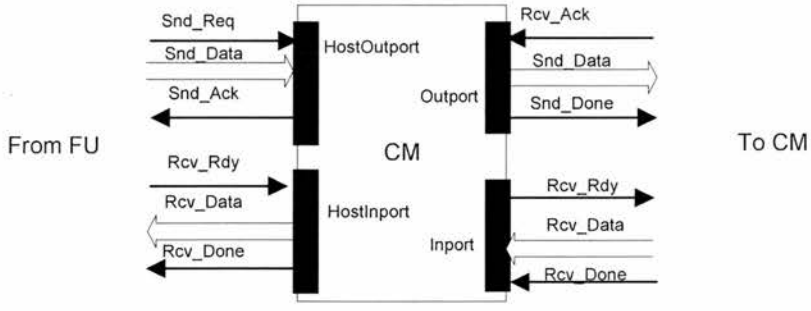


Figure 3-8: Communicating Microagent signals.

A bundled data design uses a worst-case model delay, designed to exceed the longest path through the subsystem. This delay may be an inverter chain or a replicated portion of the critical path. The main advantage is that a standard synchronous single-rail implementation may be used, so implementations are easy to design, and have low power and limited area. However, the key disadvantage is that completion is

fixed to worst-case computation, regardless of actual data inputs. Figure 3-9 shows the block diagram of CM input and output channels using bundled data method.

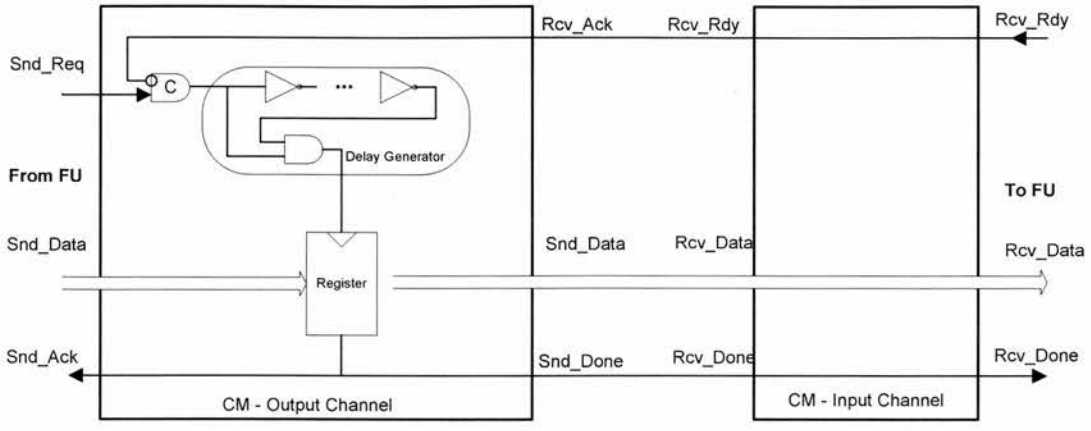


Figure 3-9: Communicating Microagent model using bundled data method

A computation detection method detects when computation is actually completed. The datapath is typically implemented in dual-rail, where each bit is mapped to a pair of wires, which encode both the value and validity of the data. The key disadvantage, in many applications, is that a completion detection network is usually required, adding several gate delays between completion and its detection. Furthermore, the increased wiring and switching activity often results in much greater area and power consumption. Figure 3-10 shows the block diagram of CM input and output channels using computation detection method.

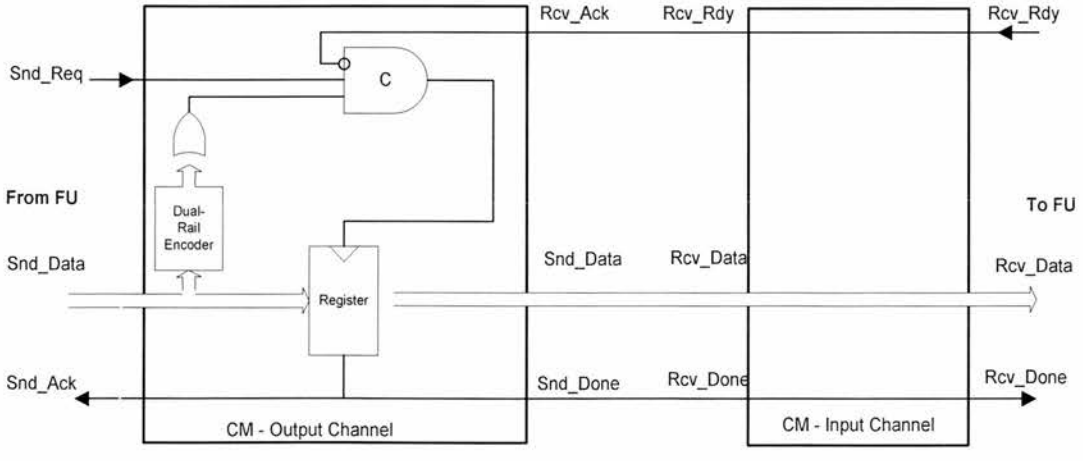


Figure 3-10: Communicating Microagent model using completion detection method

3.4.2 Fetch Unit and Branch Unit

The fetch unit takes the program counter as an instruction address and reads the corresponding instruction from the instruction buffer, and outputs it to the instruction register for decoding. In addition, the increment of the PC is also handled by the Fetch Unit. The control transfer instructions could cause the execute stage to stall and have a detrimental impact on the concurrency of the datapath. As a result, a branch unit has been adopted into the TPU architecture, which is responsible for processing control transfer instructions.

When a PC-related instruction is fetched from the buffer, it is passed directly to the Branch Unit. Previous research [52][53][54] has demonstrated that static and dynamic branch prediction techniques can reduce execution stage stall time and improve the efficiency of a data path. In the static prediction, the compiler identifies branches that are likely to be taken or not, and tags them. Then, when a branch instruction is fetched that is likely to be taken, then its target is placed in the PC, and when one is probably not taken, then normal PC increment is performed. However, some branches vary their behaviour in ways that are predictable. Therefore, a simple dynamic branch predictor based on the bimodal branch prediction technique is chosen for the MAPS architecture, which uses run-time information to predict further branch outcomes. A table of counters in the predictor is indexed by the low order address bits in the program counter. For each taken branch, the appropriate counter is incremented. Likewise, for each not-taken branch, the appropriate counter is decremented. The most significant bit determines the prediction. Repeatedly taken branches will be predicted to be taken, and repeated not-taken branches will be predicted to be not-taken.

The Fetch Unit and the Thread Issue Unit are decoupled by an asynchronous Fetch Buffer which stores the predecessor's results. It relaxes the synchrony between the fetch unit and the issue unit, allowing each stage to proceed at its own rate without hindering the other, until the buffer becomes either full or empty. The Fetch Unit continuously fetches instructions and places them in the buffer until either the buffer is full or the unit stalls waiting for the Branch Unit to resolve a conditional branch.

3.4.3 Thread Issue Unit

The Thread Issue Unit (TIU) is responsible for obtaining instructions from the Fetch Unit and deciding which functional unit to issue the instruction to. The type of the instruction is decoded and the value of the operands will be fetched from the Register File via the X_BUS and the Y_BUS. Also the destination register of the instruction is locked and a tag for the results of the instruction is set to the register. In addition, the TIU checks the result that is currently being written back because the updated value is not yet available from the register file. Finally, the decoded instruction is queued into the appropriate Functional Unit's latch.

Instructions are issued in-order from the fetch buffer, but can be executed out-of order as soon as resources are available. To ensure out-of order issuing and maximum concurrency between each of the issue processes, selection logic has to be implemented independently. An arbiter is required for each issue port to provide a decision on whether an instruction is ready to be issued to a functional unit. The TIU block diagram is shown in Figure 3-11. A similar structure is used in the instruction length decoder of the asynchronous CISC architecture- RAPPID [55].

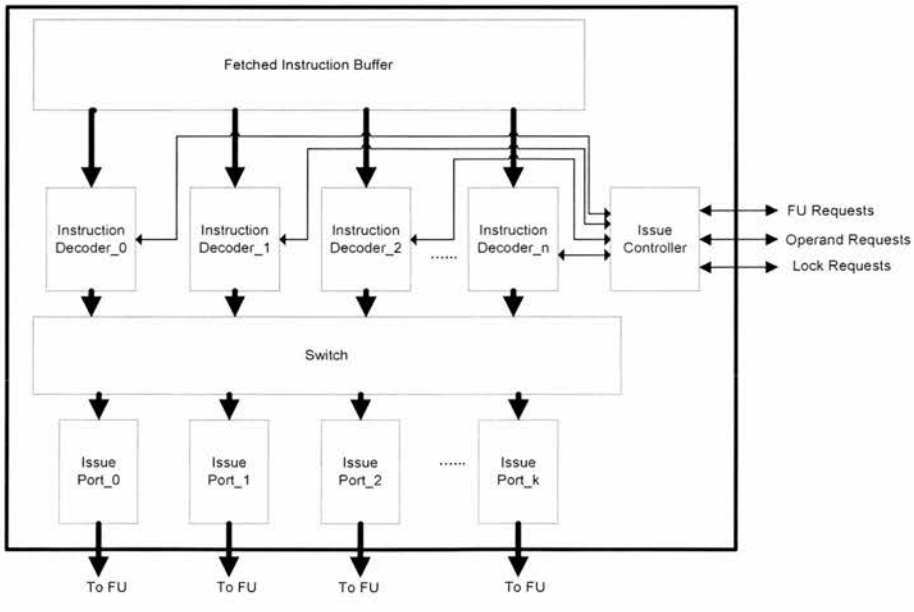


Figure 3-11: Block diagram of the Fetch Unit

The Thread Issue Unit contains five core components: fetched instruction buffer, instruction decoders, issue controller, switch network, and issue ports. The fetched instruction buffer receives 32-bit MIPS-like instructions from the Fetch Unit. Then the instruction decoder breaks a macro-instruction into multiple micro-operations, whenever the instructions operates on more than two sources, or when the nature of the operations requires a sequence of unrelated operations. The issue controller checks all the specific conditions before macro-operations can be issued, e.g. whether a specific functional unit is ready, the required operands are ready, and the destination register is locked. As soon as the pre-requisite conditions are met, the control signals of micro-operations will then be routed to the output port of the corresponding functional unit.

The instruction decoder sub-module decodes an instruction into micro-operations depending on the opcode of the instruction. The MIPS-like [57][58] instruction set has been chosen for the MAPS+ architecture, which keeps instruction size constant, and bans the indirect addressing mode. As a result, the instruction decoder is simpler than that used in CISC architecture. With additional of multithreaded, RFU and FPU instructions to the basic MIPS instruction set, the MAPS+ instructions can be classified into six groups according to their coding format.

- R-Type - This group contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand. This includes arithmetic and logic operations with all operands in registers, shift instructions, and register direct jump instructions ***jalr*** and ***jr***. All R-type instructions use opcode '000000'.
- I-Type - This group includes instructions with an immediate operand, branch instructions, and load and store instructions. All opcodes, except '000000, 00001x, and 0100xx', are used for I-type instructions.
- J-Type - This group consists of the two direct jump instructions. These instructions require a memory address to specify their operand. J-type instructions use opcodes '00001x'.

- FPU Instructions - This group consists of the floating-point unit instructions. FPU instructions use '010001' opcode.
- Multithreaded instructions - This group consists of the thread-related instruction, e.g. ***frk***, ***wat***, ***psg***, ***sstp*** and ***cmt***. Thread instructions use '010010' opcode.
- RFU Instructions – This group consists of the reconfigurable functional unit instruction. RFU instruction uses '010011' opcode.

Multithreaded instructions are used for thread spawning, inter-thread communication, and thread synchronisations. In this thesis, a subset of thread instructions defined in [59] has been chosen. Descriptions of the detailed thread instruction and operations in pseudo-C language are summarised in Table 3-2 and Table 3-3.

Instruction	**Description & operations**
frk **$d, L**	Fork a new thread to execute target label L. Return d=TRUE, if successful. *REG frk (address L) {* *if (Find a free TPU) {* *d = TRUE;* *Associate start address L to new Thread;* *Update thread information table;* *Broadcast thread information to other TPU;* *Acknowledge Scheduler;* *}* *else $d = FALSE;* *return $d;* *}*
psg **$s1,$s2**	If guard register $s1 is not set, ignore current operation; Otherwise, the register s1 contains the destination thread number, and data in register s2 will be passed to destination thread. The received data is stored in a temporarily buffer. *void psg(REG $s1,REG $s2)* *{if ($s1) Send data in $s2 to destination thread $s1;}*

Table 3-2 : Descriptions and operations of multithreaded instructions

Instruction	Description & operations
***wat* $s1, $s2**	If guard register $s1 is set, wait until data is received. The received data is written in register $s2. *void wat(REG s1, REG s2) {* *if (s1) {* *Wait until data received;* *Write Data to register s2;* *}* *}*
***sstp* $s1, $s2**	If guard register $s1 is not set, the current operation will be ignored. If current thread contains children threads, send synchronisation stop signals to all of them. Until synchronisation acknowledgement signals received, current thread stops. Then activate the head thread of $s2 *void sstp(REG s1, REG s2) {* *if (s1) {* *if (Contains children threads)* *{* *for (all the children threads)* *{* *Send synchronisation signals;* *}* *Wait synchronisation signals;* *}* *Stop current thread;* *Activate head thread of s2* *Update thread information table;* *Broadcast information to other TPU;* *Acknowledge Scheduler;* *}* *}*
***cmt* $s1**	If guard register $s1 is set, wait for synchronisation signal from scheduler and commit speculative store to memory. *void cmt(REG s1) {* *if (s1) {* *Wait for synchronisation signal;* *Commit speculative store;* *}* *}*

Table 3-3 : Descriptions and operations of multithreaded instructions

Further, the format of the fields of the multithreaded instructions has been illustrated in Table 3-4. When a multithreaded instruction is decoded, the TIU will issue such instruction to the Thread Control Unit (TCU). In the ***frk*** instruction, the register field specifies the destined register number for holding the thread spawning result. In current MAPS simulator, a sixteen-bit immediate field is used to specify the target address of the new thread, therefore code blocks sizes are limited to 64 KB. In order to support large code blocks size, the target address length have to be extended. For example, a thirty-two bit field can support up to 4GB code blocks size.

The rest of the multithreaded instructions have the same format for the fields. The *GUARD_REG* guarantees the correctness of multithreaded instructions, e.g. if the *GUARD_REG* is set to be *TRUE,* then the instruction will be issued to TCU; otherwise, the TIU ignores such instructions and proceeds to decoding the next available one. As a thread spawning operation might fail due to lack of TPU resources at run-time, the guarding mechanism ensures the sequential operations are performed correctly.

Instruction	**opcode (6)**	**function (5)**	**rs(5)**	**immediate (16)**	
frk $d,L	010010	00001	REG_NUM	TARGET_LABEL	
Instruction	**opcode (6)**	**function (5)**	**rs(5)**	**rt(5)**	**reserved(11)**
psg $s1,$s2	010010	00010	GUARD_REG	DATA_REG	N/A
wat $s1,$s2	010010	00011	GUARD_REG	DST_REG	N/A
cmt $s1	010010	00100	GUARD_REG	N/A	N/A
sstp $s1,$s2	010010	00101	GUARD_REG	HEAD_REG	N/A

Table 3-4 : Format of multithreaded instructions with fields

In the MAPS+ architecture, the RFUs are tightly-coupled to the TPU architecture. To configure the RFU to perform a specific computation, the configuration needs to be loaded into the RFU, and the input to the RFU is fetched from TPU register file. By grouping several instructions into single Reconfigurable Functional Unit Operation (***ROP***), there will be a number of source and destination registers in the ***ROP***. Therefore a normal 32-bit MIPS instruction is not adequate to hold the information for a ***ROP*** instruction. The length of a ***ROP*** is 128 bits and the '010011' opcode is chosen. The opcode is reserved for coprocessor instructions in MIPS instruction set. The reason

choosing a 128-bit ***ROP*** instruction instead of using variable length is to simplify the decoding logic. Also based on our observation of current simulated benchmarks, a 128-bit ***ROP*** is able to hold the information of grouped instructions. However, the implementation limits the flexibility of ***ROP*** instruction.

Details of the fields are listed in Table 3-5. The ***seq*** field specifies the order of the four words, and the issue controller has to guarantee that all four words are fetched and decoded in the correct order. The ***function*** field specifies the ***ROP*** sequence number (*ROP_NUM*), which is used to load the corresponding RFU configuration from memory. The *ROP_NUM* is a unique number for different RFU configuration bit streams. With a lookup table, the sequence number can be translated into a memory address, where the configuration binary can be loaded into RFU. The ***rs*** and ***rt*** fields provide up to eight operands for an ***ROP*** instruction, and the ***rd*** field provides up to four results that can be written back to register files. Zeros are filled in any of the unused fields.

As the MAPS+ TPU architecture has two on-chip bus (X_BUS, Y_BUS), the operands into the RFU have to be fetched sequentially. The throughput can be improved by increasing the number of on-chip buses and operand fetch ports. In the current two on-chip bus model, the order of operands fetched is controlled by the issue controller, and the corresponding handshaking signals to synchronise RFU operations is also controlled by it. The lock signal for the destination register will be issued to the register file to avoid any incorrect write-back order.

Instruction	opcode (6)	seq (5)	rs(5)	rt(5)	rd(5)	function (6)
rop_1 $d1,$d1,$d2,$d4,$s1,$s2,$s3,$s4,$s5,$s6,$s7,$8	010011	00000 (SEQ_NUM)	SRC_REG1	SRC_REG0	DST_REG0	00001(ROP_NUM)
	010011	00001 (SEQ_NUM)	SRC_REG3	SRC_REG2	DST_REG1	00001(ROP_NUM)
	010011	00010 (SEQ_NUM)	SRC_REG5	SRC_REG4	DST_REG2	00001(ROP_NUM)
	010011	00011 (SEQ_NUM)	SRC_REG7	SRC_REG6	DST_REG3	00001(ROP_NUM)

Table 3-5 : Formats of the fields for reconfigurable functional unit instructions.

3.4.4 Reconfigurable Functional Unit

A number of architectures [63][64][65] have coupled RFUs in their datapath. The fabric of the RFU is based on multiple SRAM-based Field Programmable Gate Arrays (SRAM-FPGAs), and the RFU customises the hardware to a specific application by mapping it to the reconfigurable logic. The RFU is able to customise the hardware to a specific application by mapping it to the reconfigurable logic and the fabrics is based on multiple SRAM-based Field Programmable Gate Arrays (SRAM-FPGAs). Traditionally, reconfigurable logic for a RFU is implemented in a synchronous manner.

Although designs for asynchronous FPGAs [63][64][65] have been proposed, the commercial CAD tools are still designed for synchronous FPGAs. Several factors have influenced the choice of a synchronous FPGA fabric for the RFU in the MAPS+ architecture, e.g. the lack of tools for designing and verifying an asynchronous FPGA, manual check for the worst case timing of data path, and the difficulties in automatic hardware code generation.

The RFU architecture for the MAPS+ is shown in Figure 3-12. Asynchronous wrapper around the synchronous reconfigurable arrays provides the necessary asynchronous-to-synchronous input interfaces and the synchronous-to-asynchronous output interfaces. The implementation is similar to the coarse bundled data interface used by Philips for their 8051 [126][127] memory interface.

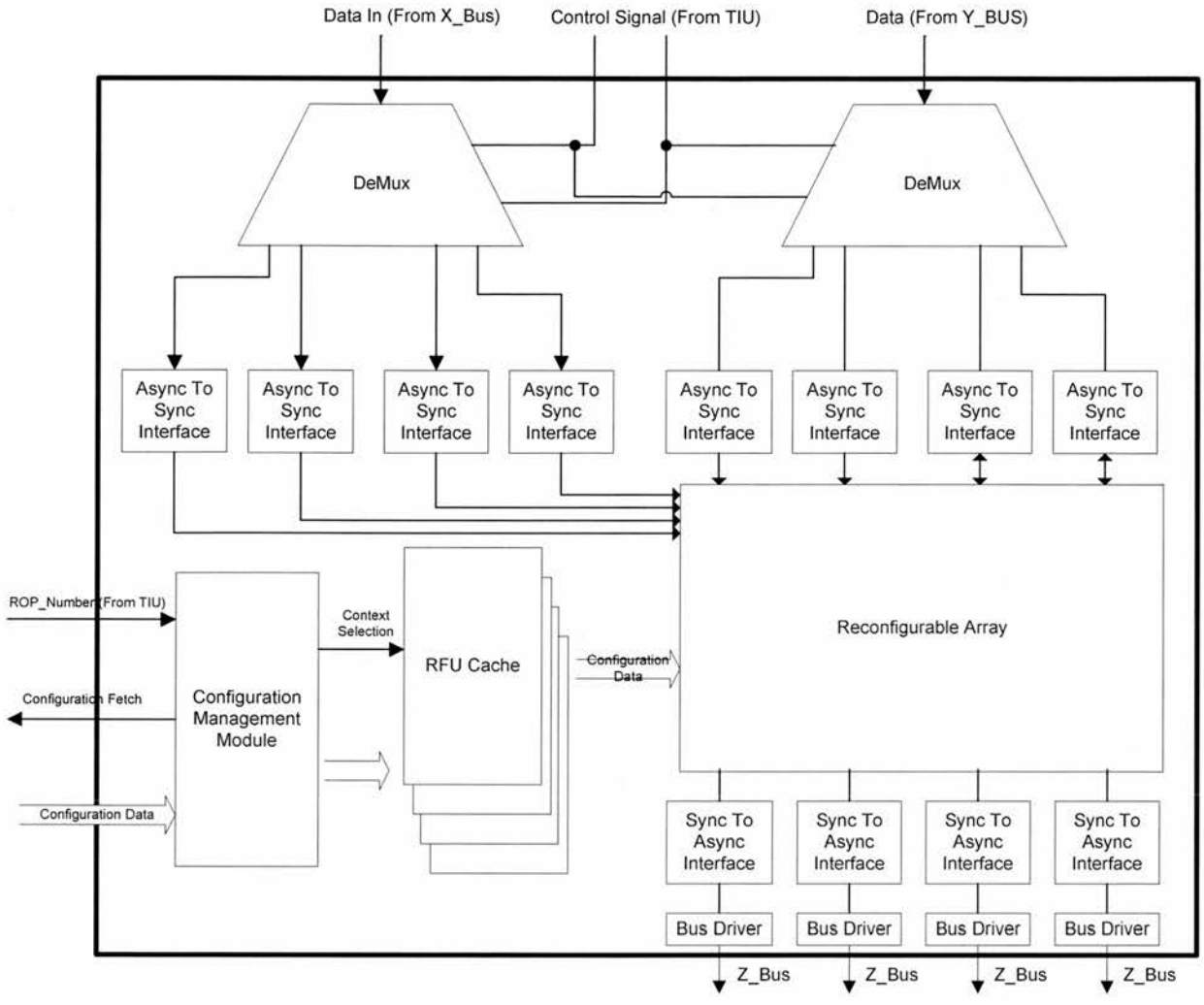


Figure 3-12: Block diagram of the Reconfigurable Functional Unit

The core of the RFU is the synchronous Reconfigurable Array, which contains synchronous Logic Blocks similar to the Configurable Logic Blocks (CLBs) used in Xilinx Virtex II FPGA [66]. As shown in Figure 3-13, each logic block comprises four similar slices with fast local feedback within the Logic Block. Two programmable 4-input LUT, carry arithmetic and logic gates, function multiplexers and two storage elements are included in the logic block. The logic blocks can be programmed to implement the functionality of basic logic (e.g. *AND, OR, XOR, INVERT*) or more complex combinatorial functions. Furthermore, a hierarchy of programmable interconnect allows the logic blocks of the reconfigurable array to be interconnected and controlled by the configuration bitstream.

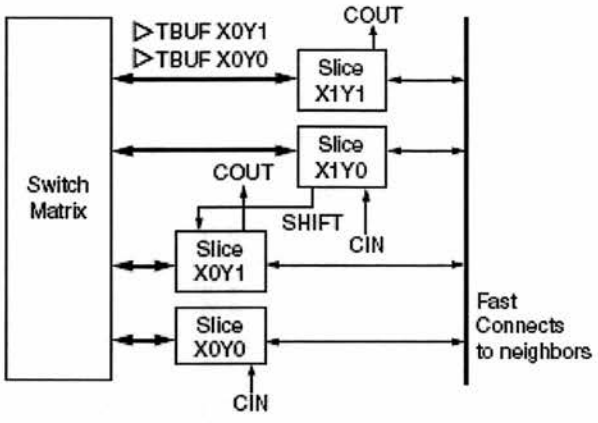


Figure 3-13: Logic block diagram (reproduced from [66]).

The RFU is triggered by the Thread Issue Unit (TIU) control signals. When the Configuration Management Module (CMM) receives the ROP_NUM of current active RFU context, it performs a search on the table. If a matched context already exists in the RFU cache, the CMM triggers a context switching operation by activating the context layer in the RFU cache. RFU cache in the MAPS+ architecture is a multi-context implementation, which can hold one or more entire configurations for the reconfigurable array. As a result, the reconfiguration overhead can be reduced. In the case of mis-matching of configuration context, the bitstream will be downloaded and reconfiguration operations are then triggered. Therefore, the ***ROP*** suffers from reconfiguration overhead, which might stall the MAPS+ processor. The configuration bitstream is loaded from the shared memory via the fetching interface. After receiving the configuration bitstream, the CMM is responsible for inserting the 32-bit configuration chunks at the correct physical location in the reconfigurable cache. If the cache is not large enough for holding the new configuration bitstream, a memory replacing operation will need to be performed. The RFU supports the download of full and partial configurations for any of the physical contexts.

The I/O interface to the Reconfigurable Array in the RFU component in Figure 3-12 shows that eight input operands and four output results are available. The control signals from the TIU controls which input port of the RFU will be used to store the current operands fetched via the X_BUS and the Y_BUS. After the execution in the

Reconfigurable Array is complete, results are written back to the register file via the Z_BUS. The Bus Driver controls the arbitration access to the Z_BUS.

Given that the synchronous reconfigurable array is embedded in the asynchronous MAPS+ architecture, interface wrapper circuits are required to permit communication between a locally synchronous module and the other asynchronous module. The I/O interfaces are based on the GALS point-to-point communication scheme used in [67].

An interface between an asynchronous producer and a synchronous consumer is shown in Figure 3-14 . When no data is being presented, the output clock is inverted and then fed to a calibrated delay line to one of the inputs of an arbiter. This oscillatory process continues and a stable clock signal is produced. After the arbiter grants the input request, data is latched, and the synchronous reconfigurable logic will be presented with a rising clock edge which latches the incoming data in the final set of latches.

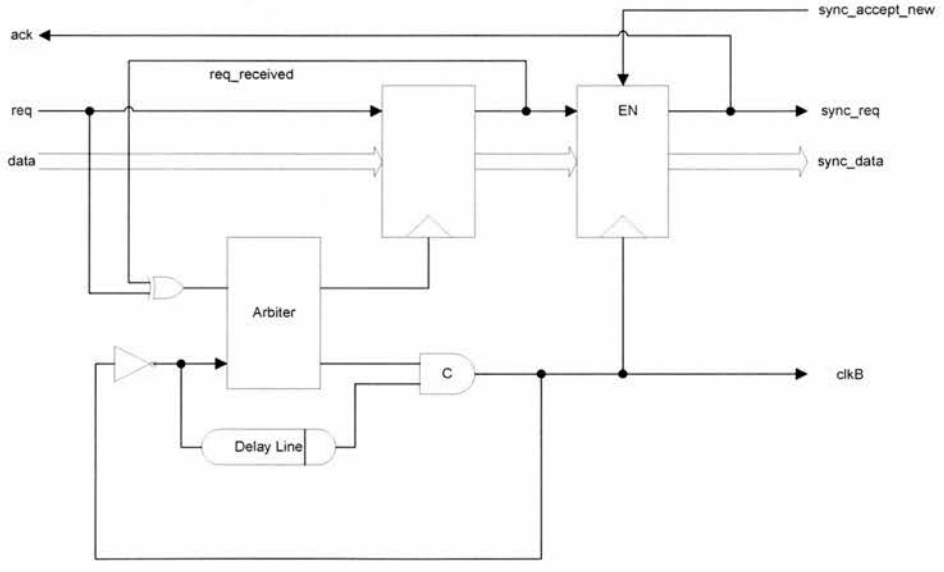


Figure 3-14: Interface between an asynchronous producer and a synchronous consumer (reproduced from [67])

On the other hand, the results need to be written back. The interface between a synchronous producer and an asynchronous consumer is shown in Figure 3-15. Handshaking signals with the data are required to indicate to the asynchronous system when new data is available. The synchronous state machine always waits for a

sync_ack signal between sending data items, the *req* signal will always toggle and data will be sent correctly.

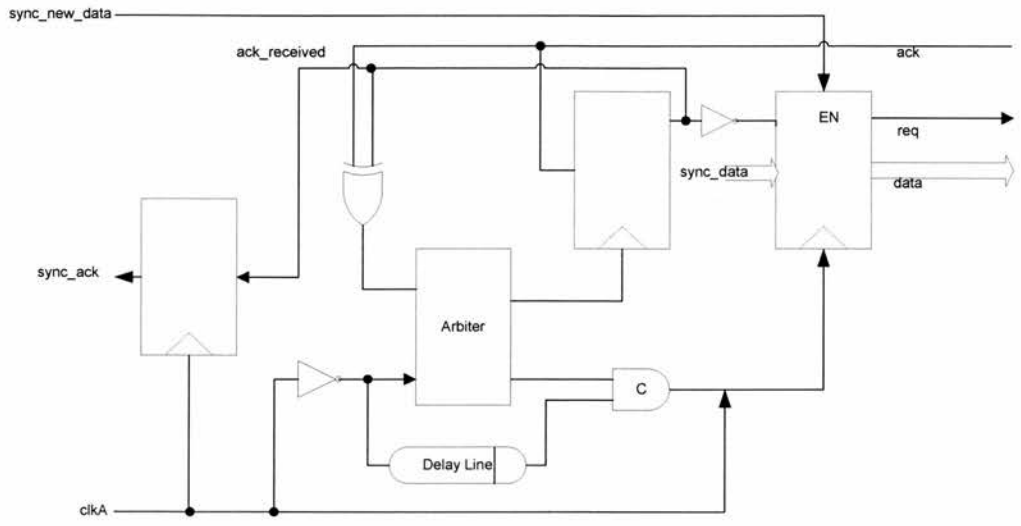


Figure 3-15: Interface between an asynchronous producer and a synchronous consumer (reproduced from [67])

3.4.5 Thread Control Unit

The Thread Control Unit (TCU) controls the execution of the multithreaded instructions and inter-thread communication and synchronisation. The MAPS+ architecture relies on the compiler to identify the threads which can execute concurrently. The block diagram of the TCU is shown in Figure 3-16. After the TIU decodes the instruction, thread control signals are passed to the TCU, and the corresponding threading operations will be performed.

The Finite State Machine (FSM) Controller of the TCU provides the model for state transitions triggered by control signals from TIU. A state register in the FSM stores the current state of the TCU, which reflects the history of input changes. For example, if a *WAT_REQ* signal goes up (triggered by TIU), current state will be *WAT_STATE*. As a result, the corresponding signals for MUX, I/O registers and asynchronous handshaking will be generated. Asynchronous state machine design is unlike synchronous synthesis, as logic must be implemented without hazards, and state codes

must be chosen carefully to avoid critical races. The design of asynchronous FSM is presented in [68].

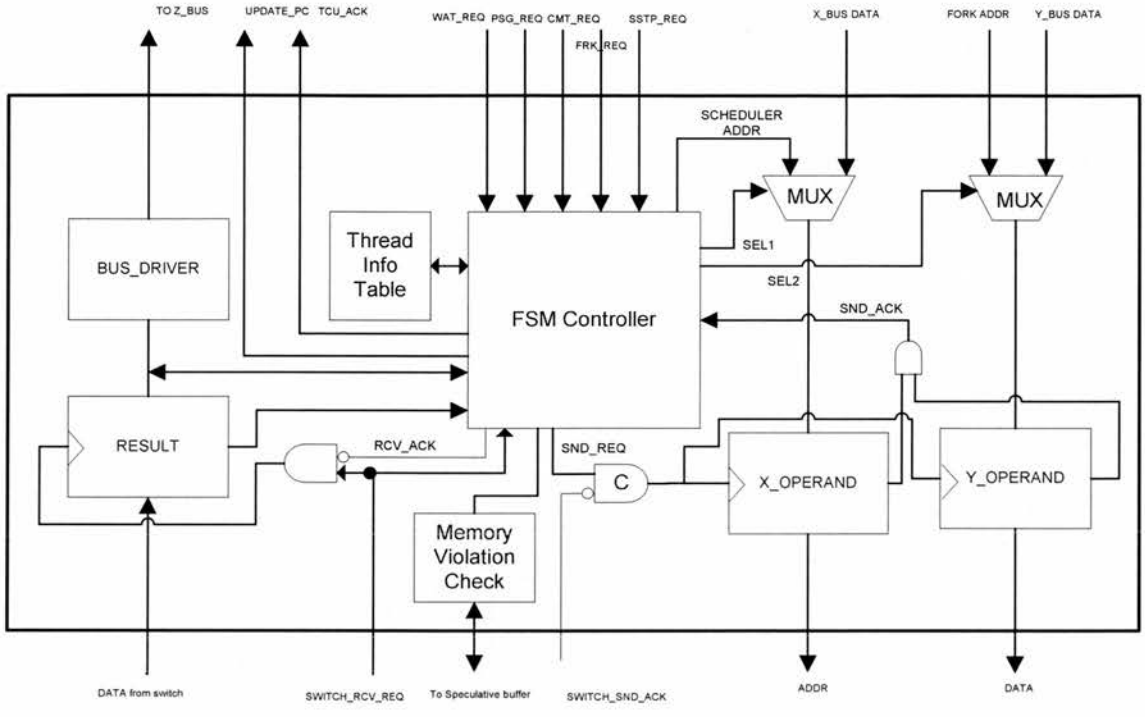


Figure 3-16: Block diagram of Thread Control Unit

The *FRK_REQ* signal to the FSM controller triggers a fork operation, which attempts to create a new thread in the next available TPU at the start address of the target label. First, a preliminary check in the Thread Information Table is performed. If no TPU is available, then the FSM controller will direct *NACK* signal back to the TIU controller, and an *INVALID* (e.g. -1) value will be written back to the Z_BUS. If one or more TPUs are available, then selection signals (via *SEL1, SEL2*) will be forwarded to the MUX. The ***fork_addr*** is a 16-bit immediate value issued by TIU, which is the target thread start address, which is next forwarded to the scheduler via the switch. Finally, the result of the fork result will be written back to the register.

For the destination TCU, the fork operations are different. The scheduler triggers the destination TCU to start a new thread context from the target address. If the *SWITCH_RCV_REQ* is raised, then the FSM controller raises the *RCV_ACK* signal to

enable the fork command moving into the result register. The FSM controller updates the PC and starts a new thread context.

When a TPU performs unsafe memory access operations, it operates speculatively and the speculative buffer is enabled by setting the tag. Data is written into the speculative buffer, instead of to the shared memory. When a *CMT_REQ* signal is received, the TCU starts to check for violations of any speculative memory accesses. If none is found, then these stores are flushed to the memory. If the thread stops without committing these stores, then the speculative buffer is simply cleared. Any failure of the ***cmt*** (COMMIT) operation causes the program counter to restore the last successful committed point, and the register image will be rolled back.

While the tag of the speculative buffer is set for a load operation, the TPU should see the latest version of the data as if the program has been executed in the sequential order. It checks the load address in its own buffer. If the address is not found, then it keeps looking in the predecessors' buffers via the switch. Finally, if the address is not found in any of the predecessors' buffers, then the data will be loaded from the memory. The information as to which threads are the predecessor of the current thread is stored in the Thread Information Table in the TCU, which has similar data structure and operations to the one described in Scheduler section 3.5 (p76). The scheduler will always broadcast the updated thread information whenever any thread structure changes.

In a non-speculative design, all thread-level parallelism is exposed by the compiler using techniques such as data dependence analysis, and memory disambiguation. Due to the lack of run-time information, compile-time thread partitioning will be conservative, i.e. two instructions will be considered to be dependent unless proven otherwise. Furthermore, for a speculative design, each speculative thread created is then checked at runtime. If it is detected that a speculative thread reads a memory location that is later written by a thread that comes earlier in the sequence, then that memory violation would be spotted and the speculative thread is squashed. As a result, hardware support for speculative thread operations is required. In the MAPS+ architecture, the hardware memory violation check block validates all speculative

memory operations. In the event of a violation, FSM controller triggers further rollback operations.

As shown in Figure 3-16, the *SSTP_REQ* signal triggers a synchronisation stop for the current thread context when the guard register is set. And the FSM controller sets the current state to idle. More details of the synchronisation stop will be described in the scheduler section 3.5. The pair ***wat*** and ***psg*** instructions perform inter-thread communications. *PSG_REQ* starts a data passing operations to the destination TPU. The FSM controller chooses the data on the X_BUS and Y_BUS as the *X_OPERAND* and *Y_OPERAND* inputs, respectively. The *Y_OPERRAND* data will be written to destination TPU, indicated by the TPU number stored in *X_OPERAND*. The overall architecture is shown in Figure 3-16, which is highlighted in Figure 3-17.

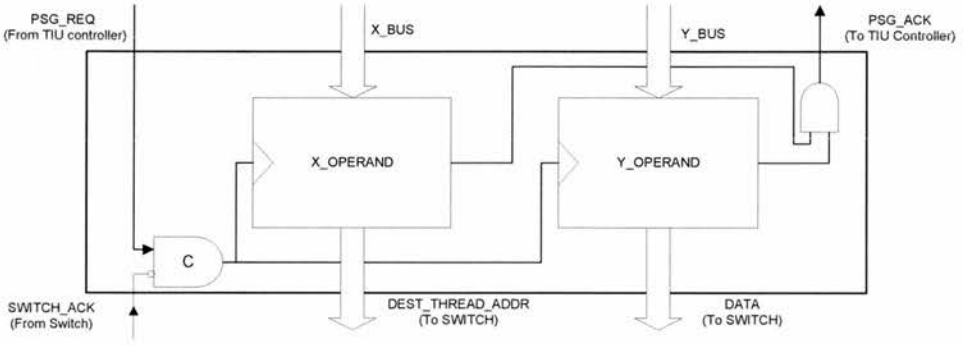


Figure 3-17: Logic for the *psg* instruction.

WAT_REQ triggers a stall to wait for the incoming data, and the implementation is shown in Figure 3-18.

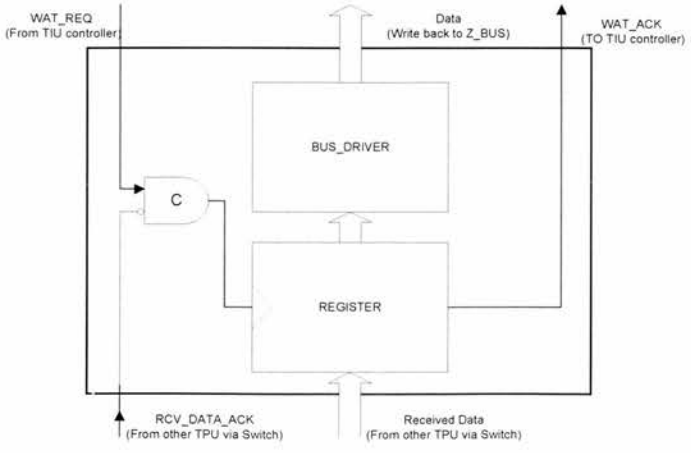


Figure 3-18: Logic for the *wat* operation.

3.4.6 Data Value Predictor

Thread level parallelism is limited due to the true data dependencies between producer and consumer threads. Figure 3-19 shows a data dependency between the producer and consumer threads, which causes the consumer to stall waiting for the result.

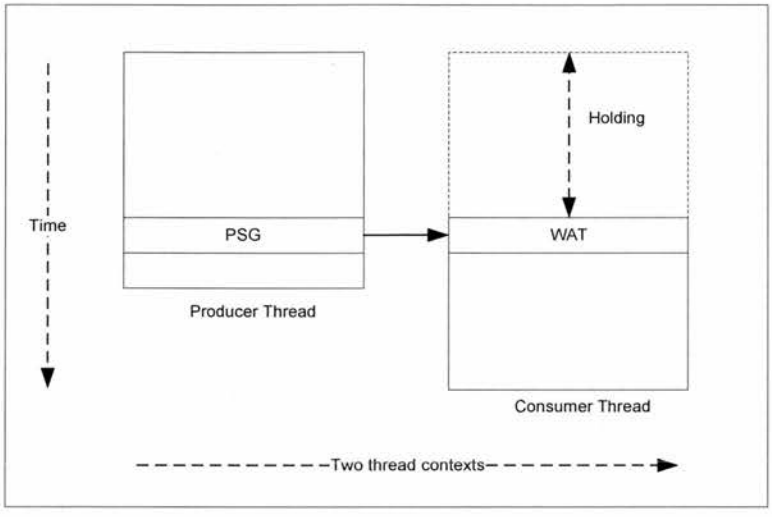


Figure 3-19: Data dependency between the producer thread and consumer thread

One possible way to overcome the limitations imposed by data dependences is to use data value prediction. The result of the ***wat*** instruction is predicted based on the history of data value patterns, and passed on to subsequent instructions of the consumer thread that depends on the result. When the value is finally available in the producer thread via the ***psg*** instruction, the consumer compares the correct result with the value predicted earlier. In Figure 3-20 (a), the values match, and the results of subsequent dependent instructions are confirmed. On the other hand, as shown in Figure 3-20 (b), if the values do not match, then a rollback operation is triggered in the consumer thread, and the correct result is forwarded to instructions that required this value, and those instructions are re-executed.

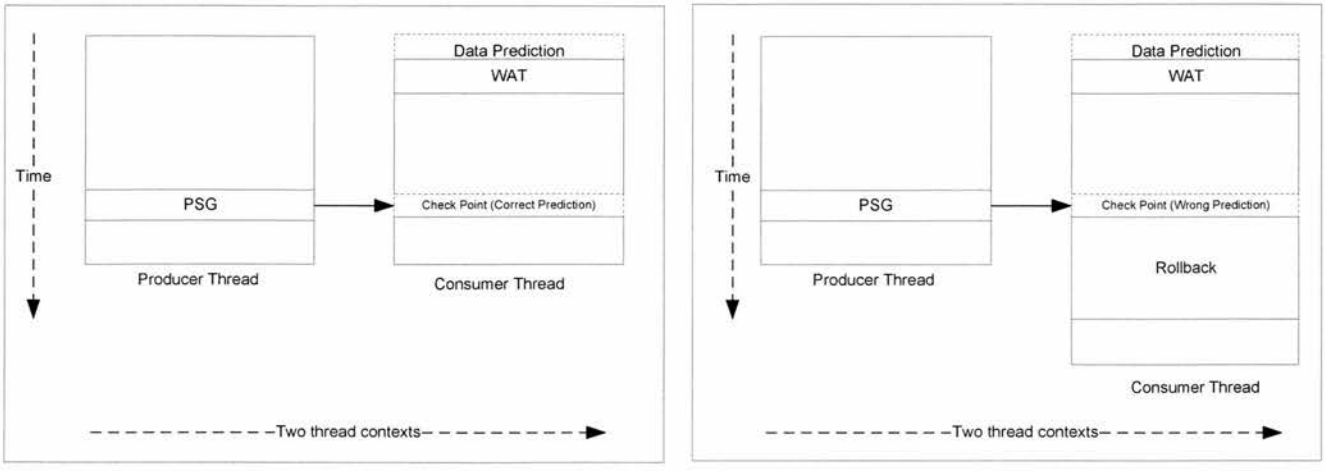


Figure 3-20: (a) Correct data value prediction (b) Wrong data value prediction

In order to support data value prediction in the MAPS+ architecture, a hardware implementation of the data value predictor was integrated in every TPU, which is used to predict the result of data-producing instructions based on their past behaviour. It records the recent results produced by previous ***psg*** instruction, and predicts the result of the ***wat*** instruction's next instance based on history results.

Several data value predictor have been proposed. The simplest form is the Last Value predictor (LVP) [74][75] scheme. It stores the result produced when the instruction was executed for the last time, and predicts the same value when the same instruction is executed next. A Stride Value Predictor (SVP) [72][73] scheme assumes that the stride between two consecutive values is constant, and works in the case of loop induction variables and programs stepping through arrays in a regular fashion. A Context-based Value Predictor (CVP) [76][77] scheme links a value to a context (an ordered sequence of recent values) and predicts this value when the same context occurs again. It is also capable of capturing constant and stride patterns to some degree, but its learning phase is longer than the last value predictor or stride predictor. Hybrid predictors [78][79] obtain good value prediction accuracy by combining multiple value prediction schemes that exploit different data value locality patterns.

In this thesis, a hybrid version of the stride- and context-based predictors was chosen, which is identical to the approaches used in [79][80][81]. As shown in Figure

3-21 , the data value history table contains seven fields- ***Tag, State, Stride, Value,*** Correctness of Stride Predictor (***S_correctness***) and Correctness of Context Predictor (***C_correctness***), and the size of the table is 32 K. The ***Tag*** field stores the identity of the ***wat*** instruction that is currently mapped to that entry, and the ***Value*** field stores the last results. The State field has one of three values – ***Init, Transient*** and ***Steady***. The pattern decoder follows the history patterns and predicts one of the values when the context repeats. When a ***wat*** instruction is received by the predictor, the hash function will convert the program counter of the instruction into a unique tag. After receiving the tag, the hybrid controller then generates a selection signal for the prediction scheme with higher correctness ratio, either stride-based prediction or the context-based prediction. The predicated data value will be stored in a temporary register, and the value will be compared with the correct value received from *PSG* value. Finally the prediction valid signal indicates whether current prediction was correct, and the controller will update the data value history table.

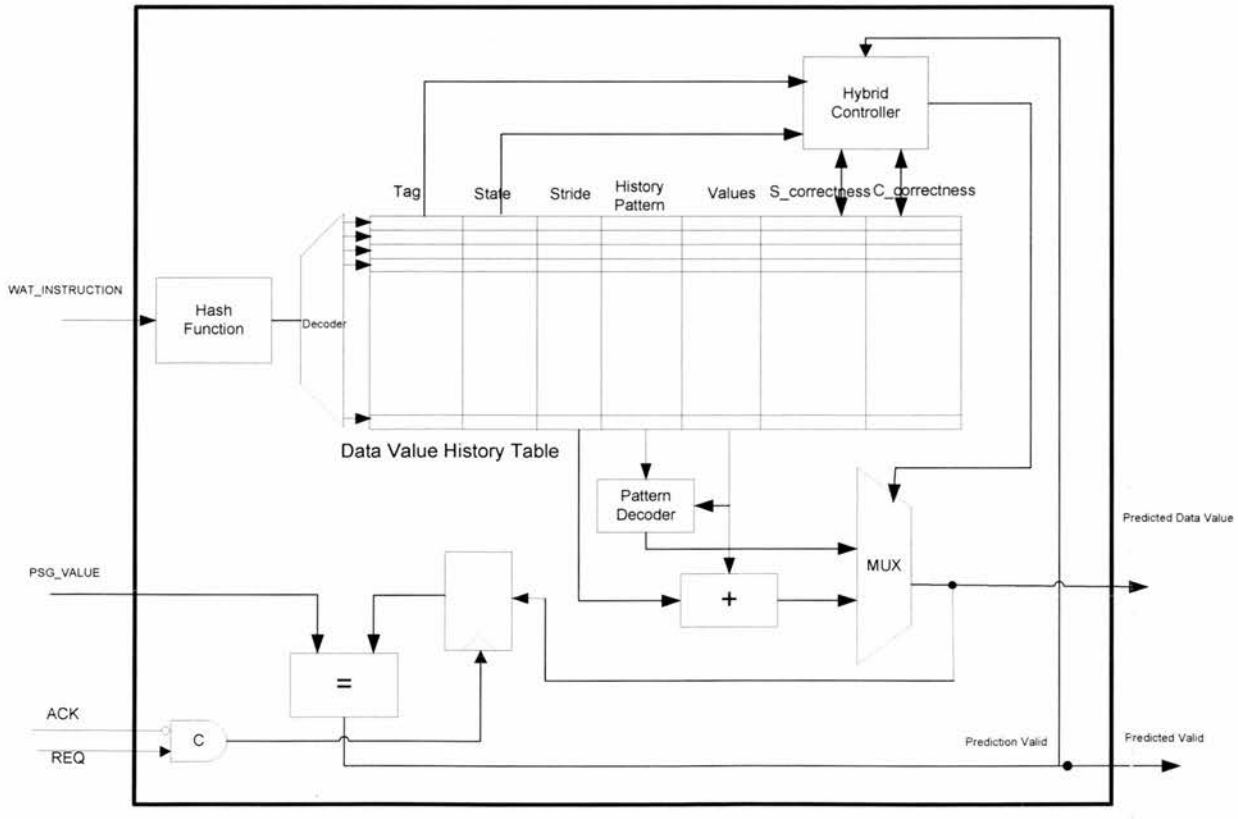


Figure 3-21: Block diagram of hybrid of stride and context predictor.

3.5 Scheduler

The MAPS+ architecture relies on both hardware and compiler support to extract thread level parallelism, which provides the ability to simultaneously process multiple threads. Each of these threads could correspond to different part of a program and runs on one of the multiple hardware contexts available in the multiple TPU architecture. A top scheduler is required to help schedule multiple threads spawning operations, synchronise inter-TPU communications, sequence memory write orders and context switching.

A Thread Information Table (Figure 3-22) is maintained in the scheduler to keep track of the program order of the threads; threads are not necessary spawned in the order in the original program, and speculative threads can spawn other threads themselves. The scheduler will broadcast changes in the thread structures to the local TCU of the TPUs.

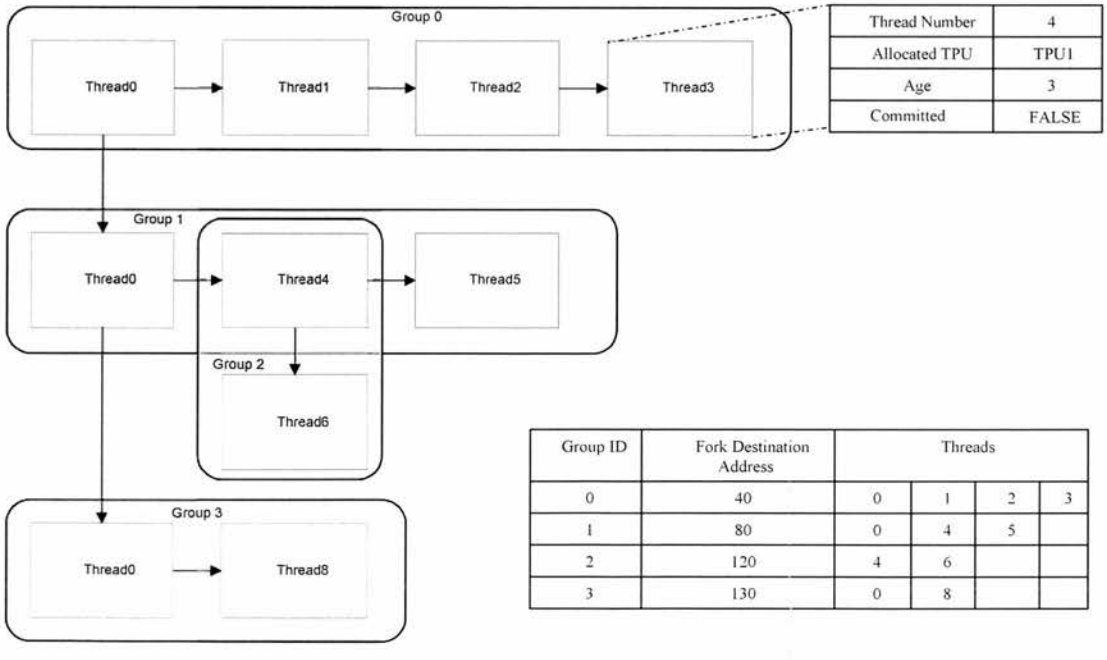


Figure 3-22: Thread Information Tables in Scheduler.

When a request to spawn a thread is received by the scheduler, the availability of TPUs is first checked. If all TPUs are busy, then the scheduler returns *INVALID* value

back to the source TPU, which results in a serial execution of the following code rather than failure; otherwise, the scheduler allocates a free TPU to the new thread context, which is assigned a unique thread number. Simultaneously, the thread number is returned back to the source TPU. Once the source TPU receives the new thread number as a result of successful fork, it will send its register context to the destination TPU (apart from the Stack Pointer register) via the switches. The scheduler allocates a section of shared memory space for the new thread, and the new Stack Pointer (SP) points to the top of the new memory space, and the SP register value will be passed to the destination TPU. Once the initialisation is complete, the new thread node will be added into the corresponding group and updated thread information will be broadcast to all the active TPUs. Finally, a *RESET* signal triggers the destination TPU to start the new thread context.

Two fields in the thread node are required for ***cmt*** operations: the *age* field and the *committed* field. An *age* field is a sequence number assigned to a new thread. The successor thread has an increment *age* field of one from its predecessor. When a thread is committed successfully, the *age* field of a successor thread is incremented by the total number of threads in the current group. This method is particularly useful for sequencing the order thread executions. Figure 3-23 illustrates the sequence for updating these two fields.

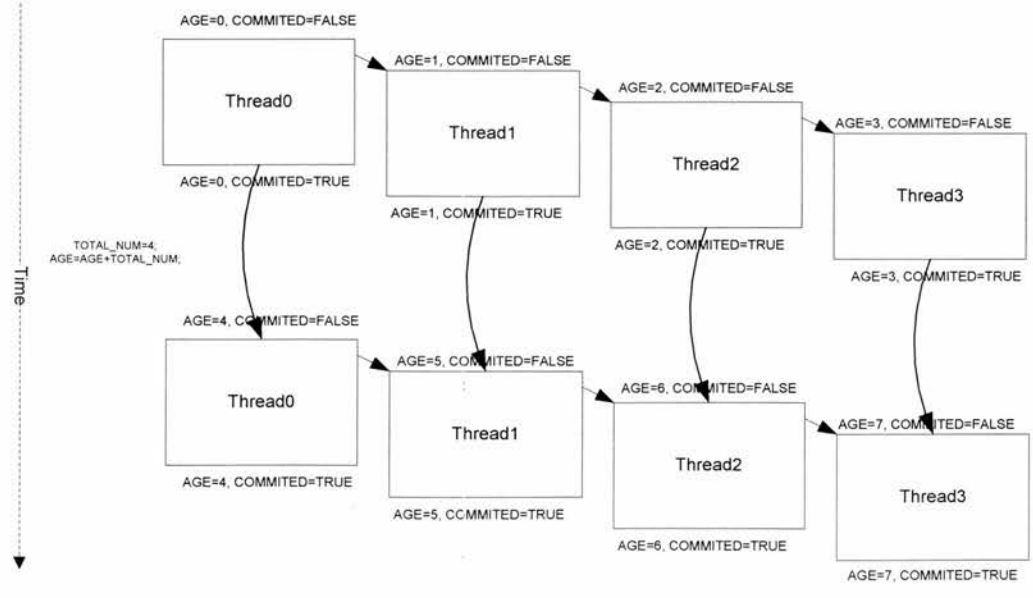


Figure 3-23: Thread Information Table updating sequence.

Thread synchronisation is achieved through the ***cmt*** operations and the synchronisation stop (***sstp***) operations. If a ***cmt*** request is received by the scheduler before an acknowledgement can be granted, then it needs to guarantee that the successors of this thread are already committed. As shown in Figure 3-23, the *committed* flag specify the status of current thread, and the *age* field sequence the order of the commit operation. As a result, if there are threads with younger *age* fields which are not committed, then the request will be placed in a holding queue. Once the requirement is met, the acknowledgement signal will be granted and its *committed* flag will be set. The ***sstp*** operations are similar to commit operations and are required to follow the sequence order of *age* fields. When receiving a ***sstp*** acknowledge from the scheduler, the TCU will update the Thread Information Table and broadcast the updated information to all active TPUs.

3.6 Discussion

The chapter has described the asynchronous MAPS+ architecture which supports multithreaded execution, thread speculation, and reconfigurability provided by the FPGA fabric in each node.

3.6.1 Micronet design style

A micronet-based [28][29][30] asynchronous hierarchical design style had been adopted in the design of the MAPS+ architecture. Micronets are networks of entities that compute concurrently and communicate asynchronously. The entities are themselves networks in a recursive fashion, leading to a hierarchical approach to system design. Self-timing logic is integrated into the switches and data communications between any two TPUs strictly follows a full handshaking protocol. The input channel of each port receives the data and the output channel is responsible for feeding outgoing data into other switches. Within every TPU, the communicating microagents effectively allow the functional units to transfer handshaking signals between themselves and their neighbours.

3.6.2 Speculative threads and data predictions

The MAPS+ architecture exploits thread-level parallelism via multiple Thread Processing Units, which has proved to be an effective way to improve the performance in several synchronous designs [32][33][34][35]. Speculative threading is supported by the MAPS+ architecture, which provides higher degree of parallelism and a significant gain in performance by removing constraints due to data dependencies. Successful speculation leads to the speedup in the execution speed; if unsuccessful, the speculative threads are squashed and the results discarded. Additional hardware components are provided to trace memory and data dependencies to help recover the correct state. A speculative buffer is integrated in each TPU to handle memory access violations. If a speculative thread has read a memory location that has later been written to by a thread that comes earlier in the sequence, then memory violation is spotted. A hybrid data predictor helps to break the sequence order of the threads due to data dependencies, allowing the speculative consumer threads to proceed without waiting for these values to be produced by the producer threads. If the data value prediction was correct, then the code is parallelised with gain in performance. Otherwise, the thread is squashed and rerun with correct values. The benefit of speculation has to be weighed against the requirement of a more complex hardware and extra energy consumption for thread speculations. (See Chapter 6 for simulation results)

3.6.3 RFU with asynchronous wrapper

A Reconfigurable Functional Unit (RFU) has been introduced in previous architectures [63][64][65], which demonstrated the potential for achieving higher performance by grouping several instructions into Reconfigurable Functional Unit Operation (***ROP***) and executing on the FPGAs. But all these design have been used in synchronous systems. Although asynchronous FPGAs been investigated in the past, e.g. AFPGAs [69], Montage [70], PCA-1 [71], the lack of commercial asynchronous FPGA implementation and availabilities of asynchronous FPGA CAD tools, have influenced

our choice of a synchronous reconfigurable fabric design for the RFU. Asynchronous wrapper around the FPGA results in islands of synchronous blocks in the asynchronous MAPS+ architecture.

3.7 Summary

The MAPS+ architecture integrates the features of asynchronous design, multithreaded architecture, thread speculation, data prediction and reconfigurable functional units for the first time. The concurrency exposed in the architecture has to be exploited by a compiler which is able to extract independent threads in the application software, and instruction-level parallelism in individual threads. The design of the compiler is the topic of the next chapter.

Chapter 4
Compiler Design for MAPS+ Architecture

4.1 Introduction

The MAPS+ architecture exploits both thread-level parallelism and instruction-level parallelism, and combines a multithreaded instruction set architecture and field programmable logic. To take advantage of the MAPS+ architecture, the compiler is required to generate concurrent threaded code and partitioning between software and hardware from the source program.

Most embedded applications are written in a combination of assembly code and sequential high level languages such as C or C++ with complicated data structures and control flows. Parallelisation methods in compilers have to make conservative assumptions regarding data dependencies, which can reduce run-time performance substantially. MAPS+ architecture provides the necessary hardware support to perform run-time dependency checking and speculation on the control or data dependencies. Therefore, compiler techniques are needed to take advantage of speculation. The compiler can partition a program into parallel speculative threads without having to prove their independence, while at runtime the underlying hardware checks whether inter-thread data dependencies are preserved, and threads affected by any such violation are re-executed. This chapter describes conventional loop threading compilation and speculative thread partitioning and transformation techniques.

MAPS+ architecture also requires the compiler to assemble RFU operations for the reconfigurable logic. One possible solution is to manually extract hardware code from sequential programs. Manual circuit description will create high-quality circuit designs. However, it demands significant hardware knowledge of a programmer. Automatic compilation techniques provide quick and easy ways to program the MAPS+ architecture, and makes reconfigurable hardware more accessible to general application programmers. In this chapter, compilation techniques for generating reconfigurable functional unit operations to perform the arithmetic and logic operations within the program have been investigated.

4.2 Thread Analyser

Program constructs such as loop and repeatedly executed parts of programs are prime candidates for thread partitioning. The sequential program is converted into the Static Single Assignment (SSA) form representative, which helps the compiler in thread identification.

4.2.1 Static Single Assignment (SSA)

A SSA [85] form can be employed as an efficient intermediate representation of sequential programs for their analysis and optimisation. Many advanced compilation techniques have been developed for optimisation [86] and parallelisation [87][88][89] based on the SSA form. Each variable read in the SSA form is directly linked to one definition. In order to allow this, ϕ-nodes are inserted at program points where more than one definition to a variable merges. A ϕ-node has a ϕ-function on its right-hand side (RHS). The ϕ- function has one operand for each merged definition, and each variable has exactly one definition. Variables are renamed to ensure that each variable has exactly one definition, which is usually done by adding a version number as the subscript of the original variable.

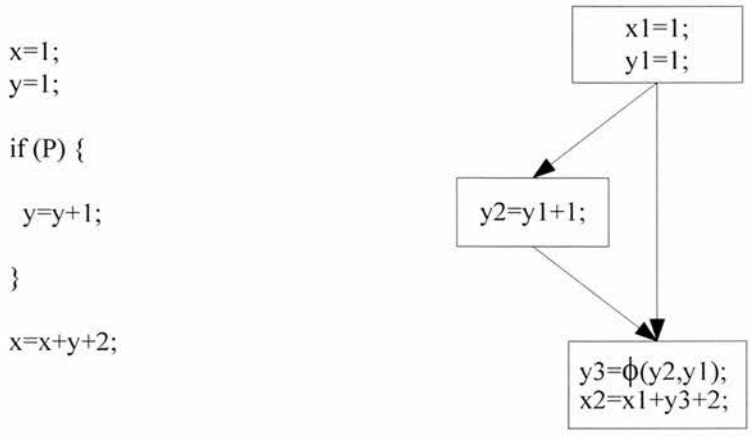


Figure 4-1: A sample program and its SSA form.

An example program and its equivalent SSA form are illustrated in Figure 4-1. In the program, there are two regular definitions that reach the reading of the variable y. One definition reaches it at the top of the program with the assignment of "y=1", and the other definition reaches the read via the if-then-branch. In order to indicate that both these definitions reach the read, a ϕ-node "y3 = ϕ(y2, y1)" is inserted at the merge point of the constant assignment, and the if-then-branch. This ϕ-node has a ϕ-function to represent a merging of the two definitions of y. A version number is also added to each appearance of the original variable so that each version of the original variable is defined only once, and the reading of a variable and the definition of the variable that directly reaches the read use the same version number.

The benefit of using an SSA form is that it encodes control-flow information and supports incremental updates to the SSA form during optimizations, such as unreachable branch elimination. Furthermore, with the SSA form, all the data dependences are illustrated clearly on the entry node of the loop body by using ϕ–nodes. The ϕ–nodes allow the compiler to identify data dependences because there is no aliasing in accessing these values, since all accesses must explicitly refer to the same variable name and static instructions that access these values occur only in the body of the loop being optimised, never in the procedures called from the loop. Therefore, it is straightforward to identify all accesses to these values and to determine their last

definitions and their first uses with SSA definition, resulting in an efficient flow-insensitive dataflow analysis.

4.2.2 Loop Partitioning

Loop-level parallelism is exploited for generating threads for the multithreaded MAPS+ architecture. Paralleling loops to extract concurrency in sequential programs has been studied in several previous researches, such as the Agassiz compiler [61], and the Hydra compiler [60]. In general, the multithreaded instructions for loop bodies are generated in two steps. Firstly, the benefit of parallelisation for each loop nest is evaluated, and the most profitable loop levels for parallelisation will be selected. Secondly, multithreaded code for spawning, committing, and synchronising threads is generated, for executing loop nests in parallel on the MAPS architecture.

Loops are good candidates for thread generations. Loops constitute a significant portion of the program execution time, and then it can have a large impact on program performance. The steps for parallelising loops are summarised below:

1. Label each basic block with execution time provided the profiling information in the SSA graph. Profiles provide more accurate expected number of iterations and dynamic instructions in the loop.
2. The execution time of entry nodes of each loop are sorted in descending order.
3. A threshold of 10% of loop coverage is set. All the loops above threshold are chosen as possible parallelising candidates.
4. Further refinements are used to filter the candidate loops. A small loop body might not achieve much speedup when parallelised. If the total number of instructions per loop invocation is less than 30, the loop will be taken out of the candidate threads.
5. For partitioning nested loops, the compiler considers both the inner loop and the outer loop for parallel execution. Depending on the available parallelism, the structure of the loop bodies, and the load balancing, either the inner loop, or the outer loop, or both can be designated as threads.

6. Finally, the compiler will perform loop unrolling or loop blocking to increase the size of each thread, and to provide each TPU with sufficient workload to exploit instruction-level parallelism.

Once the candidate loops are identified, multithreaded code is next generated based on the algorithm shown in Figure 4-2. After performing induction variable analysis, the identified induction variables are placed into a vector with annotation. A loop may have several induction variables, which are variables incremented or decremented by a constant value in each iteration and cause loop-carried data dependencies. Several induction variable recognition techniques have been proposed [90][91][92]. An algorithm based on [90] was implemented in our compiler framework. Strongly connected regions (SCR) in the SSA graph are used to find the basic linear induction variable, which is required to satisfy the following constraints:

- The SCR contains only one ϕ-function at the header of the loop.
- The SCR contains only addition and subtraction operators, and the right operand of the subtraction is not part of the SCR.
- The other operand of each addition or subtraction is loop invariant.
- Any loads and stores are to unsubscripted variables.

A SCR that satisfies all of these constraints defines a family of basic linear induction variables. After recognising the induction variables, the compiler selects one induction variable as the primary one, and replaces other induction variables with linear functions of the primary induction variable. By applying such induction variable substitution, the compiler need only generate one target store for the primary induction variable. Furthermore, the induction variable can be replaced by speculative operations on the thread processing unit, which can greatly reduce the overhead for data forwarding and synchronisation.

```
Loop_partitioner ()
{
  for each loop_body in candidate_loop_queue
  {
    /*Find the entry block of current loop body,  append a fork instruction into the entry block, the destination address i s
    the FRK_AFF_ADDR*/
    f1= NEW_FORK_INST(FRK_AFF_ADDR);
    b_entry = ENTRY_BLOCK(loop_body);
    APPEND_INST(b_entry,f1);
    b_aff=GENERATE_AFFILIATED_FORK_BLOCK(FRK_AFF_ADDR);
    APPEND_BLOCK(loop_body,b_aff);
    for each phi-node in loop_body
    {
      v_lhs=LHS(phi_node); /*Choose the left hand side variable of the phi_node*/
      if (v_lhs is induction_variable)
      {
        /*If v_lhs is an induction variable, then the linear induction instruction with the number current available TPU
          will be placed in the fork affiliated block, and the PSG/WAT synchronise overhead will be reduced.*
        OPCODE = GET_OPCODE(induction_inst);
        ind1 = NEW_INST(OPCODE, TPU_NUMER);
        INSERT_FRONT(b_aff,ind1);
      }
      else {
        /*If v is an not induction variable, then a pair of PSG/WAT instructions are needed to synchronise data passing.*/
          wat= NEW_WAT_INST(v_lhs);
          /*Then go through the right hand side of the phi_node,  then PSG instructions are needed
            to place after the instruction where the RHS variable is defined*/
          for each variable v_rhs in the RHS(phi_node)
          {
            inst=FIND_INSTUCTION_DEFINED(v_rhs);
            /*Here we still pass the left hand side variable v_lhs, because after SSA restoration, a compensation
              assignment instruction will be placed before the PSG instruction*/
            psg = NEW_PSG_INST(v_lhs);
            INSERT_AFTER_INSTRUCTION(inst,psg);
          }
          INSERT_AFTER(phi_node,wat);  /*Finally insert the WAT instruction after the phi_node*/
      }
    }
  }
  /*After the SSA restoration, assignment instruction to LHS variables will be placed, phi_nodes removed */
  RESTORE_SSA ();
}
```

Figure 4-2: Loop partitioning algorithm for the MAPS+ architecture.

When generating the multithreading code, the compiler will append a *FORK* affiliation block to the end of the loop body, which controls the spawn operations of the current thread. Furthermore, the induction thread instructions are also added to the block, which control the distance of the threaded loop body to match the available TPU numbers. With the ϕ–node, the compiler places a thread synchronisation instruction in proper place for variable definitions and usage. As mentioned in the previous chapter, two thread synchronisation instructions are used: ***psg*** and ***wat***. For the right hand side of a ϕ–node, the compiler can identify the definition places for the current variable, where the ***psg*** instruction for the producer thread will be placed. For the left hand side of the ϕ–node, the compiler can identify the usage place of current variable, where a ***wat*** instruction will be placed. The ***wat*** instruction stalls execution until the value is produced by the previous thread when prediction is disabled. When data value prediction is enabled, the ***wat*** value could be produced by the data value predictor. Pairs of instruction of ***psg*** and ***wat*** insertions need to meet a number of constraints listed below, which are similar to those defined by B.Zheng [93]. A correct program can be created by trivially placing all ***wat*** instructions at the beginning of each thread and all ***psg*** instructions at the end of each thread to satisfy the first three constraints. However, such a transformation would completely serialise execution. To remedy this situation, two additional constraints, are used to improve performance.

1. A ***wat*** instruction must occur before any use of the variable on any path.
2. A ***psg*** instruction must occur after the last definition of the variable on any path.
3. A ***psg*** must occur for all synchronised variables on all possible execution paths.
4. A ***wat*** instruction should be placed as late as possible.
5. A ***psg*** instruction should be placed as early as possible.

Figure 4-3 shows a sample loop program in SSA form and its multithreaded version using the loop partitioning algorithm in Figure 4-2. Obviously, the algorithm can be improved by further optimisation techniques, such as loop unrolling, loop normalisation, or loop skewing. Such techniques increase the granularity of parallel

loops and increase the size of the computation stage, which will be executed in parallel, and thus enable larger portions of threads to be overlapped.

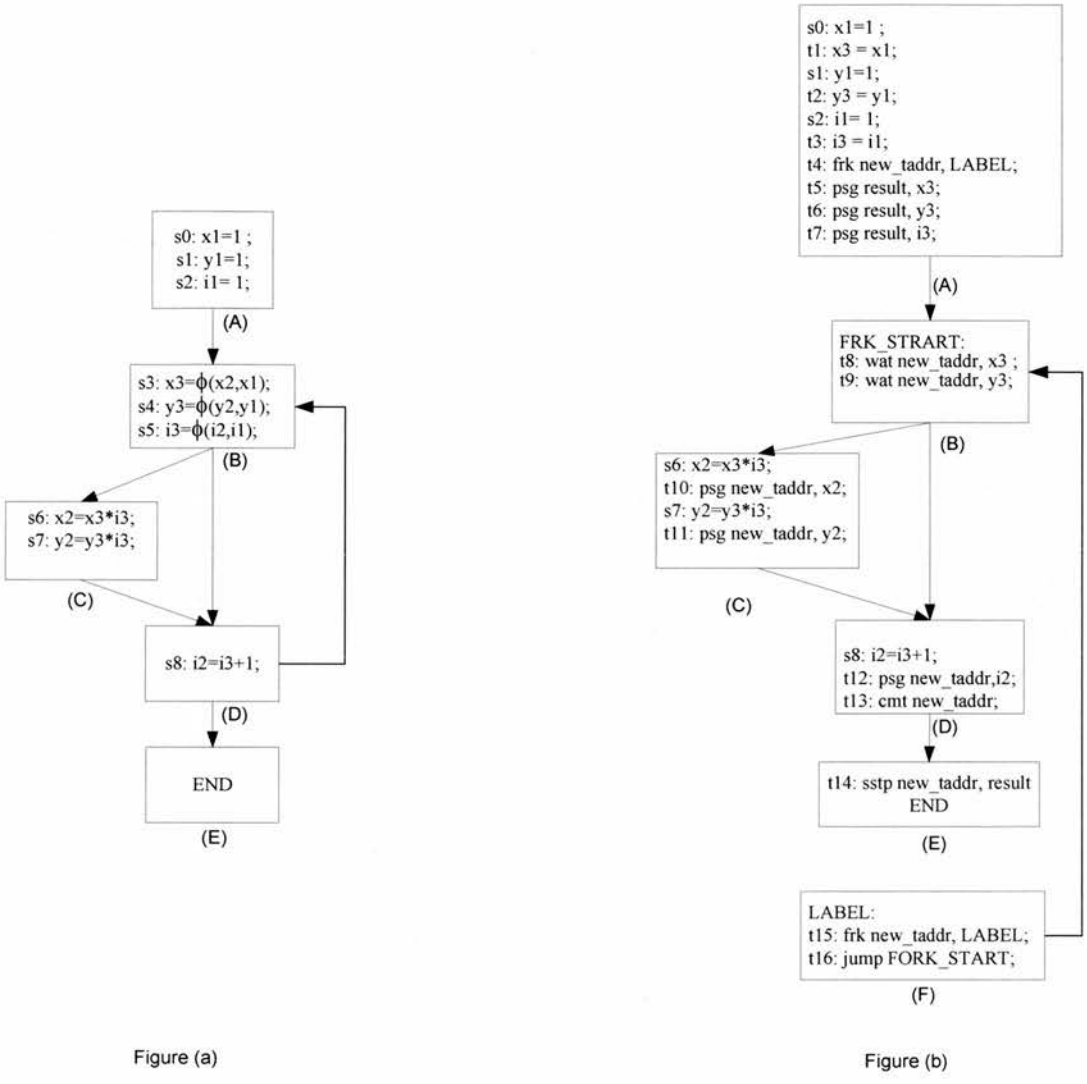


Figure 4-3: (a) A loop body in SSA form. (b) Its multithreaded version code.

4.2.3 Sequential Control-Code Partitioning

The sequential control-code partitioning algorithm is also based on the SSA intermediate representation (IR) and the execution information obtained from the point/edge profiling module. The granularity of partitioning is the basic block, i.e., either all the instructions inside a basic block are included in a thread, or none of them are included. It means a basic block is not split across multiple threads. The SSA IR

being used is the extended one with factored Use-Def [82] chains. An extended SSA form has the following distinguishing properties:

1. Every use of a variable in the program has exactly one reaching definition
2. At confluence points in the CFG, merge function call ϕ-functions are introduced. A ϕ-function for a variable merges the values of the variable from distinct incoming control flow paths, and has one argument for each control flow predecessor.
3. A linked list of variables and a list of modification cited for each variable are added to each node. Such links are called Use-Def chains.

By using profiling techniques [83], one can obtain all the edge execution information, including the run time of each instruction, and execution possibilities of each edge. Figure 4-4 shows a pseudo-C program fragment translated into extended SSA form with factored Use-Def chains and the execution possibility of each edge.

```
x=1;
y=1;
z=1;

if (P) {
    y=y<<1;

    if (Q) {
        x=x+1;
    }
    z=100 * y;

}
x=y+2;
z=z*2+x;
```

Figure (a)

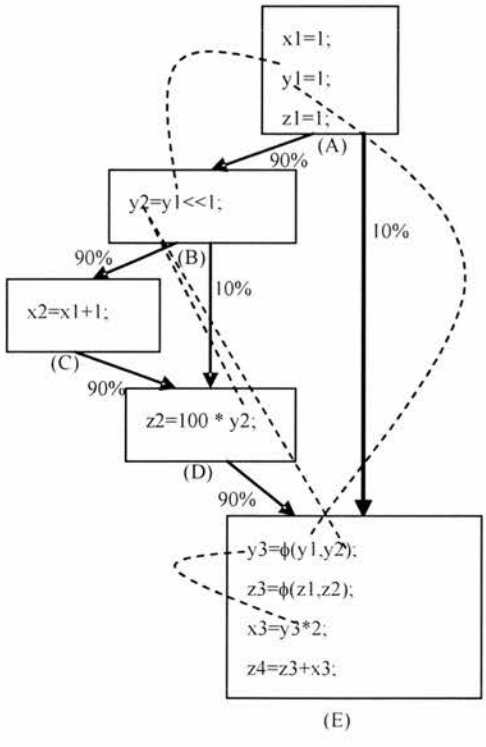


Figure (b)

Figure 4-4: (a) C program fragment in normal form; (b) Extended SSA with Use-Def Chains of Variable y.

The main problem of generating threads with data dependence is that if the producer is encountered late and the consumer is encountered early, then many cycles may be wasted waiting for the value to be communicated. The goal of data dependence driven thread generation selection is that for a given data dependence extending across several basic blocks, either the dependence is included within a thread or it is exposed such that the resulting communication does not cause stalls. During the selection of a starting point of speculative thread, the data dependence heuristic steers the exploration of control flow paths to those basic blocks that are dependent on the basic blocks that have been included in the thread. The data dependence heuristic includes a basic block only if it is dependent on other basic blocks that are below the threshold chosen. An approach of quantising the data dependencies can be found in [84]. Our approach is differentiated by implementing extended SSA to evaluate the data dependencies, using profiling information, and providing estimated delay of each instruction. Our implementation provides more accurate quantified data dependence information.

With the extended SSA, one can obtain comprehensive data dependences information. The data dependence count is the number of Use-Def chains coming into a current basic node. If the dependence count is small, then this block is less dependent on data from other blocks and may be a good target to begin a thread at the start of the basic block. The dependences from distant threads are likely to be resolved earlier and hence the current thread is less likely to wait for the data generated at that thread. Based on these observations, an algorithm to calculate the Data Dependences Factor (DDF) of each basic block in sequential program fragments was developed. A block with a DDF of less than 1 should be a suitable choice as a thread starting point.

$DDF(B_i) = 1/DD\_Length(V_0) + 1/DD\_Length(V_1) + \ldots. 1/DD\_Length(V_n)$:

$V \Leftarrow \{ V_0, V_1 \ldots. V_n \}$ is a set of variables used in basic block B_i;

$DD\_length(V_j)$ is the data dependency path length of variable V_j.

The pseudo code for the algorithm to calculate the DD_Length is shown in Figure 4-5. The choices of path lengths are based on the estimated execution times of each instruction for the MAPS architecture.

```
double DD_Length(BasicBlock b)
{
  double length = 0;

  for  (Each used variable V_i  in basic block)
  {
      for  (Each incoming use-def chain of Vi)
     {
       for  (each Path Pj in the use-def chain)
       {
         length=length+Path_length(Pj)* Possibility;
          /* The possibility of  executing this
              path, which  is obtained from
              path profiling pass*/
       }
     }
  }
}
```

```
double Path_length (Path p)
{
   for (each instruction I inside the Path p){
     switch (Type of instruction I)
     {
       case Register Asignment:
            return 1; /*MOV,LI.. */
       case Memory Access:
            return 4;    /*SW/LW */
       case Logical Operation:
            return 1; /*SLL/XOR/AND..*/
       case SUB/ADD Arithmetic:
            return 2;
       case Multiply/Division:
            return 3;
       }
     }
}
```

Figure 4-5: Pseudo code for calculating Data Dependent Length.

The C program fragment in Figure 4-4 can be used to illustrate the algorithm. The variable y1 used in block B is defined in block A by the instruction "y1=1" by traversing back the Use-Def chain. There is one register assignment instruction along the chain, the path length of the instruction "z=1" is 1. The possibility of executing path A→ B, is 90 %, so the data-dependent path of y1 is 1×90% = 0.9. As a result, the data dependent factor DDF(y1) = 1 /DD_Length(y1) = 1/0.9 = 1.1, which is greater than 1, and this basic block is not a good candidate for spawning a new thread. A full list of the DDF of each node is shown below:

A: Entry Node, ignore

B: DDF (Block_B) = 1/DD_Length(y1) = 1/(1*90%) = 1.1

C: DDF(Block_C) = 1/DD_Legnth(x1) = 1/(2+1*90%) = 0.34

D: DDF(Block_D) = 1/DD_Length(y2) = 1/(0*10% + 1 * 90%) = 1.1

E: DDF(Block_E) = 1/DD_Length(y3) + 1/DD_Length(z3)=

=1/(1*10%+ 2 * 90% * 90 % + 90% * 3) +

1/(3 + 0 * 10% + 0 * 90%) = 0.55

The DDF of block C and block E are both less than 1, and are chosen as thread starting points. The threaded pseudo code for program fragment in Figure 4-4(a) is shown in Figure 4-6. Thread 2 and thread 3 are spawned from block A. However, thread 2 is inside a conditional block, and if such a path is not taken, the context of the whole thread will be discarded.

```
Thread_1:
x=1;
y=1;
z=1;
if (P) {
    y=y<<1;
Thread_2:
    if (Q) {
        x=x+1;
    }
    z=100 * y;
}
Thread 3 :
x=y+2;
z=z*2+x;
```

Figure 4-6: Multithreaded version of the sequential control code in Figure 4-4(a).

4.3 Hardware-Software partitioning

The MAPS+ architecture combines field programmable logic and a multithreaded instruction set architecture, which requires a compilation module to perform hardware-software partitioning between each TPU. Programs are partitioned into sections to be executed on the reconfigurable hardware and other sections for executing in software on the multithreaded MAPS ISA. In general, complex control sequences, such as variable-length loops are more efficiently implemented in software, while configurable logic is very effective in speeding up regular, repetitive data-path computations.

4.3.1 Profile guided HW/SW partitioning algorithm

Programmer-directed compiler directives have been used to mark sections of program code for hardware compilation, e.g. the NAPA C [97] language provides pragma statements to allow a programmer to specify whether a section of code is to be executed in software on the Fixed Instruction Processor (FIP), or in hardware on the Adaptive Logic Processor (ALP). Most automatic hardware and software partitioning algorithms are based on mapping a loop construct onto configurable architectures. The Garp [94][95][96] compiler attempts to accelerate loops by using pipelining techniques similar to those used in VLIW compilers. The hardware blocks are chosen from hyper blocks that have loops embedded within them.

In this thesis, automatic hardware-software partitioning methods have been explored. The compiler uses cost functions based upon the amount of acceleration gained through the execution of a code fragment in hardware to determine whether the cost of configuration is overcome by the benefits of hardware execution. Loop computations provide an opportunity for parallelising the computations on reconfigurable architectures. But a loop-based analysis has some constraints. The loop statements which can be executed on configurable logic should be "well behaved", i.e.:

- Constant step size for the index
- No functions call in loop body
- No pointer operations or arithmetic operations
- Statically computable memory accesses

It is not easy for loop constructs to satisfy all these conditions, which might limit the utilisation of RFUs. A simple example CFG in Figure 4-7 is used to illustrate the function call constraint.

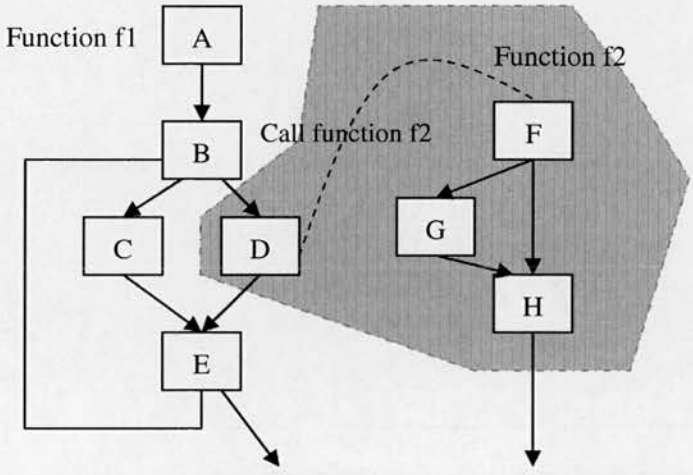


Figure 4-7: Diagram shows a function call in a loop body (blocks in the shadow are not included in hardware candidates).

Due to the function-call constraints of well-behaved loops, block D is not included as a possible hardware candidate. The whole function f2 does not even loop inside the function body, but the whole function will not be taken into account. Though function inlining might be a possible solution, it could bring unpredictable explosion in the program size. A point profile reports execution frequencies for nodes or edges in the graph. Information provided by the path profile provides an important tool for scalability, helping compilers to locate the regularly-used computation blocks.

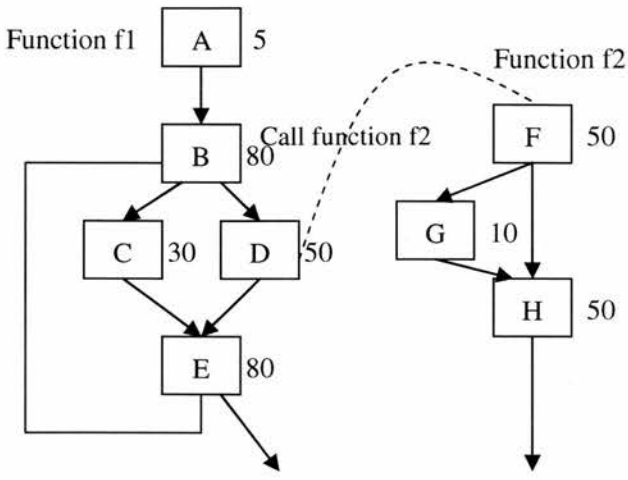


Figure 4-8: Point profiles with function inlining. (Numbers adjacent to nodes indicates frequency of execution)

The function inlining technique consists of inserting the called function's code rather than using a function call run-time mechanism. This is an optimising technique that avoids the overhead created by a run-time stack-based function-calling mechanism. But the drawback of function inlining is usually the increase in code space, which is affected by the size of the inlined function, the number of call sites that are inlined, and the opportunities for additional optimisations after inlining.

A further improvement is region-based compilation. This is a generalised trace selection approach that partitions a program into units of compilation, or regions, based on profile information. Using function inlining and restructuring a program into regions, the region-based compiler has more freedom to perform code motion and other analyses and optimisations across functions, while maintaining control over the compilation unit size and content. Unlike traditional function inlining techniques, region-based techniques provide a method for bounding the size of the unit of compilation to better control optimisation costs. The core of the technique is the region formation phase which partitions the program into regions using profile-guided heuristics. Thus, the quality of the generated code depends greatly upon the ability of the region formation phase to create regions that a global optimiser can effectively transform in isolation for improved instruction-level parallelism in the RFU. The potential benefits of region-based compilation include runtime performance

improvements, greater scope for the instruction scheduler, improved control of code growth, and better profile homogeneity. The benefits of region based formation mirror those of full inlining with the added potential for further reducing code growth.

An algorithm used to locate hardware blocks for configurable logic in MAPS+ architecture using a path-driven method as shown in Figure 4-9, which is similar to Hank's profile-sensitive region formation algorithm [98][99][100] used for exposing instruction-level parallelism in VLIW architecture. One of the similarities between VLIW architecture and reconfigurable computing architecture is the ability to exploit instruction level parallelism. Therefore, algorithms used in VLIW can be adapted for reconfigurable computing architecture.

```
void HWLocality (BasicBlocks BS)
{
    HardwareCandidates HWC;
    medianFreq = The median frequency of all basic blocks in the whole program.
    for   (Basic Block bi in set of Basic Blocks BS)
    {
      exeFreqi = The execution frequency of block bi;
      if (bi == MostFrequentBlock(BS) && exeFreqi >= medianFreq)   {
          seed = bi;
          /* Remove seed block from basic block set BS.*/
          BS = BS - seed;
          /* Add seed block in to set of Hardware candidates set.*/
          HWC = HWC + seed;
          FunctionCallExtension(seed);
          /* Choose the most frequent predecessor of seed inside set of Basic Blocks BS*/
          pred = MostFrequentPredecessor(seed);
          exeFreqPred = The execution frequency of block pred;
          while (exeFreqPred > medianFreq)   {
              /* Remove seed block from basic block set BS.*/
              BS = BS - pred;
              /* Add predecessor block into Hardware candidate set HWC. */
              HWC = HWC + pred;
              FunctionCallExtension(pred);
              temp = pred;
              pred = MostFrequenctPredecessor(temp);
              exeFreqPred = The execution frequency of block pred;
          }
          /* Choose the most frequent successor of seed inside set of Basic Blocks BS*/
          succ = MostFreqenctSucessor(seed);
          exeFreqSucc = The execution frequency of basic block succ;
          while (exeFreqsucc > medianFreq)   {
              /* Remove seed block from basic block set BS. */
              BS = BS - succ;
              /* Add successor block into Hardware candidates set. */
              HWC = HWC + succ;
              FunctionCallExtension(succ);
              temp = succ;
              succ = MostFreqenctPredecessor(temp);
              exeFreqSucc = The execution frequency of basic block succ;
          }
      }
    }
}
```

```
void FunctionCallExtension(BasicBlock b)
{
    if (b contains function calls of FS)
    {
            for (each function Fi in set FS)
            {
              BSi = Basic blocks of function Fi;
              /* A recursive call to extend the function call contained in the in Basic Block. */
               HWLocality (BSi);
             }
    }
}
```

Figure 4-9: Algorithm for locating hardware blocks using profile-driven method.

The algorithm comprises of the following steps, which are performed until all the possible blocks in the program have been included as hardware candidates.

Step 1: Find the median frequency – The median execution frequency is calculated for the basic blocks in the program.

Step 2: Seed selection - From among all the basic blocks not yet included in the set of hardware candidates, the block with the highest execution frequency is selected, and the frequency must be greater than the median frequency. This is removed from the set of basic blocks and inserted into the set of hardware candidates.

Step 3: Seed's function calls expansion – If function calls reach into the seed, then they are expanded using a recursive procedure.

Step 4: Predecessors expansion – If the execution frequency of the predecessor is greater than the median frequency, then it is removed from basic block set and added to the hardware candidates set, and the recursive method is used to expand function calls.

Step 5: Successors expansion – If the execution frequency of the successor is greater than median frequency, then it is removed from the basic block set and added to the hardware candidates set, and function calls are expanded using recursive methods.

The control flow graph of Figure 4-8 is used to illustrate the procedure of locating hardware blocks. First the median frequency is calculated; next the seed block is chosen from all the basic blocks, which is block B, because it has the highest execution frequency. B's predecessor block A does not exceed the median frequency, so A is not included in the hardware block candidates. D is the successor of seed block A that exceeds the median frequency, and D contains a function call to function f2. So recursively, blocks of F and H are added to hardware candidates. D's successor E also has an execution frequency that is greater than the median frequency, so E is also included in the hardware candidates set. The hardware extraction is shown in Figure 4-10.

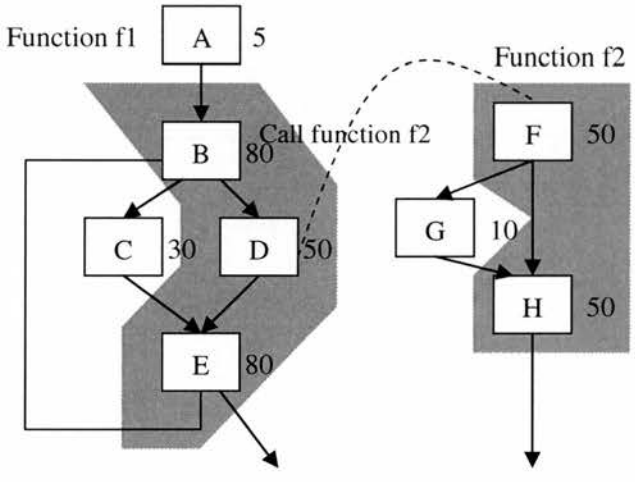


Figure 4-10: Hardware extraction by using profile-driven algorithm. Shadows are the hardware candidates.

As reconfigurable functional units in the MAPS+ architecture only support one entry point for hardware blocks, re-entry blocks located in hardware needs to be duplicated in software as depicted in Figure 4-11. The successors of block C and G are blocks E and H, respectively. However, blocks E and H are chosen to generate

hardware code, therefore these two blocks need to be duplicated to avoid re-entries in hardware blocks. E1 and H1 are duplicated blocks of E and H respectively. Then MIPS-like instructions are generated for the software parts, and the FPGA netlist files are generated for the hardware intermediate parts. The profile information for hardware-software partitioning is based on the execution of a sequential application. However, the decisions to determine if the right choice has been made are not profiled, which can be improved in the future by using HW/SW profiling information to refine the partitioning decisions.

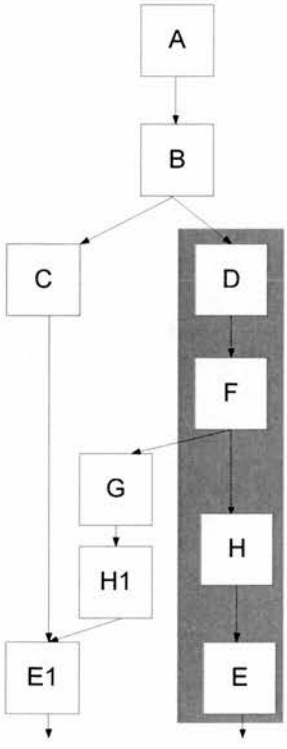


Figure 4-11: The hardware and software blocks after code duplication.

4.3.2 Control Data Flow Graph generation and synthesis

After the hardware blocks have been chosen from the program SSA intermediate form, further optimisations are performed on the IR. A Data Flow Graph (DFG) is a low-level, non-hierarchical and asynchronous program representation. DFGs can be viewed as abstract hardware circuit diagrams without timing or resource contention taken into account. From the SSA, the variables of chosen hardware blocks can be mapped to edges in a dataflow graph, while primitive operations map to nodes. The dataflow graph makes data dependences explicit, and is a convenient representation for circuit generation. Data flow graphs are then mapped to circuits using the component library in Xilinx Netlist Format (XNF) format by mapping edges onto wires and nodes onto components. For the sake of simplifying implementations, we do not generate a pipelined implementation for the RFU, nor does it support internal loop constructs. Only combinational logics are supported in the current compiler version, which limits the choice of hardware blocks deployed in hardware.

A component library has been implemented in the mapping stage during compilation, which is similar as the approach used in Chimaera [63]. The use of a component library greatly simplifies and speeds up the binding process. By pre-designing commonly-used structures such as adders, multipliers, and counters, circuit creation for configurable logic becomes largely an assembly of high-level components, and only application-specific structures require detailed design. The actual architecture of the reconfigurable device can be abstracted, provided only library components are used, as these low-level details will already have been encapsulated within the library structures. The benefit of using library macros is fast compilation. Because the library structures have been pre-mapped, pre-placed, and pre-routed (at least within the macro boundaries), the actual compilation time is reduced to the time required to place the library components and route between them. An added benefit of the architectural abstraction is that the use of library components can also facilitate design migration from one architecture to another, because designers are not required to learn a new architecture, but only to indicate the new target for the library components.

However, this does require that a circuit library contain implementations for more than one architecture. In the mapping stage, the compiler analyses the data flow graph of the part that should be implemented in hardware. If a component matches a portion of the graph, the corresponding macro is used for that part of the configuration. Finally the netlist file for reconfigurable hardware is mapped from the DFG, and a MIPS-like assembly code is generated.

A CRC-32 table construction program is used as an example to illustrate the hardware netlist generation process, which is shown in Figure 4-12. The C program is first pre-compiled by the SUIF2 compiler. Details of the MAPS+ simulation and compilation framework will be introduced in Chapter 5. The output for the hardware is a netlist file for a Xilinx FPGA. The output for the software part is a MIPS-like assembly code. While calculating the CRC for a string, one can simply look up the CRC table for each ASCII character and perform an XOR operation.

```
int main()
{
   unsigned int c,j,i,l;
   unsigned int k = 31;
   unsigned int m = 1;
   unsigned int poly = 0x4c11db7;
   unsigned int crc_table[256];

   for (i=0;i<256;i++)
   {
      c = i;
      for (j=0;j<32;j++)
      {
         l = (c>>k) * poly;
         c = (c<<m) ^ l;
      }
      crc_table[i]=c;
   }
}
```

Figure 4-12: The CRC-32 table generation program

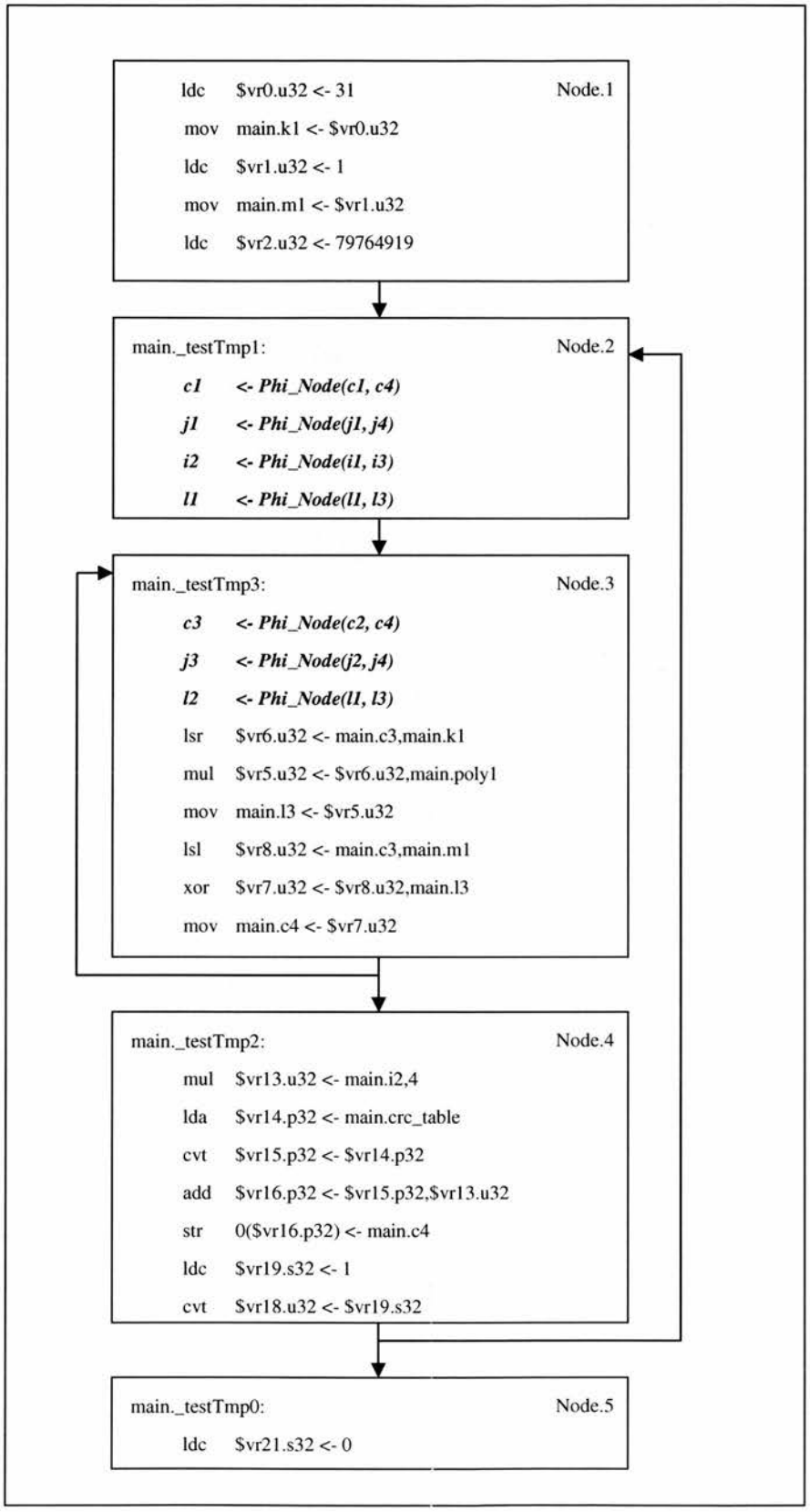


Figure 4-13: Corresponding SSA form in SUIF_IR format, with ϕ nodes inserted at the top of the blocks, for the CRC-32 table generation problem.

Figure 4-13 illustrates the SSA representation of the program fragment in SUIF_IR which is generated automatically. Two loops have been identified by the compiler, and based on the profiling information the inner loop is executed more frequently and will benefit from execution in hardware, and therefore chosen for mapping to hardware. The DFG for the inner-most loop is illustrated in Figure 4-14, which excludes the phi-functions and other loop-related functions. The primary input arcs entering the graph (m1, c3, k, poly1) represent the values passed as input to the function. The DFG shows function nodes, which denote operations that are members of the set of library functions. Operations in the SSA intermediate representation are mapped to these nodes in the graph. The DFG representation provides clues for synthesis of the function in a modular fashion. It is automatically mapped to a Xilinx FPGA netlist. The optimisation of this translation has not yet been investigated, which includes the refined mapping procedure from macro libraries to hardware implementation, and fine-grained choices of HW/SW partitioning.

Figure 4-15 illustrates the hardware structures from the component library that instantiate the behavioural elements. The component library had many functional design elements (primitives and macros) which were dependent on the device architecture. New functionality was assembled by using these basic components, which were implemented as C++ classes, e.g. a MULT32 component included input pins, output pins definition and I/O signal for each pin. The external I/O pads for the whole netlist file were also required. The only signals which were specified as external I/O signals are those which connect to the external pins of the I/O symbols, such as input pads poly1, k1, c3, m1 or output pad c4. This information was produced when generating the Xilinx Netlist File.

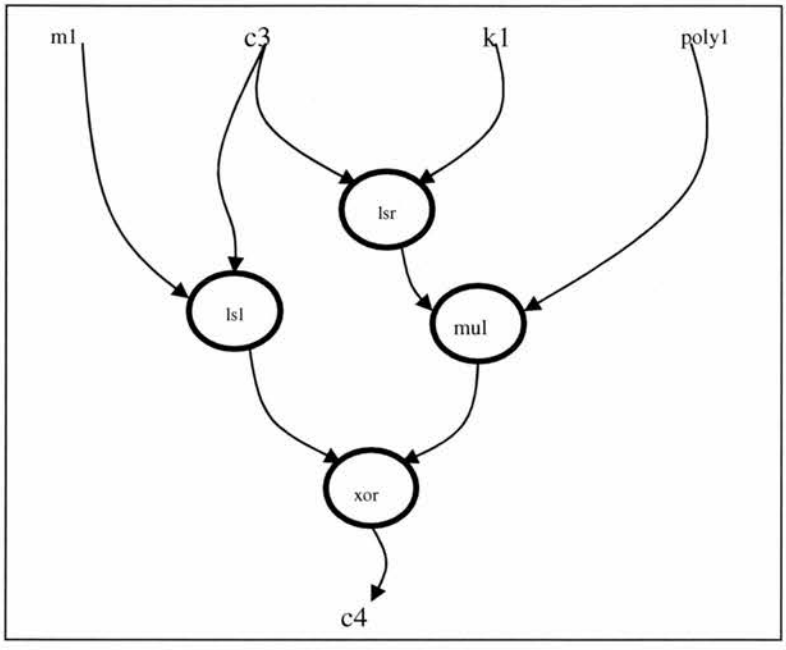


Figure 4-14: The corresponding DFG for the inner-most loop.

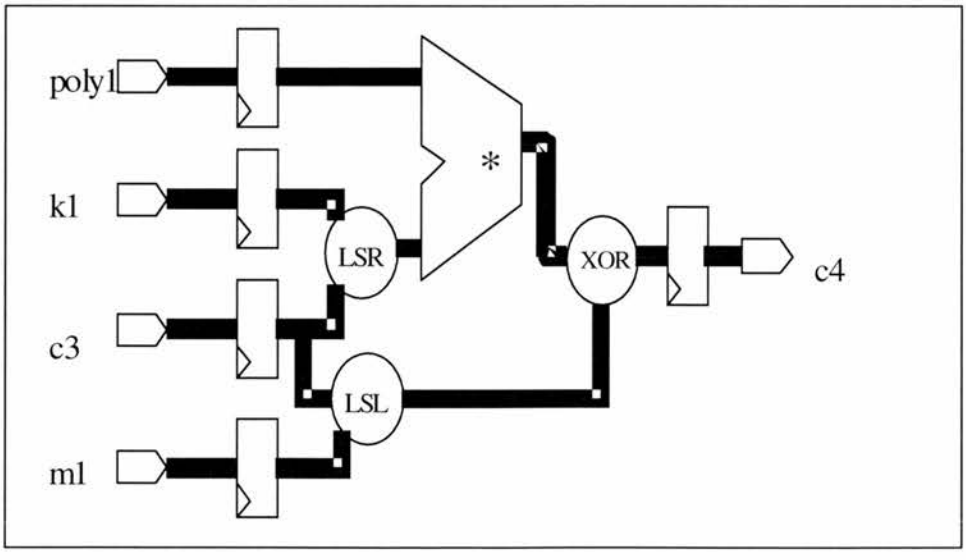


Figure 4-15: Synthesised implementation of DFG.

The hardware netlist file was mapped from the DFG of Figure 4-14, which contained mapped look-up tables and carry logic components, each annotated with a LOC location constraint, and fed to the Xilinx ISE tools. The Xilinx Flow Engine

merged all the input netlists and translated the netlist file into the Xilinx Native Generic Database (NGD) format, which contained a logical description of the design and macro library (NMC) files. Then the flow engine performed a logical Design Rule Check (DRC) on the design in the NGD file. It then mapped the logic to the components in the target Xilinx FPGA. The output design was a Native Circuit Description (NCD) file that physically represented the design mapped to the components in the Xilinx FPGA. The Xilinx flow engine then placed and routed (PAR) the design and output an NCD file which was used by the bitstream generator. The PAR design used a combination of cost-based and timing-driven methods. After the design had been completely routed, the Flow Engine configured the device so that it would execute the desired function and produced a configuration bitstream, a binary file that can be downloaded into the target device. The design of the CRC-Table generator in Xilinx FPGA is displayed by graphical application in the FPGA Editor in Figure 4-16, in which probes could be inserted to examine the signal states.

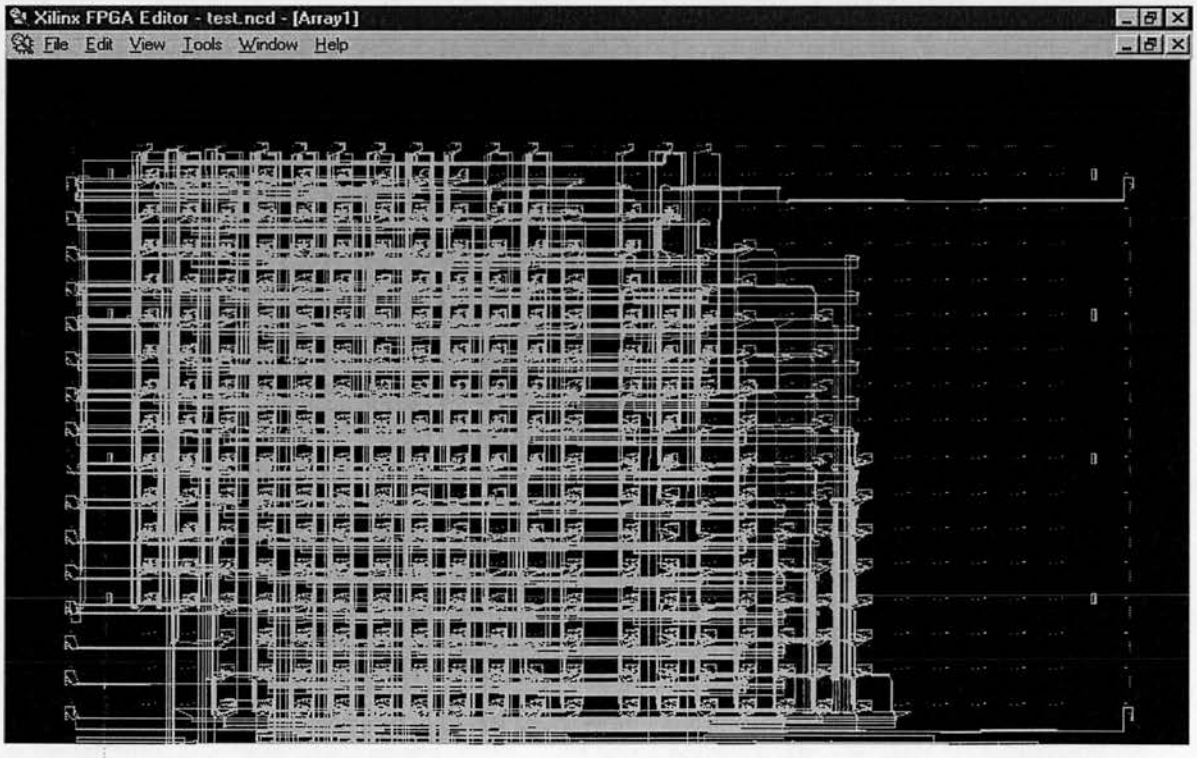


Figure 4-16: The graphical display of the CRC-32 table construction implemented in Virtex FPGA.

From the Xilinx flow engine, performance estimations are obtained. The average connection delay for this design was 1.946 ns, the maximum pin delay was 6.708 ns; the average connection delay on the 10 worst nets was 5.081 ns. All signals were completely routed. All the delay information would be fed into the MAPS+ architecture, as explained in Chapter 5.

Finally the design was translated in to a VHDL file containing a netlist description of the design in terms of Xilinx simulation primitives. It was used to perform a back-end simulation using Modelsim. The post-synthesis simulation was a timing simulation, which was important in verifying the operation of the circuit after the worst-case placement and routing delays were calculated for the design. The functionality of the CRC-Table generator design could be verified in the wave window from Modelsim. Figure 4-17 displays the CRC calculation procedure for the ASCII character "{" (ASC =123). The input included a signal 'k1' that controlled the right shift steps, m1 controlled the left shift steps, 'poly1'-the polynomial defined for CRC-32, and the input and output values were 'c3' and 'c4'. The operation completed after 32 clock cycles with the CRC for 123 being 0xCBFFD686.

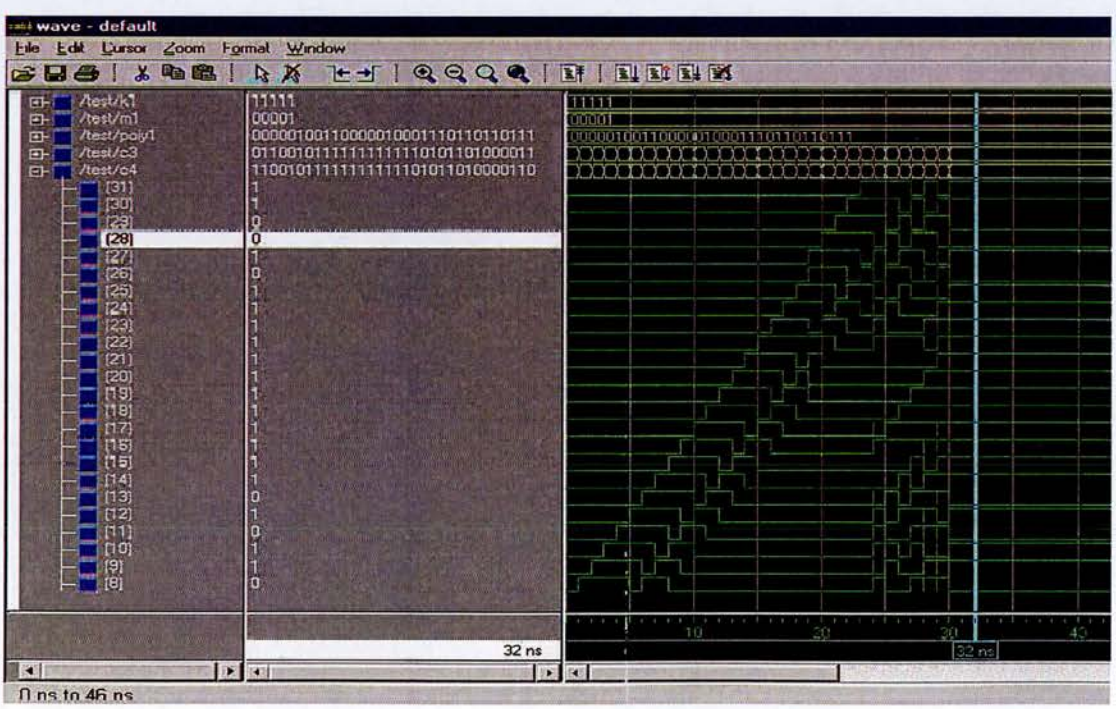


Figure 4-17: The wave form of the CRC-32 table construction simulated in Modelsim.

4.4 Summary

This chapter has described the compilation techniques for extracting threads from a sequential program, partitioning between software and hardware, and the process of compiling to the FPGA fabric. This enables the MAPS+ architecture to exploit thread-level and instruction-level parallelism in the application program. The multithreaded codes for loop bodies are generated in two steps: loop nest selection procedure, and multithreaded code generation procedure. Furthermore, sequential control codes are partitioned into speculative threads based on the data dependencies calculation algorithm using the SSA form. A hardware software partitioning algorithm based on the region-base compilation method has been investigated. Finally, a CRC-32 table generation program is used to illustrate the synthesis procedure using a data flow graph.

Chapter 5
Simulation and Compilation Environment

5.1 Introduction

This chapter presents a simulation framework for the MAPS+ architecture described in Chapter 3 and the compiler framework for implementing the algorithms described in Chapter 4.

Firstly, a MAPS+ instruction set architecture simulator called SPAMSIM2 is introduced. This consists of an event-driven simulation kernel design and the models of different functional units. Furthermore, the power/energy consumption model is also described in detail.

Secondly, the compiler framework in the thesis is based on the SUIF2 [5][6] and Machine SUIF [7] compilation framework. The MAPS+ passes, such as Software Hardware Practitioner (SHP) pass and Thread Analyser (TA) pass perform automatic code generation from sequential C benchmark programs, and produce the executable for the SPAMSIM2 simulator.

5.2 The SPAMSIM2 simulation environment

The MAPS+ architecture is modelled and simulated in SPAMSIM2 to generate timing information using execution-driven simulation. Trace-driven simulation may provide a relatively quick estimate of performance, but will often lack sufficient accuracy because of the difficulty in characterising the behaviour of real programs stochastically.

Also, generating accurate inter-thread traces is difficult, since changes in the interleaving due to timing variations, or changes in the addresses themselves due to timing-dependent program behaviour will not be reflected in the traces. As a result, an execution-driven simulator was chosen to model the MAPS+ architecture.

5.2.1 Overview

The SPAMSIM2 simulator is outlined in Figure 5-1, and was implemented in C++. The QuickThreads [2] package of the SPAMSIM2 kernel provides a portable interface to machine-dependent code that performs thread initialisation and context-switching. The simulator takes assembly code compiled for the MAPS+ architecture and simulates the execution with different configurations. During the initialisation stage of the SPAMSIM2 simulator, various parameters are read from the configuration file and can be set separately, e.g. memory size, delay of each function unit, cache size, and replacement policy. After initialisation, the parameters are set to global variables, which can be accessed by each functional block of the simulator. Contents of the register file and portions of the memory can be dumped to files on completion of the simulation. By dumping the results of the benchmark simulation to a file, the benchmark's execution can be verified.

Data cache behaviour is modelled using the Dinero IV cache simulator [3]. This provides a highly configurable model of cache behaviour. The basic idea is to simulate a memory hierarchy consisting of various caches connected as one or more trees, with reference sources at the leaves and a memory at each root. The Cache model produces performance metrics (e.g., traffic to and from the memory) and allows cache design options to be varied (e.g., write-back versus write-through, LRU vs. random replacement, demand fetching versus prefetching).

Power dissipation issues are becoming a central issue in modern processor architecture design. The power analysis and optimisation package of SPAMSIM2 simulation environment is based on Wattch[4], which is a framework for power

estimation based on a suite of parameterisable power models for different hardware structures, and on resource usage counts generated through simulation.

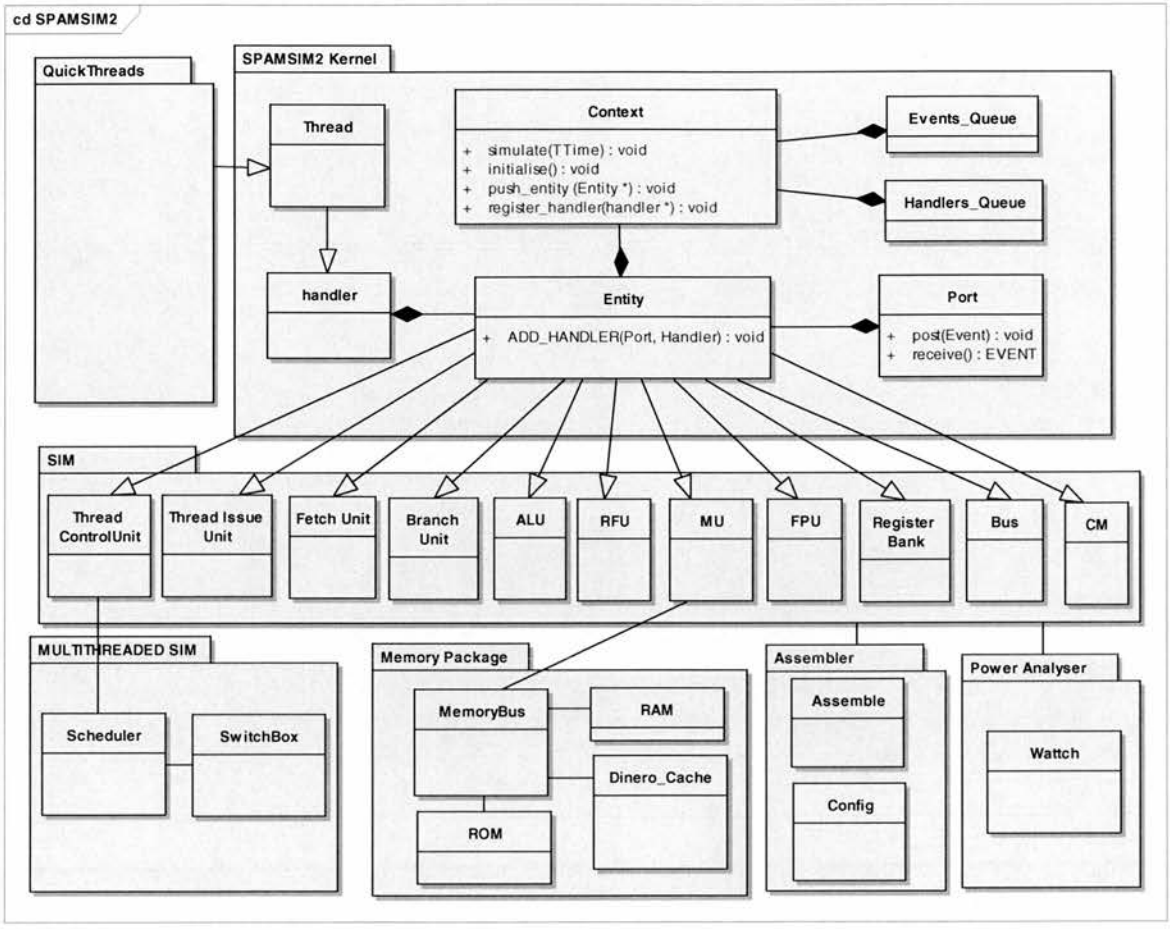


Figure 5-1: Class diagram of the SPAMSIM2 simulator.

5.2.2 Simulator kernel

The SPAMSIM2 simulator defines a computational procedure that mimics a set of properties of the MAPS+ architecture and describes the behaviour of the MAPS+ architecture in response to stimulus from outside the system. As shown in Figure 5-1, the MAPS+ architecture is modelled in a structural form, i.e. a functional description of different components and the interconnections between datapath components. It provides the platform for exchanging messages between entities and time-keeping. Instead of using a gate-level simulation of the MAPS+ architecture with greater

overhead, the functionality of the architectural components was modelled at the micro-operation and register transfer level, providing explicit description of the interaction between components of the datapath.

To model system architecture with an execution-driven simulator, two main programming paradigms dominate the simulation design. The first paradigm is flow-driven, which follows its control flow pattern and changes course at branch point. For example, the SimpleScalar [17] simulator core defines the main loop and executes one iteration for each instruction until finished. Its kernel accounts for the progression of execution time and counts the total number of clock cycles. This approach is suitable for synchronous systems, as the global clocks are required to synchronise different components within the system. The synchronous flow is easily to be mapped to the main loop body, and each instruction is processed in a lock step fashion. In flow-driven simulation, all activities in the current time step must cease before the processes of the model are allowed to advance to the next time step.

However, in an asynchronous system, there is no global centralised clock and every component operates autonomously, except to interact with other component in the system. It requires the simulation model to operate asynchronously, advancing at completely different rates. Each process maintains a local clock variable which contains the current value of the simulated time. This value represents the process's local view of the global simulated time and denotes how far in the simulated time the corresponding process has progressed.

Therefore, the event-driven approach is usually chosen to model an asynchronous system. In this approach, the driving force of the simulation which triggers actions is the availability of events to be processed. Upon receipt of an input event, the process will be activated to act upon the event and, as a result, update its local time, which is set to the minimum next event time for that process. Processes are allowed to consume and execute messages as soon as they become available, without having to wait for a global clock to tick through periods of inactivity or for other slower, but unrelated, processes to advance. Each data item carries a time stamp which indicates the time up to which data is valid, and the events which occur at different time instances are processed by the

simulation kernel. The kernel contains an event loop and looks repeatedly for information to process then perform a trigger function. As a result, the register handler will be invoked to react to the event. The SPAMSIM2 simulator kernel falls into this category. As shown in the class diagram Figure 5-1, the SPAMSIM2 kernel consists of an event-driven based scheduling function, and entities and ports for representing functional units and the communications.

5.2.3 Event scheduler

The context class implemented in the SPAMSIM2 simulation kernel contains the functions to enable event scheduling, context switching as well as time-keeping. The event scheduler of the kernel is responsible for scheduling event handlers. The kernel holds a priority queue of pending events sorted by time-stamp. Whenever an event is a trigger, its corresponding handler becomes runnable. A data structure called, ***runnable_handlers***, maintains a set of handlers, with their associated trigger time, that are scheduled for the future. Each of these handlers is guaranteed to be triggered at the associated time in the kernel. Events in the kernel are simulated chronologically and the simulation clock is advanced after simulation event to the time of the next event. Detailed flow chart of the scheduling function is depicted in Figure 5-2.

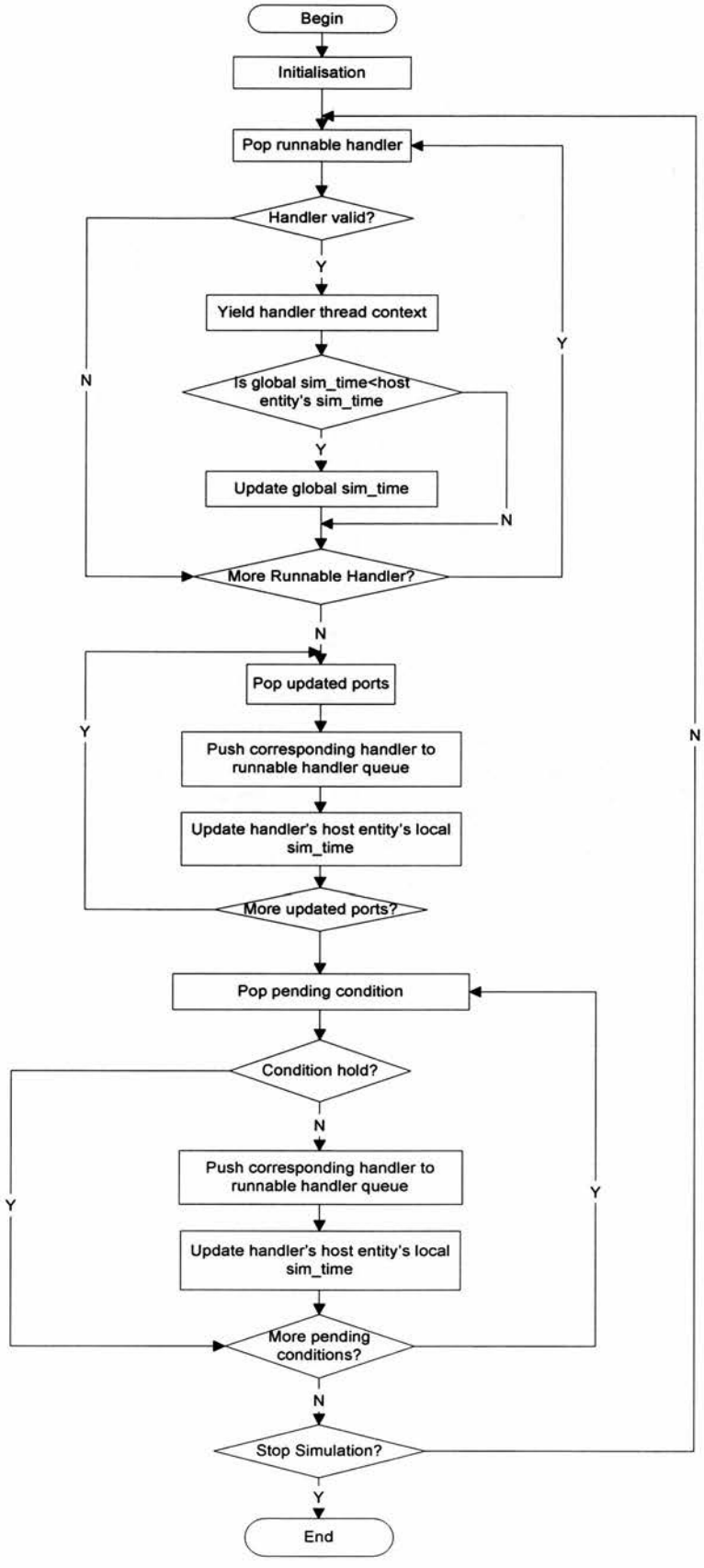


Figure 5-2: Flow diagram for the event scheduler.

Context switching of event handlers is realised by implementing a thread operations wrapper on top of a threading package of QuickThreads [2]. The QuickThreads package consists of stack spaces for handler context switching. The stack has no initialised state. While a context switch request is triggered, the stack saves the old thread state, and then switches to the scheduler stack, but no scheduler state is restored. The scheduler stack is simply used as a place to call a function on behalf of the thread that just blocked. Likewise, when the new thread is started, no scheduler state is saved. Because of the lightweight stateless thread switching scheme, the scheduler implemented in QuickThreads is faster than the heavyweight scheduler used in POSIX[152] thread packages and fulfils the requirement of the SPAMSIM2 kernel design. The top-level thread wrapper provides a machine-independent interface and operations, such as thread initialisation, execution and termination.

Each entity in the SPAMSIM2 simulator kernel maintains a local notion of time. At the top-level, the global time depicts the total elapse time for the simulation. While event messages are passed from one entity to another, time stamps are associated with each event message. Whenever the handler becomes runnable, the handler's host entity's local time is set to be equal to that of the entity's time stamp which triggers the changes. The time stamp value applied to outgoing events of an entity is the total delay of processing this event and the communication overhead. The global simulation time will be updated to the local time of the entity with the greatest value.

5.2.4 Asynchronous communication model

Asynchronous communication model is realised in the SPAMSIM2 by implementing a class of communication port, and template class EVENT represents the event message passing between entities. A port provides an entity with a means for connection and communication with its surrounding entities. In SPAMSIM2, ports are able to handle bi-directional communication (input, output). The C++ class for an asynchronous port is illustrated as follows:

```
template <class EVENT> class port {
        vector <EVENT> m_events;
    public:
        void post (EVENT e);
        EVENT receive();
        void add_event (EVENT e, TTime t);
};
```

Whenever a ***post*** function within the sender port is invoked, the event message will be passed to the connected receiver port. The messages are placed in the event list, ***m_events*** via the ***add_event*** function. Along with the event message, the local time stamp of the sender entity is also passed, and the event list will sort event messages by the order of time stamp. The receive function will remove the top message from the ***m_events*** event list, and the actually received message will be passed back to the receiver entity by the returning value.

In SPAMSIM2, the asynchronous communication channels are modelled as bundled data encoding with the four-phase handshaking protocol. All the models are wrapped into the Communication Microagents (CMs), which is illustrated as follows:

```
class cm : public Entity
{
    cm *otherCM;
    Entity *host;
     void DealAck();
     void DealReq();
    protected:
     virtual void DealReqHigh();
     virtual void DealAckHigh();
     virtual void DealAckLow() ;
     void DealData();
    public:
     port<bool> ack_port;
     port<bool> req_port;
     port<Data> data_port;
    void Connect(cm *other);
};
```

This is the basic class interface for CMs. In the construction stage of a simulation node (TPU), the connection between the receiver CM and the sender CM is defined by the ***Connect*** function. Therefore, the corresponding ***ack_port***, ***req_port*** and ***data_port*** of the pair of CMs will be linked together. After the CM class is initialised, the corresponding port handler will be registered to the kernel by the ADD_HANDLER macro. The following is a sample to register the request event handler ***DealReq***:

```
ADD_HANDLER(req_port, DealReq);
```

Once a request signal is received, the handler of the receiver CM will be switched to current context. If the request signal is high, the ***DealReqHigh*** function will be invoked and the associated delay of handling the high request signal will be added to the time

stamp of the CM entity. Then an acknowledge signal is fed back to the sender CM. As a result, the sender will pass the data to the receiver, and bring down the request acknowledgement once the data transmission is finalised. Finally, the receiver acknowledges back a low signal as a response to receiving the data successfully. The following message sequence chart in Figure 5-3 illustrates the event messages exchange sequence of implementing the 4-phase handshaking protocol in the CM model.

Figure 5-3: Sequence diagram of implementing 4-phase handshake protocol in the CM model.

5.2.5 Modelling function units

Entities are the basic building block within SPAMSIM2 simulator. Entities break complex systems into more manageable components. Functional units are inherited from entity classes. Each functional unit is a module, which allows internal data representation and algorithms to be hidden from other functional units. This modular

design makes the entire simulator easier to change and to maintain. Functional units may be instantiated inside other units to create a hierarchy. A functional unit requires that a string name be provided as part of the instantiation. The string name is used by SPAMSIM2 kernel to assign a hierarchical name to the instance automatically. This hierarchical name is formed by the concatenation of the parent's hierarchical name and the string name of the child.

Functional units contain asynchronous ports, event handlers, and internal data. Through these asynchronous ports, different functional units are able to communicate and synchronise with each other. The entity model contains local simulation time information, which indicates the time-stamp of the most recent event processed, and a list of time-stamped events that have not yet been processed. Functional unit react when they perceive a change at one or more input ports. An event message carrying the same value as before is not considered to be an event. The incoming events are processed in time-stamped order. A functional block model contains I/O ports, register event handlers. The following C++ class illustrates the modelling of an ALU.

```
Class ALU : public Entity {
    public:
        port <Data> x_port,y_port,exe_port;
        BusDriver *x_bus, *y_bus, *z_bus;
        void x_port_handler();
        void y_port_handler();
        void exec_port_handler();
        void eval ( ) ;
}
```

The ALU class contains execution port, and two data input ports x_port and ***y_port***. During the initialisation stage, the ***x_bus*** and ***y_bus*** are connected to ***x_port*** and ***y_port*** respectively. As the ***z_bus*** is for data writeback only, no ***z_port*** is defined in the ALU class. The ***exe_port*** obtains decoded instructions from the TIU. And the ***x_port*** and

y_port are used to receive fetched data from Register Bank via the ***x_bus*** and the ***y_bus***. When all of the instructions have been evaluated, the results are written back to the Register File via the ***z_bus***. Figure 5-4 illustrates a sample interaction among the CMs and the FU.

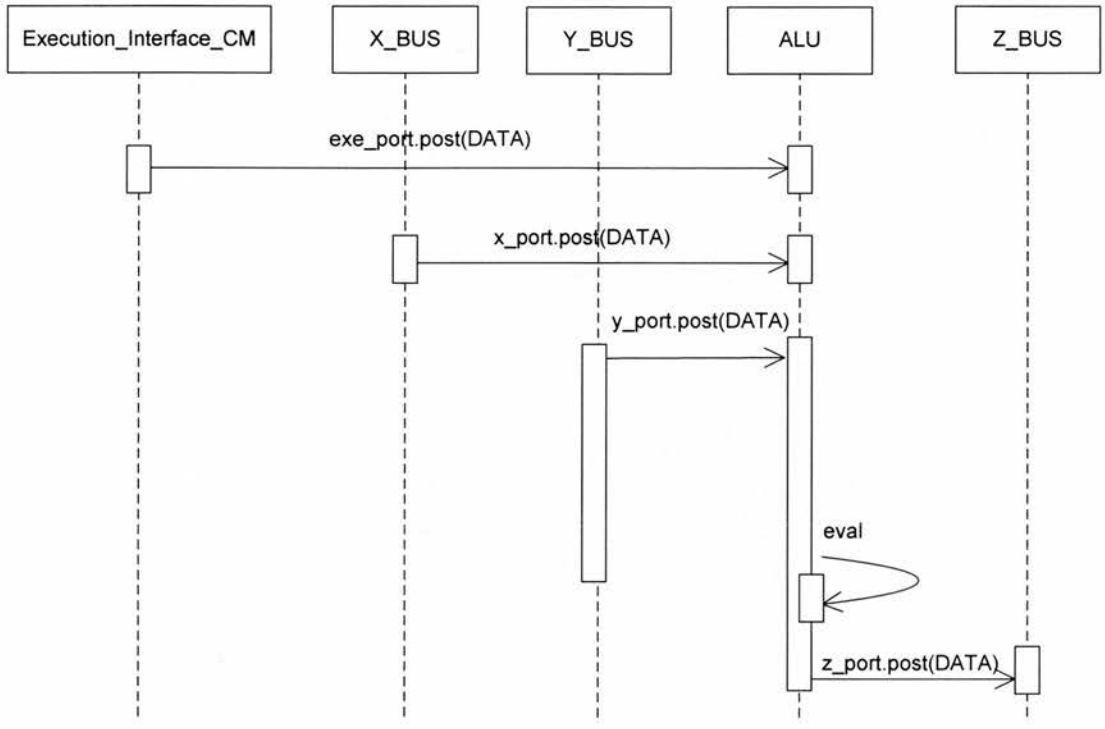


Figure 5-4: Sequence diagram of interactions among buses, CM and ALU module.

5.2.6 Architectural power model

Power consumption is a major concern for the architectural design. Firstly, most embedded architectures are implemented in mobile devices, which have limited length of battery life available. Secondly, the heat resulting from power dissipation has to be removed using cooling methods.

Integrating a power estimator for programs executing on MAPS+ architecture simulated in SPAMSIM2 is essential. Currently, most digital circuits are manufactured in CMOS technology, such as microprocessors, microcontrollers, and static RAMs.

Figure 5-5 illustrates a CMOS inverter which only uses significant power when its transistors are switching between the on and off states.

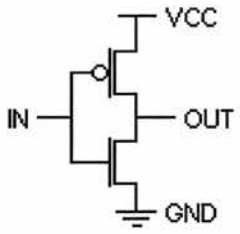


Figure 5-5: Static CMOS inverter.

CMOS circuits have three main sources of dissipation [182]: dynamic current, short-circuit currents, and leakage currents. The total power dissipated during operation can be expressed as: $P_{total} = P_{static} + P_{dynamic} + P_{short}$.

- **Static Dissipation**: The static dissipation can be expressed as: $P_{static} = I_{leak} \times V_{dd}$, where I_{leak} is the leakage current and V_{dd} is the supply voltage. Static dissipation is due to leakage current in the transistors when they are not switching. Ideally, CMOS circuits have very little leakage current; however, the leakage current does become significant for submicron feature sizes.

- **Dynamic Dissipation**: The dynamic dissipation is given by: $P_{dynamic} = \alpha \times f \times C \times V_{dd}{}^2$. In the equation, α term is an activity factor that captures how many devices are active, f is the clock frequency, C is the total switched capacitive load, and V_{dd} is the voltage supply.

- **Short Circuit Dissipation**: The short circuit dissipation is expressed as: $P_{short} = V_{dd} \times \int_T I_{short}(\tau) d\tau$, where V_{dd} is the voltage supply, T is the switching period, τ is the flow between the supply voltage and ground when a CMOS logic gate's output switches, and I_{short} is the short circuit current. This is the power consumed during the short time that both the pull-up and pull-down networks are

conducting when the CMOS gate is switching. It typically increases the total power consumption by 10% or less, unless the edge rates in the circuit is very slow. This is usually ignored in back-of-the-envelope power calculations.

In this thesis, we have only considered the dynamic power dissipation. The Wattch framework [4] has been integrated to provide power and energy consumption estimation. Wattch provides switching capacitance models for structures in a processor. Basic components have been developed, such as array structures, content-addressable memories (cache), combinational logic and wires. These components are then used to build a parameterised model for the MAPS+ architecture. Rather than using a power density and area-based model, capacitances are calculated from wire delays and then used to generate a cost for each activity. These costs can then be scaled by usage as indicated by the activity counters. The counters are configured to measure events which are significant to the energy consumption, and a model interprets these results to estimate the total MAPS+ architecture power consumption. The accuracy of the model is therefore determined by the amount of information available. Such a model has the advantage of being used on-line efficiently, allowing the information obtained to be used by power management algorithms. Current assumption of CMOS technology for the MAPS+ processor architecture is 0.18um.

In the synchronous case, the power model assumes that the MIPS-like baseline processor architecture are arranged into four groups: array structures, CAM, combinational logic and wires, and clocking. While in the asynchronous MAPS+ architecture, the globally clock distribution network is replaced by communication micro agents handling synchronisations between functional units. A CM contains c-elements, added delay gates, and registers, which are used to handle 4-phase handshaking protocol and bundled data communications. These extra asynchronous control units add extra area overhead [11][12]. Figure 5-6(a) shows a CMOS schematic diagram of the C-element and Figure 5-6 (b) shows a schematic diagram the double-edge flip-flop. The C-element implemented in static CMOS topology is first introduced by Sutherland in his Micropipelines [48] research. The double-edged flip

flop chosen in our simulator is identical to Tin Wai Kwan and Maitham Shams's implementation [183]. It contains two opposite polarity level sensitive latches and a multiplexer. Two latches operate in either transparent mode or capture mode in response to the level of the clock signal. Their simulation results show a 56% power saving and 10% area overhead over static CMOS doubled-edge flip-flop implementation. The power figures of these components were fed into the Wattch power model for the SPAMSIM2.

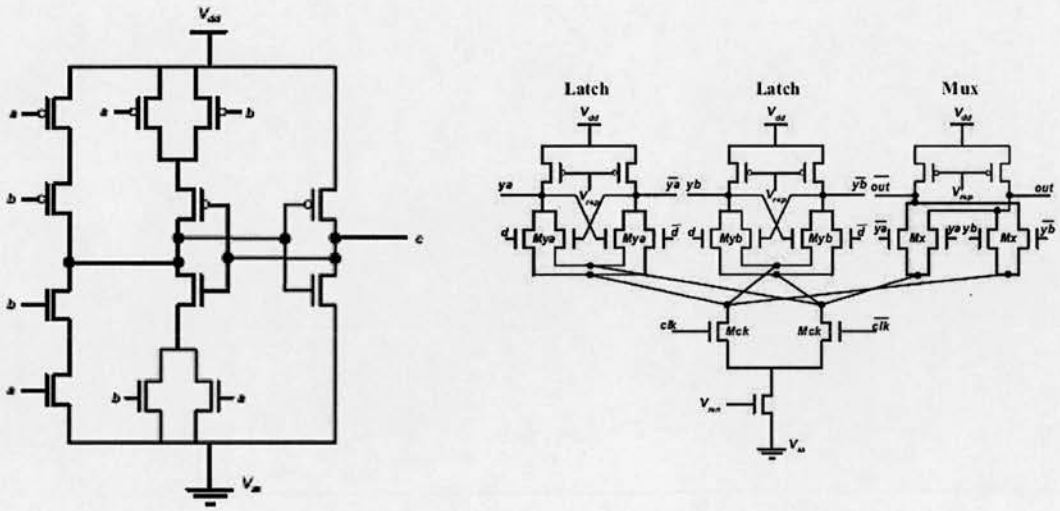


Figure (a) Figure (b)

Figure 5-6: (a) Schematic diagram of CMOS C-element (reproduced from [48]) (b) Schematic diagram of CMOS double edge flip flop (reproduced from [188]).

Using the power-cost model, the SPAMSIM2 was able to compute power and energy consumption based on the statistics of operations of the FUs, CMs, buses, and the Memory module. The entities can update the power statistics by invoking ***log_power*** function via the power_logger object. The following sequence diagram in Figure 5-7 illustrates the detailed power logging process. While an event is triggered in an entity, the corresponding handler invokes the log_power function with the elapsing time of current operation, and then power consumption cost is fetched from the Wattch power model. Finally, the power and energy consumption of the specific entity and overall architecture are updated.

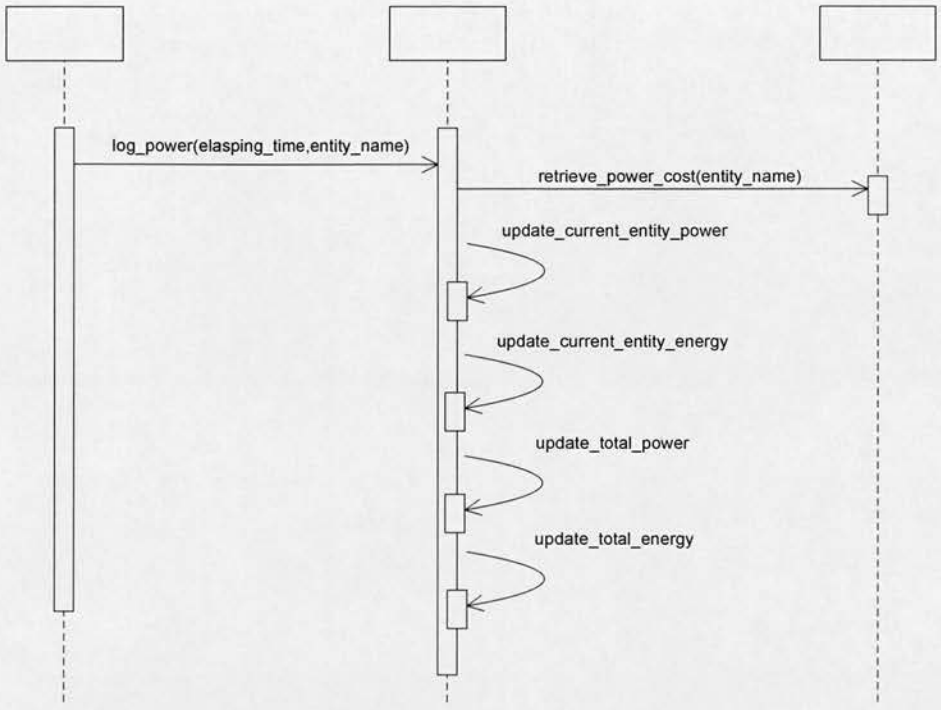


Figure 5-7: Sequence diagram of logging power and energy consumption in SPAMSIM2 simulator.

5.3 The MAPS+ compilation flow

The MAPS+ compilation framework was built on top of the Stanford SUIF2 [5] and Harvard Machine SUIF [7] compiler infrastructure. The MAPS+ compiler takes a C program as the source language. With different compiler passes, the source program is then transformed into Intermediate Representations (IRs) and optimised for the MAPS+ hardware architecture, to finally generate the target code. The compiler IRs can communicate with each other with annotations, which are items of data associated with nodes in the IR. The modular compiler passes allows a number of analysis routines for different optimisations to exist simultaneously. For different MAPS+ architecture configurations, different compiler passes can be chosen.

5.3.1 Overview

The compilation flow of the MAPS+ compiler is depicted in Figure 5-8, which performs the following functions in order: (1) the front-end SUIF parser to generate language independent SUIF2 IR from C source code; (2) Backend Machine SUIF passes to convert SUIF2 IR to virtual hardware platform SUIFvm; (3) CFG and SSA generation; (4) multithreaded code generation based on the profiling information in the SSA; (5) hardware and software code partitioning and hardware netlist block mapping; (6) register allocation and code finalisation for the software part.

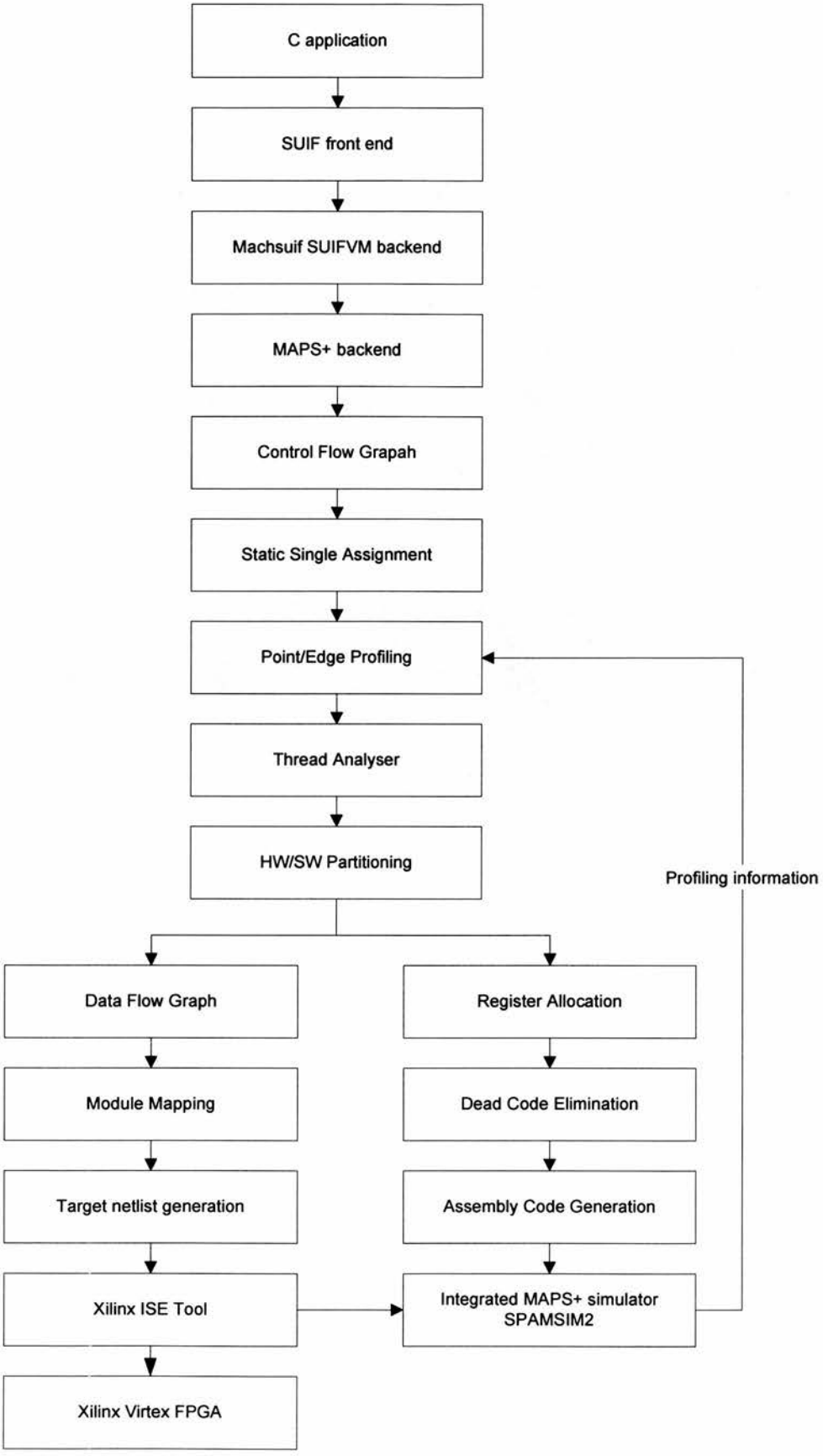


Figure 5-8: MAPS+ architecture compilation flow

5.3.2 SUIF compiler

The Stanford University Intermediate Format (SUIF) [5] system is a compiler infrastructure designed to support collaborative research and development of compilation techniques. SUIF is able to take programs written in high level languages such as C, FORTRAN and Java and transform them into assembly instructions while permitting detailed program transformations based upon program representations. The SUIF system implemented in the MAPS+ complier was the second version- SUIF2, which was released in 1999.(In the context of this thesis, references to SUIF refer to SUIF2.) The modular design of SUIF allows different program representation and program analyses to be combined easily. Also the IRs enable the SUIF compiler to be extensible into new areas of compiler and architecture research, which allows users to create new instructions for new program construct semantics and new program analysis.

The SUIF IR [184] file represents a number of basic programming constructs, and a set of SUIF object nodes. Object nodes in SUIF IR are arranged in tree structures. The SUIF IR object hierarchy is shown in Figure 5-9. At the top level, the Object, ***SuifObject*** and ***AnnotableObject*** represent abstractions of the SUIF IR node. The object class provides metaclass information or an IR node. SuifObject provides user-level functions, e.g. printing, cloning, data structure traversals. ***AnnotableObject*** allows derived information to be passed between different IRs. While the source program was transformed by SUIF, the top level IR is represented by the ***FileSetBlock***, which contains global information of the program, symbol tables, and the procedure definitions. Computation nodes derived from ***ExecutionObjects*** are the ***Statement*** and ***Expression*** subclasses. ***Statements*** represent the changes to the execution path, such as ***ForStatement***, and ***IfStatement***. ***Expressions*** capture different program semantics, such as type definition, and address symbols.

Due to these loops and structures IRs cannot be directly translated into machine code, a SUIF IR file will need to be transformed by replacing all the complex high-level structural blocks with machine code like statements. Dismantler passes are used to

conduct the tasks, e.g., the dismantler pass, ***dismantle_for_statements*** converts any for statement in the intermediate code into a series of tests and branches that can later be translated into machine instructions.

SUIF passes are compiled into shared object files and can be dynamically loaded by the SUIF driver. The SUIF script file named c2suif specify the orders to load SUIF passes object files to create a SUIF IR file from the input C source file. Also the native c pre-processor front-end and the SUIF file converter, and some dismantler needs to specify in the script file.

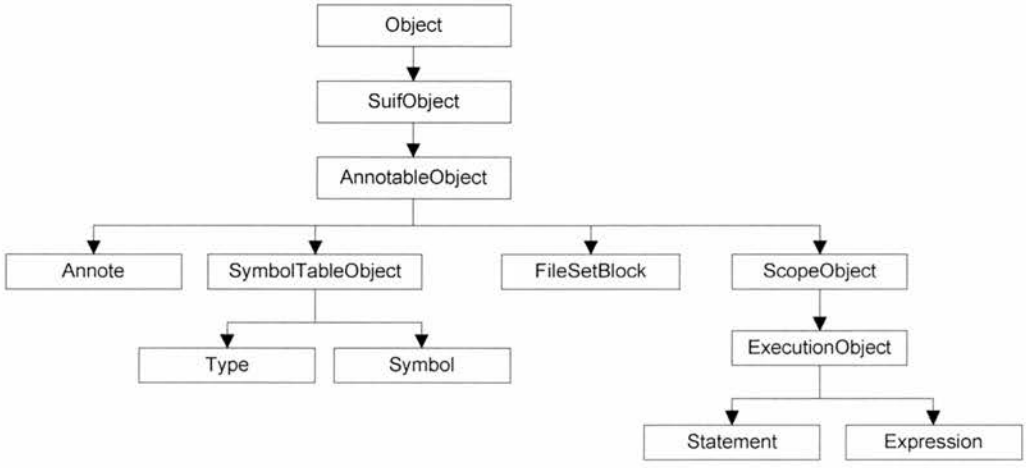


Figure 5-9: SUIF IR object hierarchy

5.3.3 MachSUIF compiler backend

Machine SUIF (MachSUIF) [7] is an extensible infrastructure for constructing compiler back ends based on the SUIF compiler framework. It is designed to ease the process of developing a compiler backend for a new architecture. MachSUIF uses SUIF's infrastructure to apply its passes. Because MachSUIF only deals with machine instructions, it reduces SUIF's complexity by limiting accessible functionalities. MachSUIF provides machine-specific compile-time optimisations to exploit the underlying computer architecture. This is realised by a one-to-one correspondence between MachSUIF IR instructions and machine instructions. As a result, MachSUIF

provides a pass named do_s2m to translate the IR into the SUIF Virtual Machine (SUIFvm). SUIFvm models instruction sets for a virtual RISC architecture. At this stage, unlimited registers are available and architecture details are omitted, which ease the implementation of the optimisation passes such as control flow analysis, and dead code elimination.

5.3.4 MAPS+ target library

MachSUIF supports different target architectures by de-coupling the SUIFvm from target-specific back ends. In order to support the MAPS+ architecture, a new target library-*maps* was developed, which transformed the SUIFvm. The *maps* library interprets the contents of SUIFvm, and the virtual instructions are then translated into MIPS-like instruction set for the MAPS+ architecture. In addition to the MIPS ISA, multithreaded instructions and reconfigurable function unit operations are defined, which can be used by the optimisation passes, such as the thread Analyser pass, and the HW/SW partitioning pass.

Apart from the instruction translation, another important task of the maps library is to instantiate MAPS+ specific data structures. This includes the number and types of register files, and the widths of registers. Machine-independent passes perform the optimisations based on the MAPS+ architecture descriptions. In the maps library, the RegInfo object describes how the MAPS+ architecture manages registers and stack space. This contains registers for storing the arguments, registers for storing procedure call results, and constant register. The thirty-two registers in the MAPS+ architecture are classified in Table 5-1:

Category	Descriptions	Registers
Constant zero	The register always stores constant zero value.	$0
Temporary registers	These are temporary placeholders for execution. A procedure needs not to these temporary registers before modifying.	$1-$8, $22-$28 (not include $26)
Saved registers	Saved registers are used to store values that a caller may need after a procedure call.	$9-$15
Argument registers	Arguments to a procedure are stored in the register set. If the arguments number exceeds six, the rest will be stored in the stack.	$16-$21
Procedure result	For a procedure with return result, the value or the address of the data structure is stored in the register.	$26
Stack pointer	The register is initialised to be a pointer to a block of reserved memory. The value grows from bottom to top, which is used for dynamic allocated data structures at run time.	$30
Heap pointer	The register is initialised to be a pointer to a block of reserved memory. The value grows from top to bottom, which is used for allocating memory space for data structures at design time.	$29
Return address	Whenever a procedure is called, the return address is stored in this register. It indicates where execution should resume after completion of the procedure.	$31

Table 5-1 : Register categories of the MAPS+ target library

5.3.5 CFG/SSA generation

Many MachSUIF passes need a CFG as the basic data structure. The MachSUIF CFG passes provides a useful compile-time data abstraction for graph-based transformations. It packs a procedure's instruction list into basic blocks, which enables

efficient block reordering without having to splice the instructions into a new linear order, which allow programs to insert instructions into nodes, to move instruction around, and to add new nodes to CFG IR.

On top of the MachSUIF CFG pass, a custom-designed SSA pass has been implemented, which provides useful information for data-flow analysis and optimisations as described in Section 4.2.1. In the SSA pass, phi-nodes are implemented in the phi_node class inherited from MachSUIF Instr class. Each source variable and destination variables defined in phi-nodes have a unique name. The phi_node class maintains a matching table of the variables to restore the SSA. The SSA pass takes a CFG IR as input, and by invoking the function SSA_Build, the optimisation pass can generate a SSA form. For further variable usage and definition analysis, the get_use_link function provides traversal Use-Def link. After all the optimisations are completed, the phi-nodes need to be removed and the SSA IR will be restored into a normal CFG IR, which is done by invoking the function SSA_Restore.

5.3.6 Software-Hardware Partitioner pass

The Software Hardware Partitioner (SHP) pass is an implementation of the algorithm described in Section 4.3. Profiling information is generated by the SPAMSIM2 by running the original MIPS code (without HW/SW partitioning and thread partitioning). The run-time instruction count information is written into file "pathprofile". The SHP pass is able to characterise specific basic blocks as hot execution path, by using the information from "pathprofile". The rest of the basic blocks represent non-critical parts of the application. With the grouping algorithm, one or several candidate basic blocks are grouped together to form the hardware blocks.

Then data-flow analysis is performed on these hardware blocks by using the data flow graph generation class DFG and the SSA pass. The SSA IR provides the information of available expressions, live variables, and reaching definitions. Also each variable in the hardware block just has a unique name, which eases the difficulty of

hardware netlist generation. The DFG class determines how these hardware blocks manipulate data flowing in and what the output data for these blocks are.

Once the DFG is generated, the SHP will perform a one-to-one mapping to a hardware netlist from the macro library. The macro library of the different functional blocks for the RFU simulations are generated by the CORE generator and then synthesised on the Xilinx ISE tool flow. The power consumption and delay information are taken into the SHP passes and provided a macro-lib for the compilation tool flow. The black box descriptions of hardware operations for RFU are summarised in the following Table 5-2.

Category	Descriptions
Input Pins	Describe the input pins of current hardware black box.
Output Pins	Describe the output pins of current hardware black box.
ROP	Group of the data flow graphs descriptions, which will be evaluated by the RFU in SPAMSIM2 simulator.
Size	Estimated logic cells numbers of current hardware block size from information provided by the macro library.
Delay	Estimated delay of current hardware block in nano-seconds.
Current	Estimated current to drive current hardware block. This information is obtained from Xilinx XPower tool, and will be used by SPAMSIM2 to estimate power consumption and energy consumptions.

Table 5-2 : Hardware black box description.

After the hardware blocks are extracted, the corresponding ROP instructions will be placed into the original source program body. Each ROP instruction has a unique sequential number, which provides a reference for the SPAMSIM2 simulator to load the corresponding hardware block box. The input and output pins of the hardware black box are mapped to the source variables and destination variables of the ROP instruction. The rest of the software codes remain the same as MIPS ISA form.

5.3.7 Thread Analyser pass

The Thread Analyser pass (TA) is the pass of the implementation of the threaded partitioning algorithms described in Section 4.2. The TA pass exploits loop-level parallelism and speculative threads, which requires underlying MAPS+ architecture hardware support. Along with the loop information and Control Flow Analyser (CFA) pass [186], the TA is able to partition the sequential C code into multithreaded MIPS assembly code for the multithread SPAMSIM2 simulation environment.

From the profiling information, the TA pass is able to evaluate the benefit of parallelisation for each loop nest, and choose the most profitable loop nests for parallelisation. Loop induction variables are identified via the control flow analyser pass. The loop-carried control dependencies can be speculated. From the data dependence analysis, the TA will also try to locate the appropriate places for speculative threads in sequence code body. Data value predictor supported in MAPS+ architecture provides the possibilities to improve thread-level parallelism among speculative threads.

5.3.8 Register Allocation

The MachSUIF provides a register allocation pass (Raga), which uses graph colouring algorithm [185] for register allocation. Algorithm implemented in raga creates a graph to represent variables and conflicts between variables. A colouring is then a conflict-free assignment. If the number of colours used is less than the number of registers, then a conflict-free register assignment is possible. If the conflictive register assignment happens, the Raga pass has to insert register spills to save values in memory. The Raga does not determine the position of each spill on the stack, and the code finalisation pass will resolve the memory address allocation.

5.3.9 Code finalisation

Code finalisation pass (Fin) is responsible for the final translation work. It allocates the stack frame for each compiled procedures. The Fin pass also adds code to save and restore callee-saved registers. The symbolic reference to the stack-frame locations are replaced by effective-address expressions with specific frame offsets. Also prologue code and epilogue code for the MAPS+ architecture are introduced.

5.3.10 Code Generation

After the code is finalised, the MAPS+ compiler needs to print the assembly code. An assembly code printer named m2s performs this task, which is a modification of original MachSUIF m2a pass. The code generation pass has been extended to handle the multithreaded instructions and ROP instructions and the pass is capable of generating header bootstraps to invoke the main functions.

5.3.11 Standard C library emulation

The benchmark programs invoke several C library functions which are not provided in the SUIF/MACHSUIF compilation framework. The detlibc[16] package was chosen for the MAPS+ compiler to provide the statically-linked standard C library, which contains the system call wrappers and commonly used standard C functions with small binary size. The dietlibc source codes are compiled by the SUIF C compiler into SUIF IRs. With the SUIF link tool, the library and the benchmark programs will be linked into one big SUIF IR, which can be analysed and optimised by the SHP/THP passes. Finally the C I/O functions are compiled into low-level system calls. The SPAMSIM2 provides a similar mechanism to execute these system calls as the one used in the SimpleScalar[17] simulator. SPAMSIM2 simulator emulates the system calls by translating them to equivalent host operating-system call and executes the calls on the benchmark program's behalf. For example, a fwrite() function is executed in the following order: 1) When fwrite() is invoked, the source code body provide by dietlibc

will be executed. 2) The control is passed to write function by the dietlibc function call. 3) Then a write() system call function is emulated in the SPAMSIM2 simulator, write buffer data are passed from SPAMSIM2 memory space to operating system memory space and the result of execution will be written back to the result register of the SPAMSIM2 simulator, which will be accessed by the benchmark program.

5.4 Summary

The simulation and compilation framework have been described in detail in this chapter. The SPAMSIM2 simulator have been introduced, which models different aspects of an MAPS+ architecture, such as handshaking protocols conducted by CMs, event-driven communication kernel for the asynchronous design, and different functional units for performing execution. Then the power and energy consumption model based on the Wattch power analysis tool has been introduced in detailed.

Furthermore, the compilation framework implementing the algorithms described in Chapter 4 are also introduced. The framework based on the SUIF2 and MachSUIF infrastructure. The passes perform the software hardware partitioning functions for deploying codes into RFUs and other functional units, and thread partitioning functions for spreading code into different TPUs. The next chapter describes the results of executing the benchmarks on MAPS+ architecture.

Chapter 6
Experimental Results

6.1 Introduction

This chapter presents experimental results from the SPAMSIM2 simulator, which provides performance, power and energy consumption evaluation of the MAPS+ architecture for the execution of benchmark programs.

Firstly, the benchmark programs are summarised. Then the simulation environment setup for the synchronous baseline and the asynchronous MAPS+ architecture are introduced. The performance and energy consumption results are presented, which are divided into four sections. With different configuration files for the SPAMSIM2 simulator, the results show the trade-off between performance, power and energy consumption in a synchronous MIPS baseline architecture, asynchronous MAPS architecture, asynchronous MAPS+RFU architecture, multithreaded MAPS architecture, and multi-threaded MAPS+RFU architecture, respectively.

6.2 Benchmark programs

There is a growing demand for mobile devices to support multiple services, such as mobile office applications, multimedia services, wireless communications, video/audio codecs, compression and error correction algorithms. Therefore, ten benchmark programs were chosen to represent these aspects of embedded applications.

Dijkstra's algorithm for shortest path calculation was chosen as it is used for routing packets in networking applications; the Patricia program represents routing

tables in networking applications. Two programs were included from the security applications category: BLOWFISH – a symmetric block cipher algorithm, and SHA – an algorithm for secure hashing. Telecommunication programs are another important category in our benchmark programs. The final category represents program commonly used in communications, such as Fast Fourier Transform (FFT) algorithm, the GSM and ADPCM audio codec algorithms, the frequency hopping generator for Bluetooth [14] baseband, and the CRC32 error correction algorithm, and a data compression algorithm – GZIP. These benchmark programs were obtained from different sources, mainly from the MiBench [1] benchmark set, the frequency hopping program from Bluetooth baseband test bed and GZIP program from SPEC2000[15]. More details on the benchmark programs are listed in Appendix A.

The C source files of the benchmarks (with minor modifications for input and output purposes) were compiled into MIPS like assembly codes for execution on the SPAMSIM2 simulator. For the simulations to be tractable, medium-sized datasets were chosen as input for the MiBench. The benchmarks were executed until completion with the average number of the instructions in each benchmark being around 90-120 million. The average simulation time for a benchmark was between 50 to 60 minutes on a Pentium IV 3.2 GHz desktop with 2GB memory.

6.3 Simulation setup

The SPAMSIM2 can be configured to run in different modes, such as synchronous mode, asynchronous mode, RFU mode, multithreaded mode, and a combination of these modes. Details of the configuration file and the invoking script can be found in Appendix B.

The parameters of the baseline synchronous processor shown in Table 6-1 are based on the MIPS 4K processor [8] using the 180nm CMOS process. The speed chosen for the synchronous baseline process is 250 MHz. It uses a 5-stage pipeline with a configurable instruction issue width – choice of 1, 2 and 4. The baseline processor contains thirty-two, 32-bit general-purpose registers used for scalar integer operations

and address calculations. In addition, there are two extra registers for the Program counter (PC) and Thread Number (TNUM). The register file consists of two read ports and one write port and is fully bypassed to minimise operation latency in the pipeline. The Dinero IV Cache model provides a highly configurable cache module. In the baseline processor model, the size of instruction cache is 8K bytes and 4-way set associative, with a line size of 16 bytes and a LRU replacement policy. The data cache is 16K bytes in size and has similar setup as the instruction cache. The same 180nm CMOS process was chosen for the asynchronous MAPS+ architecture, with similar setup for different functional units. Whilst clock frequency is not applicable for the MAPS+, a mean operating rate was measured on SPAMSIM2 simulator, which is about 320 MHz with an average latency of 3.125 ns. In the comparison [11] of a synchronous Manchester carry chain adder to asynchronous Manchester carry chain adder, the asynchronous case has a 40% premium in extra. Alex Branover et al.[12] state that the area overhead for converting synchronous circuits into asynchronous ones is relatively smaller for larger circuits. In their experiments, the asynchronous circuits were about 16% larger than the original synchronous one. Based on these researches, we estimate there is a 20-30% area overhead for the asynchronous MAPS+ over the original synchronous MIPS.

	Synchronous Baseline	Asynchronous MAPS+
Clock Frequency	250MHz (4 ns per cycle)	320 MHz (Average latency is 3.125 ns)
Instruction Cache	16K, Replace-LRU, Write back-Always, Delay –(1 cycle)	16K, Replace-LRU, Write back-Always
Data Cache	8K, Replace-LRU, Write back-Always, Delay- 1 cycle	8K, Replace-LRU, Write back-Always
Register	Number – 34, Delay-1 cycle	Number- 34
Arithmetic logic unit	Number- 1/2/3/4, Delay-1/2/3 cycles	Number- 1/2/3/4
Floating point unit	Number – 1, Delay -2/3/4/5 cycles	Number – 1
Reconfigurable Functional Unit	1 – Delay estimated via Xilinx ISE Tool,	1 – Delay estimated via Xilinx ISE Tool,
Superscalar features	FETCH_WIDTH 1/2/4 , ISSUE_WIDTH 1/2/4	FETCH_WIDTH 1/2/4 , ISSUE_WIDTH 1/2/4
Memory	Size- 32Mb, Delay - 20 cycles	32Mb
Data Forwarding	Enabled	Enabled
Thread Processing Unit	1	1/2/3/4/5/6/7/8

Table 6-1 : Configuration of synchronous multithreaded MAPS+ vs. baseline

The delay parameters used in asynchronous MAPS+ simulator are listed in Table 6-2. The model uses the same memory/cache delay values and register files accesses values as the synchronous baseline processor. The delay parameters are based on the 180nm CMOS models used in [9][10], with assumption of improvement in technology. The execution of instructions on the MAPS+ model consists of different levels of micro-operations, e.g. Instruction Issue, register requests, ALU/FPU operations, memory access, and register write back. Programmable logics for RFU are fabricated in the same 180nm CMOS process as the rest of MAPS+ architecture. And parameters for RFU were obtained using Xilinx Alliance tool for a Virtex II FPGA. The latencies of different micro operations are randomised delays generated by the MAPS+ simulator within the range of minimum and maximum delay, and taking into account data dependent execution speeds. The values of the simulation time for the same benchmark might vary due to the randomised delay, but the asynchronous handshaking protocol via communication microagent in the MAPS+ architecture still guarantees repeatable

simulation results, whereas the execution time for a simulation depend on several factors: such as instruction type, data, and resource dependencies, and the sequence of instructions. In the asynchronous case, the dependencies between successive instructions introduce stalls in the issue units, by waiting for the results of previous instruction, whereas in the synchronous model, the availability of these results availability can be predicted exactly.

	Delay- Min Value (ns)	**Delay- Max Value (ns)**
Communication Micro-agent	0.5	1.0
X_BUS/Y_BUS/Z_BUS	0.5	1.0
Register Delay	1	2
Control Unit	2	4
Arithmetic logic unit	3	10
Floating point unit	6	20
Memory Management Unit	2	4
Thread Issue Unit	2	4
Cache Delay	4	4
Memory Delay	80	80

Table 6-2 : Asynchronous MAPS+ Delay Models for 0.18μm CMOS process.

Four sets of simulations have been chosen to test the impact of small parameters changes on the relative performance of MAPS+ architecture simulator. The parameters of the first set has 2 ns increase over the original setting presented in Table 6-2, followed by 4 ns, 6 ns and 8 ns increases respectively. The normalised simulation results against the original results are summarised in Figure 6-1, which shows a linear increment in simulation time. For each set with 2 ns, 4 ns, 6 ns and 8 ns increases over the original setting, the average normalised simulation times against original ones are 2.1, 3.1, 4.5 and 5.7 respectively. And the relative performance differences are 105.9%, 102.5%, 145.1%, 116.3%. It demonstrates that the increases in parameters settings impact the relative performance linearly.

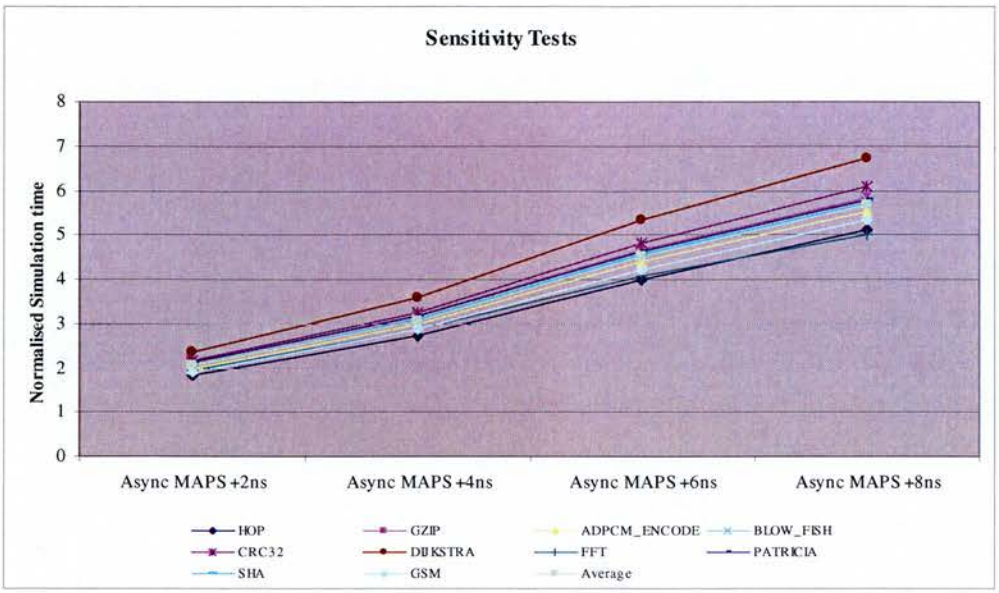


Figure 6-1: Sensitivity analysis of the SPAMSIM2 simulator

6.4 Single-threaded asynchronous MAPS

This section investigates the performance improvements for the ten benchmarks executing on the asynchronous MAPS architecture normalised against the synchronous MIPS baseline. The corresponding performance, power efficiency, power consumption break-down, energy dissipation diagrams and the trend in energy growth are also analysed.

6.4.1 Benchmark analysis

The distribution of instructions in the ten benchmarks is listed in Table 6-3. These figures depend on the application, the processor's instruction set and the compiler, but not on any architectural parameters of the processor such as the number of functional units, or cache sizes. The integer arithmetic instructions dominate, as a majority of the MiBench programs perform arithmetic-intensive signal processing relying on fixed-point data. The cryptography algorithms – Blowfish and SHA, and the frequency hopping generator HOP are characterised by code operating on bit-level data, which are performed using repeated logic operations, which explains the higher percentage of logic instructions. The audio codec, ADPCM encoder, and the

compression algorithm, GZIP are control intensive, as demonstrated by the higher number of branch instructions. Both the FFT and Patricia employ floating point operations.

Instruction classes

Benchmarks	Load	Store	Arithmetic	Logic	Multiply Divide	Branch	Floating Point	Other
HOP	1.29	0.62	55.18	25.87	1.64	15.33	0.00	0.07
GZIP	13.67	7.74	54.42	6.26	3.56	14.36	0.00	0.00
ADPCM_EN	4.40	0.75	61.52	5.56	2.92	24.84	0.00	0.00
BLOWFISH	11.94	4.62	61.66	8.35	3.27	10.16	0.00	0.00
CRC32	15.74	6.31	58.27	4.71	0.79	14.18	0.00	0.00
DIJKSTRA	21.54	6.02	51.36	0.41	6.14	14.52	0.00	0.00
FFT	9.94	5.76	48.23	3.33	14.70	5.41	12.36	0.28
PATRICIA	13.19	7.45	58.33	2.63	0.45	16.80	1.14	0.01
SHA	12.36	5.74	60.84	7.66	3.21	10.19	0.00	0.00
GSM	11.83	2.11	62.52	0.54	16.48	6.52	0.00	0.00
Average	11.59	4.71	57.23	6.53	5.31	13.23	1.35	0.04

Table 6-3 : Instruction distribution for the benchmarks

6.4.2 Performance and power efficiency

The performance and power consumption were analysed based on the Wattch power model. For fair comparison, the synchronous MIPS, without, and with clock gating technology, were compared to the asynchronous MAPS architecture. Clock gating technology is an effective means for reducing average power consumption in synchronous processors. For example, the MIPS M4K [188] core contains fine grained clock gating technology, and Azuro's PowerCentric [187] provides fully-integrated clock gating, clock-tree synthesis, and vectorless power analysis solution for ARM embedded processor cores.

Three types of architectures were compared in Table 6-4. For the Clock distribution tree model, a one-level H-tree is a common clock distribution topology, was implemented. The wire lengths of the tree was obtained from the MIPS 4K [8] specification (2.25 mm^2 in a 180 nm process). Wattch tool provides functions to estimate the clock-tree power based on the clock-tree length. For the power model of synchronous MIPS architecture without clock gating, it was assumed that the full

clock power is consumed every cycle, irrespective of activity in the architecture. The second model is a synchronous MIPS architecture with similar clock-tree model using clock gating, which contains an idle factor for representing the ratio of power consumed in the combinational logic when idle. An idle factor of 10% is assumed in the simulation, with no reduction in performance when clock gating is introduced in the architecture. The third model is the asynchronous MAPS architecture, which employs handshaking protocols as implemented in the SPAMSIM2 simulator, and therefore the components consume no power when idle. The factor is set to 0%. The average speed and power consumption of the three models therefore summarised in Table 6-4 were obtained by executing the ten benchmarks introduced in Section 6.2.

The asynchronous MAPS architecture has an average speed of 200 MIPS, which is a 27.6% improvement over synchronous MIPS. The power consumptions for the three architectures are 214 mw for a synchronous MIPS, 72 mw for a synchronous MIPS with clock gating techniques, and 77 mW for asynchronous MAPS. From these figures, one can conclude that both clock gating and asynchronous designs are able to reduce power consumption. Taking into account that the errors in the absolute power figures are likely to be at least 10%, we would consider the clock gated MIPS and asynchronous MAPS power requirement are similar. Traditionally, asynchronous designs are considered to consume less power than the corresponding synchronous designs. However, this will not always be true when fabrication processes improve and clock gating techniques mature.

In the simulations, the clock-gated MIPS has a 2182 MIPS/W power efficiency, which is an improvement of 198% over the synchronous MIPS. The asynchronous MAPS has a 2585 MIPS/W power efficiency, which is an improvement of 253% over the synchronous MIPS and 18.4% over the synchronous clock gated MIPS. In the context of the thesis, the synchronous baseline refers to the synchronous clock gated MIPS architecture.

	Synchronous MIPS architecture without clock gating	Synchronous MIPS architecture with clock gating (Synchronous MIPS baseline)	Asynchronous MAPS architecture
Process	180 nm	180 nm	180 nm
Supply voltage	1.8 V	1.8 V	1.8 V
Average speed	157 MIPS	157 MIPS	200 MIPS
Average power consumption	214 mW	72 mW	77 mW
Power efficiency	732 MIPS/W	2182 MIPS/W	2585 MIPS/W

Table 6-4 : Performance and power efficiency comparisons of synchronous MIPS and asynchronous MAPS architecture

6.4.3 Performance speedup

Figure 6-2 shows the performance improvement for the ten benchmarks executing on the asynchronous MAPS architecture normalised against the clock-gated synchronous MIPS baseline. The processors, both synchronous and asynchronous, were configured to explore the impact of increasing the number of ALUs from one to four. Multiple instructions could be fetched and issued in the same cycle. The data dependencies between instructions limit the exploitable parallelism with average speedups in the synchronous architecture limited to 12.2%, 14.6%, and 15.1% for 2 ALUs, 3 ALUs and 4 ALUs, respectively.

The asynchronous MAPS architecture exploits fine-grained concurrency by executing micro-operations concurrently. The improvements of the asynchronous MAPS over the baseline for the average speedup are 35.0%, 41.5%, 42.8% and 42.9% for 1 ALU, 2 ALUs, 3 ALUs, and 4 ALUs, respectively.

The Dijkstra benchmark has a relatively low improvement in the asynchronous MAPS with an average improvement of just 14.6%. The reason is due to the high frequencies of load- and store-memory operations in the Dijkstra benchmark. It has a distribution of 21.5% load instructions (read data from memory or cache) and 6.02%

distribution rate of store instructions (write data to memory or cache). Given that fully asynchronous memory systems are difficult to implement, the cache and memory are the same in both asynchronous MAPS and the synchronous MIPS baseline, although the latter has an extra asynchronous handshake wrapper to handle communication between the asynchronous data path and the synchronous memory system.

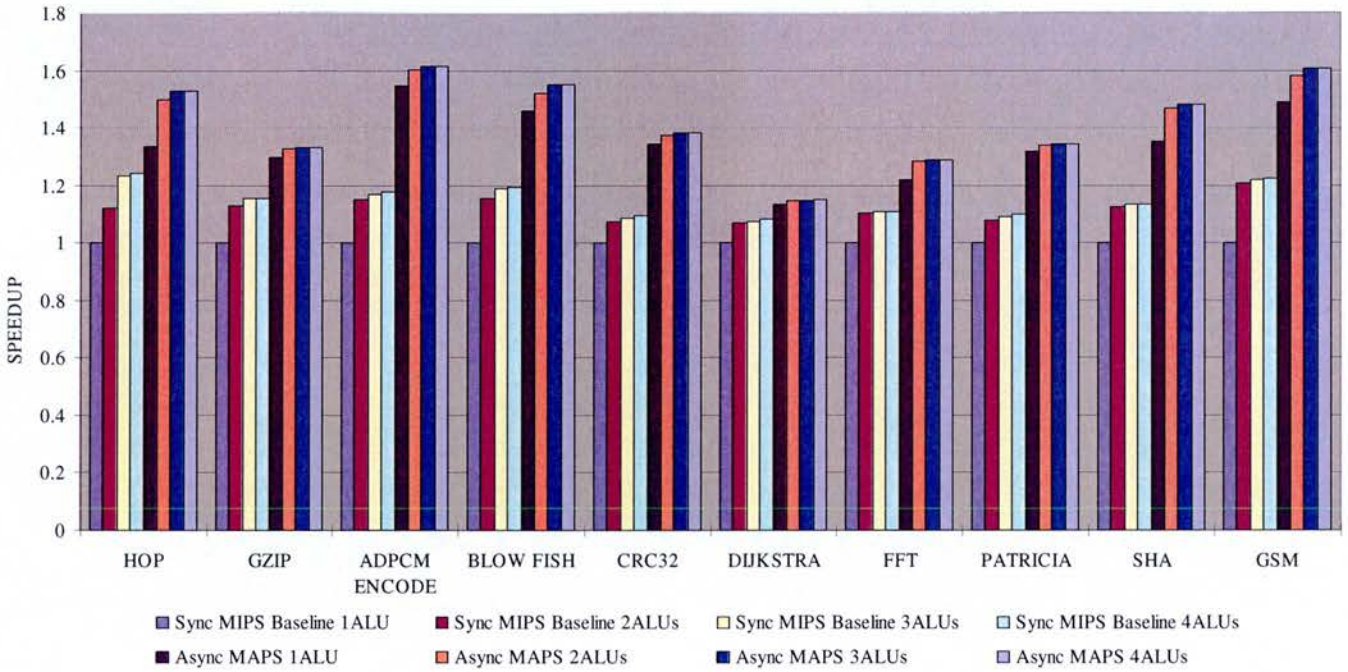


Figure 6-2: Speedup of asynchronous MAPS compared to synchronous MIPS baseline with variable number of ALUs.

6.4.4 Energy consumption

As stated in Section 5.2.6, the power and energy analysis tool in the SPAMSIM2 simulator is based on the Wattch power tool. The normalised energy breakdown for each benchmark is shown in Figure 6-3, Figure 6-4, and Figure 6-5. On average, about 33.2% of energy in the synchronous clock gated MIPS baseline with one ALU configuration is spent in the instruction cache and 14.1% in the data cache. In contrast, the asynchronous MAPS architecture spends about 32.9% of the energy in the instruction cache, and 12.3% in the data cache. The energy spent in the cache contributes a substantial proportion of the total energy in both synchronous and asynchronous cases.

Around 13.5% of energy in the synchronous baseline is spent in the clock. In the MAPS architecture, the global clock is replaced by CMs for synchronisation. The CMs consume 2.3% of the energy in the MAPS model, which is much less than the clock energy in the synchronous baseline. Floating-point units are expensive in terms of both area and power consumption, and their high latencies are usually difficult to hide. In the simulation, two floating-point benchmarks were executed. In the case of FFT, the proportion of energy spent in FPU is 32.7% in the synchronous clock-gated MIPS, and 21.5% in MAPS, and in the case of Patricia, the proportions are 39.4% in synchronous MIPS baseline, and 37.2% in MAPS, respectively.

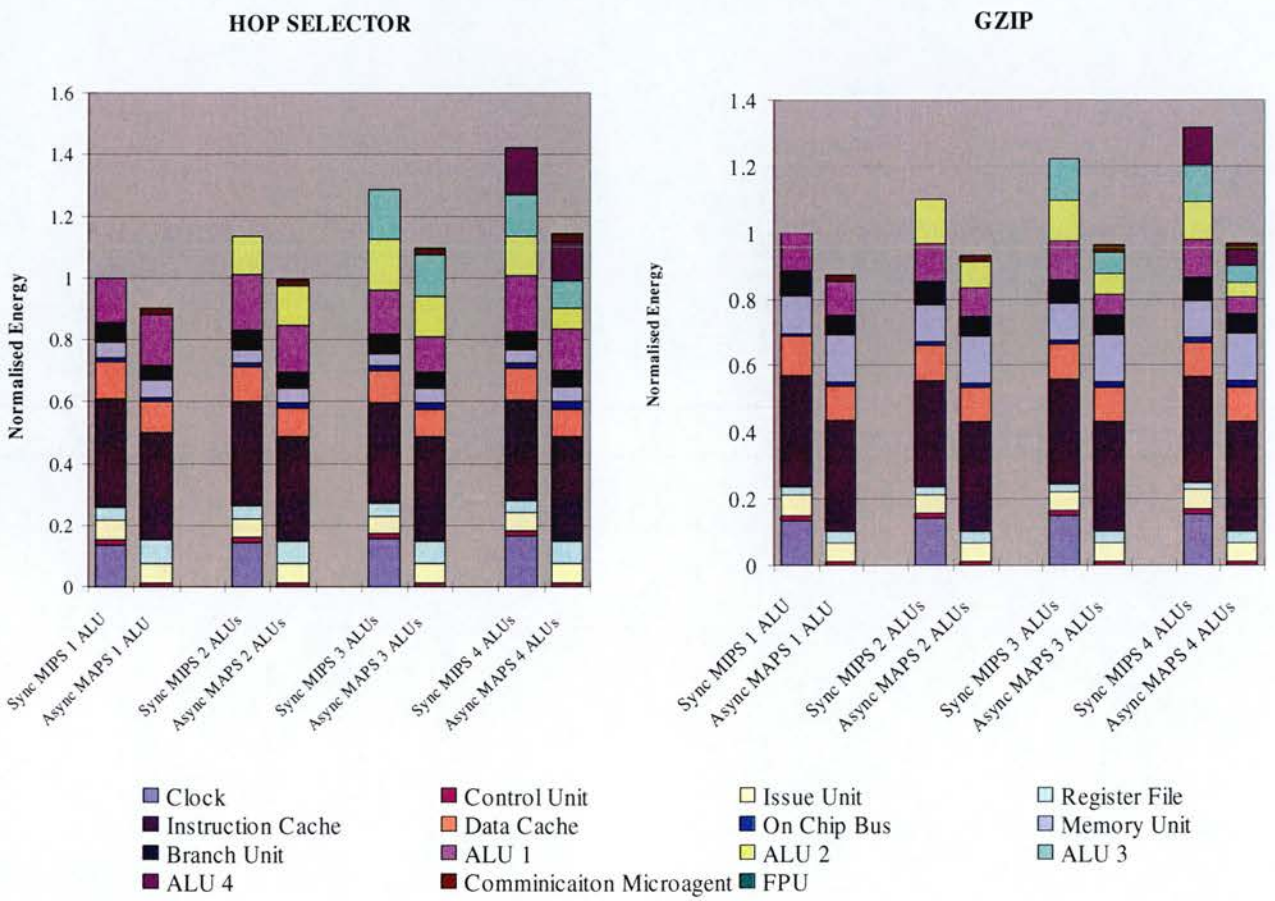


Figure 6-3: Normalised energy dissipation of the synchronous clock gated MIPS and the asynchronous MAPS architecture

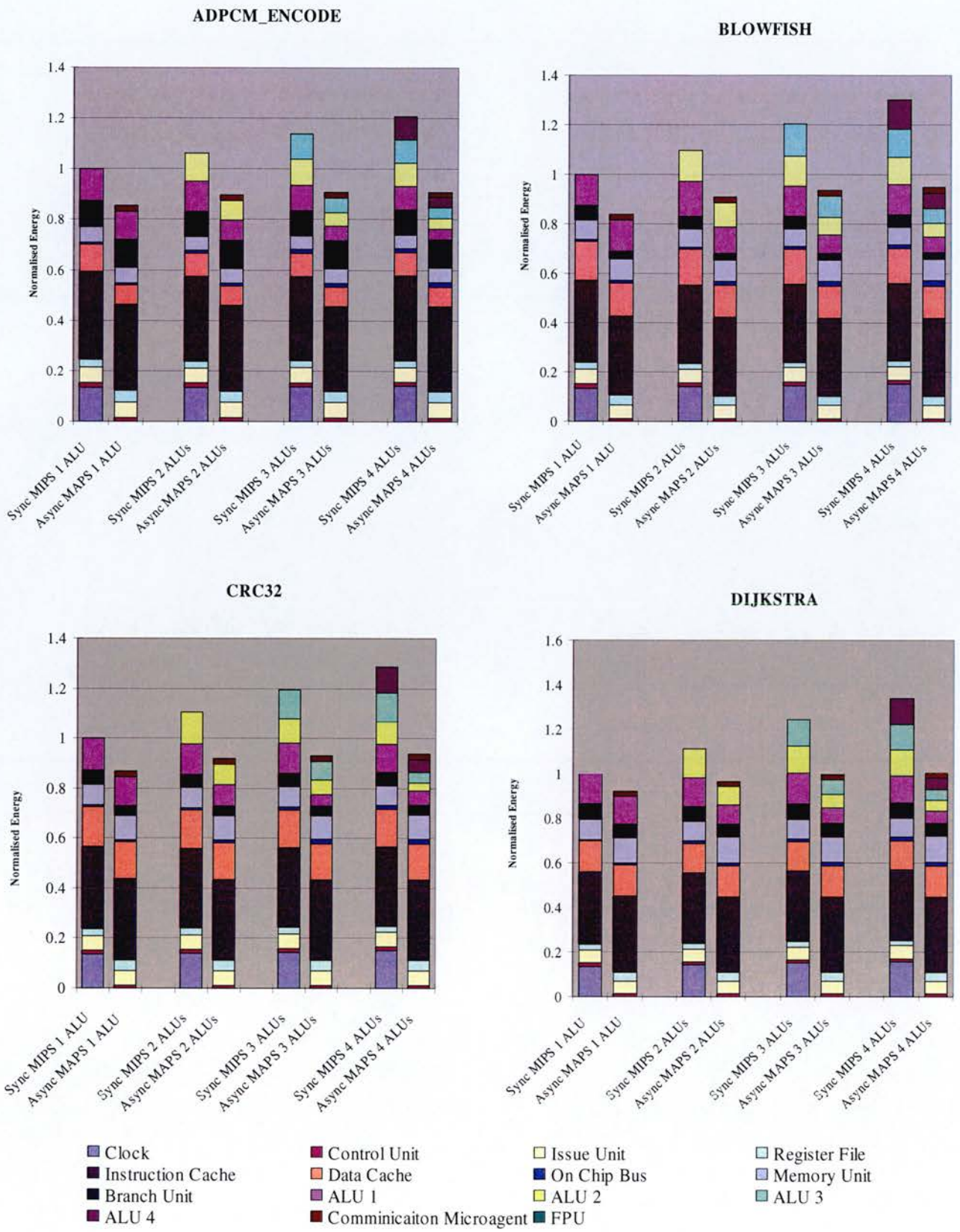


Figure 6-4: Normalised energy dissipation of the synchronous clock gated MIPS and the asynchronous MAPS architecture

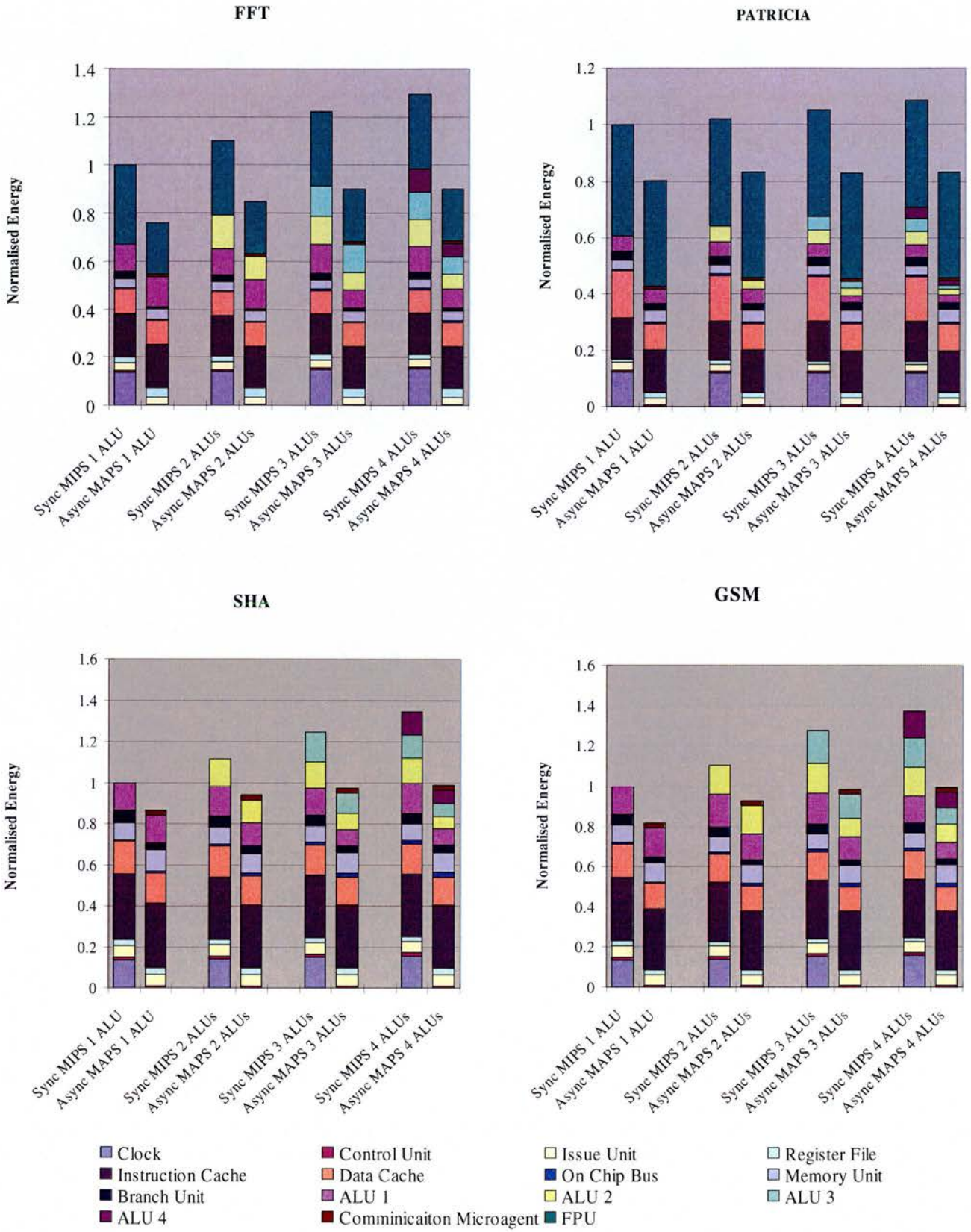
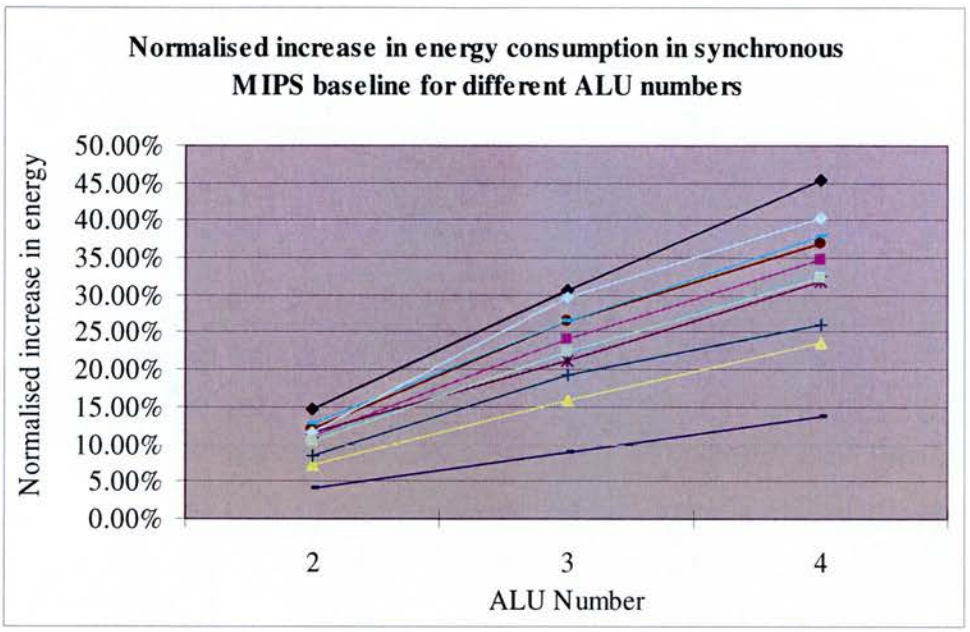


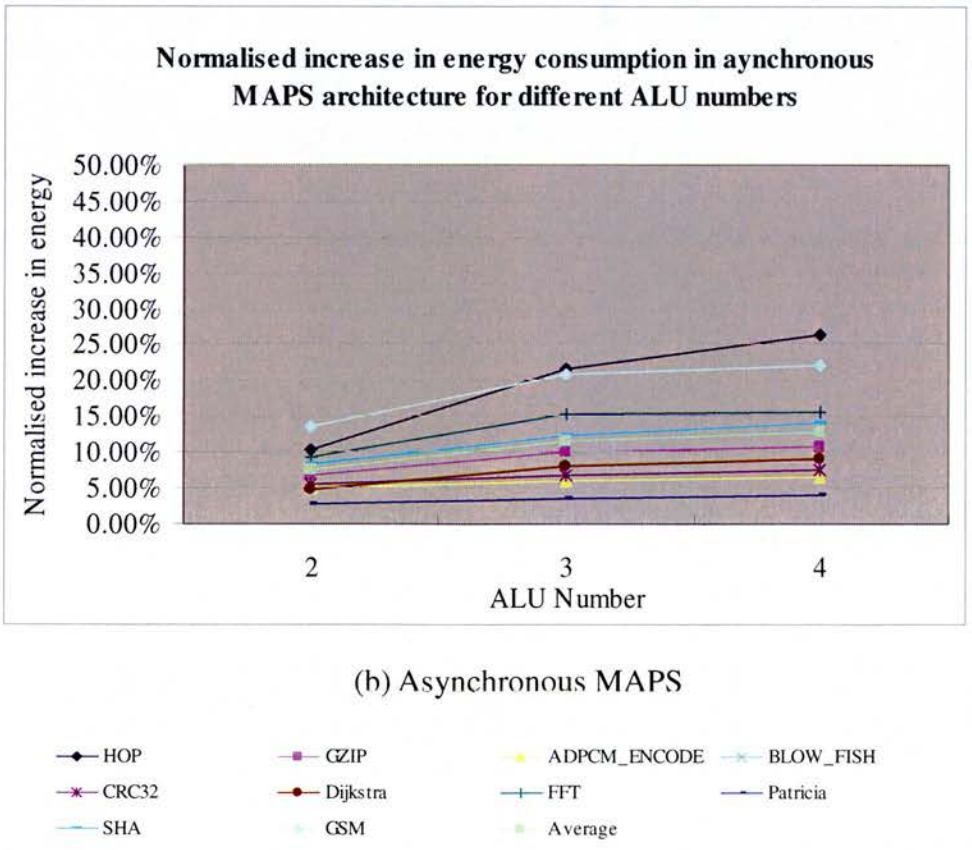
Figure 6-5: Normalised energy dissipation of the synchronous clock gated MIPS and the asynchronous MAPS architecture

As shown in Table 6-4, the power consumption of MAPS is similar to synchronous clock-gated MIPS, within 10% margin of error. However, MAPS provides the potential to explore average case performance instead of worst case performance. Therefore, the MAPS simulation demonstrates less energy consumption and the average energy savings are 9.3% for one-ALU case, 12.2% for two-ALU case, 20.3% for three-ALU case, and 28.4% for four-ALU case.

Furthermore, for investigating the scalability issues of synchronous and asynchronous architectures, Figure 6-6 depicts the normalised increase in energy consumption for the synchronous MIPS and asynchronous MAPS with different ALU numbers. The average increase in energy for the MAPS architecture is 2.7% per ALU compared to 11.0% for synchronous MIPS. These results show that asynchronous MAPS architecture scales better.



(a) Synchronous clock gated MIPS



(b) Asynchronous MAPS

Figure 6-6: Normalised increase in energy consumption (a) Synchronous MIPS baseline (b) Asynchronous MAPS.

6.5 Single-threaded asynchronous MAPS+ RFU

The asynchronous MAPS architecture with integrated Reconfigurable Fictional Unit (RFU) has been described on Chapter 3 and the ten benchmarks outlined in section 5.6.2 were chosen to exercise the performance of the single-thread asynchronous MAPS+RFU architecture. Programs were compiled via the SHP pass, which implements the algorithms described on Chapter 4.

6.5.1 Benchmark analysis

Instruction distribution for the benchmarks with hardware and software partitioning is listed in Table 6-5. With the reconfigurable functional unit, group of arithmetic, shift and logic operations of the sequential code within one basic block or across several basic blocks were converted into a single ROP, and internal data parallelism of the RFUs will accelerate the execution speed of the program. In the HOP benchmark, hand-coded optimisation arranged several shift and logical operations in one single basic block, leading to a higher ratio of hardware code extraction. However, extracting floating-point operations have been avoided given that complexity of implementing them in field programmable logic.

Instruction Classes

Benchmarks	Load	Store	Arithmetic	Logic	Multiply Divide	Branch	Floating Point	ROP	Other
HOP	0.77	0.48	28.95	1.95	0.75	11.93	0.00	55.12	0.05
GZIP	11.21	6.79	47.34	4.76	2.42	12.95	0.00	14.55	0.00
ADPCM_EN	1.27	0.65	49.23	3.54	0.00	21.40	0.00	23.90	0.00
BLOWFISH	10.08	4.54	52.82	6.97	2.85	10.06	0.00	12.68	0.00
CRC32	13.17	5.44	49.64	3.10	0.78	13.96	0.00	13.92	0.00
DIJKSTRA	14.51	4.62	29.42	0.30	1.29	11.46	0.00	38.40	0.00
FFT	9.58	5.72	46.38	3.29	13.49	5.37	12.27	3.64	0.27
PATRICIA	11.37	7.36	50.25	0.73	0.37	16.45	1.12	12.34	0.01
SHA	10.74	5.33	58.12	6.40	2.81	9.67	0.00	6.93	0.00
GSM	11.03	1.41	56.72	0.23	15.29	6.24	0.00	9.08	0.00
Average	9.37	4.23	46.89	3.13	4.00	11.95	1.34	19.06	0.03

Table 6-5 : Instruction distribution of benchmarks with HW/SW partition

The HOP kernel consists mainly of bit manipulation and shift operations and has a simple control flow as shown in Figure 6-7, which leads the SHP compiler pass to extract group of instruction to form ***ROPs***. In order to improve the performance of programs running on the MAPS+ architecture, hand-coded optimisation of the source code is sometimes needed to group more arithmetic and logic operation into ***ROPs***.

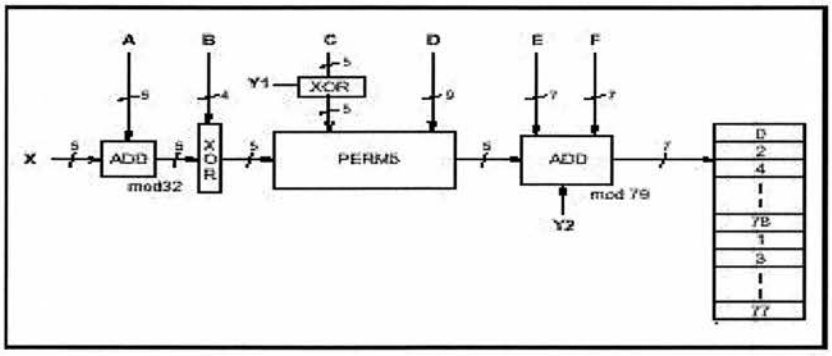


Figure 6-7: Block diagram of HOP selection kernel for the Bluetooth baseband.

The SHP extracts the hardware code from the sequential C automatically. While the ROP instructions are inserted into the MIPS assembly code, dummy hardware block descriptions will also be generated by the compiler. Table 6-6 shows the binary code size after hardware and software partitioning. The estimated bitstream size of FPGA logic blocks are based on the Virtex [20] configuration information provided by Xilinx.

	Baseline binary size (bytes)	HW/SW binary size (bytes)	ROP Number
HOP	1548	55884	4
GZIP	217031	607471	40
ADPCM_EN	151097	164649	2
FFT	152208	295828	5
GSM	231071	448055	30
CRC32	152932	165716	3
BLOWFISH	191179	215976	9
Dijkstra	191179	274699	13
PATRICIA	159117	362029	15
SHA	152177	186357	5
GSM	231071	448055	30

Table 6-6 : Baseline binary size and HW/SW binary size.

As mentioned in Table 5-2 in Section 5.3.6, the hardware parameters for a ROP are obtained from Xilinx Alliance and XPower tools. The information for a specific ROP is loaded into the SPAMSIM2 simulator. Data parallelism provided by RFUs comes with the extra overhead, which includes time to load configuration from memory, and time to deploy configuration on FPGA's SRAM. For a specific ROP, the normalised overhead is the ratio of time of loading and deploying the configuration onto the FPGA to the time of executing the instruction. Equation 6-1 shows how one measures the average normalised overhead in a benchmark. $\boldsymbol{TIME_{loading\_rop\_i}}$ is the time to load $\boldsymbol{ROP_i}$ from memory into RFU, $\boldsymbol{TIME_{Deploying\_rop\_i}}$ is the time to deploy $\boldsymbol{ROP_i}$ to RFU's configurable fabric, and $\boldsymbol{TIME_{Executing\_rop\_i}}$ is the time for executing the $\boldsymbol{ROP_i}$.

$$Overhead = (\sum_{i=0}^{n} \frac{TIME_{Loading\_rop\_i} + TIME_{Deploying\_rop\_i}}{TIME_{Executing\_rop\_i}}) / n$$

Equation 6-1 : Average normalised overhead of reconfiguring ROP

In a synchronous system, the normalised overhead is an illustration of cycles of reconfiguring a ***ROP*** instruction. Figure 6-8 illustrates the normalised overhead measure on the synchronous MIPS baseline with RFU. There are no RFU caches and multi-context switching enabled in the simulation.

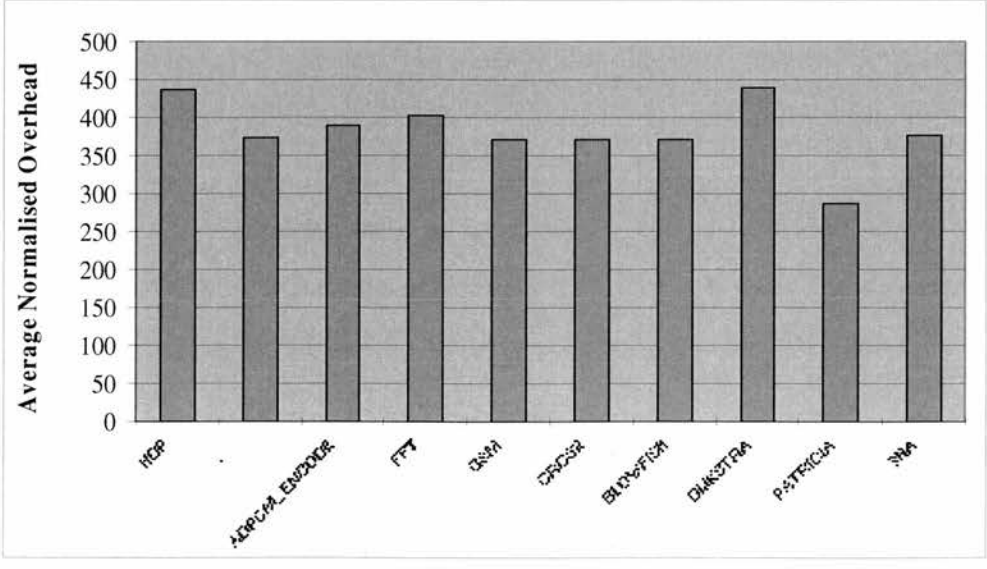


Figure 6-8: Quantised reconfiguration overhead of the benchmarks

From this figure, it is clear that the ***ROP*** operations incur higher overheads for loading and deploying. Without mechanisms to hide the overhead, reconfigurable functional unit is unable to bring any benefit to accelerating the execution speed. The actual time for executing a ***ROP*** is illustrated in Equation 6-2. We assume the configuration of a ***ROP*** is loaded once. In order to match the ROP_i actual executing time ($TIME_{Actual\_Executing\_rop\_i}$) with the ideal executing time $TIME_{Executing\_rop\_i}$, it is required to execute the ROP_i as much as possible. The execution number of such ROP_i is $EXEC\_NUMBER_{rop\_i}$. However, this execution number is controlled by the program, and can not be optimised at run-time.

$$TIME_{Acutal\_Executing\_rop\_i} = TIME_{Executin\_rop\_i} + \frac{TIME_{Executing\_rop\_i} \times Overhead_{rop\_i}}{EXEC\_NUMBER_{rop\_i}}$$

Equation 6-2: Actual time for executing a ROP

Combining Equation 6-1 and Equation 6-2, one can conclude that reducing loading and deploying times of a ROP results in lower overhead. Caches can be used to hide the latency of loading data access time. RFU caches are introduced into the MAPS+RFU architecture. Furthermore, multiple contexts are used to reduce the deploying time.

6.5.2 Performance and power efficiency

In this section, performance and power efficiency are analysed on the MAPS+ architecture with RFU. From Table 6-7, the average speed of the benchmark is 212.1 MIPS, which is a 35.4% improvement over the synchronous clock-gated baseline. However, the improvement over an asynchronous MAPS with one ALU is reduced by 6.1%. It shows that the RFU contributes a small amount to the speedup and to the overall performance. As extra hardware is introduced, such as the RFU and RFU cache, more power is consumed. In the simulation, the average power consumption is 388 mW with a 439.9% increase over the synchronous clock gated MIPS and 391.5%

increase over the asynchronous MAPS. Also the power efficiency is reduced to 547 MIPS/W.

These results show that RFU is not an efficient way for accelerating normal applications on a battery-driven embedded environment. In the asynchronous MAPS+RFU architecture setup, we have to trade an extra 392% power for a 6% performance improvement. Even though with automatic HW/SW partitioning techniques in the compiler, one still needs to carefully choose suitable applications for such reconfigurable computing architecture, e.g. baseband processing, hardware optimised video and audio codecs.

	Asynchronous MAPS+ RFU architecture
Average speed	212 MIPS
Average power consumption	388 mW
Power efficiency	547 MIPS/W

Table 6-7 : Performance and power efficiency of MAPS+RFU architecture

6.5.3 Performance Speedup

The technique to hide the reconfiguration latency is important to improve the performance of a reconfigurable computing system. In previous research [18] has investigated configuration compression, configuration caching and configuration prefetching techniques for reconfigurable systems. The caching technique increases the likelihood of the required configuration being present on-chip. It is claimed that the configuration caching technique can reduce the reconfiguration overhead by a factor of 2.5 to 10. The RFU cache is similar to a data cache, which holds the configuration in the RFU internal memory to reduce reconfiguration overhead. In the simulations, LRU algorithm is used to replace the configuration at runtime. A multiple-context [21][22] RFU is integrated in the MAPS+ architecture, which has multiple layers of bitstreams and activates a different layer at different times.

In order to evaluate the impact of increasing the RFU cache on the overall performance of the MAPS+ architecture, the benchmark programs were simulated on

the SPAMSIM2 simulator with one ALU with different cache sizes (64 K, 128 K, 256 K, 512 K, 1024 K, and 2048 K).

The diversity of ROPs configuration size and latency among different benchmark programs leads to very different cache requirements. In the HOP selector benchmark, the performance of MAPS+ with 32 K RFU is 99.2% slower than the asynchronous MAPS. The performance is only improved marginally by increasing the RFU cache to 128 K, but it still exhibits a 94.6% slowdown. The main reason for the slowdown is the inadequate RFU cache provided, and the MAPS processor has to stall until completion of the RFU's configuration. For the majority of benchmarks, a cache size of 128K is adequate. Inadequate cache sizes decrease performance, whereas a large-sized cache does not improve performance either, as seen in the cases of ADPCM and CRC32 benchmarks.

A configurable cache adds flexibility to the RFU. From the simulation, one can find out the optimised cache configuration for different applications. Several methods [23][24][25][26] enable tuning of cache parameters to the needs of the applications. In Albonesi et al's design [24], there is partitioning of the data into one or more sub-arrays for each cache way. Decision logic and gating hardware for disabling the operation of particular ways, and software-visible register for signalling hardware to enable/disable particular ways are used to disable a subset of the ways in a set associative cache during periods of modest cache activity. This approach trades off a small performance degradation for energy savings. Zhang et al [26]'s configurable cache design allows the ways to be concatenated to form either a direct-mapped or 2-way set associative cache.

In the current SPAMSIM2 simulator, such techniques have not been implemented and the cache sizes are optimised manually. Implementation of these automatic cache configuration methods in the MAPS+RFU architecture is left as future work. By adapting these methods, one is able to optimise MAPS+ architecture by fine-tuning the cache parameters at runtime to achieve the best performance. The impact of increasing the RFU cache size on the energy consumption is analysed in the next section.

The normalised speedup of MAPS+RFU over asynchronous MAPS architecture with one ALU is illustrated in Figure 6-9. Average speedups for the MAPS+RFU against asynchronous MAPS are -70.5%, -43.6%, -14.7%, -13.3%, 10.5%, 10.5% and 10.5% with 32K, 64K, 128K, 256K, 512K, 1024K and 2048K RFU cache sizes, respectively. The highest speedup of 47.6% is achieved with the HOP benchmark with a 128K RFU cache. Hand-coded optimisations were performed on the HOP algorithms, which reduces the control-flow complexities and increases the basic block sizes. Excluding the HOP benchmark, the average speedups achieved are -67.3%, -37.9%, -21.6%, -20.1%, 6.4%, 6.4%, and 6.4% with 32K, 64K, 128K, 256K, 512K, 1024K and 2048K RFU cache sizes, respectively. Without the hand-coded optimisations, the average speedups are quite small.

The above simulation results show that a reasonably large sized cache and manual optimisation applications are critical for the MAPS+RFU architecture. It trades extra hardware, power and development time for improving performance. However, the price of a large size cache, the complexity of implementing the configurable cache, and the extra power consumption might inhibit chip manufacturers from introducing the reconfigurable computing elements into hand-held embedded processors. Therefore, the reconfigurable computing architecture, such as the MAPS+RFU might be more suitable for applications such as set-top boxes.

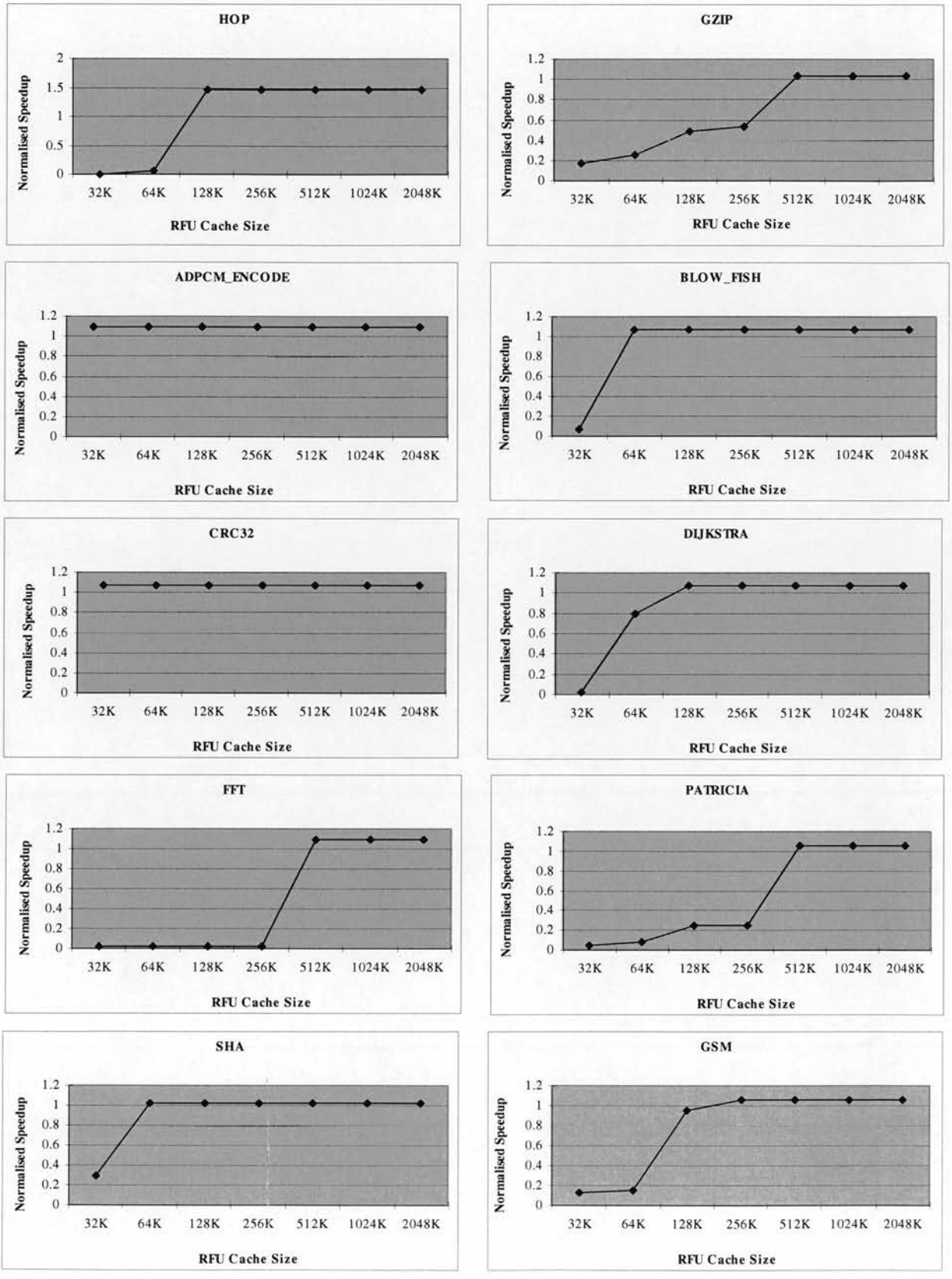


Figure 6-9: Normalised speedup of MAPS+RFU architecture for different cache sizes

Furthermore, the performance of MAPS+RFU is evaluated by comparing the MAPS+RFU performance with the synchronous clock gated MIPS baseline and the asynchronous MAPS architecture with ALUs ranging from one to four. The average performance improvements are around 49.2%, 10.5%, 5.4%, 4.4% and 4.4% for MAPS+RFU compared to the synchronous clock-gated MIPS, and MAPS with 1, 2, 3, and 4 ALUs respectively. By taking away the hand-coded optimised HOP benchmark, the average performance improvements are reduced to 43.8%, 6.4%, 2.4%, 1.7% and 1.7% for MAPS+RFU against synchronous clock-gated MIPS, and MAPS with 1, 2, 3, and 4 ALUs respectively. In the case of the SHA and GSM benchmarks, the MAPS+RFU architecture can not surpass the MAPS with two ALUs version. The reasons for the RFU performing badly are mainly because of the complex control paths in these applications, frequent memory accesses, and reconfiguration overhead.

These results show that MAPS+RFU achieves performance similar to a super-scalar MAPS architecture with multiple ALUs in most of the un-optimised applications, but it also consumes more energy (explained in more detail in the next section). Therefore, un-optimised applications are not recommended on a MAPS+RFU architecture.

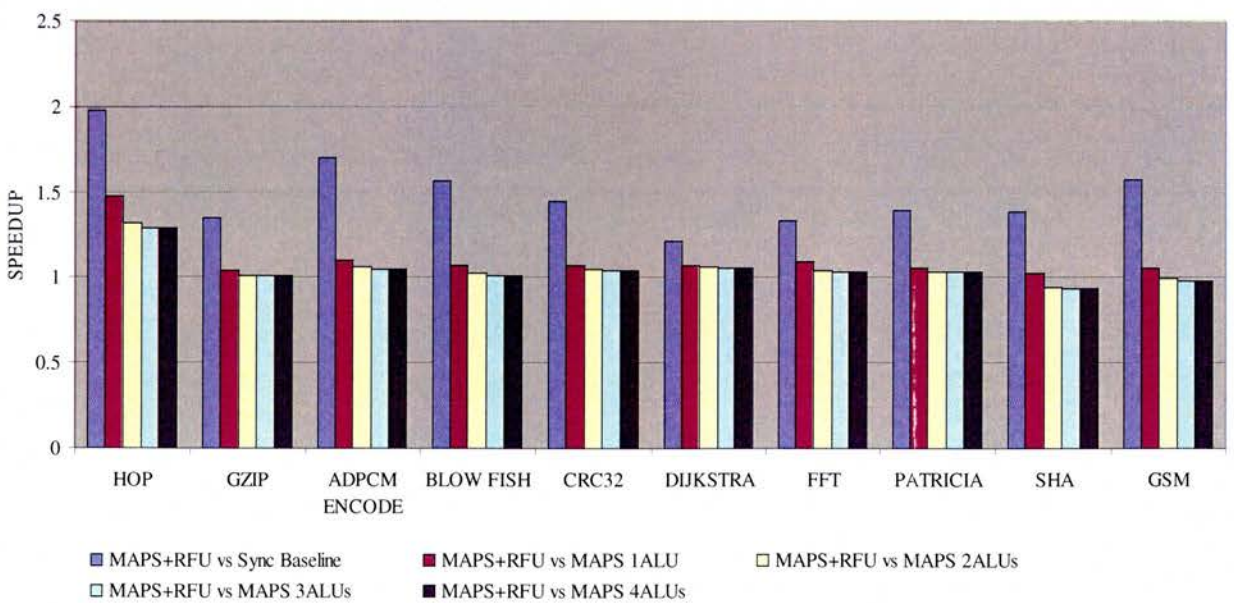


Figure 6-10: Speedup of the asynchronous MAPS+RFU architecture over the synchronous clock gate MIPS baseline and asynchronous MAPS.

6.5.4 Energy consumption

The flexibility of the RFU is provided by the programmable FPGA fabric. In the asynchronous MAPS+RFU architecture, the synchronous FPGA was integrated with CMs to support asynchronous handshaking protocols. The FPGAs trade additional silicon area and power consumption for flexibility. Routing tracks consume some amount of energy each time they switch. In addition, the programmable switch and cache memory for storing the configuration bitstreams increase the energy dissipation of FPGAs.

Detailed energy consumptions are displayed in Figure 6-11, Figure 6-12 and Figure 6-13. The RFU and the RFU cache contribute to large proportions of the total energy dissipation. Although small cache consumes less power, inadequate cache sizes significantly degrade the system performance, with a longer elapsed execution time, and the total energy consumption increase. However, when the cache size reaches an optimal size, the energy consumption will be reduced due to performance improvement.

From the observations on these energy consumptions figures, one can figure that different benchmarks have quite different requirements for RFU cache. As mentioned in the previous section, a configurable cache provides benefits by fine tuning cache parameters, and enabling/disabling cache banks. Obviously, the flexibility comes at a price, as it requires extra control logic and a heuristic learning algorithm [23][24] to teach the cache. But these algorithms are beyond the scope of this thesis, and should be studied in the future.

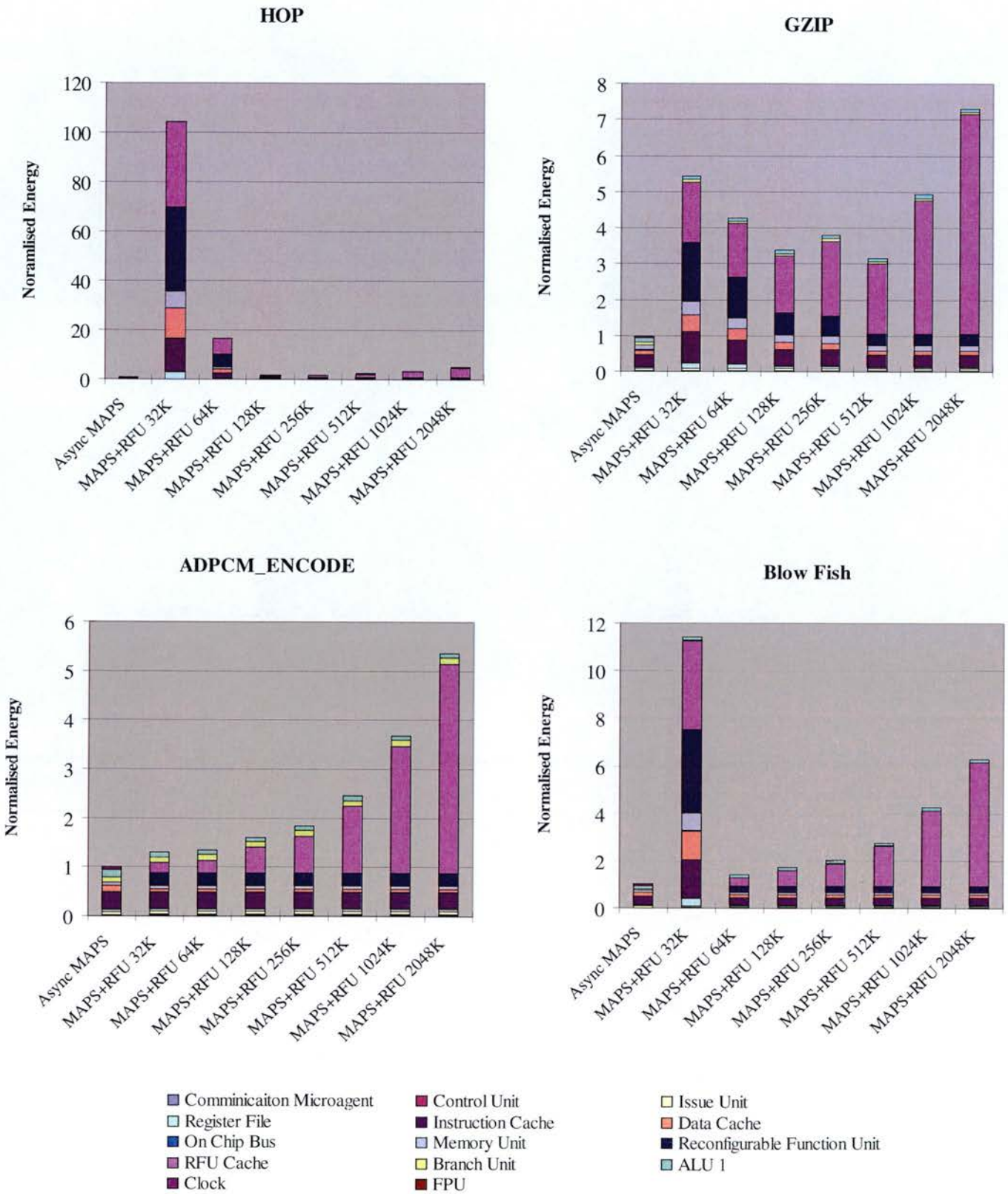


Figure 6-11: Energy dissipation breakdown by functional units in the asynchronous MAPS+RFU architecture for different cache sizes.

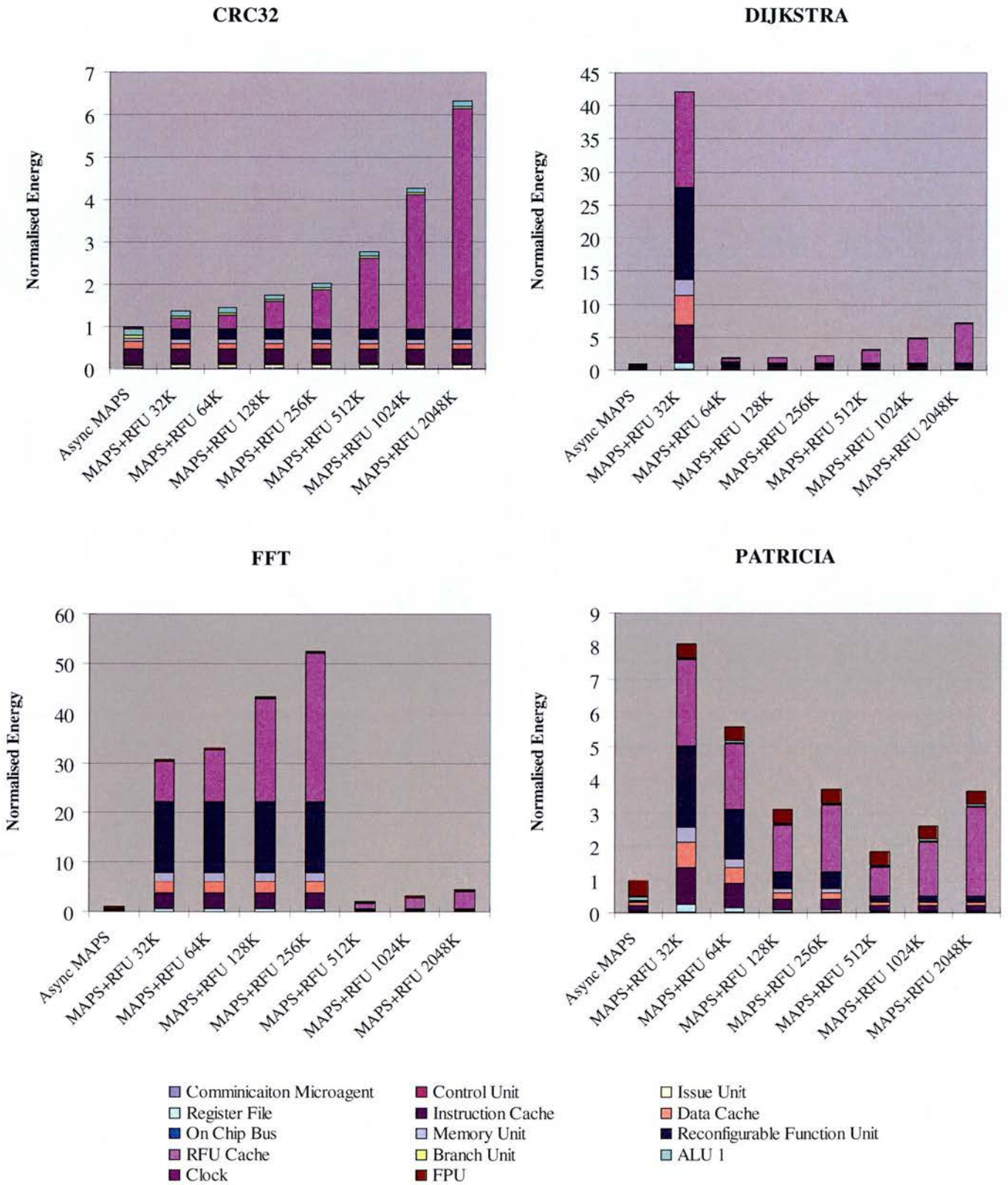


Figure 6-12: Energy dissipation breakdown by functional units in the asynchronous MAPS+RFU architecture for different cache sizes.

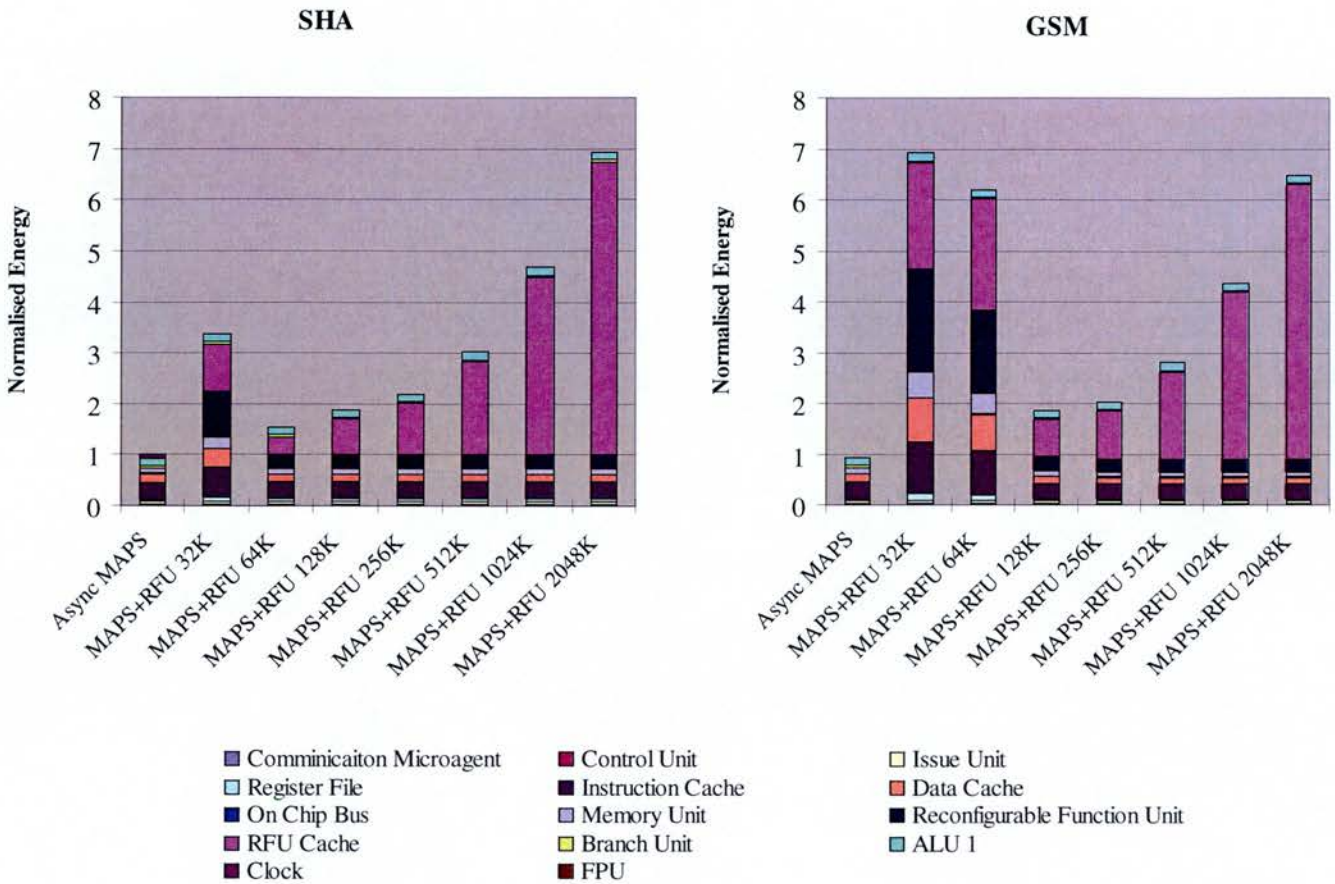
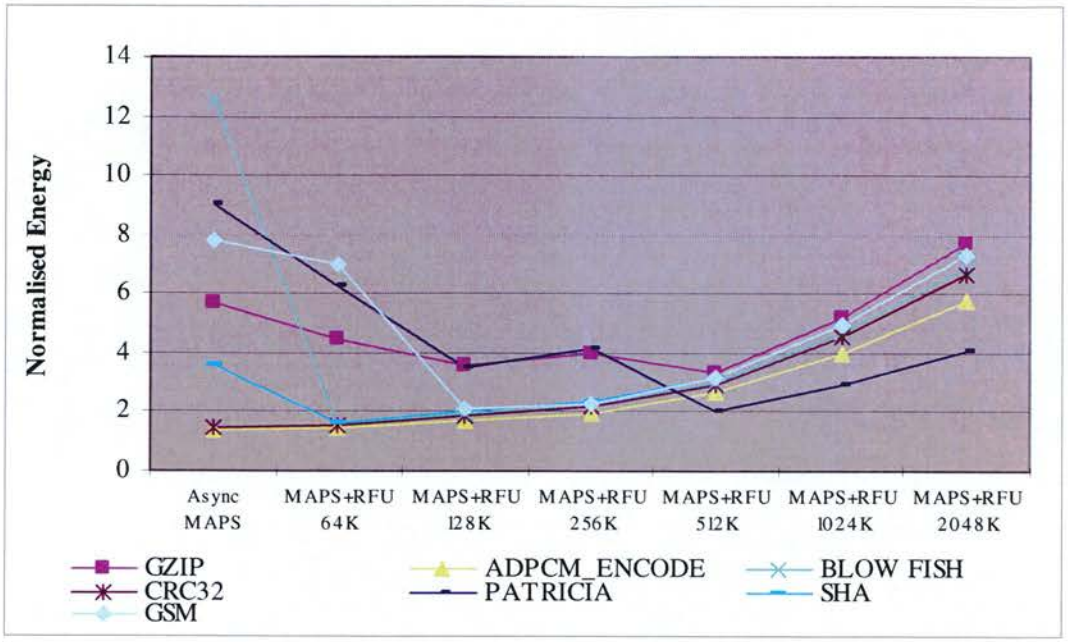
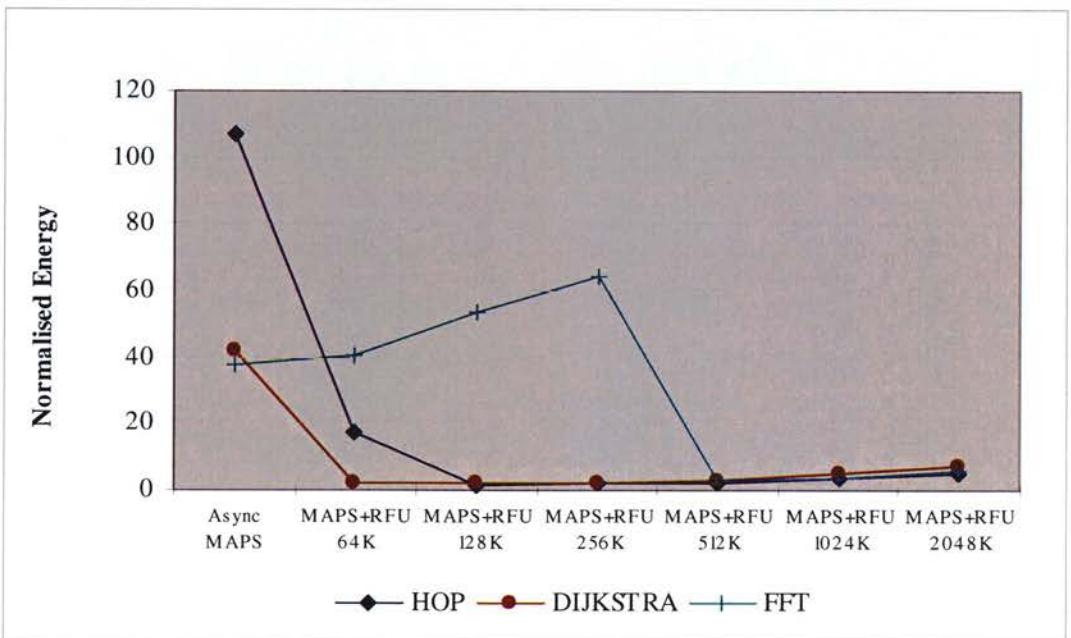


Figure 6-13: Energy dissipation breakdown by functional units in the asynchronous MAPS+RFU architecture for different cache sizes.

Figure 6-14 displays the normalised energy consumption increase for asynchronous MAPS+RFU normalised against asynchronous MAPS for different cache size. The energy consumption increase is on average 2283%, 836%, 737%, 880%, 287%, 436% and 638% with 32K, 64K, 128K, 256K, 512K, 1024K and 2048K respectively. These figures demonstrate that RFU is expensive to implement. Again, this will become factors for not introducing RFUs in embedded processors where energy consumption is critical.



(a) Benchmarks with lower variability



(b) Benchmarks with higher variability

Figure 6-14: Energy consumption for asynchronous MAPS+RFU with different cache sizes normalised against asynchronous MAPS architecture.

6.6 Multithreaded asynchronous MAPS

The aim of the multithreaded experiments was to explore the scalability of a multithreaded environment and the trade-off between performance and energy consumption by varying the number of active TPUs. The impact of speculative threads was also evaluated.

6.6.1 Benchmark analysis

In order to explore thread-level parallelism, the benchmarks were analysed with the thread partitioning compiler pass. As illustrated in Table 6-8, different numbers of multithreaded instructions are automatically extracted by the compiler. In the experiment, the ADPCM encoder provides the highest level of instruction distribution rate of multi-threaded instructions, which is about 18%. This is because the main body of the ADPCM encoder is a loop body which takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1, and does not involve complex control flows that are found in the GZIP or GSM benchmarks. The thread-level parallelism is also dependent on other factors, such as data dependencies between adjacent loop bodies. Detailed speedup results will be discussed in the next section.

Instruction Classes

Benchmarks	Load	Store	Arithmetic	Logic	Multiply Divide	Branch	Floating Point	Multi Thread	Other
HOP	1.38	0.76	46.46	23.94	6.29	14.19	0.00	6.91	0.06
GZIP	12.45	7.06	52.33	5.67	3.22	13.00	0.00	6.26	0.00
ADPCM_EN	3.61	0.62	50.45	4.56	2.40	20.37	0.00	17.98	0.00
BLOWFISH	11.94	4.57	59.11	8.18	3.50	9.96	0.00	2.75	0.00
CRC32	14.91	5.98	55.23	4.47	0.75	13.44	0.00	5.23	0.00
DIJKSTRA	20.83	5.88	51.18	0.39	5.92	14.00	0.00	1.79	0.00
FFT	10.04	5.78	46.70	3.33	14.80	5.42	12.39	1.26	0.28
PATRICIA	12.93	7.39	57.33	2.52	0.43	16.15	1.10	2.12	0.01
SHA	11.55	5.36	60.75	7.16	3.00	9.52	0.00	2.66	0.00
GSM	11.70	2.06	59.70	0.52	16.04	6.36	0.00	3.62	0.00
Average	11.14	4.54	53.92	6.08	5.63	12.24	1.35	5.06	0.04

Table 6-8 : Instruction distribution of benchmarks with multithreaded instruction

6.6.2 Performance and power efficiency of non-speculative maps

In this section, simulations have been executed on MAPS architecture with non-speculative threads with different numbers of TPU. The number of ALUs in each TPU was set to 2, as it was observed in Section 6.4 that larger numbers of ALU had limited improvement in speedup. All the benchmarks contain a certain number of loops, which were the main target for threads extracted by SHP compiler pass. No thread level control speculations and data predications were enabled.

The average speeds, power consumption and power efficiency are shown in Table 6-9. With more TPUs, the power consumption increases. The average power increase per TPU is 33 mW. Due to inefficient thread-level parallelism, the power efficiency is reduced from 2036 MIPS/W to 1154 MIPS/W. As a result, in order to fully exploit the potential of multithreaded MAPS+ architecture, more advanced threaded control and extraction techniques are needed.

	Speed	Power	Power efficiency
MAPS with 2 TPUs	234 MIPS	115 mW	2036 MIPS/W
MAPS with 3 TPUs	261 MIPS	145 mW	1798 MIPS/W
MAPS with 4 TPUs	282 MIPS	175 mW	1612 MIPS/W
MAPS with 5 TPUs	306 MIPS	210 mW	1459 MIPS/W
MAPS with 6 TPUs	322 MIPS	246 mW	1311 MIPS/W
MAPS with 7 TPUs	349 MIPS	280 mW	1244 MIPS/W
MAPS with 8 TPUs	363 MIPS	314 mW	1154 MIPS/W

Table 6-9 : Performance and power efficiency of non-speculative multithreaded MAPS architecture.

6.6.3 Contention of MAPS with non-speculative threads

As discussed in Chapter 3, the top level architecture of the multithreaded MAPS is similar to a Network-on-Chip processor. TPUs communicate with each other via switches. Each switch is connected to a local TPU, and has four directions to connect to neighbouring switches or the shared buses to access shared memory and the

scheduler. The topology of the TPUs and switches are arranged in a mesh network.

In order to evaluate how the switch buffers affect the contention in the MAPS on-chip network, we configured the switch with different buffer sizes and executed the ten benchmarks on the SPAMSIM2 simulator. The buffer sizes ranged from two words (eight bytes), four words, eight words, sixteen words, to thirty-two words. Each set of simulation doubled the buffer size of the previous set. And the TPUs numbers are set from two to eight.

A data packet is requested to be sent from one switch to the forwarding switch, and should the latter be busy to receive such a packet, then contention occurs and congestion is recorded in the simulator. Therefore, the congestion rate is the ratio of the total number of stalled packets to the total number of packets passed in the multithreaded MAPS on-chip network. The congestion rates of the MAPS architecture with different switch buffer sizes for the ten benchmarks are shown in Figure 6-15 with the numbers of TPU ranging from two to eight.

The simulation results show that congestion rates are reduced when the size of switch buffer is increased as would be expected. It is mainly due to the fact that when a switch buffer is large enough, data packets can be stored in the forwarding switch's local buffer in the event of contention, and no stall need occur. Therefore, contention is avoided with the resulting increase in performance. The average congestion rates are 9.3%, 8.65%, 7.46%, 5.10% and 0.59% for multithreaded MAPS with 2-, 4-, 8-, 16- and 32-word switch buffers, respectively. To improve the multithreaded MAPS performance and reduce the congestion rate, extra memory and power are required. This is a trade-off between memory, power and performance.

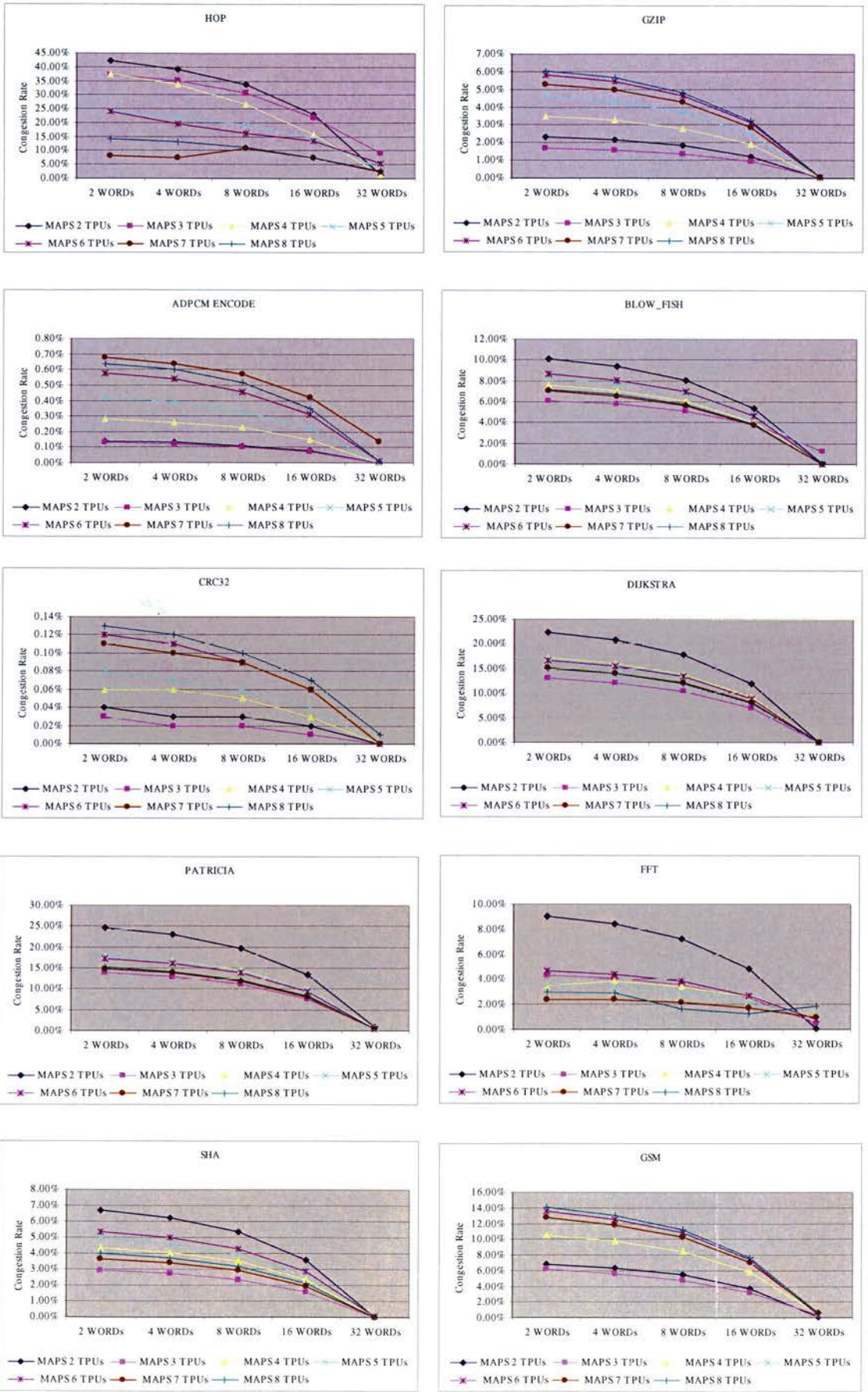


Figure 6-15: Congestion rates for multithread MAPS with different switch buffer sizes

The impact of increasing TPU numbers on the contention for multithreaded MAPS's on-chip network was also investigated. The same set of simulation results are displayed in Figure 6-16, with the TPU numbers along the X-axis and congestion rate along the Y-axis.

When the switch buffer was set to thirty-two words, most of the contention was avoided irrespective of the number of TPUs. However, when the switch buffer is smaller than or equal to sixteen words, different benchmarks behave quite differently. For example, the main trend in the congestion for HOP, Dijkstra, FFT, Patricia and SHA benchmarks is a fall or maintenance of a certain level when TPU numbers are increased. The congestion rates for the other benchmarks rise when TPU numbers are increased.

From an examination of the threaded source codes of different benchmarks, this behaviour is caused by the overhead due to inter-thread communication and shared memory accesses. The main threads generated in the HOP, Dijkstra, FFT, Patricia and SHA are from the outer loops of the applications. The inter-TPU communications occur when data is required to be passed from the current iteration (source TPU) to the next iteration (destination TPU) in the loop body. This benchmark contains weak data dependency between different loop bodies and the amount of inter-TPU communications is small. The contentions in these benchmarks are caused by shared memory access. Therefore, more communications channels are provided to access the shared memory when the TPU number is increased, and less contention occur.

On the other hand, the GZIP, ADPCM_ENCODE, CRC32 and GSM benchmarks are data encoding applications. For example, in the GZIP application, each iteration needs to search a matching string in the source data based on the previous iteration's search, and placed in the hash table, which results in large amounts of inter-TPU communication. As a result, when TPU numbers are increased, the total amount of inter-TPU communication also increases, and the probably of contention is greater.

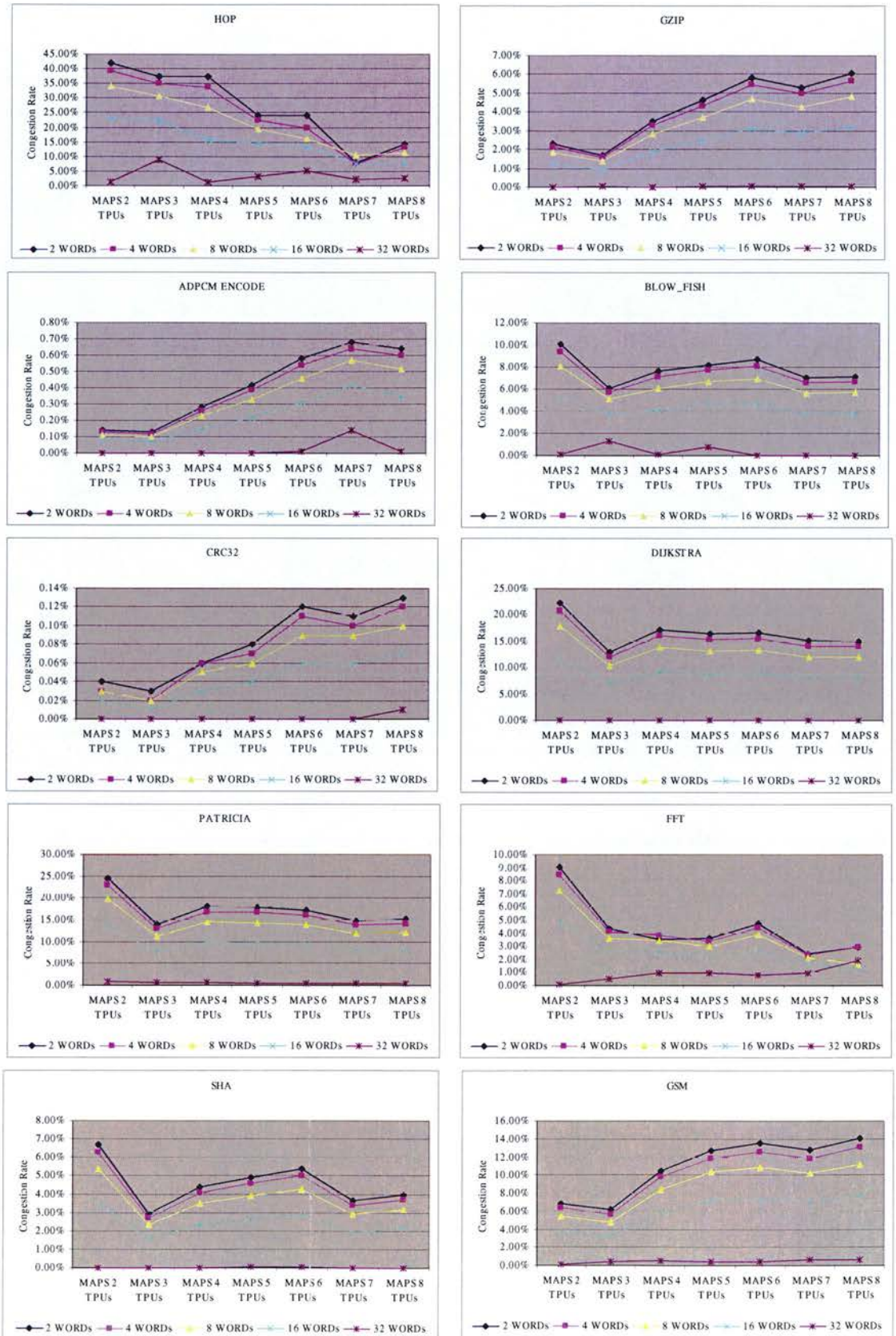


Figure 6-16: Congestion rates of multithread MAPS with different TPU numbers

6.6.4 Speedup of MAPS with non-speculative threads

The performance of all ten benchmarks for the multithreaded MAPS architecture with non-speculative threads was compared to the sequential versions on single-threaded asynchronous MAPS architecture. The switch buffer size was set to thirty-two words, which is the optimised setting to avoid contention. Normalised speedups are displayed in Figure 6-17.

Only half of the multithreaded versions of the benchmarks deliver speedups over the sequential programs. The HOP benchmark offers almost linear improvement when TPU numbers are increased. This is because the main body in HOP benchmark contains no loop-carried data and each thread size is reasonable large. The performances of benchmarks BLOWFISH and FFT saturate after 4 TPUs. The maximum performance speedups are 139.1% for BLOWFISH, and 17.5% for FFT respectively. GZIP and GSM gain a small improvement in the multithreaded version, the maximum performance speedups are 0.75% for GZIP and 3% for GSM. On the other hand, five of the benchmarks were far worse in the multithreaded cases. The performance reductions are 13.6% for ADPCM_ENCODE, 7.6% for CRC32, 11.0% for Dijkstra, 5.6% for Patricia, and 3.5% for SHA.

The simulation results show that data dependencies in the embedded applications are hazards for improving performance in multithreaded applications. Without mechanisms to resolve these dependencies either in compile- or run-time, a multithreaded MAPS architecture is not the best choice.

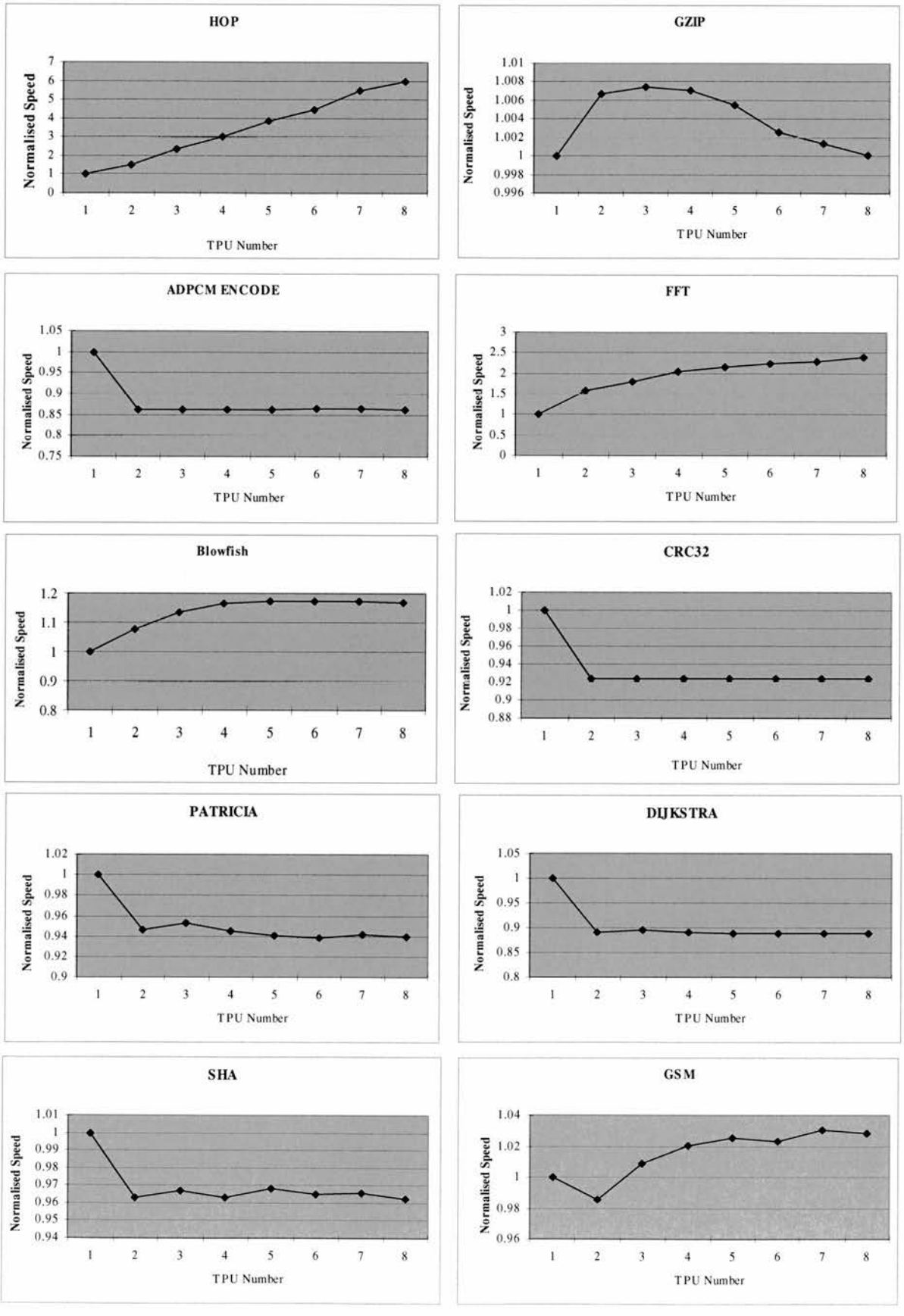


Figure 6-17: Normalised speedup for the multithreaded asynchronous MAPS architecture without speculative threads

The main reason for the performance reduction is inter-thread data dependency resulting in TPU's having to stall waiting for the arrival of dependent values. Extra overhead for thread context switching also slows down the overall performance. Figure 6-18 illustrates the control flow and inter-thread data dependency for the ADPCM_ENCODE benchmark multithreaded program.

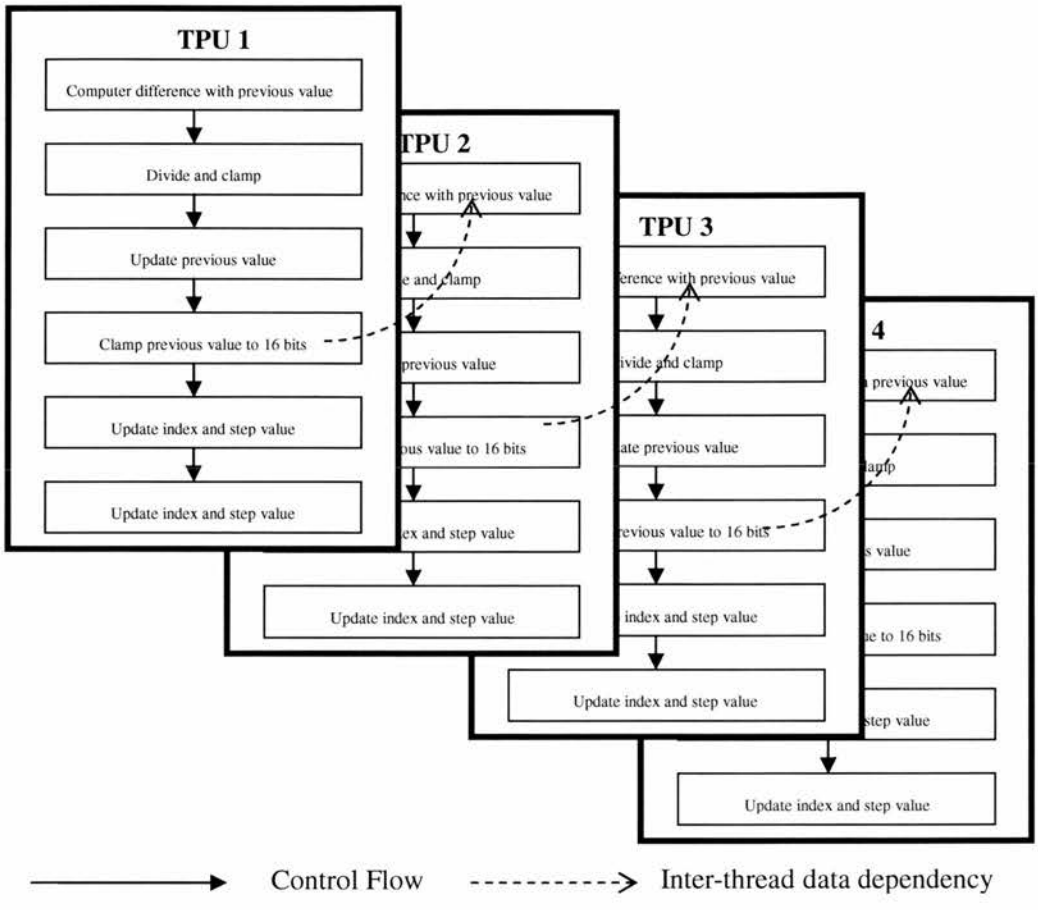


Figure 6-18: Control flow and data dependencies of ADPCM_ENCODE benchmark on multithreaded MAPS with 4 TPUs.

Figure 6-19, Figure 6-20, and Figure 6-21 provide normalised execution times (broken down into five parts), evaluation times of normal (non-threaded) instructions, evaluation times of threaded instructions, context copying and loading times, and waiting times for synchronisation. The graphs confirm that the majority of benchmarks suffer data dependencies across different threads, in which synchronisation times dominate the total execution times. For example, in the case of

the GZIP benchmark, average synchronisation time for stalled TPUs is 47.9% of the total execution time; in the case of the ADPCM_ENCODE benchmark, the stall time due to inter-thread data dependency is 89.2%. As a result, compiler techniques and hardware architectures are required to extract more thread level parallelism from traditional sequential benchmark programs.

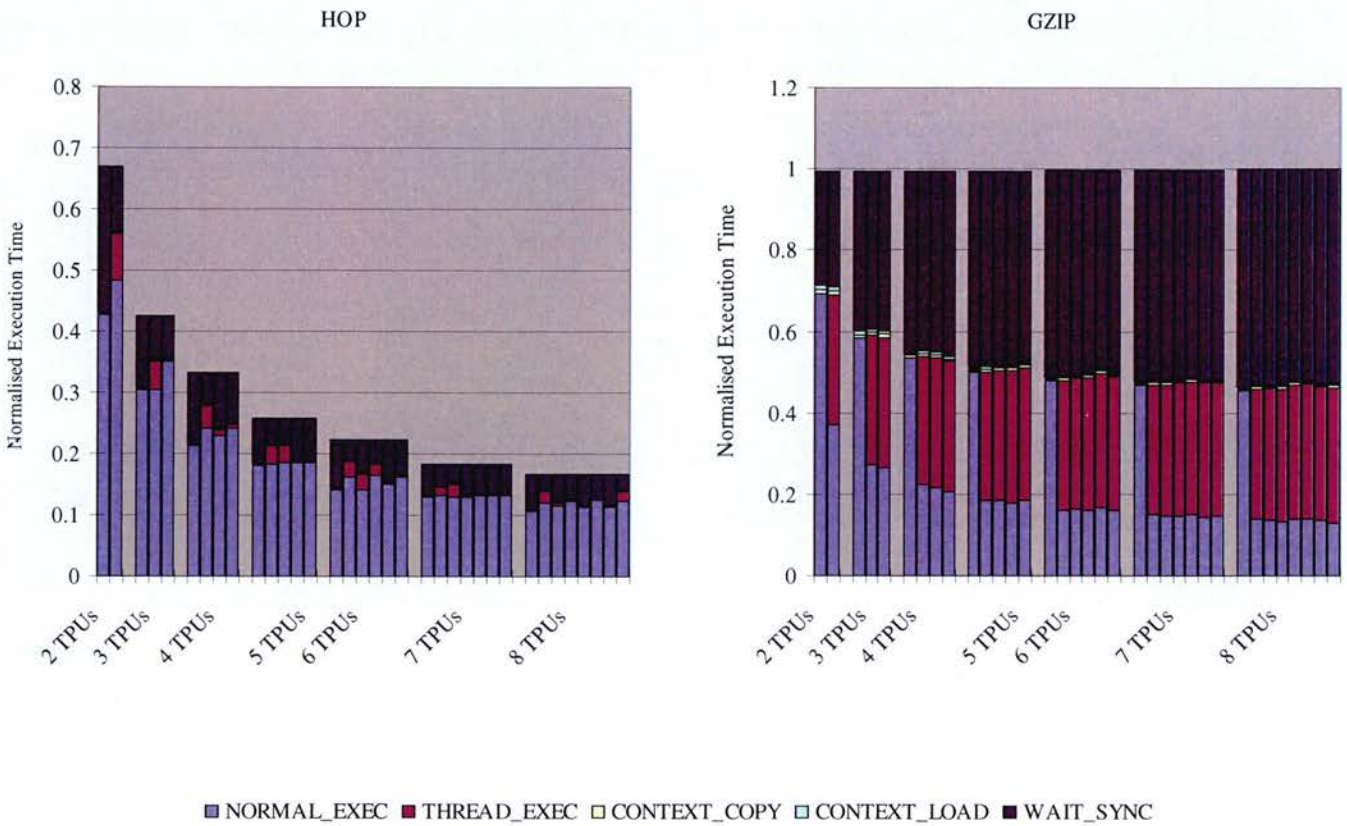


Figure 6-19: Execution time breakdown for TPUs without speculation.

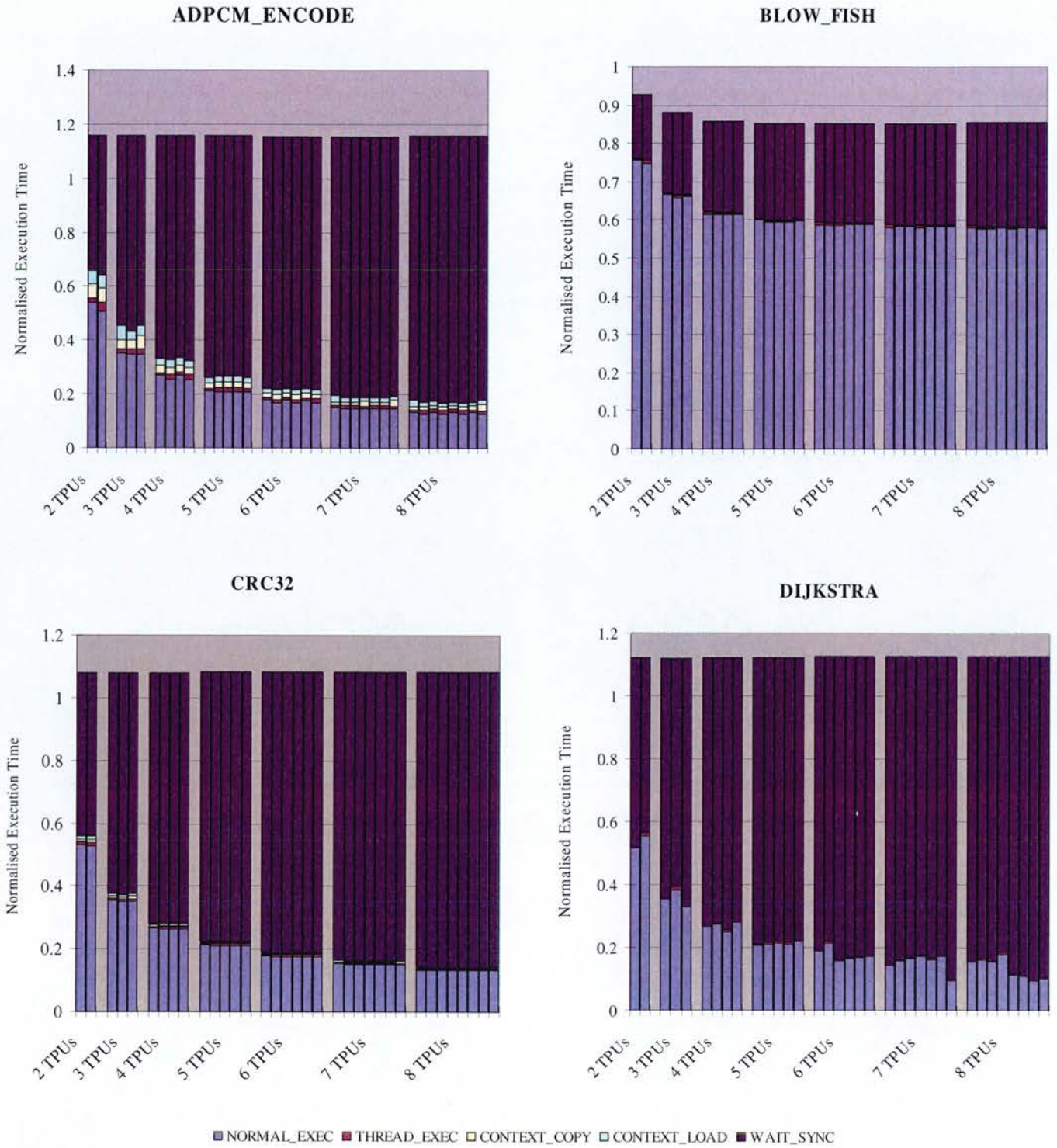


Figure 6-20: Execution time breakdown for TPUs without speculative threads.

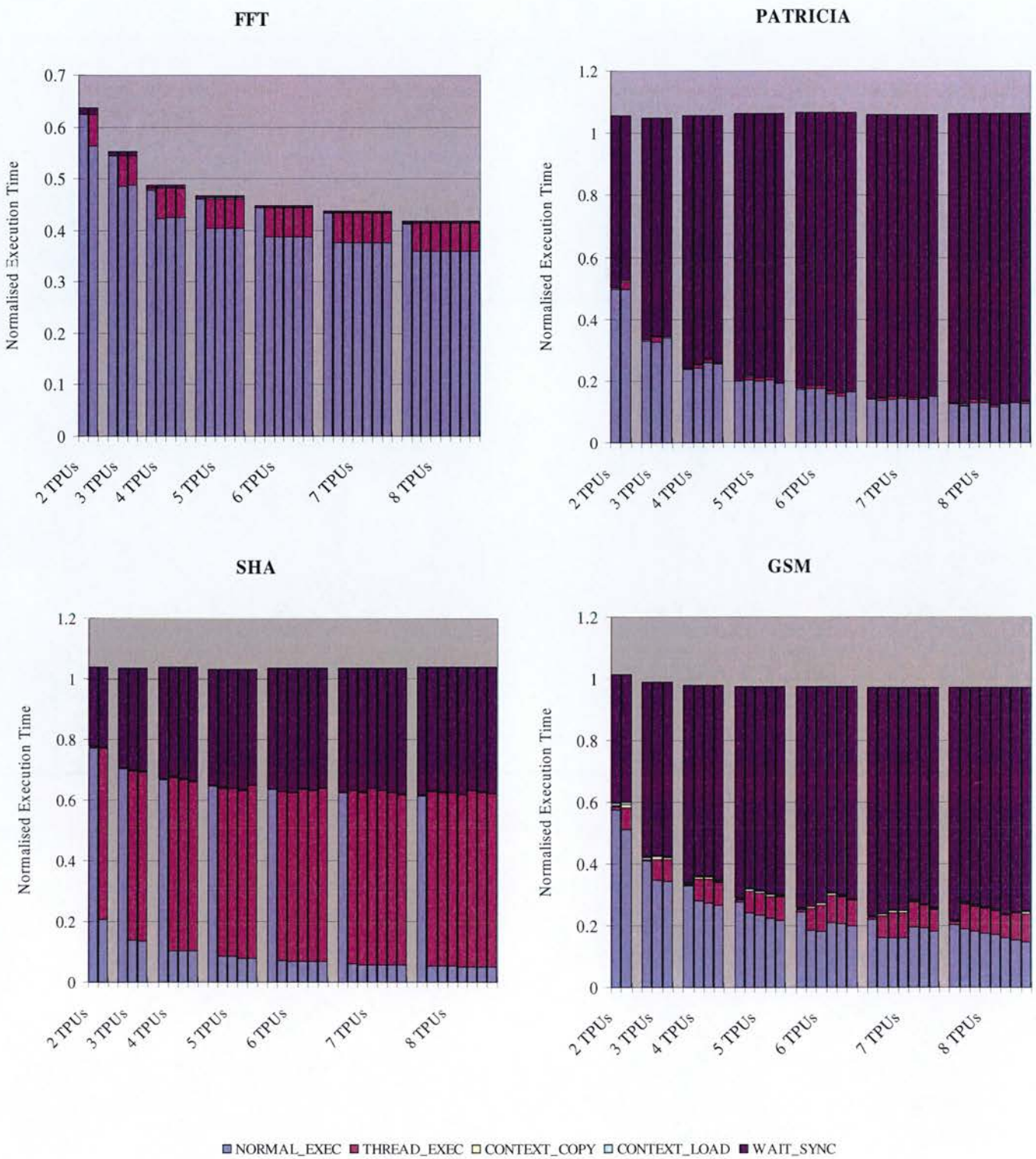


Figure 6-21: Execution time breakdown for TPUs without speculative threads.

6.6.5 Energy consumption of MAPS without speculative threads

The normalised energy consumptions of the multithreaded versions compared to the asynchronous single-threaded MAPS architecture is shown in Figure 6-22, which shows a trend of linear increase in energy for all the benchmarks. The two-TPU version shows the maximum rate of increase of 62.6%, due to resource duplication, resource sizing and extra overhead for context switching. For greater than two TPUs, the average overall energy increases by 25.7 %.

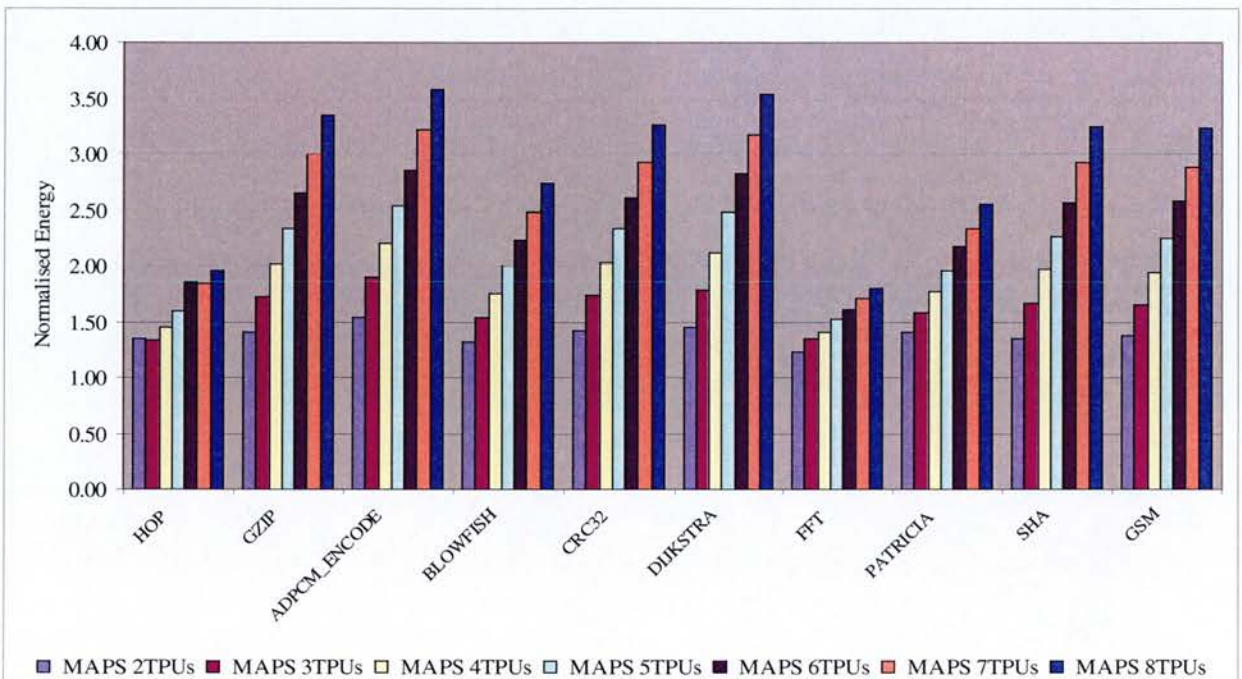


Figure 6-22: Normalised energy consumption of the multithreaded benchmarks without speculative threads

More detailed energy breakdown for each TPU and inter-thread communication are shown in Figure 6-23, Figure 6-24 and Figure 6-25. In most of benchmarks, thread executions is distributed fairly to different TPUs, and the energy contribution of each TPU are similar. However, for the GZIP and SHP benchmarks, TPU-1 contributes more to energy consumption than other TPUs. This is because the bodies of the

spawned threads are much smaller than the main thread bodies. For GZIP, the energy consumptions of TPU-1 are 56.7%, 40.2%, 32.1%, 26.4%, 22.4%, 19.5% and 17.1% for multithreaded MAPS with 2, 3, 4, 5, 6, 7, and 8 TPUs respectively. These figures match the execution time distributions presented in the previous section. One of the benefits of asynchronous systems is that no power is dissipated when TPUs are in an idle state. Therefore, asynchronous design can potentially help multithreaded architectures to reduce power consumption.

Inter-thread communication energy consumptions are mainly contributed by the MAPS on-chip buses and switches. The inter-thread communication energy increases as the number of TPU increase. For example, the energy dissipation in inter-thread communication for the HOP benchmark is 8.7%, 9.0%, 15.3%, 23.5%, 33.5%, 33.9%, and 37.2% for multithreaded MAPS, with 2, 3, 4, 5, 6, 7, and 8 TPUs, respectively.

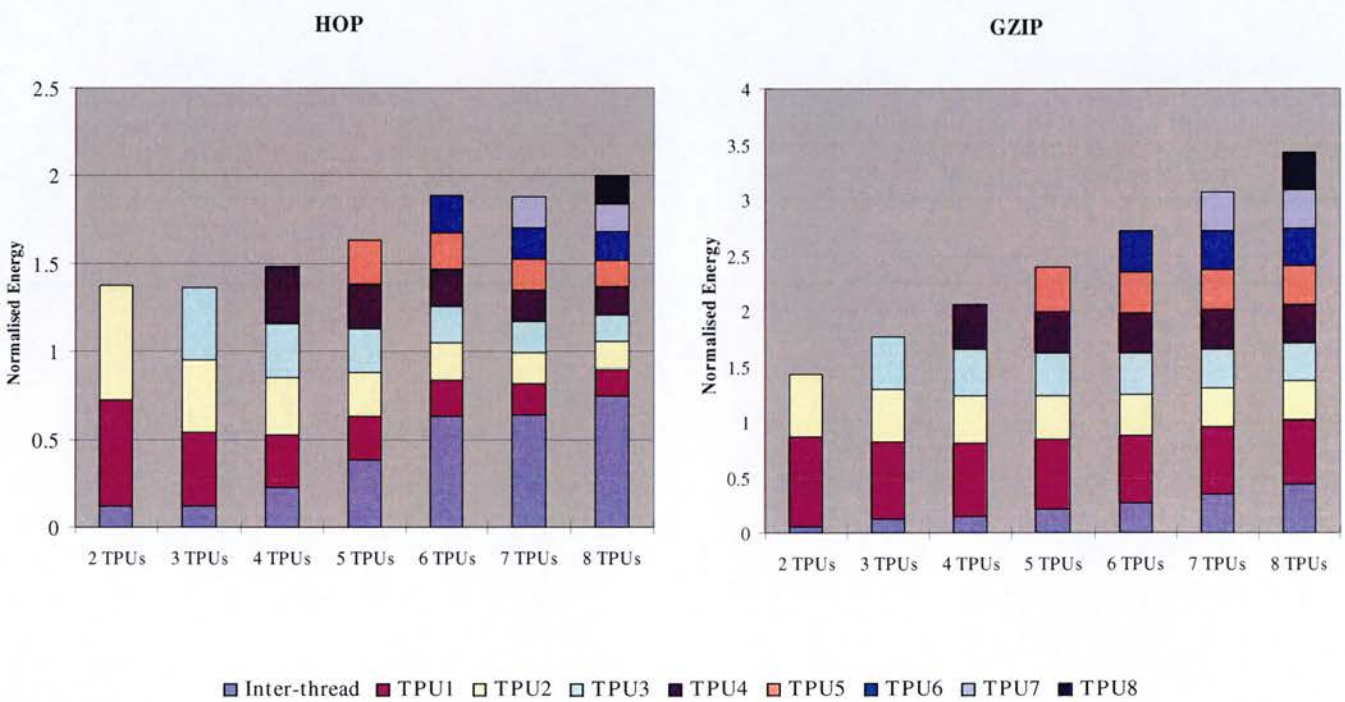


Figure 6-23: Normalised energy breakdown of the multithreaded benchmarks without speculative threads

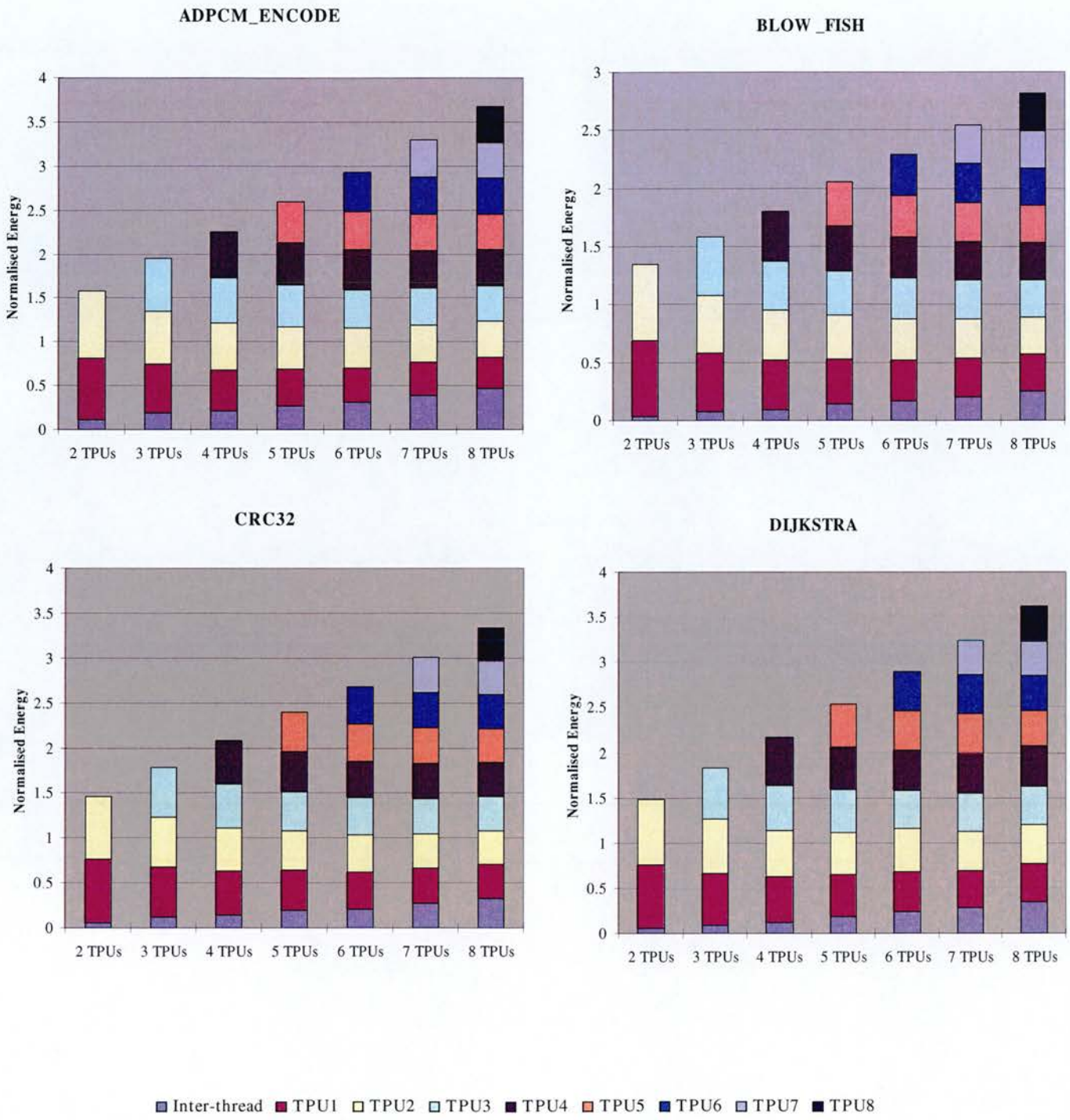


Figure 6-24: Normalised energy breakdown of the multithreaded benchmarks without speculative threads

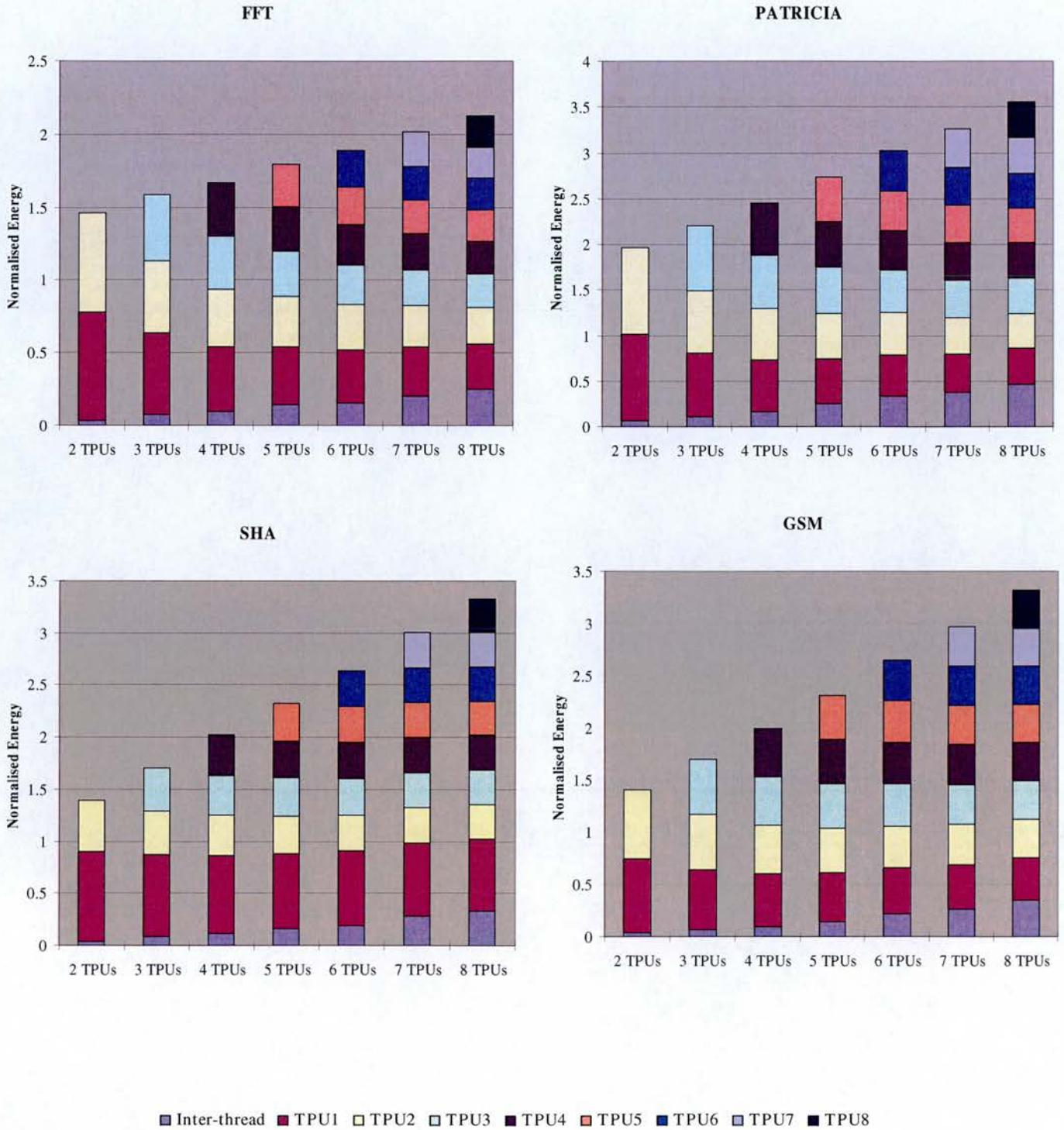


Figure 6-25: Normalised energy breakdown of the multithreaded benchmarks without speculative threads

6.6.6 Performance and power efficiency of speculative threads

In Section 6.6.4, the performance of the multithreaded asynchronous architecture shows limited speedup. Due to control and data dependencies, large portions of the execution time are wasted in synchronisation while executing the benchmarks non-speculatively. To fully explore the potential of the multithreaded MAPS architecture and extract parallelism from sequential programs, further optimisations are required. In Chapters 4 and 5, techniques for extracting speculative threads in control flows and data value predictions were implemented as the SHP pass in the MAPS+ compiler. The multithreaded benchmarks were executed on the MAPS architecture with corresponding hardware support.

As shown in Table 6-10 , the average speed of the two TPUs case is 338 MIPS, which has a 69% improvement over the asynchronous MAPS. As the number of TPU increases, the average speedup improves linearly. However, the speedup comes at the price of extra hardware and power consumption. More complicated hardware logic to control predictions and extra memories for storing predicated data are introduced, resulting in extra power being consumed. In the two-TPU case, the average power consumption is 161 mW, which is an increase at 108%. Due to linear increase in speed and power consumption, the power efficiency is maintained at a constant level with increase in the number of TPUs. The average power efficiency is 1874 MIPS/W.

	Speed	**Power**	**Power efficiency**
MAPS with 2 TPUs	338 MIPS	161 mW	2104 MIPS/W
MAPS with 3 TPUs	453 MIPS	228 mW	1988 MIPS/W
MAPS with 4 TPUs	555 MIPS	293 mW	1894 MIPS/W
MAPS with 5 TPUs	657 MIPS	354 mW	1857 MIPS/W
MAPS with 6 TPUs	737 MIPS	410 mW	1797 MIPS/W
MAPS with 7 TPUs	836 MIPS	471 mW	1775 MIPS/W
MAPS with 8 TPUs	911 MIPS	532 mW	1710 MIPS/W

Table 6-10 : Performance and power efficiency of the speculative multithreaded MAPS

6.6.7 Contention in MAPS with speculative threads

The impact of speculative thread execution on the multithreaded MAPS architecture is investigated in this section. Contention within the MAPS on-chip network is quantised, and congestion rates are measured on the multithreaded MAPS with, and without speculation. Similar to the simulation on the non-speculative MAPS, the speculative version runs on the SPAMSIM2 simulator by configuring the switch's buffer size from an init size of two words, and doubling up to thirty-two words. The results are shown in Figure 6-26.

The total amount of data packets transmitted on the network affects the congestion rate. As more packets are transmitted on the network within a period of time, greater is the probability of contention. Simulation results show that speculative threads can have both positive as well as negative impact on the contention within the network. For example, in the case of mis-speculation, the spawned thread codes are required to be rolled back and the correct piece of codes re-executing. If the mis-spawned codes contain memory access operations, then extra overhead is brought to the network, and increasing the chances of network contention. On the other hand, a thread speculated correctly has a positive impact on the network, with reduced number of data packets and resulting lower congestion.

In our simulation, HOP, GZIP, FFT and GSM have similar curves for both the non-speculative and speculative cases. For the ADPCM_ENCODE and CRC32 benchmarks, congestion rates are increased in the speculative case. BLOWFISH, Dijkstra, Patricia and SHA benchmarks have reduced congestion rates for the speculative multithreaded MAPS.

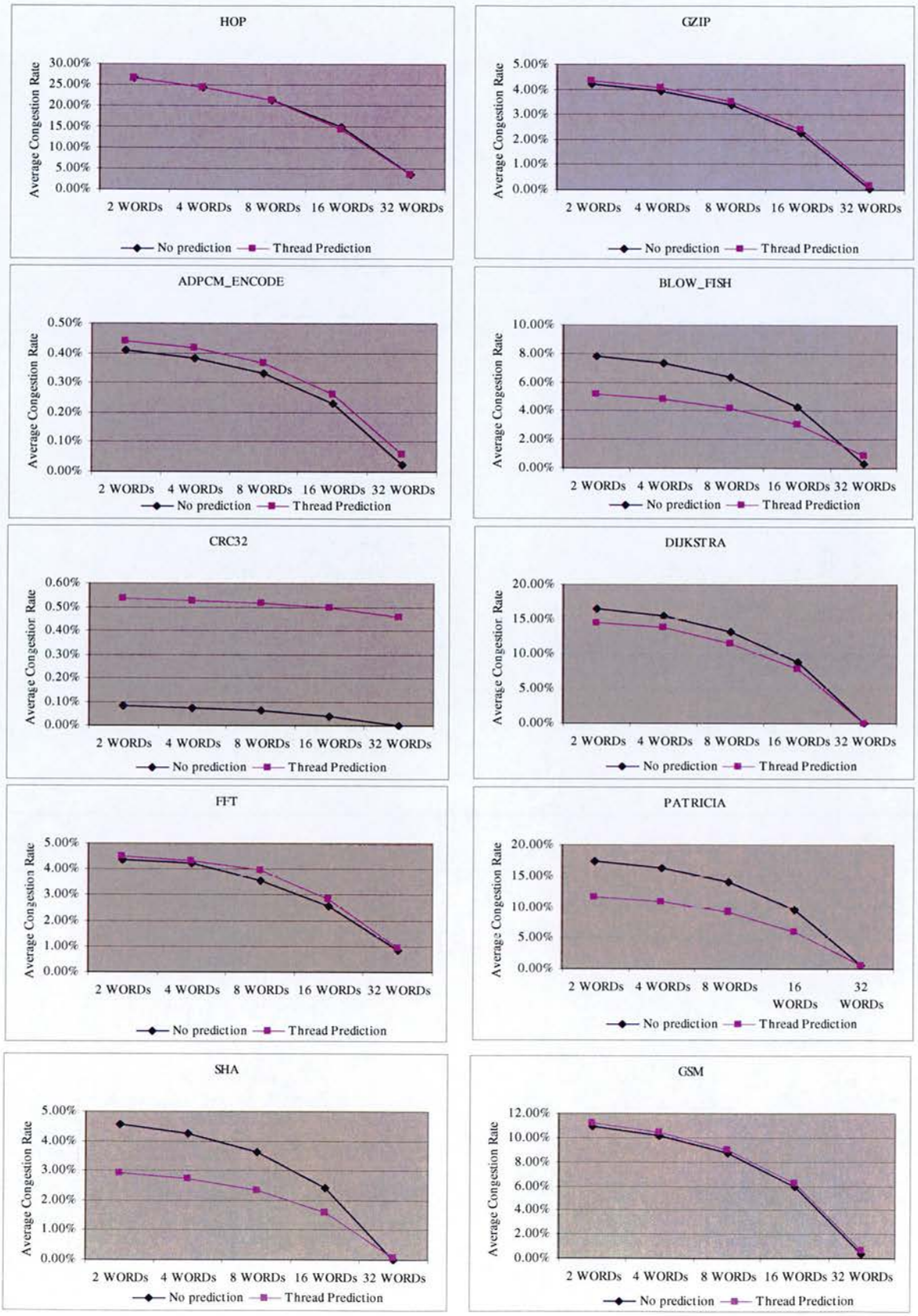


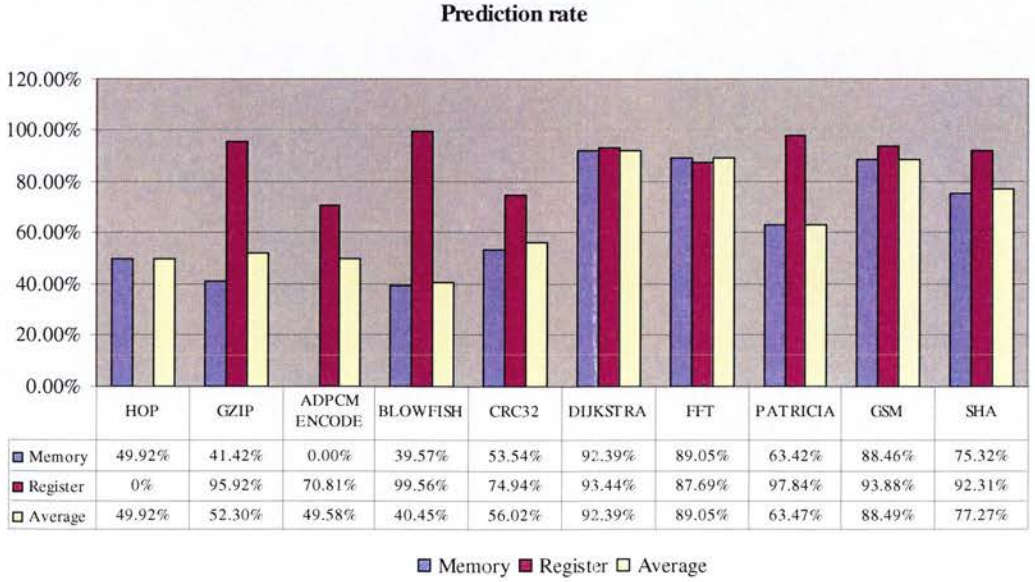
Figure 6-26: Comparison of congestion rates of multithreaded MAPS with non-speculative threads and speculative threads

6.6.8 Speedup of MAPS with speculative threads

The performance of all the ten benchmarks for the multithreaded MAPS architecture with speculative threads and data predictions were compared to the sequential version executing on asynchronous MAPS architecture.

First the performance of the value predictor was evaluated and the simulation results are illustrated in Figure 6-27. Two sets of value prediction behaviour were simulated. Value predictor handles the shared memory accesses by using a history buffer to record the data patterns. Data can also be passed between TPUs via registers. The register value predictor uses a history table similar to the memory value predictor. Two predictors share the same history buffer of size 32K.

The HOP benchmark has a zero prediction rate for the register values is because no data passes between TPUs. ADPCM_ENCODE has zero prediction rate on the memory value prediction because the generated thread body contains memory access of PCM audio data and the data patterns cannot easily be captured by the history table. In most cases, the register value prediction performs better, and the correctness of prediction is above 70% (excluding the HOP benchmark), varying between 39.6% and 92.4%. The input data also affects the behaviour of the value predictor.



	HOP	GZIP	ADPCM ENCODE	BLOWFISH	CRC32	DIJKSTRA	FFT	PATRICIA	GSM	SHA
Memory	49.92%	41.42%	0.00%	39.57%	53.54%	92.39%	89.05%	63.42%	88.46%	75.32%
Register	0%	95.92%	70.81%	99.56%	74.94%	93.44%	87.69%	97.84%	93.88%	92.31%
Average	49.92%	52.30%	49.58%	40.45%	56.02%	92.39%	89.05%	63.47%	88.49%	77.27%

Figure 6-27: Predication rates of value predictor

The normalised speedup for different benchmarks is illustrated in Figure 6-28. All the benchmarks demonstrate different levels of speedups in the speculative multithreaded MAPS over the single-threaded asynchronous MAPS.

The maximum speedup exhibited by GZIP is 43.73% in speculative multithreaded MAPS with eight TPUs, and 60.1% for SHA on eight-TPU MAPS. These two applications have reasonable high accuracy in value predictions, e.g. 41.4% for memory value prediction of GZIP, 95.9% for register value prediction of GZIP, 75.3% for memory prediction of SHA, and 92.3% for register valued prediction of SHA. However, the threaded program bodies are not long enough to the spread over different TPUs fairly, and the main thread on TPU-1 dominates the execution time. Therefore, the speedup achieved on these two benchmarks is quite limited. HOP has no data dependencies between threads, and therefore the performances on speculative multithreaded MAPS and non-speculative MAPS are quite similar. For ADPCM_ENCODE, FFT, Patricia, CRC32, Dijkstra and GSM, the accuracy of value predictors are quite different, with the value predictor for helping MAPS to break the data dependencies between TPUs.

The average speedup for the speculative multithreaded MAPS is 55.5%, 109.0%, 157.4%, 204.2%, 240.4%, 286.1% and 320.40% for 2-, 3-, 4-, 5-, 6-, 7-, and 8-TPU versions, respectively.

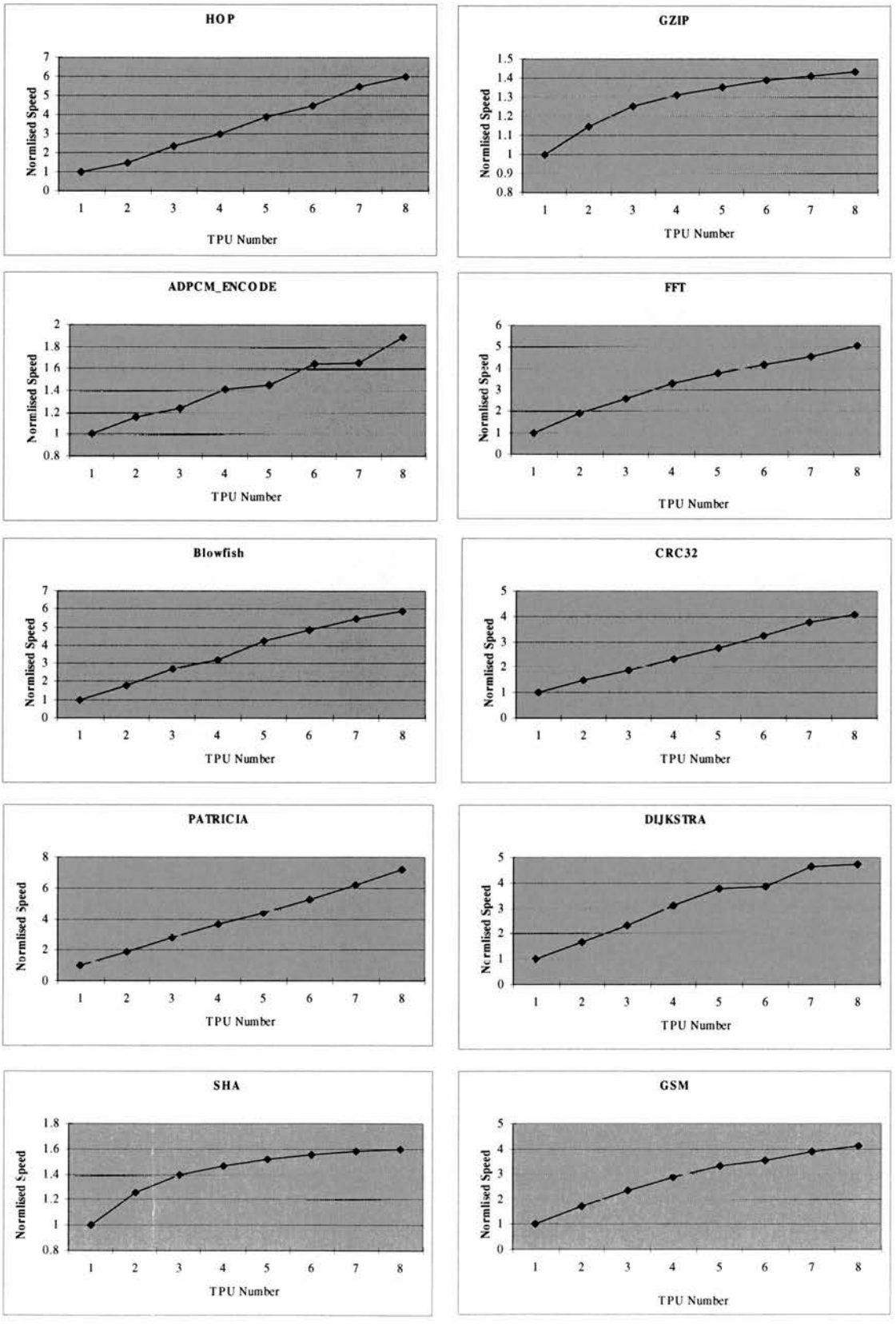


Figure 6-28: Normalised speedup for the multithreaded asynchronous MAPS architecture with speculative threads

The breakdown in the normalised execution times in Figure 6-29, Figure 6-30 and Figure 6-31 gives insights into the effectiveness of the MAPS architecture. The compiler-optimised multithreaded benchmarks perform better than the non-speculative cases on all the experiments. Our simulations demonstrate speculative threads and data value prediction techniques are important to extract parallelism from traditional sequential programs. In the case of a mis-speculation, the current context of the mis-speculated thread has to be squashed and its memory and register states will be discarded. However a correctly-speculated thread will commit its memory and register states to the parent threads and guarantee progress. A new category called SQUASH is introduced for timing information for handling squashed operations, including squashed thread codes, restarting TPU pipeline, registers' states, and reload cache buffers.

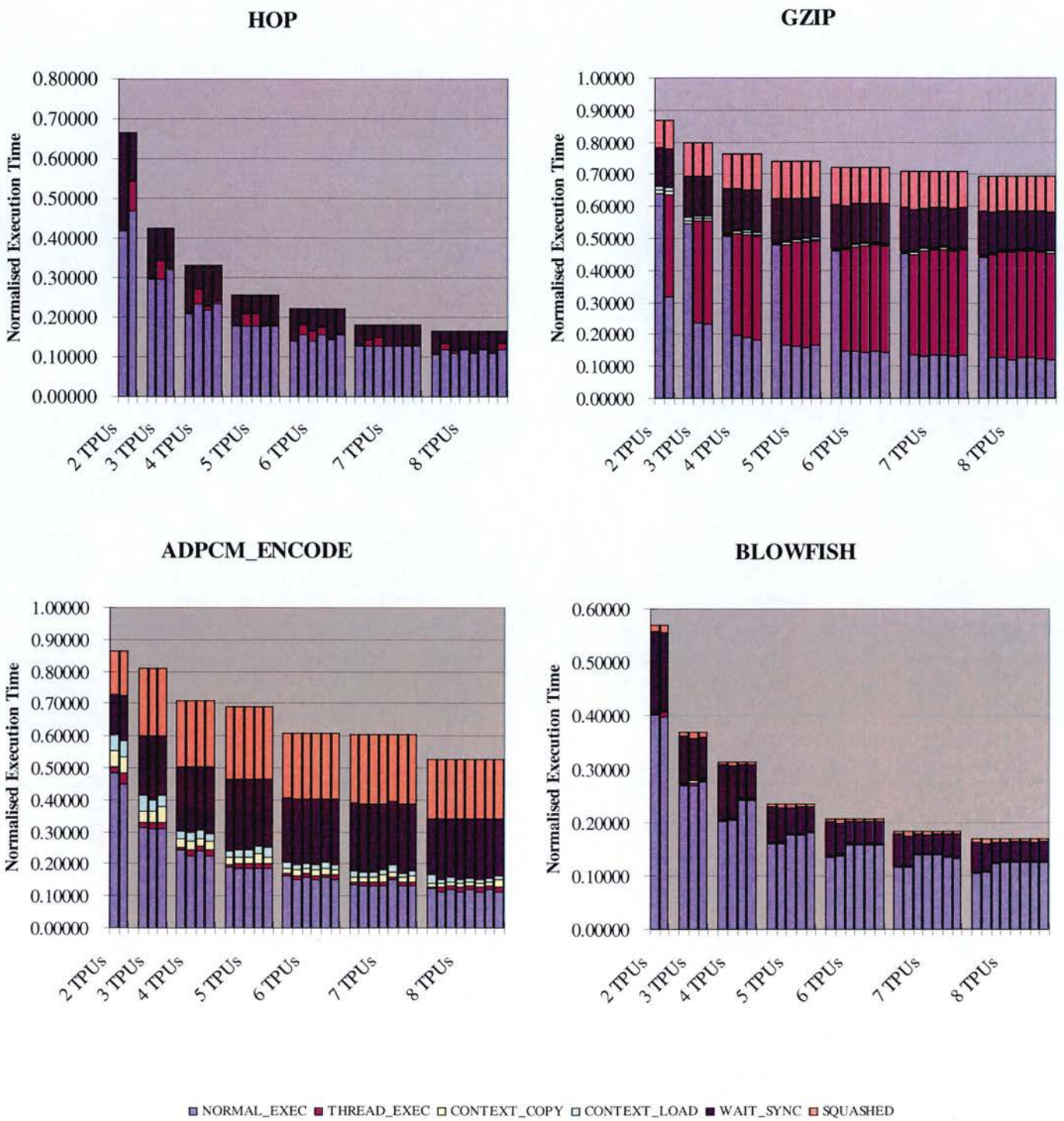


Figure 6-29: Execution time breakdown for TPUs with speculative threads.

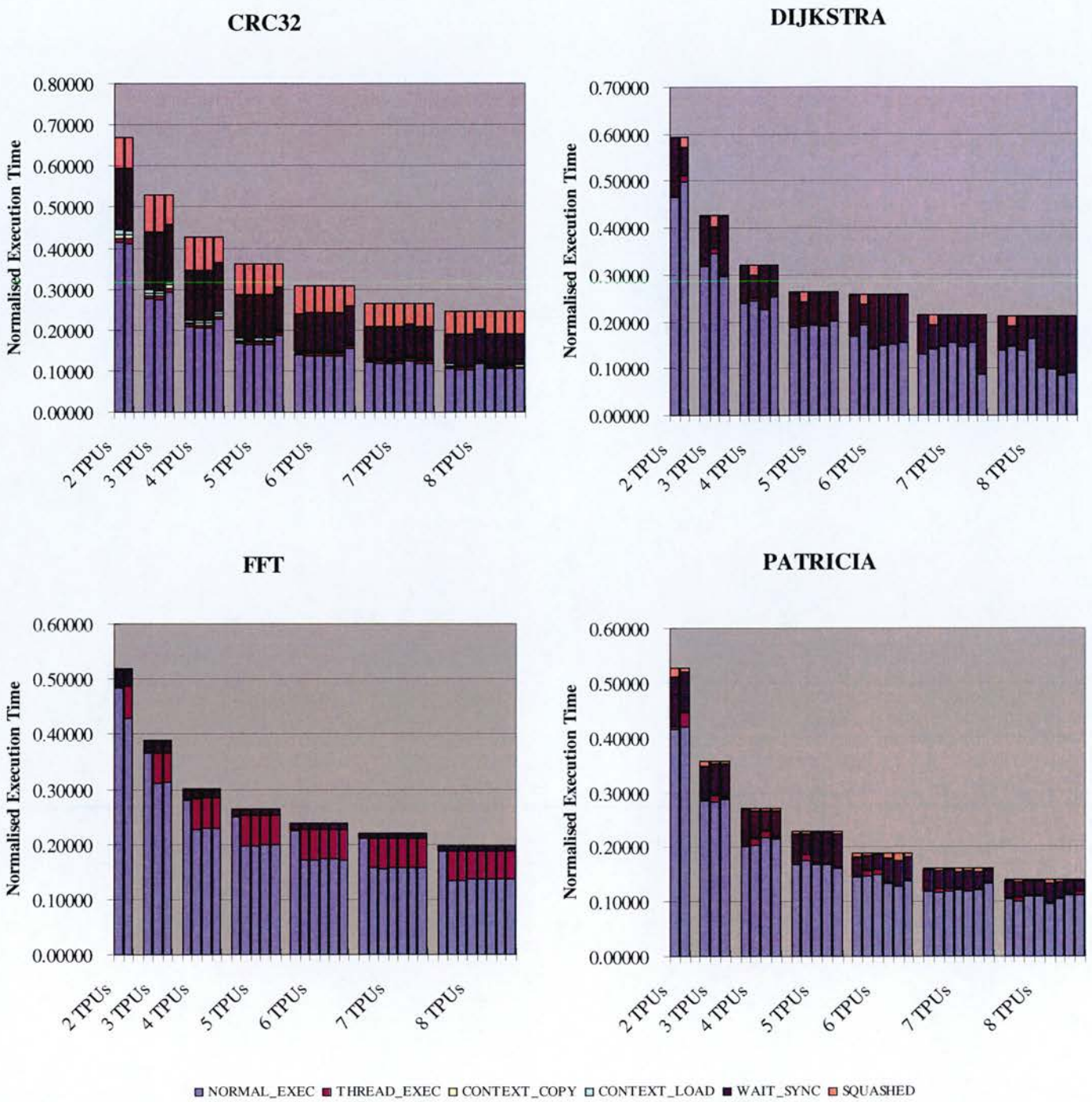


Figure 6-30: Execution time breakdown for TPUs with speculative threads.

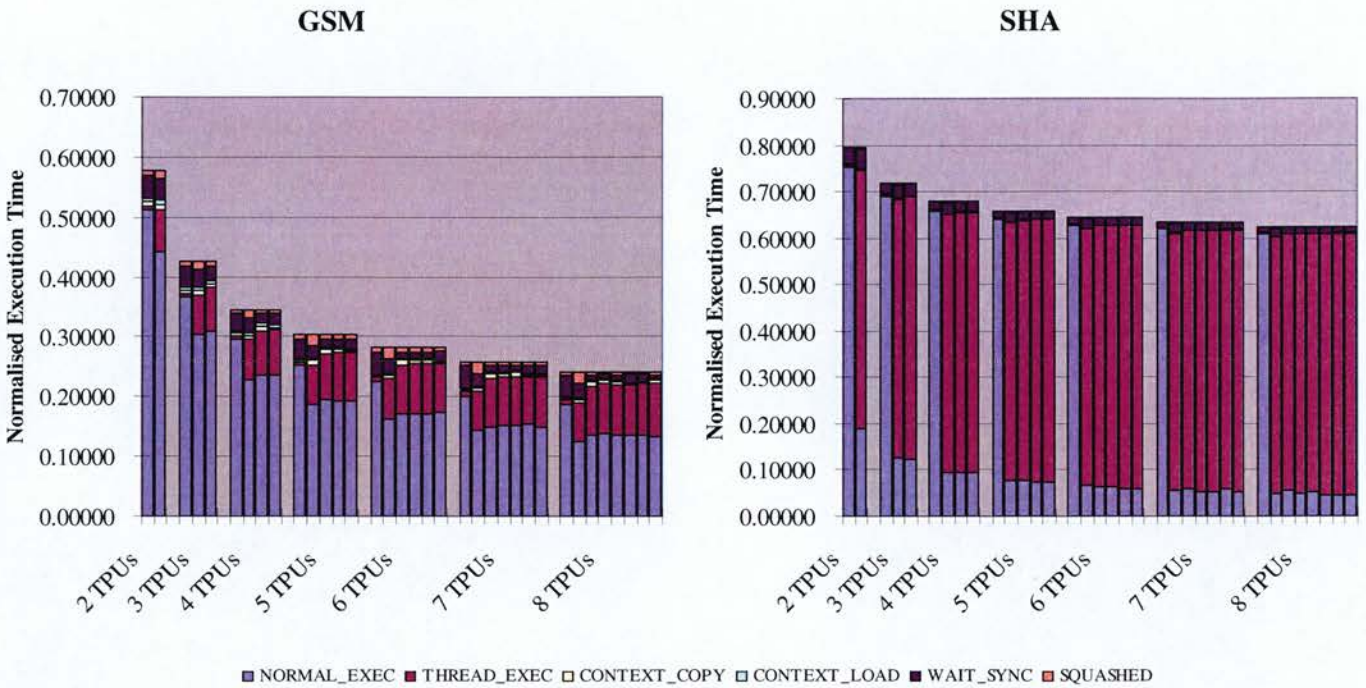


Figure 6-31: Execution time breakdown for TPUs with speculative threads.

6.6.9 Energy consumption of MAPS with speculative threads

The performance improvement in the multithreaded MAPS architecture with speculative threads comes at a price. The speculative techniques require hardware support for run-time speculation, longer time for run-time optimisation, which result in higher power consumption. The power and energy consumption of the predictors, e.g. speculative-thread predictor, data value predictor, memory space for holding history tables are modelled in the SPAMSIM2 simulator based on the models provides by CACTI toolset [27].

Normalised energy consumption of ten benchmarks on the speculative multithreaded MAPS against the single-threaded asynchronous MAPS are shown in Figure 6-32. The average increase in energy consumption is 40%, 50%, 59%, 71%, 82%, 92% and 103% for speculative multithreaded MAPS with 2, 3, 4, 5, 6, 7, and 8 TPUs, respectively. Energy consumption increases by 11% for each TPU.

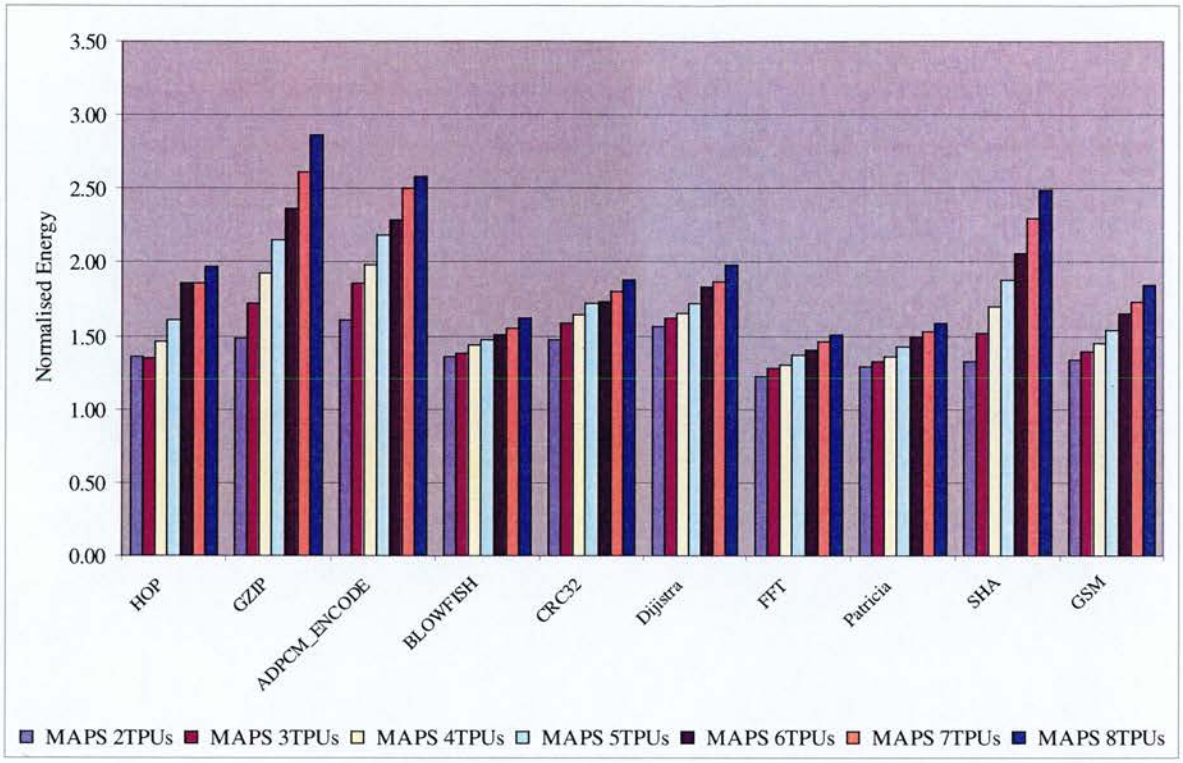


Figure 6-32: Normalised energy consumptions of the multithreaded benchmarks with speculative threads

Comparisons between normalised energy breakdown for the multithreaded benchmarks in non-speculative MAPS and speculative MAPS are shown in Figure 6-33, Figure 6-34 and Figure 6-35. Energy is spent on the value predictor, which includes that spent on read/write history table, searching history table for a specific memory address or register value and predicting the possible values. Total buffer size for memory and register value predictor is set to 32K. The GZIP benchmark spends 7.5% of the energy on value predictor and the SHA benchmark spends 4.8% energy on value predictor.

In the HOP benchmark, the non-speculative and speculative versions spend similar levels of energy. This is because no inter-thread data dependencies exist. The other nine benchmarks spend less energy in the three-TPU cases and above. The applications run faster, and although extra power is required for value prediction, it still benefits the system leading to lower energy consumption, overall.

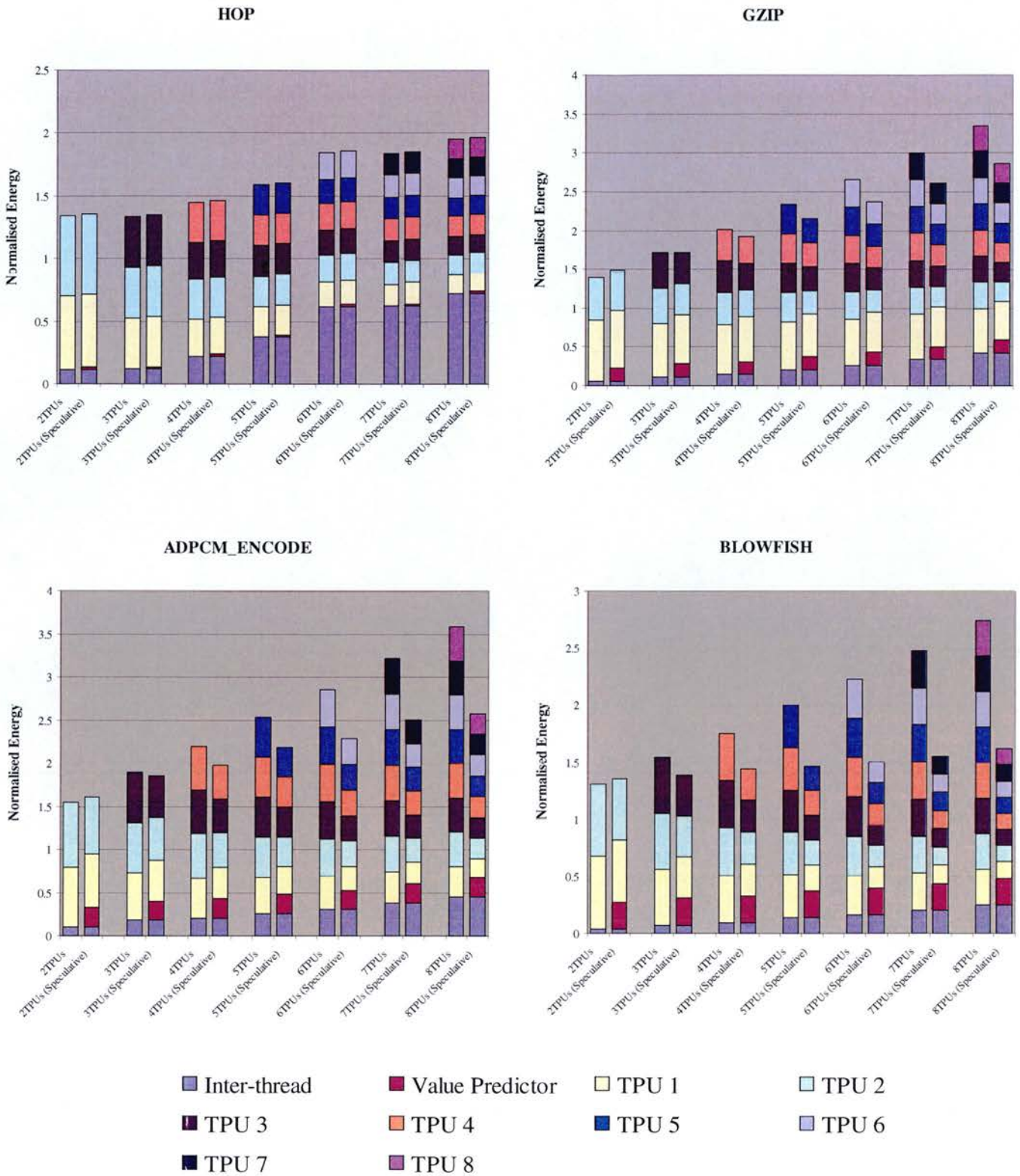


Figure 6-33: Comparison of normalised energy breakdowns of the multithreaded benchmarks on non-speculative MAPS and speculative MAPS.

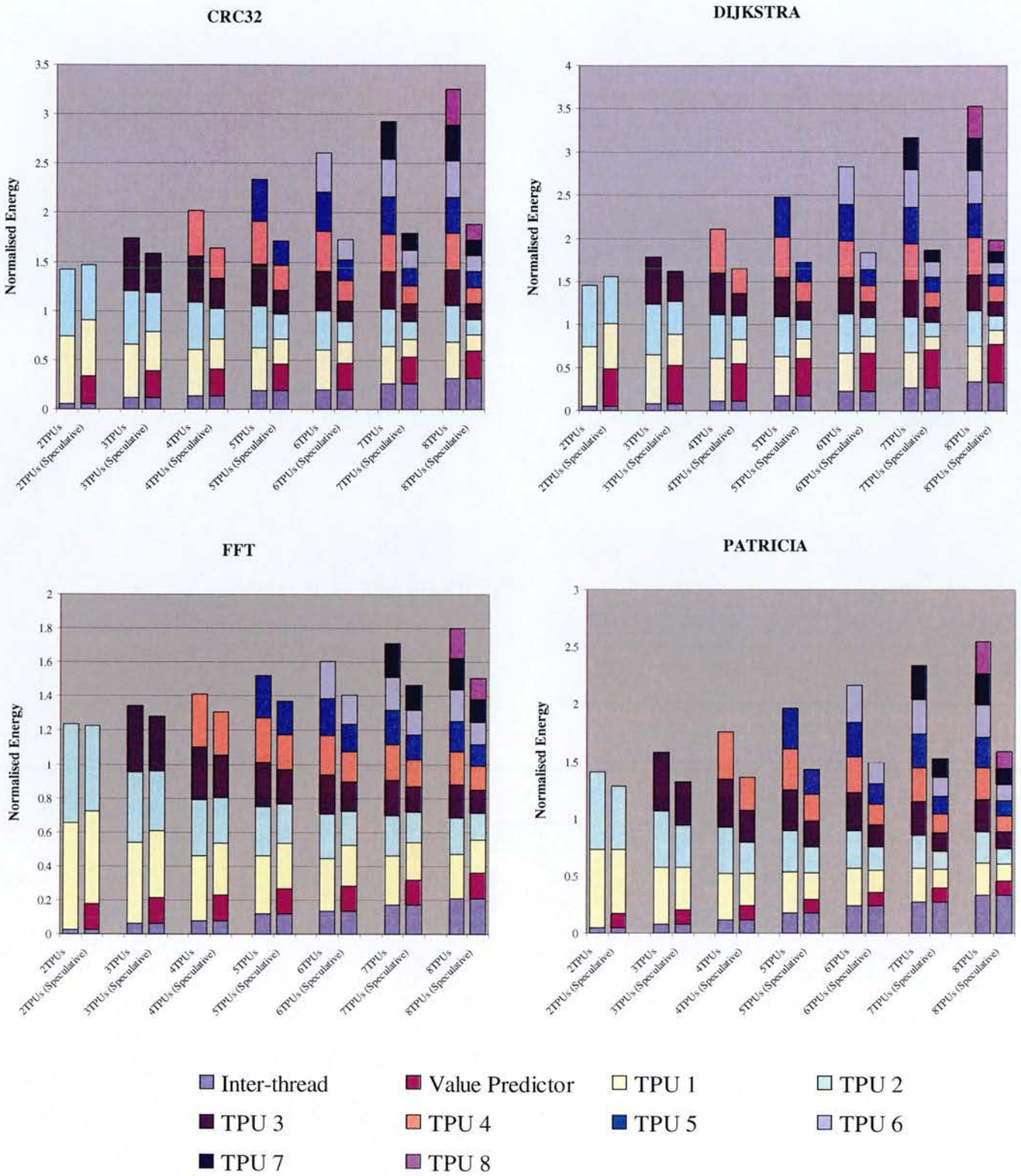


Figure 6-34: Comparison of normalised energy breakdowns of the multithreaded benchmarks on non-speculative MAPS and speculative MAPS.

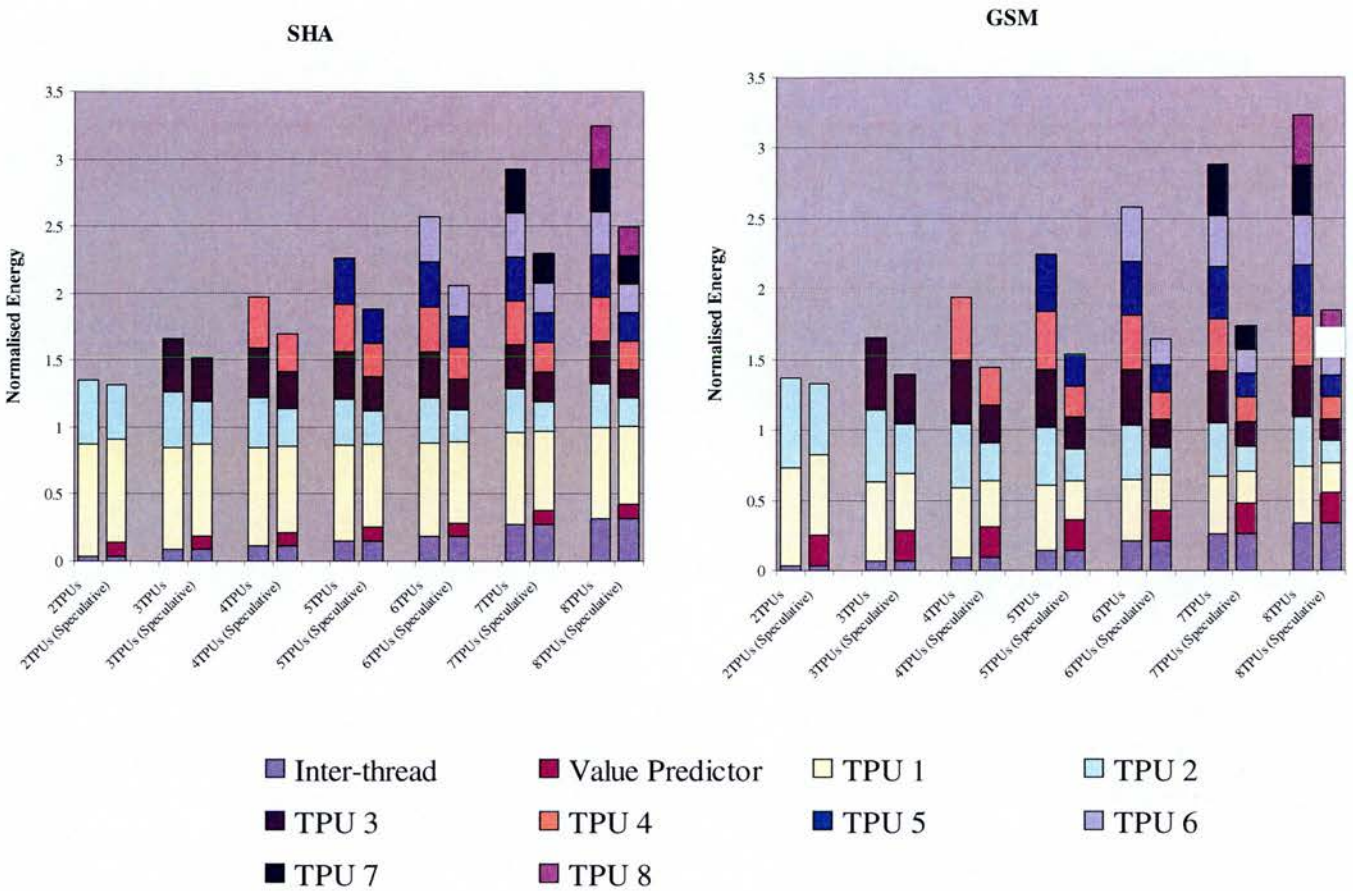


Figure 6-35: Comparison of normalised energy breakdowns of the multithreaded benchmarks on non-speculative MAPS and speculative MAPS.

6.7 Multithreaded asynchronous MAPS+ RFU

In this section, all the compiler optimisation passes were enabled to explore thread-level and data-level parallelism simultaneously.

6.7.1 Benchmark analysis

The instruction distribution of nine benchmark programs is listed in Table 6-11. Because of the large size of the GSM program and several unexpected bugs while compiling the GSM program with thread partitioning and the hardware/software partitioning pass, the GSM program has been omitted in our final set of benchmarks.

	Load	Store	Arithmetic	Logic	Multiply Divide	Branch	Floating Point	ROP	Multi Thread	Other
HOP	0.60	0.41	16.68	1.27	3.12	7.73	0.00	66.39	3.76	0.03
GZIP	10.44	6.30	45.46	4.41	2.27	12.00	0.00	13.33	5.78	0.00
ADPCM_EN	1.00	0.51	38.55	2.77	0.00	16.76	0.00	25.61	14.79	0.00
BLOWFISH	10.39	4.49	52.91	6.83	3.36	9.87	0.00	9.41	2.72	0.00
CRC32	12.40	5.12	47.47	2.91	0.73	13.15	0.00	13.10	5.11	0.00
DIJKSTRA	12.01	4.20	26.58	0.27	1.16	10.30	0.00	44.18	1.32	0.00
FFT	8.89	5.30	40.03	0.77	12.51	4.98	11.38	14.80	1.10	0.25
PATRICIA	11.23	7.28	49.39	0.72	0.37	16.02	1.09	11.78	2.10	0.01
SHA	10.18	5.17	53.75	3.48	2.76	9.19	0.00	12.91	2.57	0.00
Average	8.57	4.31	41.20	2.60	2.92	11.11	1.39	23.50	4.36	0.03

Table 6-11: Instruction distribution of benchmarks with HW/SW partition and multithreaded instruction

6.7.2 Performance and power efficiency

All the architectural modes described in the previous sections were integrated to maximise the speedup in asynchronous MAPS+RFU architecture. Two levels of optimisations were executed to extract thread-level parallelism for multiple TPUs and instruction-level parallelism supported by RFUs. With the same speculative-thread predictor and data-value predictor as implemented in the previous section, speculative threads and rollback operations were also supported.

The average speeds, power consumption and power efficiencies of the nine benchmarks on the multithreaded MAPS+RFU are listed in Table 6-12. The average speeds are 386 MIPS, 551 MIPS, 716 MIPS, 885 MIPS, 1056 MIPS, 1201 MIPS, and 1403 MIPS for speculative multithreaded MAPS+RFU with 2, 3, 4, 5, 6, 7 and 8 TPUs, respectively. The speedups against single-threaded asynchronous MAPS are 93%, 176%, 259%, 343%, 428%, 506% and 602% for 2, 3, 4, 5, 6, 7 and 8 TPUs, respectively, which provides the best performance among all the experiments.

However, more complex hardware logic and higher power consumption are the prices to the paid for performance gains. The average increase in power consumption compared to the asynchronous MAPS are 339%, 613%, 827%, 1045%, 1266%, 1466% and 1716% for 2-, 3-, 4-, 5-, 6-, 7-, and 8-TPU versions, respectively. The high power consumption of integrating reconfigurable functional unit brings great barrier

for introducing such design in embedded processor architecture for most the applications.

	Average speed	Power consumption	Power efficiency
MAPS+ RFU with 2 TPUs	247 MIPS	386 mW	640 MIPS/W
MAPS+ RFU with 3 TPUs	275 MIPS	551 mW	498 MIPS/W
MAPS+ RFU with 4 TPUs	296 MIPS	716 mW	414 MIPS/W
MAPS+ RFU with 5 TPUs	316 MIPS	885 mW	357 MIPS/W
MAPS+ RFU with 6 TPUs	329 MIPS	1056 mW	311 MIPS/W
MAPS+ RFU with 7 TPUs	321 MIPS	1210 mW	265 MIPS/W
MAPS+ RFU with 8 TPUs	384 MIPS	1403 mW	274 MIPS/W

Table 6-12: Performance and power efficiency of asynchronous multithreaded MAPS architecture plus RFU.

6.7.3 Contention

The contention in the on-chip network for multithreaded MAPS+RFU architecture are quantised by using the congestion rates, as summarised in Figure 6-38. These diagrams compare the congestion rates of the nine benchmarks in the speculative multithreaded MAPS, and the speculative multithreaded MAPS+RFU. Switch buffer sizes are set to two, four, eight, sixteen, and thirty-two words, respectively. Results shown in the diagram are average numbers of the different settings. In order to illustrate the impact on the network performance due to changes in TPU numbers, the average congestion rates are summarised for increasing TPU numbers. Most of the benchmarks have higher congestion rates executing on the multithreaded MAPS+RFU compared to the multithreaded MAPS. Extra network loadings are needed for ROP configuration fetching.

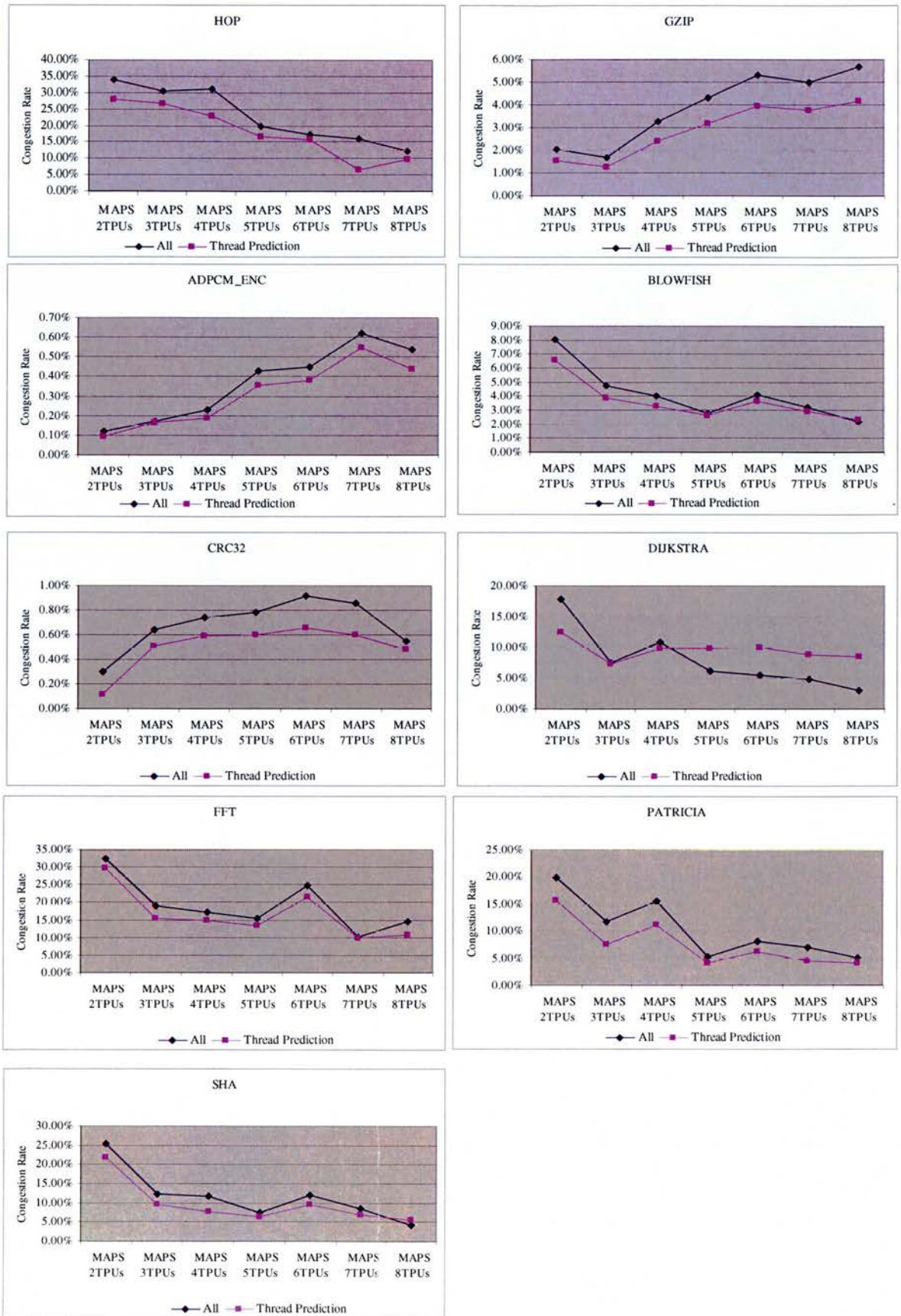


Figure 6-36: Comparison of congestion rates of the speculative multithreaded MAPS and the speculative multithreaded MAPS+RFU

6.7.4 Speedup

Individual speedup of each benchmark is analysed in this section. Due to the register-value passing and memory access patterns in speculative multithreaded MAPS+RFU which are similar to the multithreaded MAPS, the accuracies for value predictors on both architectures are quite similar. Details of the value prediction behaviour can be referred to Section 6.6.8.

Comparisons of performance for the benchmarks on multithreaded MAPS+RFU with different TPU number configurations are presented in Figure 6-37. Most of benchmarks perform better on the speculative multithreaded MAPS+RFU than the MAPS architecture. However, the improvement achieved is quite limited, when the extra hardware needed and power consumption overhead are also considered.

For example, the ADPCM_ENCODE benchmark speedup on the multithreaded MAPS+RFU over the single-threaded asynchronous MAPS are 28%, 36%, 56%, 60%, 82%, 85% and 109% on 2-, 3-, 4-, 5-, 6-, 7-, and 8-TPU versions, respectively. The performances are just slightly better than the multithreaded MAPS, e.g. 16%, 23%, 41%, 45%, 64%, 65% and 89% on 2, 3, 4, 5, 6, 7 and 8 TPUs, respectively. The actual improvement provided by the RFU operations is quite limited.

The FFT benchmark performs worse on the speculative multithreaded MAPS+, compared to the speculative multithreaded MAPS. The performance improvements by increasing TPU numbers are quite small. It is because the thread analyser and the software/hardware partitioning compiler searching for the hot spots on the same sequential code of a program, and parts of the threaded code are executed in the hardware blocks. Therefore, the threads generated on the multithreaded MAPS+RFU are less than in the case of the multithreaded MAPS architecture. Therefore, even with extra ROPs, the FFT performance on multithreaded MAPS+RFU is not as good as the one on multithreaded MAPS.

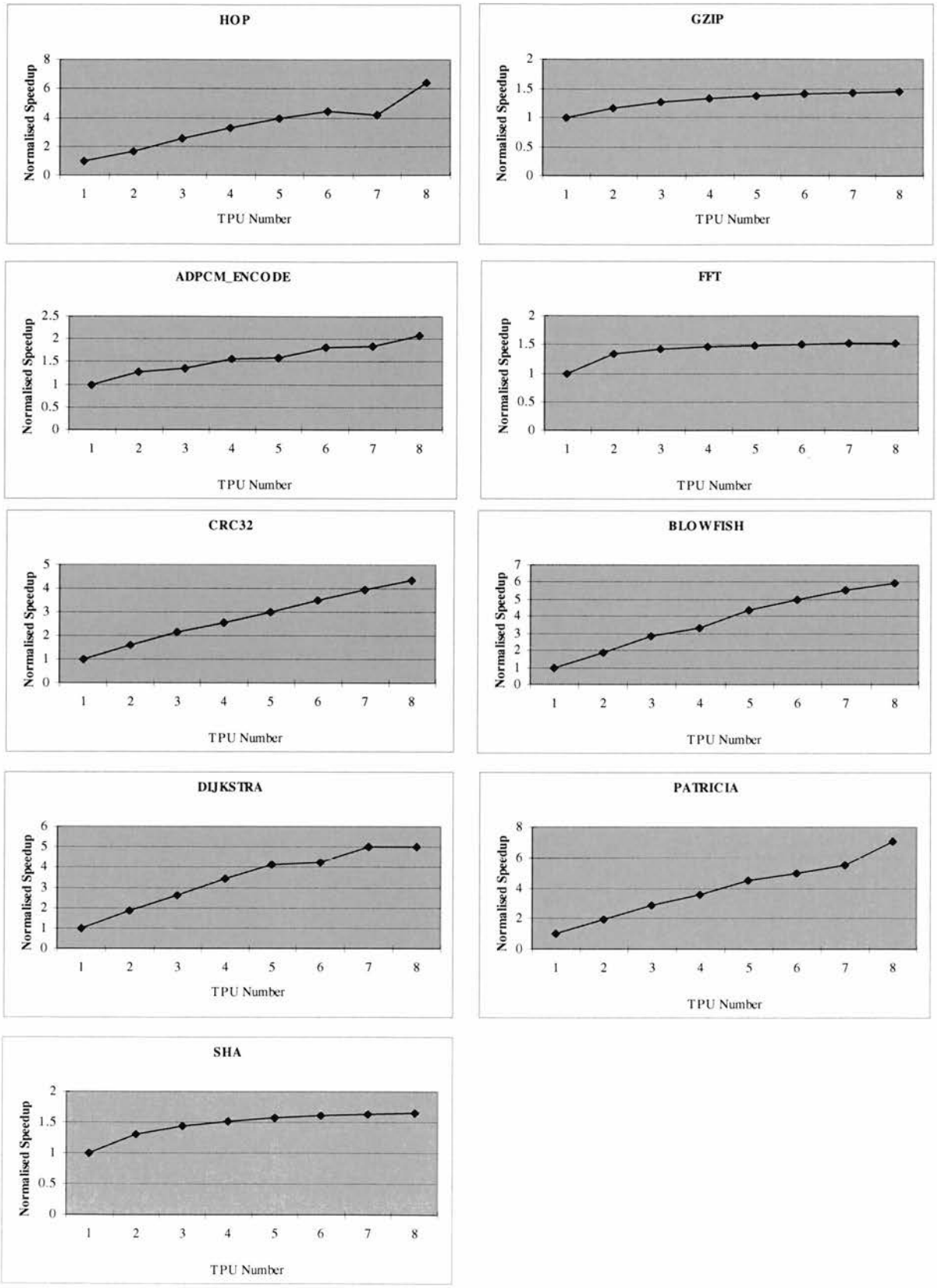


Figure 6-37: Normalised speedup for the multithreaded MAPS+RFU architecture

6.7.5 Energy consumption

The improvement in speed on speculative multithreaded MAPS+RFU comes at a price of extra energy dissipation in the reconfigurable logic and caches for storing configuration data. The implementation of reconfigurable functional unit was based on synchronous FPGA fabric which scales poorly with the number of TPUs. As a result, the energy efficiencies for multithreaded MAPS+ are the worst among all the versions. The normalised energy consumption of the nine benchmarks is displayed in Figure 6-38. The average increases in energy consumption in the speculative multithreaded MAPS over single-threaded asynchronous MAPS is 367%, 542%, 720%, 902%, 1088%, 1277% and 1455% for the 2-, 3-, 4-, 5-, 6-, 7- and 8-TPU cases respectively.

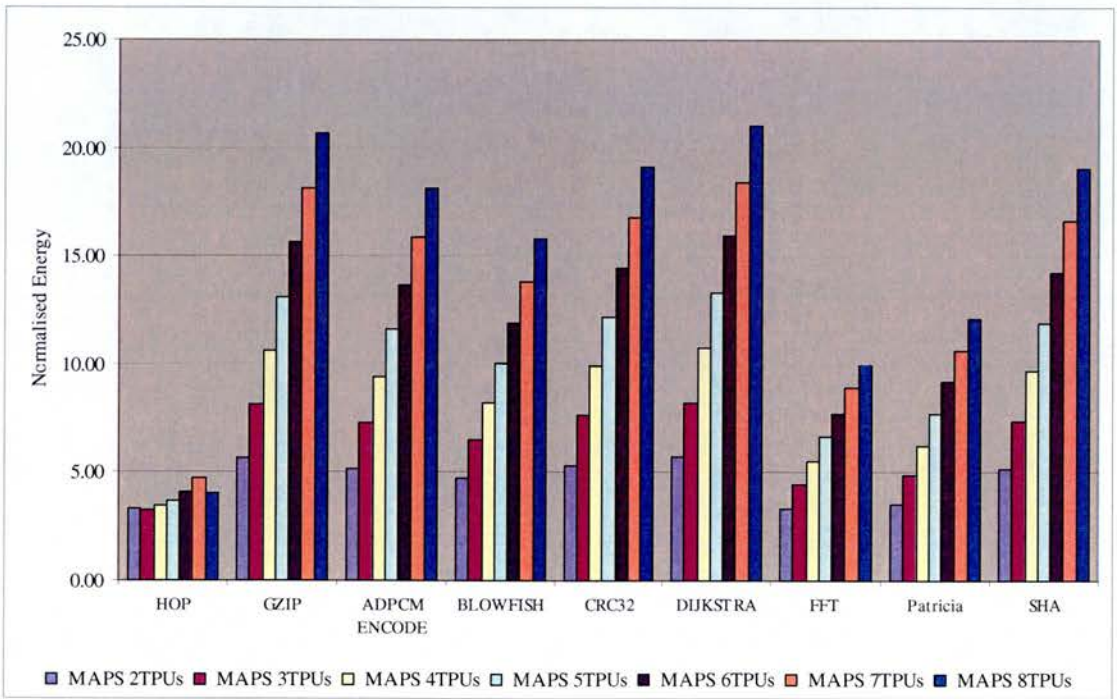


Figure 6-38: Normalised energy consumption of the benchmarks on multithreaded MAPS+RFU architecture

6.8 Summary

This chapter has presented simulation results for the asynchronous MAPS+

architecture. The SUIF/MACHSUIF compiler was chosen to perform thread partitioning and hardware code extraction, and assembly machine codes were generated for the asynchronous MAPS+ based processor. The MAPS+ architecture and the compiler algorithms have been evaluated by compiling a set of C benchmarks and simulating their output in the execution-driven instruction-level simulator. These benchmarks cover algorithms commonly used in embedded platforms, e.g. error correction, audio codecs, compression and encryption.

From our simulations, the average power consumption of synchronous clock-gated MIPS is 72 mW, and the power of asynchronous MAPS is 78 mW. Taking account that the error in the estimated power figures is at least 10%, one can conclude that the power consumption of synchronous clock-gated MIPS and asynchronous MAPS are similar. The normalised power performance of asynchronous MAPS against the synchronous clock-gated MIPS is shown in Figure 6-39. In both cases, the power performance is reduced when increasing ALU numbers. Due to lack of instruction-level parallelisms, the ALUs are saturated as numbers reach three or four.

The improvement of the asynchronous MAPS over the synchronous clock-gated MIPS is 8% for ALU-1 cases, 11% for ALU-2, 19% for ALU-3, and 27% for ALU-4. Therefore, the asynchronous MAPS architecture provides better scalabilities when the system reaches a saturated state. In the asynchronous architecture, the idle components will not consume power until it is activated. This is why the asynchronous cases provided slightly better power performance over the synchronous clock gated MIPS in saturation. However, the asynchronous MAPS imposes extra area overhead due to handshaking logic. Based on the MAPS architecture, the estimated area overhead will be 30-40% over the corresponding synchronous MIPS architecture.

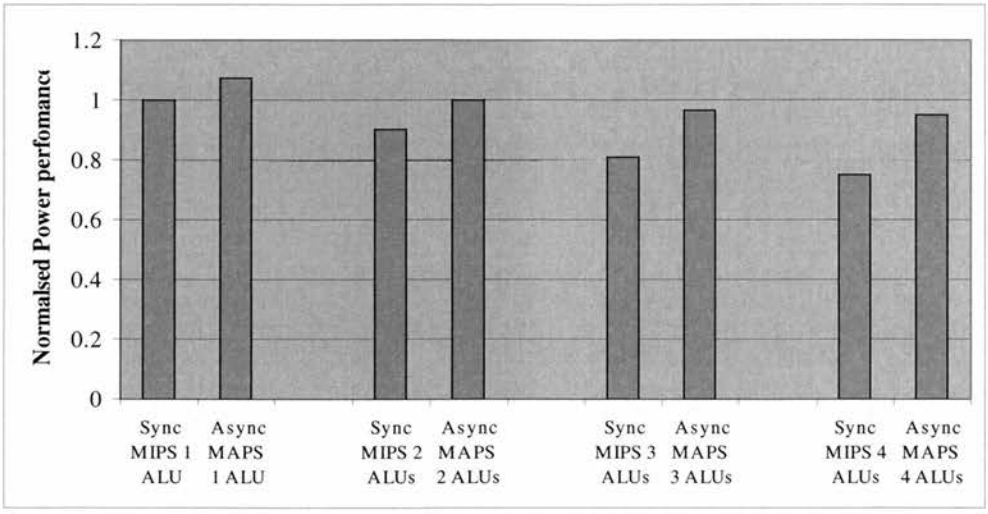


Figure 6-39: Normalised power performance of the asynchronous MAPS against synchronous clock gated MIPS

SHP compiler module converts groups of arithmetic, shift and logic operations into RFU operations by using the path-profiling information. The RFU cache configurations are also significantly affect the overall performance of a MAPS+ architecture. Analysis of the impact of scaling RFU cache on MAPS architecture shows that an under-size RFU cache will greatly slow down the performance and increase energy consumption due to the overhead of reconfiguring RFU. A 512K RFU cache with four contexts provides an optimal performance speedup and energy consumption. However, the price of introducing large sized cache into embedded processors, high power consumptions, and difficulties of designing high performance HW/SW partitioning algorithms are the factors for not introducing RFU in the power-critical embedded environment.

The normalised power performance of the asynchronous MAPS+RFU against synchronous clock gated MIPS, asynchronous MAPS with 1, 2, 3, and 4 ALUs are shown in Figure 6-39. The MAPS+RFU is not considered as a power efficient approach when compares to all the architecture setting mentioned above. The average power performance of MAPS+RFU are reduced to 39% of synchronous clock-gated MIPS, 36% of asynchronous MAPS with 1 ALU, 38% of 2 ALUs MAPS, 39.7% of 3 ALUs MAPS, and 40.2% of 4 ALUs MAPS.

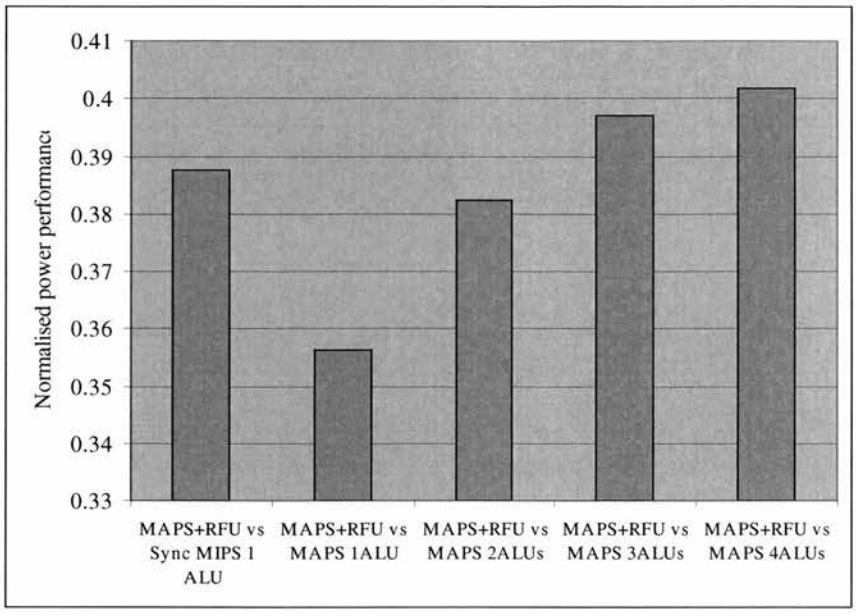


Figure 6-40: Normalised power performance of the asynchronous MAPS+RFU against different architecture settings

Multithreaded MAPS architecture relies on the thread partitioning compiler module to extract thread-level parallelism. However, loop-carried dependencies force most of the threads to execute sequentially. Extra overhead for thread synchronisations and context switching also slows down the multithreaded version. Hardware components are implemented in the multithreaed MAPS+ architecture and provide speculative threads, coupled with control-flow and data-value predictors to break the dependence among loops. Buffers are required for squashing mis-speculated threads and restore memory and register states. All the configurations are brought together in the asynchronous multithreaded MAPS+RFU architecture with enabled speculative-threads and data value predictor.

Summaries of the normalised power performance of the speculative multithreaded MAPS and the speculative multithreaded MAPS+RFU against asynchronous single-threaded MAPS are displayed in Figure 6-41. The speculative multithreaded MAPS has better power performance than the multithreaded MAPS+RFU. The power performance of multithreaded MAPS over single-threaded MAPS are 68%, 64%, 60%, 57%, 53%, 50% and 48% for 2-, 3-, 4-, 5-, 6-, 7- and 8-TPU cases, respectively.

Because reconfigurable logic is power hungry, the power performance drops

significantly when RFUs are integrated into multithreaded MAPS. The normalised power performance of multithreaded MAPS+RFU against single-threaded MAPS are 21%, 16%, 13%, 11%, 9%, 8% and 8% for 2-, 3-, 4-, 5-, 6-, 7-, and 8-TPU cases, respectively. The approach of reconfigurable computing is therefore not a good solution for power critical embedded approaches.

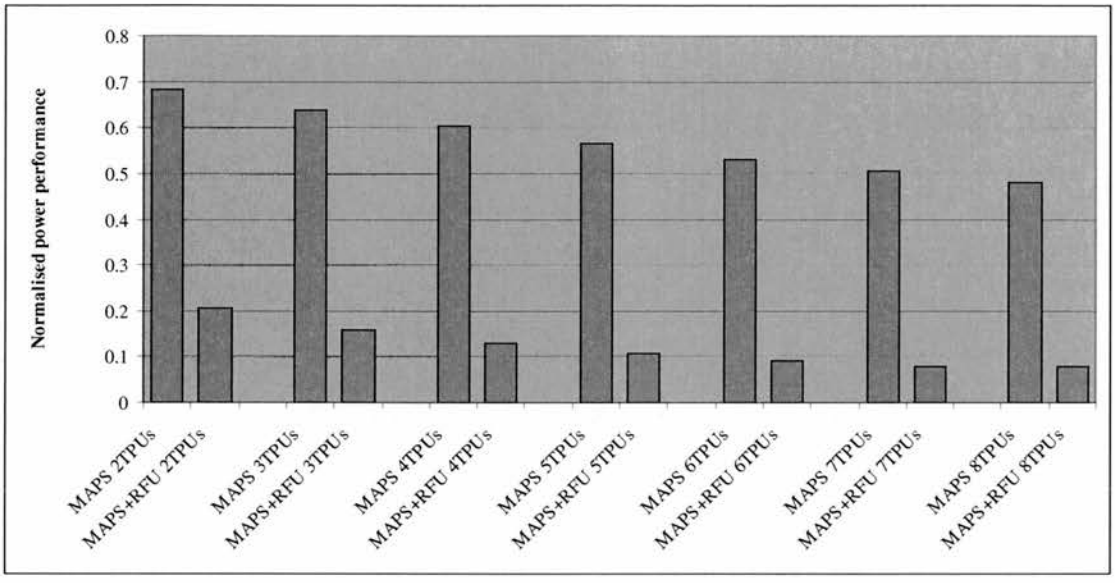


Figure 6-41: Comparison of the normalised average power performances of the speculative multithreaded MAPS and the speculative multithreaded MAPS+RFU

Chapter 7
Conclusions and Future work

7.1 Summary

This thesis brings together different areas of research: asynchronous architecture design, multithreading, and reconfigurability for hardware-software partitioning. The resulting MAPS+ architecture has been modelled and simulated in an instruction-level simulator for benchmarking programs representative of mobile wireless applications. The adoption of a micronet-base model offers potential advantages of a recursive design style of a network of TPUs, where each TPU is a network of functional units. Multithreaded execution is effective in exploring coarse-grain parallelism at the thread-level using speculative methods to minimise the effect of data dependencies on concurrent thread execution. Reconfigurable architectures promise to become a valuable alternative to conventional computer devices by accelerating frequently-used programs in reconfigurable arrays.

The MAPS+ architecture explores concurrency at different levels: between threads, within threads and between instructions. Coarse-grained thread level parallelisms are mapped to TPUs. Threads can execute concurrently with other threads, allowing parallelism to be expressed. Data predictions for speculative threads are supported by a hardware hybrid data predictor using stride-based and context-based prediction techniques. Fine-grained instruction level parallelism is mapped to the RFU. The RFU accelerates applications by customising reconfigurable fabric for compute-intensive tasks and executing several operations in parallel.

Within each TPU, the functional units are connected via communication

microagents, which handle the flow of information among the FU and with the Register Bank. The CMs negotiate local data transfers by using a four-phase, request-acknowledge handshaking protocols. At the top level, TPUs are connected via switches, which route messages from source TPUs to their destination TPUs and shared memory. Each switch is able to be connected to four other neighbouring switches or shared memory blocks via four interconnect ports. The handshaking signals among TPUs are also negotiated by the Switches.

The compiler framework based on SUIF2 generates code for the MAPS+ architectures. The MAPS+ architecture provides hardware components to perform run-time dependence checking and speculation on the control or data dependences, e.g. the data value predictors and the thread speculative buffer. Loop bodies are always hotspots for thread generations. An algorithm for thread partitioning and data dependence instruction insertions based on the static single assignment has been developed. For the sequential control paths, compiler techniques to partition a program into parallel speculative threads are investigated. Based on the profiling information, the algorithm tries to minimise the control dependence and data dependence among speculative threads. Due to the support from underlying hardware checks, the compiler does not need to guarantee the independence of threads.

The MAPS+ architecture requires the compiler to aid in the creation of configurations for RFU operations using reconfigurable logic. Automatic compilation techniques for generating reconfigurable functional unit operations to perform the arithmetic and logic operations within the program have been investigated, which provides a quick and easy way to program the MAPS+ architecture and makes the use of reconfigurable hardware more accessible to general application programmers. The algorithm to locate blocks for configurable logic is based on a profile-sensitive region formation algorithm. After the hardware block selection, a data flow graph is generated within the basic blocks, and are mapped into reconfigurable array using the hardware component library.

The SPAMSIM2 simulation environment is used to model the MAPS+ in C++. The QuickThreads package of the SPAMSIM2 kernel provides a portable interface to machine-dependent code that performs thread initialisation and context-switching. The simulator takes assembly code compiled for the MAPS+ architecture and

simulates execution with different configurations. During the initialisation stage of the SPAMSIM2 simulator, various parameters are read from the configuration file and can be set separately, e.g. memory size, delay of each function unit, cache size, replacement policy, etc. After initialisation, the parameters are set to global variables, which can be accessed by each functional blocks of the simulator. Contents of Register File and portions of Memory can be dumped to files on completion of the simulation. By dumping the results of the benchmark simulation to a file, the benchmark's execution can be verified. Data cache behaviours are simulated by using the Dinero IV cache simulator and the power analysing and optimising is based on Wattch power estimation tool.

The experimental results based on the SPAMSIM2 simulator have demonstrated the trade-offs between performance and energy consumption for different configurations of MAPS+ architectures: synchronous MIPS baseline, asynchronous MAPS, asynchronous MAPS+RFU, multithreaded MAPS, and multithreaded MAPS+RFU architecture.

7.2 Future Work

There are several issues have not been addressed in this thesis, which can be extended in future research.

7.2.1 Architecture Development

The following sections summarises possible future works to improve the MAPS+ architecture performance and evaluate the architecture in actual hardware.

Extend architecture to SMT

The current MAPS+ architecture is based on a CMP design, which is divided into distributed multiple thread processing unit, and each running a separate concurrent thread. Though the current architecture is relatively simple to implement in hardware and easy to extend and reuse the threaded processing unit, the resources are not fully

shared among logical threads. From the experimental results in the singled thread MAPS+ architecture section, the average performance improvement from two ALU to three ALU is limited, which is mainly due to the relatively simple pipeline design of the threading processing unit restricts the ability to exploit deeper level of ILP. It's believed that the performance of the MAPS can be improved by introducing SMT design into the thread processing unit, similar to the commercial processor introduced in Chapter 2, such as IBM'S POWER5 [145], Sun's UltraSPARC T1 [176], which forms a global CMP architecture, inside each TPU. Therefore, unused issue slots in TPU can be decreased by eliminating both vertical and horizontal waste. In order to support SMT in TPU, the size of register file need to be increased to hold several logical threads states.

Implement hardware registers renaming logic

In the current MAPS+ architecture, the register renaming is performed by the compiler, but this approach still has some drawbacks. It is difficult for the compiler to avoid reusing register without large code size increases. For example, in a loop, a successive iteration would have to use different registers, which requires replicating the code. Also in the MIPS instruction set chosen for the MAPS+ architecture, the register number is limited to 32, which limit the effect of software register renaming schemes.

On the other hand, the hardware register renaming logic selects a new destination for the instruction. As a result, the subsequent references to the result are directed to the new physical destination register. Therefore, false-dependencies are removed at runtime. This logic can be implemented by maintaining an explicit register mapping table [177]. This table records the current logical to physical mapping for each register and enables a sequential state to be restored in the event of an interrupt. Instead of using register mapping table, a reservation station scheme [178] is an alternative mechanism for register renaming. In the scheme, every register referenced for reads is looked up in both the indexed future file and the rename file. The future file read gives the value of that register, if there is no outstanding

instruction yet to write to it. When the instruction is placed in an issue queue, the values read from the future file are written into the corresponding entries in the reservation stations. Register writes in the instruction cause a new, unready tag to be written into the rename file. The tag number is usually serially allocated in instruction order.

Comparisons with asynchronous reconfigurable array

In current MAPS+ architecture, the reconfigurable functional unit is implemented in synchronous reconfigurable logic array with asynchronous wrapper, in order to reuse an existing FPGA architecture and CAD tools. However, current synchronous FPGA are not designed to produce hazard free signals, and hazards may be introduced by the circuit decomposition performed by technology mappers for synchronous FPGA architectures. Arbitration is a common function used in asynchronous circuits and current FPGA architectures provide no support for building the special circuitry needed in arbiters for providing clean output signals from the meta-stable states. Arbiters can be built in synchronous FPGAs, but require careful design and have a finite chance of failure. Therefore asynchronous reconfigurable fabrics are alternative options to implement the asynchronous RFU. MONTAGE [70] and AFPGA [69] provide the base architecture designs. Therefore performance comparisons can be made to investigate the chip area, performance, energy consumption trade-offs between synchronous RFU implementation and fully asynchronous RFU.

Hardware Implementations

The SPAMSIM2 is a C++ simulator for the MAPS+ architecture. The functions and handshaking protocols are modelled by event-driven simulation kernel of the SPAMSIM2. With the detailed simulation of the MAPS+ architecture, we believe that it is possible to construct processor architecture based on the MAPS+ design. Balsa [179] is framework from University of Manchester for synthesising asynchronous hardware systems and a language for describing such systems. The implementation of

MAPS+ on Balsa can be compiled into an asynchronous communicating network from a set of handshake components. The functions can be verified and deadlock during execution of a program could be avoided. Furthermore, the design can be mapped to different cell libraries for silicon foundries or libraries for programmable gate arrays. Therefore, the evaluation of the potential performance bottlenecks in a silicon design can be in investigated.

7.2.2 Compiler Improvement

The following sections summaries some possible future work to improve the MAPS+ compiler design.

Improve Hardware code generation

Due to current limitations of our hardware/software partitioning pass design, we are not able to generate pipelined implementation and loop construction in RFU. Therefore limited speedups are achieved from our simulation. So improving the hardware code generation by using pipelined logics for loop constructions is one possible future research. Further, in our MAPS compilation frameworks, the hardware codes generated for RFU are in XNF format, which is proprietary netlist description language and is becoming obsolete. Electronic Design Interchange Format (EDIF) [181] is a much more common format used to exchange design data between different CAD systems, and between CAD systems and Printed Circuit fabrication and assembly. Its syntax has been designed for easy machine parsing and is similar to LISP. By their very nature, the EDIF standards are used by most EDA vendors. Replacement with EDIF generation would provide broader FPGA output suitable for latest FPGA CAD tools.

Scheduling algorithm

In order to improve the ILP performance, one could implement fine-grained

scheduling algorithms in the compilation framework. This is done by re-ordering instructions of a program at run-time. A scheduling algorithm for asynchronous architecture called Penalise True Dependency was presented in [10]. The algorithm aims to minimise stalls due to data dependencies and resource contentions by scheduling instructions within basic blocks. This method could be extended to global optimisation using this metric on techniques such as code motion, code and tail duplication, and blocks merging.

Migrate to more realistic compilation framework

Current compilation framework is based on the SUIF/MACHSUIF compiler system, which is well suited for the implementation of the compiler prototype, because various specialised functions can be implemented separately and communicate via internal program and annotations. Basic structures, such as procedures, loop and branched are easily recognised from the intermediate format. However, the complex intermediate format design passed among different compiler modules slows down the overall compilation speed. A possible solution is to port the thread partitioning and hardware software partition algorithms to more realistic open source compiler framework, such as GCC [180]. MIPS target code and several optimisations, such SSA generation, Register Allocation, Profiling are also supported in GCC, which would improve the compilation speed while still maintaining the functionality of the thread partitioning and hardware and software partitioning for the MAPS+ architecture.

7.3 Conclusion

The main contribution of the thesis is the experimental evolution of microprocessor-framework consisting of advanced design techniques, e.g. asynchronous design techniques, multithreaded execution and runtime reconfiguration. As well as the compilation framework for thread generation, hardware and software partitioning are also investigated in detail.

In the MAPS+ architecture design, solutions are presented for the following

aspects: an asynchronous switch model based on the existing synchronous NOC incorporating asynchronous handshaking protocol; a hierarchical architecture based on the Micronet-based asynchronous architecture; a RFU design that integrates synchronous Field Programmable Logic in the asynchronous architecture, with interfaces for asynchronous to synchronous and synchronous to asynchronous inter-connections; a detailed Thread Control Unit for inter-TPU communications; a Data Value Predictor based on existing hybrid data predictor; a scheduler design for synchronising TPUs.

Compiler techniques for MAPS+ architecture are investigated on SUIF2 and Machine SUIF compiler environment. Loop parallelising techniques are implemented based on existing loop-partitioning techniques. Also mechanisms of identifying speculative threads by analysis data dependence between basic blocks are also investigated based on data dependence analysis. Solutions of partitioning hardware and software codes using regional based block formation algorithms are also presented. By using macro libraries, hardware blocks are mapped to FPGA netlist. Estimated power consumption and delay are parameters for ***ROP*** black box.

An asynchronous multithreaded simulation framework with a RFU is presented, and the performance of different processor configurations is evaluated. The simulation results show that asynchronous and clock-gated synchronous design consumes a similar level of power based on the 180nm process technology. There is an estimated 20-30% area overhead of the single-threaded asynchronous MAPS over the synchronous counterpart. However, based on the similar architecture setup (e.g. the same number of ALUs, same memory/cache sizes) the asynchronous case provides better performance. The asynchronous case also demonstrates better scalabilities when increasing the number of ALUs, and the energy increase per ALU is less than that in synchronous case.

Due to the limitations of our HW/SW partitioning algorithms design (no pipeline and loop structured support in RFU), the simulation results of the MAPS+RFU architecture show a small performance improvement over the asynchronous case. An under-size RFU cache greatly slows down the performance and increase energy consumption due to the overhead of reconfiguring RFU. Reconfigurable logic is power hungry, the power performance drops significantly when RFUs are integrated

into the multithreaded MAPS. It is still not economic to adopt reconfigurable computing techniques in power-critical embedded processors under current technology.

The multithreaded MAPS degrades its performance when increasing TPU numbers. The main reason for the performance reduction is inter-thread data dependency resulting in TPU's having to stall waiting for the arrival of dependent values. Extra overhead for thread context switching also slows down the overall performance. The simulation results show that data dependencies in the embedded applications bring greater hazards for improving performance of multithreaded applications. Without mechanisms to resolve these dependencies, a multithreaded MAPS architecture is not the best choice.

Speculative multithreaded MAPS achieve good performance improvement over the single-threaded case. But it comes with prices of by replicating the TPUs, adding hardware logics for data prediction, and using more memory for buffering history data. It is a trade-off between hardware complexity, power consumption, and performance.

Appendix A

List of Benchmark programs

Category	Benchmark name	Floating point	Descriptions
Network	Dijkstra	No	The Dijkstra benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm. Dijkstra's algorithm is a well known solution to the shortest path problem and completes in O(n2) time.
Network	Patricia	yes	A Patricia tries is a data structure used in place of full trees with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the tree to reduce traversal time at the expense of code complexity. Often, Patricia tries are used to represent routing tables in network applications. The input data for this benchmark is a list of IP traffic from a highly active web server for a 2 hour period. The IP numbers are disguised.
Security	BLOWFISH encrypt/decrypt:	no	Blowfish is a symmetric block cipher with a variable length key. It was developed in 1993 by Bruce Schneider. Since its key length can range from 32 to 448 bits, it is ideal for domestic and exportable encryption. The input data sets are a small ASCII text file.
Security	SHA	no	SHA is the secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures.
Telecommunication	FFT	yes	This benchmark performs a Fast Fourier Transform and its inverse transform on an array of data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given

			input signal. The input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components.
Telecommunication	GSM encode/decode	No	The Global Standard for Mobile (GSM) communications [18] is the standard for voice encoding/decoding in Europe and many countries. It uses a combination of Time- and Frequency-Division Multiple Access to encode/decode data streams. The input data is small and large speech samples.
Telecommunication	ADPCM encode/decode	No	Adaptive Differential Pulse Code Modulation (ADPCM) is a variation of the standard Pulse Code Modulation (PCM). A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The input data are small speech samples.
Telecommunication	CRC32	No	This benchmark performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission. The data input is the sound files from the ADPCM benchmark.
Telecommunication	HOP	no	Hop selector generates a hopping sequence that uses the address and clock from the master as inputs. This sequence determines the "pseudo-random" hopping order at which transmissions occur.
Compression	GZIP	no	GZIP (GNU zip) is a popular data compression program written by Jean-Loup Gailly for the GNU project. GZIP uses Lempel-Ziv coding (LZ77) as its compression algorithm. SPEC's version of gzip performs no file I/O other than reading the input. All compression and decompression happens entirely in memory.

Table A-1: List of Benchmark programs

Appendix B

Power consumption breakdown

	Average power consumption of synchronous MIPS architecture (without clock gating)	Average power consumption of synchronous MIPS architecture (with clock gating)	Average power consumption of asynchronous MAPS architecture
Clock	52.41 mW	5.77 mW	0.00 mW
Communication Microagent	0.00 mW	0.00 mW	2.28 mW
Control Unit	1.77 mW	1.32 mW	1.24 mW
Issue Unit	7.28 mW	4.89 mW	5.60 mW
Register File	4.62 mW	2.17 mW	4.08 mW
Instruction Cache	50.70 mW	27.05 mW	31.96 mW
Data Cache	46.93 mW	10.91 mW	11.57 mW
On Chip Bus	0.97 mW	1.00 mW	1.47 mW
Memory Unit	17.13 mW	6.36 mW	8.86 mW
Branch Unit	12.95 mW	5.50 mW	5.11 mW
ALU	17.80 mW	6.81 mW	5.12 mW
Total	212.55 mW	71.78 mW	77.29 mW

Table B-1: Power consumption breakdown

Bibliography

[1] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, Richard B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, (December 2001).

[2] David Keppel, "Tools and Techniques for Building Fast Portable Threads Packages", University of Washington Technical Report UWCSE 93-05-06, (May. 1993).

[3] Jan Edler, Mark D.Hill , "Dinero IV Trace-Driven Uniprocessor Cache Simulator", http://www.cs.wisc.edu/~markhill/DinernoIV

[4] David Brooks, Vivek Tiwari, and Margaret Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," 27th International Symposium on Computer Architecture (ISCA), (June 2000).

[5] Stanford SUIF Compiler Group. "The SUIF Compiler Systems." http://suif.stanford.edu/

[6] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam, David L. Moore, Brian R. Murphy, Constantine Sapuntzakis, Computer Systems Laboratory,Stanford University, "An Overview of the SUIF Compiler System".

[7] Michael D. Smith's Research Group on Compilation and Computer Architecture. "Machine SUIF", http://www.eecs.harvard.edu/hube/software.

[8] MIPS Technologies, Inc. "MIPS32 4K Processor Core Family Software User's

Family", (Sept, 2002).

[9] Robert. D. Mullins, "Dynamic Instruction Scheduling and data forwarding in asynchronous superscalar processors", Ph.D. Thesis, University of Edinburgh, (2001).

[10] Salvador Satelo Salazer, "Instruction Scheduling in Micronet-based Asynchronous ILP processors", Ph.D. Thesis, University of Edinburgh, (2002).

[11] D. W. Parent, W. C. Lin, H. Rattanasonti, "Comparison of an Asynchronous Manchester Carry Chain Adder to a Synchronous Manchester Carry Chain Adder ", EE Department, College of Engineering, San Jose State University, (Sept 2004).

[12] Alex Branover, Rakefet Kol, Ran Ginosar. "Asynchronous Design By Conversion: Converting Synchronous Circuits into Asynchronous Ones", Date, p. 20870, Design, Automation and Test in Europe Conference and Exhibition Volume II ,DATE'04, (2004).

[13] A. Varma, E. Debes, I. Kozintsev, and B. Jacob, "Instruction-level power dissipation in the Intel XScale embedded microprocessor." ,Proc. SPIE's 17th Annual Symposium on Electronic Imaging Science & Technology, San Jose CA, (Jan 2005).

[14] Bluetooth Baseband Specification V1.2 (Nov 2005).

[15] SPEC CPU2000 V1.2 http://www.spec.org/osg/cpu2000/

[16] DIETLIBC, "diet libc - a libc optimized for small size", http://www.fefe.de/dietlibc/

[17] Doug Burger, Todd M.Austin, "The SimpleScalar Tool Set, Version 2.0",

http://www.simplescalar.com, (1997).

[18]Zhiyuan Li, "Configuration Management Techniques for Reconfigurable Computing", North-western University, Ph.D. Thesis. (June, 2002).

[19]R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," Journal of VLSI Signal Processing, vol. 28, no. 1, pp. 7--27, (May,June 2001)

[20]"Virtex-II Platform FPGA User Guide", UG002 (V2.0), pp 269-359, Xilinx Data Sheet, (March 2005).

[21]Andre DeHon. "DPGA-coupled microprocessors: Commodity ICs for the early 21st century." In Duncan A. Buell and Kenneth L. Pocek, editors, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp 31--39, (April 1994).

[22]Steve Trimberger, Khue Duong, and Bob Conn. "Architecture issues and solutions for a high-capacity FPGA." Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp 3--9, (February 1997).

[23]Ann Gordon-Ross, Frank Vahid, Nikil Dutt. "Automatic Tuning of Two-Level Caches to Embedded Applications", Date, p. 10208, Design Automation and Test in Europe Conference and Exhibition Volume I , DATE'04, (2004).

[24]Albonesi, D.H. "Selective cache ways: on demand cache resource allocation." Journal of Instruction Level Parallelism, (May 2002).

[25]Malik, A., Moyer, W., Cermak, D. "A low power unified cache architecture providing power and performance flexibility.", International Symposium on Low Power Electronics and Design, (2000).

[26]Zhang, C., Vahid, F., Najjar, W. "A highly-configurable cache architecture for embedded systems". 30th Annual International Symposium on Computer Architecture, (June 2003).

[27]Glen Reinman and Norman P. Jouppi, "CACTI 2.0: An Integrated Cache Timing and Power Model", WRL, Research Report, (July. 2000).

[28]Arvind, D.K., And Rebello, V. E. F. "Instruction-level parallelism in asynchronous processor architectures". In Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures (Leuven, Belgium), M. Moonen and F. Catthoor, Eds., Elesevier Science Pusblisers, pp. 203-215, (August 1994).

[29]Arvind, D.K., And Rebello, V. E. F. "On the performance evaluation of asynchronous processor architectures". In Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication System (MASCOTS'95) (Durham, NC, USA), P. Dowd and E. Gelenbe, Eds,. IEEE Computer Society Press, pp. 100-105, (January 1995).

[30]Arvind, D.K, and R.D.Mullins. "A fully asynchronous superscalar processor". In International Conference on Parallel Architectures and Compilation Techniques. Newport Beach, California, (October 1999).

[31]Timothy J. Callahan and John Wawrzynek. "Instruction Level Parallelism for Reconfigurable Computing". In Hartenstein and Keevallik, editors, FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinin, Estonia, volume 1482 of Lecture Notes in Computer Science. Springer-Verlag, (September 1998).

[32]Stefanos Kaxiras, Alan D. Derenbaum, and Girja Narlikar. "Simultaneous

multithreaded DSPs: Scaling from high performance to low power." Technical report, 10009639-001024-06, Bell Laboratories, Lucent Technologies.

[33] Daniel Towner and David May. "The Uniform Heterogeneous Multi-threaded Processor Architecture". In: Communicating Process Architectures -- 2001, Alan Chalmers, Majid Mirmehdi and Henk Muller, editors, pages 103--116. IOS Press, (September 2001).

[34] Patrick Crowley, Marc E. Fiuczynski, and Jean-Loup Baer. "On the performance of multithreaded architectures for network processors". Technical report, Department of Computer Science, University of Washington, Seattle, WA 98195, (Oct, 2000).

[35] Panit Watcharawitch, "MulTEP: A MultiThreaded Embedded Processor", Technical Report, UCAM-CL-TR-588, ISSN 1476-2986, University of Cambridge, Computer Laboratory, (May 2004).

[36] Terry Tao Ye, "On-Chip Multiprocessor Communication Network design and analysis", Ph.D. Thesis, Stanford University (December 2003).

[37] Shashi Kumar, Axel Jantsch, Mikael Millberg, Johny Oberg, Juha-Pekka Soininen, Martti Forsell, Kari Tiensyrja, Ahmed Hemani. "A Network on Chip Architecture and Design Methodology," isvlsi, p. 0117, IEEE Computer Society Annual Symposium on VLSI, (2002).

[38] Willam John Bainbridge, "Asynchronous System-on-chip Interconnect", Ph.D Thesis, Department of Computer Science, University of Manchester (March 2000).

[39] Giovanni De Micheli, Luca Benini, "Networks on Chip: Technology and Tools (Systems on Silicon)", Morgan Kaufmann Publishers Inc, US, ISBN-10-0123705215, (30 Augest, 2006).

[40]Lionel M. Ni, Philip K. McKinley. "A Survey of Wormhole Routing Techniques in Direct Networks," Computer, vol. 26, no. 2, pp. 62-76, (February 1993).

[41]J. Wu; "A deterministic fault-tolerant and deadlock-free routing protocol in 2-D meshes based on odd-even turn model", Proceedings of the 16th international conference on Supercomputing, pp. 67-76, (2002).

[42]W. Dally and C. Seitz, "Deadlock-free Message Routing in Multiprocessor Interconnection Networks," in IEEE Transactions on Computers, pp. 547–553, (1987).

[43]W. J. Dally, H. Aoki, "Deadlock -free adaptive routing in multicomputer net works using virtual channels", IEEE Trans. on Parallel and Distributed Systems, pp. 466-475, (April 1993).

[44]Liang, J.; Swaminathan, S.; Tessier, R. "aSOC: A Scalable, Single-Chip communications Architecture". In: IEEE International Conference on Parallel Architectures and Compilation Techniques, pp. 37-46, (October 2000).

[45]Ni, L. M.; McKinley, P. K. "A Survey of Wormhole Routing Techniques in Direct Networks". IEEE Computer Magazine, v.26(2), pp. 62-76, (February, 1993).

[46]Mark B. Josephs andJ elio T. Yantchev. "CMOS design of the tree arbiter element". IEEE Transactions on VLSI Systems, 4(4), pp. 472–476, (December 1996).

[47]Thomas Villiger, Hubert Kaslin, Frank K. Gurkaynak, Stephan Oetiker, Wolfgang Fichtner. "Self-Timed Ring for Globally-Asynchronous Locally-Synchronous Systems," async, pp. 141, Ninth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'03), (2003).

[48]I.E. Sutherland, "Micropipelines", in Communications of the ACM, vol. 32, pp.

720–738. (June 1989)

[49]G. Birtwistle and A. Davis (eds.), "Asynchronous Digital Circuit Design", Springer-Verlag (Workshops in computing series), London, (1995).

[50]A.J. Martin, "Asynchronous Datapaths and the Design of an Asynchronous Adder", in Formal Methods in System Design, volume 1:1, pp. 119–137, (July 1992)

[51]Steven M. Nowick, Kenneth Y. Yun, Ayoob E. Dooply, Peter A. Beerel. "Speculative Completion for the Design of High-Performance Asynchronous Dynamic Adders," async, pp. 210, Third International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '97), (1997).

[52]Scott McFarling, "Combing Branch Predictors", Western Research Laboratory, Technical Note TN-36, (June 1993)

[53]S. T. Pan, K. So, and J. T. Rahmeh. "Improving the accuracy of dynamic branch prediction using branch correlation" Proceedings of ASPLOS V, pages 76--84, Boston, MA, (October 1992).

[54]David M. Koppelman, "The Benefit of Multiple Branch Prediction on Dynamically Scheduled Systems", Workshop on Duplicating, Deconstructing, and Debunking Held in conjunction with the 29th International Symposium on Computer Architecture, pp. 42-51, (May 26, 2002).

[55]Shai Rotem, Ken Stevens, Charles Dike, Marly Roncken, Boris Agapiev, Ran Ginosar, Rakefet Kol, Peter Beerel, Chris Myers, Kenneth Yun. "RAPPID: An Asynchronous Instruction Length Decoder," async, p. 60, Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems ,ASYNC"99, (1999).

[56]J.K.L. Lee and A.J. Smith. "Branch prediction strategies and branch target buffer Design". Computer, 17(1), (January 1984).

[57]Charles Price, "MIPS IV Instruction Set", Revision 3.2, MIPS Technologies Inc, (September, 1995).

[58]Gerry Kane and Joe Heinrich,"MIPS RISC Architecture". Prentice Hall, (1992).

[59]Rangsian Marukatat, "Clustered Multithreading for Speculative execution", Ph.D Thesis, ICSA, University of Edinburgh, (2003).

[60]K. Olukotun, L. Hammond, and M. Willey, "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP," Proc. Int'l Conf. Supercomputing, (1999).

[61]B. Zheng, J.-Y. Tsai, B.Y. Zhang, T. Chen, B. Huang, J.H. Li, Y.H. Ding, J. Liang, Y. Zhen, P.-C, Yew, and C.Q. Zhu, "Designing the Agassiz Compiler for Concurrent Multithreaded Architecture," Proc. Workshop Languages and Compilers for Parallel Computing (LCPC), (1999).

[62]Dubey, P. K., O'Brien, K., O'Brien, K. M., and Barton, C. 1995. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. In Proceedings of the IFIP Wg10.3 Working Conference on Parallel Architectures and Compilation Techniques (Limassol, Cyprus). IFIP Working Group on Algol, Manchester, UK, 109-121, (June , 1995).

[63]S. Hauck, T.W. Fry, M.M. Hosler, J.P. Kao. "The Chimaera reconfigurable functional unit," fccm, p. 87, 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97), (1997).

[64]Razdan, R. "PRISC: programmable reduced instruction set computers". Doctoral Thesis. UMI Order Number: UMI Order No. GAX95-00124., Harvard University.

(1994).

[65]Rolf Enzler, "Architectural Trade-offs in Dynamically Reconfigurable Processors", Ph.D Thsis, Diss. ETH NO. 15423. Swiss Federal Institute of Technology, (2004).

[66]"Virtex II Platform FPGAs: Complete Data Sheet", Xilinx Prodcut Specification, DS031 (v3.4), (March, 2005).

[67]Simon Moore, George Taylor, Peter Robinson, Robert Mullins. "Point to Point GALS Interconnect", async, p. 69, Eighth International Symposium on Asynchronus Circuits and Systems (ASYNC'02), (2002).

[68]Fuhrer, R. M., Lin, B., and Nowick, S. M. 1995. "Symbolic hazard-free minimization and encoding of asynchronous finite state machines". In Proceedings of the 1995 IEEE/ACM international Conference on Computer-Aided Design (San Jose, California, United States). International Conference on Computer Aided Design. IEEE Computer Society, Washington, DC, 604-611, (November 05 - 09, 1995)

[69]John Teifel, Rajit Manohar. "An Asynchronous Dataflow FPGA Architecture," IEEE Transactions on Computers, vol. 53, no. 11, pp. 1376-1392, (November, 2004).

[70]S. Hauck, S. Burns, G. Borriello, and C. Ebeling, "An FPGA for Implementing Asynchronous Circuits," IEEE Design and Test of Computers, vol. 11, no. 3, pp. 60-69, (1994).

[71]R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Shiozawa, M. Inamori, and K. Nagami, "PCA-1: A Fully Asynchronous Self-Reconfigurable LSI," Proc. Int'l Symp. Asynchronous Circuits and Systems, (2001).

[72]F. Gabbay and A. Mendelson, "Speculative Execution Based on Value Prediction", Technion, Israel Institute of Technology, Technical Report 1080, (1996).

[73]J. Gonzalez and A. Gonzalez. "The potential of data value speculation to boost ILP", in Proceedings of the 12th International Conference on Supercomputers, pp. 21-28, (July 1998).

[74]M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Load Value Prediction", in Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 138-147, (October 1996).

[75]M. Lipasti and J. Shen, "Exceeding the Dataflow Limit via Value Prediction", in Proceedings of the 29th Annual International Symposium on Microarchitecture, pp. 226-237, (Dec. 1996).

[76]Y. Sazeides and J. E. Smith, "The Predictability of Data Values", in Proceedings of the 30th International Symposium on Microarchitecture, pp. 248-258, (December 1997).

[77]Y. Sazeides and J. E. Smith, "Implementations of Context Based Value Predictors", Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Technical Report ECE97-8, (Dec. 1997).

[78]K. Wang and M. Franklin. "Highly Accurate Data Value Prediction Using Hybrid Predictors", in Proceedings of the 30th International Symposium on Microarchitecture, pp. 281-290, (December 1997).

[79]B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction", in Proceedings of the 26th International Symposium on Computer Architecture, pp. 64-74, (May 1999).

[80]G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation", in 31st International Symposium on Microarchitecture, (1998).

[81]B. Rychlik, J. Faistl, B. Krug, and J.P. Shen., "Efficacy and performance impact of value prediction", in International Conference on Parallel Architectures and Compilation Techniques, (1998).

[82]Eric Stoltz, Michael P. Gerlek, Michael Wolfe. "Extended SSA with Factored Use-Def Chains to Support Optimization and Parallelism". Proc. Hawaii International Conference on Systems Sciences. (January, 1994).

[83]Reginald Clifford Young, "Path-based compilation". Ph.D. thesis, Division of Engineering and Applied Sciences, Harvard University. (January, 1998).

[84]Bhowmik, A. and Franklin, M. 2004. "A General Compiler Framework for Speculative Multithreaded Processors". IEEE Trans. Parallel Distrib. Syst. 15, 8, 713-724, (Aug. 2004).

[85]Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F.Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In ACM Transactions on Programming Languages and Systems, vol. 13, no. 4, pages 451-490, (Oct.1991).

[86]Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. "A new algorithm for partial redundancy elimination based on SSA form". In Proceedings of the ACM SIGPLAN conference on programming Language Design and Implementation, pages 273-286, (1997).

[87]M. Wolfe. "Beyond Induction Variables." Proceedings of the SIGPLAN'92, Symposium on Programming Languages Design and Implementation. SIGPLAN Notice 27, pages 162-174, (July 1992).

[88]Peng Tu and David Padua. "Gated SSA-Based Demand-Driven Symbolic

Analysis for Parallelizing Compilers". Proceedings of the 9th ACM, International Conference on Supercomputing, page 144, (1995).

[89] M. P. Gerlek, E. Stoltz, and M. Wolfe. "Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form". ACM Transaction on Programming Languages and Systems, Vol 17 No 1,pages 85-122, (January 1995).

[90] Michael Wolfe, "Beyond induction variables". In Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, pages 161--174, (1992).

[91] Liu, S., Lo, R., and Chow, F. "Loop Induction Variable Canonicalization in Parallelizing Compilers". In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques. PACT. IEEE Computer Society, Washington, DC, 228, (October 20 - 23, 1996)

[92] William Morton Pottenger, "Induction variable substitution and reduction recognition in the POLARIS parallelizing compiler", University of Illinois at Urbana-Champaign, MSc Thesis, (1995).

[93] B. Zheng. "Integrating Scalar Analyses and Optimizations in a Parallelizing and Optimizing Compiler." Ph. D Thesis, also as Technical Report 00-011, Dept. of Computer Science and Engineering, University of Minnesota. (February, 2000)

[94] John R. Hauser and John Wawrzynek. "GARP: A MIPS processor with a reconfigurable coprocessor". Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM), Napa, CA, (April 1997)

[95] Timothy J. Callahan, Phillip Chong, Andre DeHon, and John Wawrzynek. "Fast module mapping and placement for datapaths in FPGAs". In Proceedings of the ACM/SIGDA Sixth International Symposium on Field Programmable Gate

Arrays (FPGA-98), pages 123--132, New York, ACM Press, (February 22-24 1998).

[96] Timothy J. Callahan, John R. Hauser, John Wawrzynek. "The Garp Architecture and C Compiler," Computer, vol. 33, no. 4, pp. 62-69, (April, 2000).

[97] Maya B. Gokhale, Janice M. Stone. "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture," fccm, p. 126, IEEE Symposium on FPGAs for Custom Computing Machines, (1998).

[98] R.E.Hank, "Region-Based Compilation". Ph.D. dissertation, University of Illinois at Urbana Champaign, (1996).

[99] R.E.Hank, S.A.Mahlke, R.A.Bringmann, J.C.Gyllenhaal, and W.W.Hwu, "Superblock formation using static program analysis". In Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitectures, pages 247-256, (Dec. 1993).

[100] R.E.Hank, W.mei W.Hwu, and B.R.Ran. "Region-based compilation: An introduction and motivation." In IEEE/ACM International Symposium on Microarchitecture, pages 158-168, (1995).

[101] Gordon Moore, "Speaking of extending Moore's Law", International Solid-State Circuits Conference (ISSCC), February 2003.

[102] C.P.Collier, E.W. Wong, M. Belohradsky, F.M. Raymo, J.F.Stoddart, P.J.Kuskes, R.S.Willams, and J.R. Heath. "Electronically Configurable Molecular-Bases Logic Gates", Science, 285:391-394, (July 16 1999).

[103] Seth Copen Goldstein and Mihai Budiu. "NanoFabrics: Spatial Computing Using Molecular Electronics". In Proceedings of the 28th International Symposium on Computer Architectures 2001, (2001).

[104] "Intel Pentium4 encyclopedia", http://en.wikipedia.org/wiki/Pentium_4, (2005)

[105] "The ITRS report: Executive Summary, 2005 Edition", International Technology Roadmap for Semiconductors, (2005)

[106] C. L. Seitz. "System Timing. In C. Mead and L. Conway, editors, Introduction to VLSI Systems", chapter 7, pages 218–262. Addison-Wesley, (1980).

[107] J. Sparso, C. D. Nielsen, L. S. Nielsen, and J. Staunstrup. "Design of Self-Timed Multipliers: A Comparison". In S. B. Furber and M. Edwards, editors, Asynchronous Design Methodologies, volume A-28 of IFIP Transactions, pages 165-179. Elsevier Science Publishers, (1993).

[108] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V.Woods. "The Design and Evaluation of an Asynchronous Microprocessor". In Proceedings of International Conference on Computer Design, pages 217-220. IEEE Computer Society Press, (October 1994).

[109] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. "AMULET2e: An Asynchronous Embedded Controller". In Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 290-299. IEEE Computer Society Press, (April 1997).

[110] Verhoeff, T., "Encyclopedia of Delay-Insensitive Systems", Eindhoven University of Technology, The Netherlands, (1995-1998).

[111] Peeters, A.M.G., "Single-Rail Handshake Circuits". Ph.D. thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, (1996).

[112] Dean, M.E., Dill, D.L., And Horowitz, M. "Self-timed logic using

current-sensing completion detection (CSCD)". In Proceeding of International Conference Computer Design (ICCD), IEEE Computer Society Press, pp. 187-191 (Oct. 1991).

[113] Dean, M.E., Dill, D. L., And Horowitz, M. "Self-timed logic using current-sensing completion detection (CSCD)". Journal of VLSI Signal Processing 7, 1/2 , 7-16, (Feb. 1994).

[114] Martin, A.J., "The limitations to delay-insensitivity in asynchronous circuits". In W.J. Dally, editor, Sixth MIT Conference on Advanced Research in VLSI, pp263-278, MIT Press, (1990)

[115] Manohar, Rajit and Martin, Alain J. "Quasi-Delay-Insensitive Circuits are Turing-Complete". Technical Report. California Institute of Technology. CaltechCSTR:1995.cs-tr-95-11, (1995).

[116] Scott Hauck, "Asynchronous design methodologies: An overview," In Proceedings of the IEEE, 83(1):69-93, (1995).

[117] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic analysis: Concrete results." In Cryptographic Hardware and Embedded Systems (CHES 2001), volume 2162 of Lecture Notes in Computer Science, pp. 251-261. Springer, (2001).

[118] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): "Measures and counter-measures for smart cards.", In I. Attali and T.P. Jensen, Ed., Smart Card Programming and Security (E-smart 2001), volume 2140 of Lecture Notes in Computer Science, pp. 200{210. Springer, (2001).

[119] Simon Moore , Ross Anderson, Robert Mullins, George Taylor, Jacques J.A. Fournier, "Balanced Self-Checking Asynchronous Logic for Smart Card Applications", Journal of Microprocessors and Microsystems, 27(9) pp.421-430,

(October 2003).

[120] Gunther, S.H., Binns, F., Carmean, D.M., and Hall, J.C. "Managing the Impact of Increasing Microprocessor Power Consumption". Intel Technology Journal Q1 2001, pp 1-8, (2001).

[121] Christos P. Sotiriou, Luciano Lavagno, "De-synchronization: asynchronous circuits from synchronous specifications", SOC Conference, 2003. Proceedings. IEEE International [Systems-on-Chip], Vol., Iss., 17-20, pp. 165- 168, (September 2003).

[122] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Christos P. Sotiriou: "Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications." IEEE Trans. on CAD of Integrated Circuits and Systems 25(10): 1904-1921 (2006).

[123] Stephen B. Furber, P. Day, Jim D. Garside, N. C. Paver, J. V. Woods, "AMULET1: A Micropipelined ARM". COMPCON 1994, pp 476-485, (1994).

[124] Stephen B. Furber, David A. Edwards, Jim D. Garside, "AMULET3: A 100 MIPS Asynchronous Embedded Processor". ICCD 2000, pp 329-334 , (2000)

[125] Jim D. Garside, W. J. Bainbridge, Andrew Bardsley, David M. Clark, David A. Edwards, Stephen B. Furber, David W. Lloyd, S. Mohammadi, J. S. Pepper, Steve Temple, J. V. Woods, J. Liu, O. Petli, "AMULET3i - An Asynchronous System-on-Chip". ASYNC 2000, pp162-175, (2000).

[126] Hans van Gageldonk, "An Asynchronous Low-Power 80C51 Microcontroller". Ph.D. Thesis, Eindhoven University of Technology, The Netherlands, (1998)

[127] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel

Gloor, Gerhard Stegmann. "An Asynchronous Low-Power 80C51 Microcontroller," async, p. 0096, Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '98), (1998).

[128] "The UltraSPARC IIIi Processor Architecture Overview", Technical whitepaper, Sun Microsystems, Version 1.2 (April, 2004)

[129] Arjan Bink, Mark de Clercq, Richard York, "ARM996HS Processor", White Paper, Handshake Soluations, ARM Limited, (February 2006).

[130] "ARM996HS Technical Reference Manual", Revision:r0p0, ARM Limited, (2005-2006).

[131] P.B. Endecott, "SCALP: A Superscalar Asynchronous Low-Power Processor", Ph.D Thesis, Department of Computer Science, University of Manchester, (1995)

[132] Alain J. Martin, Mika Nyström, Paul Penzes, and Catherine Wong , "Speed and Energy Performance of an Asynchronous MIPS R3000 Microprocessor". Caltech Report: CSTR:2001.012, (June 2001).

[133] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri Cummings and Tak Kwan Lee. "The Design of an Asynchronous MIPS R3000 Microprocessor". Proc. 17th Conference on Advanced Research in VLSI, 164-181, IEEE Computer Society Press, (1997).

[134] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Shiozawa, M. Inamori, and K. Nagami, "PCA-1: A Fully Asynchronous Self-Reconfigurable LSI," Proc. of 7th International Symposium on Asynchronous Circuits and Systems (ASYNC 2001), (March 2001)

[135] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi, "Plastic Cell Architecture:Towards Reconfigurable Computing for General-Purpose," Proc. of

6th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98), pp.68-77, (April 1998).

[136] N. Imlig, T. Shiozawa, K. Nagami, Y. Nakane, R. Konishi, H. Ito, and A. Nagoya, "Scalable Space/Time-shared Stream-Processing on the Run-time Reconfigurable PCA Architecture,", Proc. of 8th Reconfigurable Architecture Workshop (RAW 2001) associated with 15th Annual,International Parallel & Distributed Processing Symposium (IPDPS 2001), (April 2001).

[137] N. Imlig, T. Shiozawa, K. Nagami, R. Konishi, and K. Oguri, "Communicating Logic: Digital Circuit Compilation for the PCA Architecture," IPSJ DA Symposium '99, pp.101-106, (July1999).

[138] T. Ungerer, B. Robič, J. Šilc, "A survey of processors with explicit multithreading", ACM Computing Surveys, 35(1):29-63, (2003).

[139] T. Ungerer, B. Robič, J. Šilc, "Multithreaded processors", The Computer Journal, 45(3):320-348, (2002).

[140] Dean Tullsen, Susan Eggers, and Henry Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", Proceedings of the 22rd Annual International Symposium on Computer Architecture, (June 1995).

[141] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen, "Simultaneous Multithreading: A Platform for Next-generation Processors", IEEE Micro, (September/October 1997).

[142] Luke K. McDowell, Susan Eggers, and Steven D. Gribble, "Improving Server Software Support for Simultaneous Multithreaded Processors", Symposium on Principles and Practice of Parallel Programming, (June, 2003).

[143] Lance Hammond, Basem A. Nayfeh and Kunle Olukotun, "A Single-Chip

Multiprocessor", IEEE Computer Special Issue on "Billion-Transistor Processors", (September 1997).

[144] Lance Hammond, Ben Hubbert , Michael Siu, Manohar Prabhu , Mike Chen , and Kunle Olukotun, "The Stanford Hydra CMP", IEEE MICRO Magazine, March-April 2000, and presented at Hot Chips 11, (August 1999).

[145] Ron Kalla, Balaram Sinharoy, Joel M. Tendler. "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro, vol. 24, no. 2, pp. 40-47, (March/April, 2004).

[146] Deborah T. Marr,Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, Michael Upton, "Hyper-Threading Technology Architecture and Microarchitecture", Intel Technology Journal, Volume 6, Issue 1, (February 14, 2002)

[147] "ARM11 MPCore Process", ARM Technical Reference Manul, Revision:r0p2, (2005).

[148] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. "An elementary processor architecture with simultaneous instruction issuing from multiple threads". In 19th Annual International Symposium on Computer Architecture, pages 136–145, (May 1992).

[149] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky. "Performance estimation of multistreamed, superscalar processors". In 27th Hawaii International Conference on System Sciences, pages I:195–204, (January 1994).

[150] V. Krishnan and J. Torrellas. "A Chip-Multiprocessor Architecture with Speculative Multithreading". Special Issue on Multithreaded Architecture, IEEE Transactions on Computers, (December 1999).

[151] Patrick Niemeyer, Jonathan Knudsen, "Learning Java", O'Really, Third Edition, ISBN: 0-596-00873-2 , (May 2005).

[152] Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell, "PThreads Programming, A POSIX Standard for Better Multiprocessing", O'Really, First Edition, ISBN: 1-56592-115-1, (September 1996)

[153] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, "Ramesh Menon,"Parallel Programming in OpenMP", Morgan Kaufmann Publishers, ISBN: 1-55860-671-8, (2001).

[154] William Gropp, Ewing Lusk and Anthony Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface". Cambridge, MA, MIT Press, (1994).

[155] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. "Multiscalar processors". In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 414--425, (June 22--24, 1995).

[156] T.N. Vijaykumar and Gurindar S. Sohi, "Task Selection for a Multiscalar Processor", 31st International Symposium on Microarchitecture (MICRO-31), (November/December 1998).

[157] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, Bhama Sridharan, "Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations", Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag, LNCS 757, pp.406-420, (1993).

[158] T. Nguyen, J. Gu, and Z. Li. "An Interprocedural Parallelizing Compiler and Its Support for Memory Hierarchy Research." Lecture Notes in Computer

Science, 1033, (1996).

[159] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. "Polaris: A New-Generation Parallelizing Compiler for MPP's." Technical Report 1306, Center for Supercomputing Research and Development, (June 1993).

[160] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. "An Overview of the PTRAN Analysis System for Multiprocessing." In Proceedings of the 1st International Conference on Supercomputing. (June 1987).

[161] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, (December 1996).

[162] Rene Rebe, "Compiler Rally- A look at the Intel C/C++ Compiler 9.0", Linux Magazine, Issue 59, pp.50-52, (October 2005).

[163] Eichenberger, A.E; O'Brien, K.; Peng Wu; Tong Chen; Oden, P.H.; Prener, D.A.; Shepherd, J.C.; Byoungro So; Sura, Z.; Wang, A.; Tao Zhang; Peng Zhao; Gschwind, M. ,"Optimizing Compiler for the CELL Processor",Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, Vol., Iss., 17-21, Pages: 161- 172, (September 2005).

[164] J. Tsai, P.-C. Yew. "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation". In the Proceedings of Int'l Conf. on Parallel Architecture and Compiler Techniques (PACT'96), (1996).

[165] Akkary, H. and Driscoll, "A dynamic multithreading processor". In Proceedings of the 31st Annual ACM/IEEE international Symposium on Microarchitecture (Dallas, Texas, United States). International Symposium on

Microarchitecture. IEEE Computer Society Press, Los Alamitos, CA, 226-236, M. A. (1998).

[166] "MicroBlaze performance", http://www.xilinx.com/ipcenter/processor_central/microblaze/performance.htm, (2005).

[167] Athanas, P. M. and Silverman, H. F. . "Processor reconfiguration through instruction-set metamorphosis". Computer 26, 3, 11-18, (Mar. 1993).

[168] Maya B. Gokhale, Janice M. Stone. "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture", FCCM, p. 126, IEEE Symposium on FPGAs for Custom Computing Machines, (1998).

[169] R.Razdan, M.D.Smith, "A High Performance Microarchitecture with Hardware Programmable Functional Units", Micro 27, pp 172-180, (November 1994).

[170] Ye, Z. A., Shenoy, N., and Baneijee, P. 2000. "A C compiler for a processor with a reconfigurable functional unit". In Proceedings of the 2000 ACM/SIGDA Eighth international Symposium on Field Programmable Gate Arrays (Monterey, California, United States,). FPGA '00. ACM Press, New York, NY, 95-100, (February 10 - 11, 2000).

[171] Ralph D. Wittig and Paul Chow. "OneChip: An FPGA Processor With Reconfigurable Logic". In The Fourth Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96), pages 126-135. IEEE, (March 1996).

[172] Jorge E. Carrillo E. "Evaluation of the OneChip Reconfigurable Processor". Master's thesis, University of Toronto, Department of Electrical and Computer Engineering, Toronto, Ontario, M5S 3G4, (2000).

[173] M. B. Gokhale, J. M. Stone, and E. Gomersall. "Co-synthesis to a hybrid

RISC/FPGA architecture". Journal of VLSI Signal Processing, 24(2/3):165–180, (Mar. 2000).

[174] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. "PipeRench: A reconfigurable architecture and compiler". IEEE Computer, 33(4):70–77, (April 2000).

[175] Doug Burger, Todd, M. Austin, "The SimpleScalar Toolset, Version 2.0", SimpleScalar LLC, (1997).

[176] Denis Sheahna, "Developing And Tuning Application on UltraSPARC T1 Chip Multithreading Systems", UltraSPARC T1 Architecture Group, Sun Blue Prints Online, (December 2005)

[177] M.Moudgill, K.Pingali, and S.Vassiliadis, "Register renaming and dynamic speculation: an alternative approach", Tr 93-1379, Department of Computer Science, Cornell University, (August 1993).

[178] Perry H. Wang, Hong Wang, Ralph M. Kling, Kalpana Ramakrishnan, John P. Shen. "Register Renaming and Scheduling for Dynamic Execution of Predicated Code," hpca, p. 0015, Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), (2001).

[179] Doug Edwards, Andew Bardsley, Lilian Janin, Will Toms, "Balsa: A tutorial Guide.", Version V3.4.2, University of Manchester, (Jan, 2005)

[180] "GCC Home Page", http://gcc.gnu.org

[181] "Electronic Design Interchange Format", http://www.edif.org

[182] N. H. E.Weste and K. Eshraghian. "Principles of CMOS VLSI Design - A Systems Perspective", pages 231–238. Addison-Wesley, 2nd edition, (1993).

[183] Tin Wai Kwan, Maitham Shams, "Design of High-Performance Power-Aware Asynchronous Pipelined Circuits in MOS Current Mode Logic," async, pp. 23-32, 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05), (2005).

[184] Gerald Aigner, Amer Diwan, David L. Heine, Monica S. Lam David L. Moore, Brian R. Murphy, Constantine Sapuntzakis, "An Overview of the SUIF2 Compiler Infrastructure", Computer Systems Laboratory, Stanford University & Portland Group, Inc. (1999).

[185] G.J. Chaitin. "Register allocation and spilling via graph coloring." In Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction, pages 98-105, New York, NY,. ACM, (1982).

[186] Glenn Holloway and Michael D. Smith, "The Machine-SUIF Control Flow Analysis Library", Division of Engineering and Applied Sciences, Harvard University, (2000).

[187] "PowerCentric Overview", Azuro website, http://www.azuro.com/prod/prod01.htm, (2005).

[188] "MIPS32 M4K Processor Core Datasheet", MIPS Technologies, Revision 01.01, Document Number: MD00247 (January 8, 2003).

[189] J.Bhaker, "A System C Premier", Second Edition, Star Galaxy Publish, ISBN-10-0965039129, (Feb 2004).

[190] "Handle-C Langue Reference Manaual", Celoxica Limited, RM-1003-4.2, DK version 3.0, (2004).

[191] Hutchings, B., Bellows, P., Hawkins, J., Hemmert, S., Nelson, B., and

Rytting, M. 1999. "A CAD Suite for High-Performance FPGA Design". In Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM. IEEE Computer Society, (April 21 - 23, 1999).