國立臺灣大學電機資訊學院電機工程學系

博士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Doctoral Dissertation

晶片上多核處理器與其驗證模型之設計

Design of On-chip Multi-Processor and its Verification Model

林光輝

Guang-Huei Lin

指導教授：陳少傑 博士

Advisor: Sao-Jie Chen, Ph.D.

中華民國 98 年 8 月

August, 2009

國立臺灣大學博士學位論文
口試委員會審定書

晶片上多核處理器與其驗證模型之設計

Design of On-chip Multi-Processor and its Verification Model

本論文係 林光輝 君（D87921034）在國立臺灣大學電機學系、所完成之博士學位論文，於民國九十八年七月七日承下列考試委員審查通過及口試及格，特此證明

口試委員：

陳少傑（簽名）

（指導教授）

[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	張 耀 文

系主任、所長 胡振國（簽名）

Design of On-chip Multi-Processor and its Verification Model

By

Guang-Huei Lin

DISSERTATION

Submitted in partial fulfillment of the requirement
for the degree of Doctor of Philosophy of the
Department of Electrical Engineering
at National Taiwan University
Taipei, Taiwan, R.O.C.

July 2009

Approved by：

[illegible] [illegible] [illegible]

[illegible] [illegible] [illegible]

Advised by： [illegible]

Approved by Director： [illegible]

致謝

感謝普林斯頓大學的 Ruby Lee 教授提供並且指導 PLX 指令集的使用，使我們在處理器設計有很好的入門基礎。

感謝威斯康辛大學的胡玉衡教授的指導，使我們對演算法的最佳化、多媒體應用及處理器架構等有更深入的了解，使得研究得以順利完成。

感謝蘇培陞博士的指導, 使我們得以快速掌握電子系統層級這個新興且複雜的領域，進而開啟軟硬體共同設計的研究方向。

感謝中正大學的熊博安教授與林朝聖同學設計了 PLX 的即時作業系統。

感謝曉龍在編譯器研究上的貢獻，君任在系統驗證上的貢獻，英誠在 NoC 研究上的貢獻，亞南在多媒體演算法上的貢獻，盈貝在 FPGA 製作的協助，以及伯壎在生醫電子應用的咨商，使我們的研究完整且有扎實的基礎。

感謝振哲，鴻奎及泗紋在程式撰寫與晶片佈局上的努力，使我們的實作得以完成。

再次感謝所有協助這個論文完成的所有老師和同學們。

中 文 部 分

摘要

這論文是一個研究專案的成果，旨在開發嵌入式多媒體系統使用的晶片上多核處理器架構。近年來晶片上多核處理器已經成為數位電路設計，計算機輔助設計，以及嵌入式系統開發的焦點。我們的重點是設計一種新型的基於 PLX 的單指令多資料指令集架構的晶片設計平台。本論文探討研究成果的幾個面向，包含各種單處理器與多處理器的微架構，系統層級軟硬體協同設計和協同驗證，以及平行化的方法。

目錄

第一章
簡介

隨著半導體工藝技術已經縮小到奈米級，一個晶片可以包含上億個電晶體。由於晶片設計的複雜度增加，以及應用多樣性的需求日益提高，使得晶片的市場及時性（time-to-market）與市場延續性（time-in-market）之重要性大大提昇，從而導致可程式化的晶片上多核處理器之普及。而電子系統層級 （ESL）設計驗證方法的採用降低了大型系統晶片的設計時間並提昇可靠度。

為提高處理器的性能以期能與特定應用集成電路（ASIC）相抗衡，必須從處理器的微架構，多處理器通訊與以及軟體平行化等多方面作努力。

本論文採用由下而上的設計方法。從多媒體的應用分析開始，藉由所遭遇的問題與提出解決方法，而引導出次字組平行的單指令多資料 PLX 指令集，超長指令字組，同時多執行緒，與快取間的直接資訊傳遞等設計架構。內容安排如下：第二章介紹了這些設計架構的細節，第三章介紹了系統層級設計和驗證，第四章介紹了軟體的平行化技術，以及最後第五章給出結論。

第二章
處理器設計

在多媒體應用中，大多是處理低解析度的資料。若一個 64 位元處理器能同時處理 8 個 8 位元像素，則效率提高 8 倍。次字組平行的單指令多資料多媒體延伸指令集發揮了這個特色。在嵌入式系統，我們通常用定點數取代浮點數以降低成本與功耗；為解決定點數溢位造成的精確度下降，需要特殊指令來處理。在加解密及快速傅利葉轉換常需要重新排列資料，而在單指令多資料運作下位元組間的移動也變成了排列運算；提供各種排列指令可大幅提升效率。當兩個影像重疊，各像素的亮度應該相加，當相加後亮度太高造成溢位，表現出來的反而是暗灰色。為避免溢位的失真，相加溢位後亮度應該設為飽和值。飽和運算需要許多比較與分支指令因此效率極差，提供飽和運算指令對影像處理有極大助益。

在設計處理器元件時，我們考量的是速度與面積的取捨。在元件中，暫存器組、快取記憶體控制與乘法器所佔面積最大，加法器面積不大但使用最頻繁，因此使用面積較大的進位選擇加法器可得最佳效益，並且進位選擇加法器可很容易支援不同解析度的單指令多資料操作。經由速度-面積的曲線分析，第一版 PLX 處理器的最佳操作速度是 260MHz。

由於多媒體尚有許多 32 位元純數運算，使用 64 位元處理器會浪費許多功率，因此我們使用超長指令字組的技術使兩個 32 位元指令可同時執行。但此技術需要較多的暫存器組存取埠與旁路邏輯，會拖慢整體效能。藉由同時多執行緒技術可消除旁路邏輯以及分支預測邏輯，沒有旁路邏輯的運算單元可以增加管道級數並平行化，藉由分時多工使第二版 PLX2 處理器的運算速度加倍至 520MHz.

使用 energy-delay-area 乘積比較兩版本的效率，對於能平行化且記憶體存取與計算能重疊的程式，第二版的效率較佳。對於循序演算法，第一版效率較佳。

第三章
系統級設計與驗證

在多處理器系統，處理器間通訊的頻寬決定了晶片的整體效能。多處理器系統通常連結成網路，而依照記憶體共享方式的不同有四種架構。第一種是各處理器獨立，處理器間透過網路交換訊息。第二種是完全共享，每個記憶體都有唯一的位址，可以透過直接記憶體存取方式溝通。第三種是私有快取，資料提供者須把資料寫入共用記憶體，使用者再從共用記憶體提取。第四種是分享快取，第二層快取分散在每個處理器，透過快取同步的機制溝通。前兩種方式傳送一筆資料只要用到一次網路，而後兩種方式需要用到兩到三次網路，前兩種方式較有效率。由於快取是用來暫存最近用過的資料以免重複自晶片外提取，若用前兩種方式，軟體設計者須自行管理重複資料提取，這是很大的負擔。

我們設計一個機制，可暫時把快取的一部分當作訊息的先進先出緩存區，使處理器間可以像第一種方式般直接傳送訊息以減少網路使用，但仍保留快取的優點。另外設計一個 OpenMP to TLM 的工具以減輕軟體設計負擔。它分析 OpenMP 程式的 shared 參數，依其大小與可平行化程度決定要使用直接傳送還是透過共用記憶體來溝通。所有的通訊與記憶體存取都改成 MPI 型式，各程序的運算時間也用 profiling 的方法分析出來。產生的程式可放在 SystemC TLM 平台上跑以得到較精確的時間資訊。

第四章
平行化

軟體平行化的技術從 1960 年代開始發展。藉由分析迴圈內各變數的相依性，再用變數更名、純量擴展及迴圈轉置等技術移除假相依，許多迴圈可以轉換成為單一向量表示。而向量化的程式可以套用在單指令多資料指令集而得到資料階層的平行化。

針對次字組平行的單指令多資料指令集，有許多新的挑戰要克服。首先是控制流，由於次字組無法各別執行分支指令，控制流必須轉換為資料流才能平行化。第二是記憶體位址對齊，由次字組所組成的向量，其起始位址不一定在 64 位元邊界，運算時須移位對齊。而當需要重新排序與移位時，排序指令的選擇與使用順序也大幅影響效能。這些工作需要更深入的程式行為分析。

多媒體應用通常需要大量存取記憶體，減少多餘的記憶體存取也是增加效能很重要的關鍵。當使用迴圈轉置進行向量化時，可能有許多種可行的組合，而這些組合中記憶體存取次數最少者通常效能最佳。我們採用一個系統化的方法尋找最佳組合。首先將所有可平行化的迴圈展開，將所有運算依其資料相依性重建為一個樹。依照暫存器的數目以及記憶體位址對齊等條件，從最靠近樹根的運算中找出基本運算方塊重建為最內層的迴圈。接著從剩餘運算中找出其記憶體存取位址與基本運算方塊重疊最高者，依其規則重建外層迴圈。記憶體存取位址重疊時，其資料可由先前載入的暫存器經移位得到，因而減少了記憶體存取次數。當使用同時多執行緒時，多個運算方塊可重疊而得到更高的記憶體資料重利用率。

第五章
結論

我們設計了一個 PLX 為基礎的晶片上多核處理器。通過分析多媒體應用，我們決定了次字組平行的單指令多資料指令集架構及所需的特殊指令。第一版 PLX 晶片可以運行在 260MHz。

透過超長指令字組與單指令多資料的整合設計使 32 位元程式更有效地工作在 64 位核心上。同時多執行緒技術增加運算單元管道級數和消除旁路邏輯以提高處理器的速度。改進的 PLX2 晶片可以運行在 520MHz。

為了使訊息在處理器間直接傳送以減少網路通信量，一個將快取的一部分當作訊息的先進先出緩存區的機制被提出。為了減少軟體設計的負擔，一個工具將 OpenMP 程式轉換為 MPI,並可在 SystemC TLM 平台上執行軟硬體協同設計和協同驗證。

針對 PLX 單指令多資料與多媒體程式特性提出一個平行化工具。除了向量化外，另一個重點是減少多餘的記憶體存取以及記憶體邊界對齊的額外負擔。

在奈米時代，使用有特定應用指令集可程式化的晶片上多核處理器，並致力於提高其性能功率效率是必須的。而這個結合軟硬體的設計流程對系統階層模型、處理器設計、平行化編譯器與及時作業系統的設計者都是很大的負擔。這篇論文對於相關領域的知識與整合有概略的探討，希望我們的經驗對於多核系統的研究設計有所貢獻。

ABSTRACT

This Dissertation is the outcomes of a research project aiming at developing multi-processor System-on-Chip (SoC) architecture for embedded multimedia systems. Since its inception a decade ago, SoC has captured the attentions of application specific integrated circuit (ASIC) design houses, computer aided design (CAD) companies, and embedded system developers. In particular, the immense popularity of killer multimedia gadgets, such as the iPod and smart phone, has fueled unprecedented interests in developing new generation multimedia SoC systems.

We focused on the design of a novel SoC platform based on a PLX Subword-Parallel Single Instruction Multiple Data (SWP-SIMD) instruction set architecture. Most of the materials included in this Dissertation are drawn from the outcomes of our research project. Several single-processor and multi-processor micro-architectures are deeply studied and adapted to our design. However, the high level of integration also brings great challenges to system designers. Hardware and software are necessarily becoming convergent and must be fully concurrent design endeavors. The system level hardware/software co-design and co-verification methodologies are also discussed in this Dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

CHAPTER ONE
INTRODUCTION

In last decade, performance/area efficiency is the main concern in designing chips, all commercial chips must have the highest performance and smallest area. As semiconductor process geometries have shrunken into nanometer, more and more gates can be integrated into a chip, thus logic area becomes less sensitive.

In 1965, Gordon Moore predicted that the number of transistors incorporated in an IC would increase twofold every year. This was really an amazing prediction that proved to be more accurate than Moore had believed. In the past few decades, the scale of IC integration has been soaring high. It started from *Small Scale Integration* (SSI) with around 100 transistors per IC in 1960s, up to *Very Large Scale Integration* (VLSI) accommodating more than 10,000 transistors per IC in 1980s. There is no sign that such tendency would ever cease. In recent years, the integration scale has only slightly slowed down to a factor of two for every eighteen months. The outburst of IC complexity, as predicted by Moore's Law, is driving the current semiconductor industry to challenge another cutting edge revolution: *System-on-Chip* (SoC) with the capacity of integrating more transistors in a single chip to form an entire electronic system. This concept is feasible thanks to the very exceptional manufacturing advances that bring IC nanotechnology to fruition. As Moore's Law continues unencumbered into the nanometer era, process geometries have shrunken to 65 nm, chips are reaching the giga-gate scale. A normal 32-bit multiplier may contain 8,000 gates. Adding computation components into an SoC will not increase the chip area too much.

In SoC era, energy consumption and time-in-market/time-to-market become new concerns. SoC used in portable handheld consumer electronic products is more and more popular. For handheld devices, battery endurance is an important parameter. Time-in-market is also as important as time-to-market. Any new chip production needs to pay a very high *non-returnable engineering* (NRE) cost even if there is only a little modification from the previous version. By the same reason, we wish that the chip can sustain longer and re-useable for more applications. For example, in IP-based 4G wireless communication, we would like to design a chip used for both WiMAX (*Worldwide Interoperability for Microwave Access*) and LTE (*Long Term Evolution*), while these standards are not well-defined. The key to applying a single integrated circuit to multiple applications for both time-to-market and time-in-market is *programmability*.

An *application specific instruction-set processor* (ASIP) is a software-programmable processing-element tailored for this purpose. It provides an efficient and economic way for a particular application computation. An ASIP may add some multimedia operations or encryption operations into its instruction-set to improve performance with low cost-overhead.

Traditional hardware-software partitioning is simply as: critical functions performed by specific hardware and control-oriented functions by software. Here specific hardware is defined as an *application specific integrated circuit* (ASIC) which is a special design dedicated for an application, and software means running a code on a general purpose processor. In the last century, most embedded systems need specific hardware to process multimedia applications, with the constraint of power consumption or performance. In general, specific hardware is more power-efficient than software for an application with the same performance. But specific hardware is less flexible to adapt to new features.

To compete with the performance of ASIC, many parallelization techniques are adapted into ASIP. These techniques include *data level parallelism* (DLP) in a *single instruction multiple data* (SIMD) processor, *instruction level parallelism* (ILP) in a *very large instruction word* (VLIW) processor, *thread level parallelism* (TLP) in a multi-threading processor. Armed with these parallelism mechanisms, multi-processor system-on-chip (MPSoC) becomes more and more feasible and popular in portable handheld consumer electronic products.

Therefore, a specialized parallel compiler becomes more important to optimize an application on a specific multi-core processor. This kind of parallel compiler has not only to translate high-level programming language instructions into the target ASIP codes, but also to schedule these instructions to exploit the parallelization capability of that ASIP.

Software needs to run on its target processor. While processor and compiler are designing, software is unable to design until a prototype was developed. Without verification by software, the processor is not guaranteed to meet system constraints, thus the ASIP needs to be re-designed many times. To reduce the long cycle, both developing software as early as possible and evaluating system constraints at a higher system level become very important.

Electronic system-level (ESL) design methodology has been introduced to decrease design cost and design time for large scale SoCs. Two methodologies had been introduced to implement an ESL design, a bottom-up process and a top-down process [1][2].

The top-down process begins from specification. It automatically decomposes a specification into hardware and software under cost, power and performance constraints. In this process, a system is described at a high abstraction layer by an architecture description language (ADL). Such a processor described in ADL can easily change its instruction set, pipeline stage, register file size, and issue width. The C compiler, instruction set simulator, and synthesizable RTL code can be generated automatically. The algorithm that should be processed by a dedicated hardware to satisfy performance requirement must be translated into RTL code by high level synthesis (HLS) technique.

In a bottom-up process, the fundamental building blocks of an SoC are *intellectual property* (IP) cores, which are reusable hardware blocks designed to perform the particular task regarded in a given component. An IP core could either be a programmable component like a processor such as ARM or MIPS, or a hardware entity with fixed behavior like an MPEG accelerator. Different IP cores are interconnected on an SoC by a communication structure, such as a shared bus or a *network-on-chip* (NoC), in order to establish communication among them. IP reuse is the main challenge in a bottom-up process. A reusable IP can be obtained from the third-party IP provider. While every IP provider has a large amount of in-house IPs, it is difficult to integrate IPs from different providers without a standard. Typically IP providers would not release their RTL design. The time spent to identify a third-party IP and integrate it into the designed system places this approach at an unfavorable position compared to designing the IP in-house. A higher level *Transaction Level Modeling* (TLM) description is more feasible for IP providers to protect their design.

TLM is the current promotion methodology used for hardware/software co-design before and after hardware/software partitioning. Before partitioning, TLM could be used to create a point-to-point, addressless functional yet concurrent system model, reusing IP behaviors from application engineers. After partitioning, TLM automatically wraps the behavior in the address-mapped TLM model for embedded software functional verification.

PLX [3], developed by Professor Ruby Lee at Princeton University, is a native *subword-parallel single instruction multiple data* (SWP-SIMD) instruction set architecture (ISA) [4] that supports high-performance, low-cost multimedia information processing, 3-D graphical processing and permutation instructions for security operations.

This Dissertation intends to discuss many of the above mentioned hardware-software codesign issues that we encountered in designing a PLX-based embedded multimedia SoC platform. The contents are organized as follows. Chapter 2 introduces techniques in designing ASIP. Chapter 3 describes the system level design and verification. Chapter 4 introduces software parallelization techniques specifically tailored for PLX. Finally, a conclusion is drawn in Chapter 5.

CHAPTER TWO
ASIP DESIGN

Today's growth in markets for consumer electronics, wireless electronics, and hand-held devices requires cost-efficient solutions that supply high performance computing, energy efficiency, and programmability. General-purpose processors are poorly suited to meet the requirements of energy efficiency and competitive cost. ASICs are unable to provide sufficient programmability. As a result, a variety of Application Specific Instruction-set Processors (ASIP) is emerging to meet the requirement.

General-purpose processors are the ones used in desktop PCs and servers. Development tools for desktop processor are popular, and there are millions of software developed for desktop processor. Thus, using general-purpose processor can reduce time-to-market. But this solution is not optimized on some critical metrics including performance, cost, power, and size. Many embedded multimedia systems are handheld systems, such as MP3 players, PDAs and 3G phones. A single general-purpose processor is unable to handle real-time functions such as communication, camera, video, audio, touch screen, TV, and GPS in time, or it will consume too much power.

Many embedded processors have worse performance on general applications, but have much better performance on some specific applications than the general-purpose processors. The well-known examples are *digital signal processor* (DSP) and network processor.

Embedded system devices normally embody the functionality they implemented. In other words, they are designed to run a few codes with a predictable pattern. In contrast, applications of a general-purpose system are known in advance. A traditional embedded system design flow is to select a pre-designed platform from IP provider which may satisfy power/performance constraint, then to spend most of the effort on developing software for this system.

We had designed two processors tailor for multimedia application. The first processor *PLX* utilizes subword-parallel instruction set architecture (ISA) to improve multimedia application performance. The second version *PLX2* improves energy-delay-area efficiency by including *Very Long Instruction Word* (VLIW) and Simultaneous Multi-Threading (SMT) techniques.

2.1 PLX Processor Design

In a RISC processor, all operation execution should complete in one cycle. Complex operation will increase critical path thus reduce clock rate. In deciding the instruction set architecture, we prefer the operations which can improve multimedia application performance without reducing clock rate. By this constraint, we implement subword-parallel single-instruction-multiple-data (SWP-SIMD), fixed-point, permutation, and saturation arithmetic operations into PLX ISA.

2.1.1 SWP-SIMD

Supercomputer with vector processor was developed in the 1960s to increase the scientific computation speed. Since scientific program codes contain many one-dimensional vector and two-dimensional matrix operations, using a vector processor can perform these operations simultaneously to improve performance. A vector processor is also called a single-instruction multiple-data (SIMD) machine because it can apply one instruction on many data elements. Such kind of parallelism is often called *data level parallelism* (DLP).

Multimedia applications mostly perform low-precision data, such as 16-bit audio samples and 8-bit video pixels. Today the ALU word size in a processor is mostly sixty-four bits. It is a waste to compute 16-bit data using a 64-bit ALU. If the 64-bit ALU can compute four 16-bit data simultaneously, its throughput will be higher. A processor owing an instruction set architecture (ISA) with this feature is called a *subword-parallel single instruction multiple data* (SWP-SIMD) processor [3]. It works as an SIMD vector machine, but performs in a single register. Many low-precision data are packed into a *superword* which occupies a register, and each element is called *subword* which only occupies part of a register. This feature is also called *multimedia extension* for it is specified for multimedia applications. MAX-1 is the first SWP-SIMD ISA for HP PA-RISC processor [4], introduced in January 1994.

PLX [5] is an SWP-SIMD ISA developed by Professor Ruby Lee at Princeton University. The main feature of PLX is that it is native to SWP-SIMD. Its vector function unit supports 8/16/32/64 subword widths, and its scalar function unit is just the 64-bit subword subset in a vector unit. A typical multimedia code contains many scalar operations, such as loop counter or memory index, which disable vector pipeline to execute smoothly. A native SWP-SIMD can execute scalar and vector operations in the same core to reduce scalar-vector communication overhead.

Power-aware is a benefit obtained from the SWP-SIMD feature. The term power-aware is often ascribed to any system which design has been sensitive to energy consideration; its connotation in recent work has been shown in [6]:

(1) The system allows its clients to adjust the expected quality and also the tolerable latency/throughput constraints.

(2) When such adjustments are made, the energy consumption is expected to vary accordingly, *i.e.*, higher energy dissipation is tolerated by clients for higher quality (or lower latency) and vice-versa.

There are many topics on power/performance trade-offs. At the circuit level, since the CMOS power consumption is proportional to voltage square, the core and bus buffer supply voltages usually have to be reduced to save power. At the logic level, gated clock when datapath is not working can reduce unnecessary logic switching power. At the system level, the supply power of a non-active core can be turned off. The disadvantage is its requiring a long stable time to turn on again, which may cause real-time request failure. At the algorithm level, datapath width adjustment can get the most power budget. For example, if a program performing only 8-bit operations with a value range of -128 to +127 is implemented in a 32-bit ALU, the register switching of bits 8 to 31 are meaningless and the power is thus wasted.

Most applications contain variables of different widths. An MPEG-2 video decoder [7], for example, contains fifty 1-bit Boolean variables, nine 8-bit char variables, thirty-nine 16-bits short variables, seventeen 24-bit variables, and eighty-two 32-bit variables. If implemented in a 16-bit datapath, the 24-bit and 32-bit operations cannot be completed in one cycle and the performance will be degraded, but the power spent on fewer bit operations is saved by the reduction of meaningless switching. This example showed that when the datapath width is larger than 28 bits, the performance increases little, but the power and area are still increased linearly, so the best power-efficient design occurs at 28 bits.

Most processor-based system design is unable to change the datapath width, or they need to change instruction set architecture to adjust datapath width [8], which needs extra cost for decoding the second instruction set. PLX's native subword-parallelism design extends the flexibility to change datapath width during software execution, which can improve the computation power efficiency.

Figure 2-1 demonstrates the subword parallel processing concept of PLX instruction `padd Rd,Rs1,Rs2`. Eight 8-bit data are packed into one 64-bit word. They are processed by one `padd` instruction, taking only one cycle. With the appropriate subword boundaries, this technique results in the parallel processing of subwords. The degree of parallelism is within an instruction and depends upon the size of the subword.

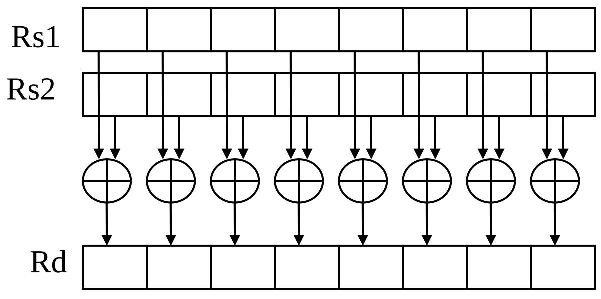


Figure 2-1. Subword-parallel execution.

Figure 2-2 shows a logic level power-aware concept. Figure 2-2(a) is a 4-bit pipelined adder. The maximum delay (T) is two half-adder delays at stage 4. The highest performance is 1/T operations per second. When the system requires only half of the performance, the clock frequency can reduce to 1/2T, and the power consumption is also reduced to half. Now the clock cycle 2T is much larger than the maximum delay, T. Figure 2-2(b) changes the pipeline registers of Stage 1 and Stage 3 into buffers, the critical path delay is one full-adder plus two half-adders plus register setup time, it is a little lower than 2T. The combinational logic propagation power is increased because it is more complex, but register power is reduced. Using well-designed combinational logic, the total power consumption can be reduced greatly. Figure 2-2(c) extends the adder to support subword parallel. Compared to Figure 2-2(a), it uses 4 extra adders, but can compute four 1-bit additions in one stage, two 2-bit additions in two stages, or one 4-bit addition in 4 stages. When data precision is low, higher stages can be gated to save power.

Adjusting the pipeline structure dynamically will increase the complexity of data dependence detection. An instruction is dependent to previous instruction needs bypass logic to forward result as described in Section 2.3.3. On a dynamic pipeline architecture, the bypass logic becomes complex. An alternate power-aware circuit implementation for SMT is described in Section 2.3.5. In SMT, bypass logic is not needed, thus no extra power waste on bypass logic.

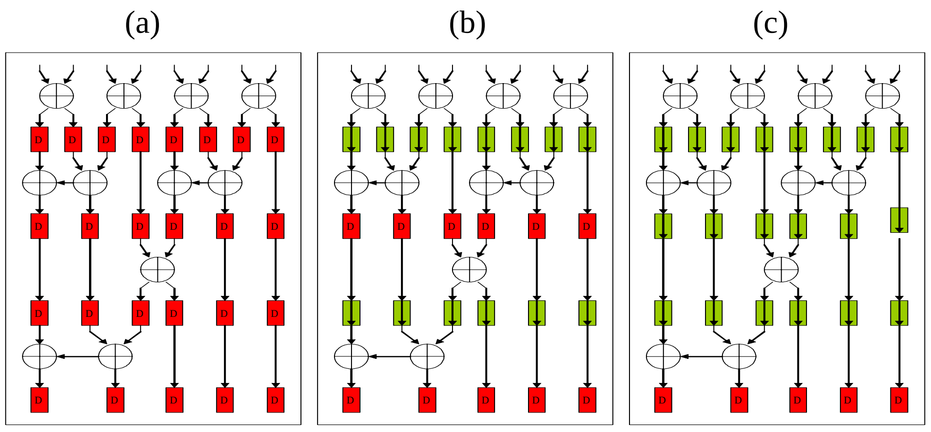


Figure 2-2. Power-aware reconfigurable pipelined adder.

Another subword-parallel ALU design is for high-performance purposes. Figure 2-3 shows a 64-bit wide carry-select adder structure, where all the subword 8-bit adders are designed to complete an addition in one clock cycle. At the beginning, two pairs of 8-bit subword additions are computed in each 8-bit ALU, one with a carry-in of 0 and the other with a carry-in of 1. Then these two obtained results are respectively stored in the two registers waiting for the select control signal to select one addition result to output. In such way, we can have eight 8-bit precision addition operations done in one cycle. For a 16-bit precision addition, the four multiplexer control signals “16” are high and the other “32” and “64” control signals are low, such that the carry-out of an even byte can pass through the multiplexer and serve as the select control signal to select the result of an odd byte, we can thus have four 16-bit subword addition results generated at one clock cycle. For a 64-bit precision addition, all the multiplexer control signals are high, and we can have one 64-bit full-word addition result generated, which datapath delay is the longest, equal to the delay of one 8-bit adder plus those of the fourteen multiplexers. In such design, all instructions are required to be completed in one cycle, making the pipeline control simpler.

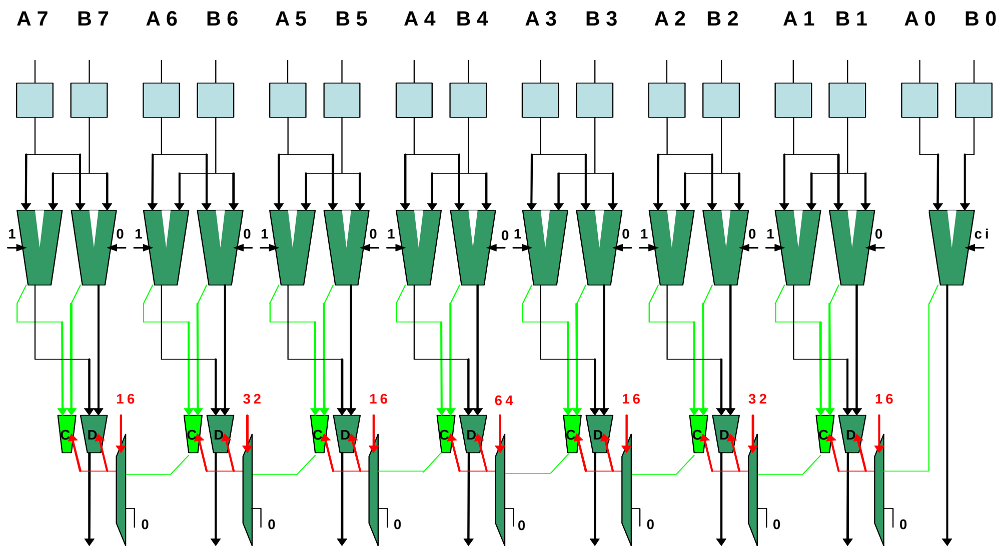


Figure 2-3. High-performance subword-parallel adder.

2.1.2 Fixed Point

Most scientific algorithms such as object rotation require floating-point operations. To implement floating-point operations by integer instructions is too slow for them. Thus, scientific requirement drives processor to integrate floating-point instructions.

Due to the limitations of human eye and ear sensitivity, some precision loss on image pixels and audio samples is acceptable. For example in a DCT algorithm, using 12-bit fixed-point to represent a cosine value is good enough for most image quality requirements. Floating-point hardware is more complex than fixed-point hardware. For cost and power efficiency, most multimedia applications use fixed-point operations. Fixed-point operation is combination of integer arithmetic operation and a shift operation. In typical integer operations, most-significant bits (MSB) are truncated when the result is overflow. The following shift operation for fixed-point cannot recover this error. For example, assume that a 32-bit fixed-point has a 16-bit decimal part and a 16-bit integer part, where a value 1.502 is represented as 0x00018083. The multiplication result 1.502×1.502=2.256≡0x00024189. By integer multiplication, 0x00018083×0x00018083 will be truncated into 0x41894309 due to the 32-bit limitation. Thus right shift the last 16 bits will derive a wrong result 0x00004189. To avoid overflow, we can right shift 8 bits on the two multiplicands before multiplication; the result will be 0x000180×0x000180 =0x00024000. This result has 2.6% loss in precision, which is not acceptable. A better solution is to right shift on the 64-bit multiplication result before writing it into a register file. It needs a specific instruction (mulshr16) which adds a shifter after the multiplier.

2.1.3 Permutation

Permutation operations are widely used in many algorithms. Datatype conversion is the basic permutation operation in many processors. Symmetric-key cryptographic algorithms such as DES and AES are based on complex permutation.

RGB components are mixed in video samples. Permutation in hardware is just a wire routing, but it is costly in software. To permute a data typically needs many shift and and/or operations.

On subword-parallel execution, moving of neighbor elements in packed register becomes a special type of permutation. For example in FFT, data should be reordered into butterfly sequences before next iteration, which becomes a permutation operation on a packed register. Thus permutation becomes more critical, it is better to offer abundant permutation instructions to reduce software effort. Each permutation instruction can be implemented by using multiplexers in an ALU which cost is much lower.

2.1.4 Saturation Arithmetic

Saturation arithmetic is useful for multimedia applications. When two image pixels or audio samples are mixed, their intensions are added. By typical integer addition, mixed white pixel will become light gray when its most significant bit (MSB) is truncated. To avoid the wrong result, software should keep the mixed intension as a maximum white value when overflow occurs. This function is called saturation arithmetic described as in the following code

```
if (a+b>255) y=255;
else y=a+b;
```

Conditional branch is an inefficient operation on RISC processor. It causes pipeline refill that wastes many cycles. While saturation arithmetic should apply on all pixels during image processing, it is better to offer specific saturation instructions to improve efficiency in multimedia applications.

2.1.5 Critical Path Analysis

Figure 2-4 shows our designed PLX chip architecture. It is a 5-stage pipelined RISC design. The IFETCH stage gets instruction from ICACHE and handles interrupt. The DECODE stage extracts operand register addresses froman instruction word. The

OPFETCH stage gets operands from the register file. The EXECUTION stage contains two units, ALU and Load/Store. The WRITE stage writes execution results into the register file.

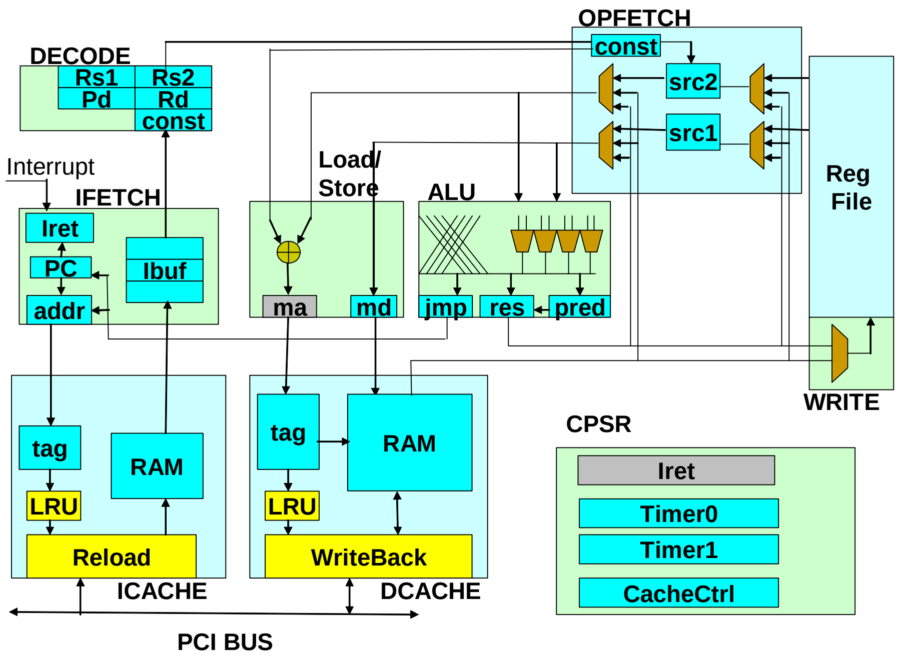


Figure 2-4. PLX chip architecture.

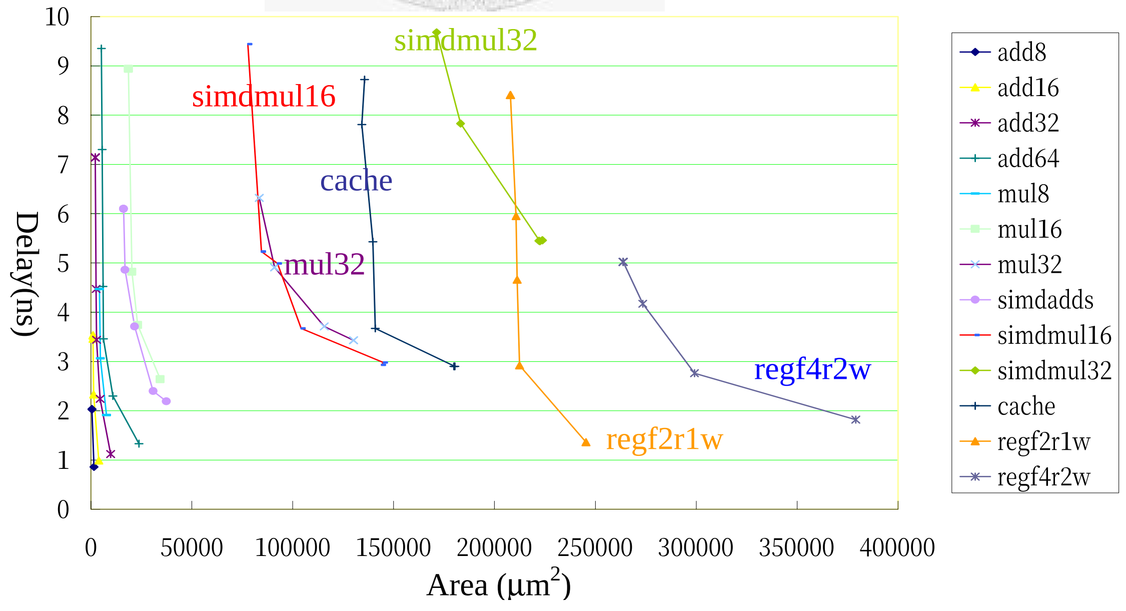


Figure 2-5. Delay-area trad-off in components.

On RISC architecture, operations in the execution stage should complete in one cycle. Figure 2-5 shows the area and speed trad-off in some components designed using TSMC 0.18 μm standard cells. Each curve represents a component designed in various structures such as carry-look-ahead or carry-select. The simdmul32 is the critical path, its minimum delay is 5.45ns. This path performs 32-bit multiplication and right shift for fixed-point, thus its delay is larger than a 32-bit multiplication. In multimedia applications, most algorithms only use 16-bit multiplication. The 32-bit multiplication is used at rate-distortion computation and some high level protocols, which are not critical for performance. To reduce critical path delay, we only implement a 16-bit multiplier, the new critical path delay is 2.93ns on simdmul16.

In Figure 2-5, we can see the area of simdmul16 and cache at 3.71ns is much lower than that at 2.93ns. For performance/area tradeoff, we implement the PLX chip at 260MHz speed.

The second critical path is in cache. A cache-hit load has 4 jobs to do: (1) generate memory address from the operand register, (2) get cache row address from the tag array, (3) get cache content from RAM, and (4) write content into register file. On a single-cycle RISC processor, only 2 cycles (EXECUTION and WRITE stages) are available for these 4 jobs. Jobs (1) and (2) should be combined. A fast 32-bit adder in Load/Store unit adds address base from operand *src2* and address offset from instruction *const* field. The generated memory address is directly sent to tag without using buffer. The path delay is 2.9ns in a 32-bit adder plus tag lookup latency. The fast 32-bit carry-look-ahead adder takes 1.12ns, and tag lookup takes 1.78ns. In designing PLX2, this path delay should be cut by SMT to execute at 520MHz.

2.2 Implementation of ME on PLX

In video encoding, motion estimation (ME) occupies more than 70% computation. This section utilizes PLX ISA to improve ME performance.

H.264 advanced video coding (AVC) is the state of the art video coding standard. The sum-of-absolute-difference (SAD) is a criterion used in block-based matching motion estimation algorithms to gauge the similarity between a given macroblock in the current frame and corresponding macroblock in a reconstructed reference frame. The displacement between these two macroblocks is a candidate motion vector. For a $K \times L$ macroblock, one has

$$SAD(m,n) = \sum_{i=0}^{K-1} \sum_{j=0}^{L-1} | C(i,\, j) - R(i+m,\, j+n) |,$$

where $C(i,\ j)$ is the luminance value of a current frame pixel and $R(i,\ j)$ is the luminance value of a reference frame pixel. Argument (m,n) is the displacement between these two blocks. The *motion vector* (*MV*) is defined as the displacement that yields the minimum SAD value:

$$MV = \arg \min_{(m,n) \in searchrange} SAD(m,n),$$

where the search range is a neighboring region in the reference frame(s) where the motion vector is to be found. For a CIF-size video frame, a search range of 16 is mostly used. By a full search strategy, the maximum number of displacements that must be evaluated will be 16^2=256.

In H.264, the 16×16 macroblock can be partitioned into 16×8, 8×16 sub-blocks, or 8×8 sub-blocks to improve the quality of motion estimation at the expense of additional computation and motion vectors. In fact, if the 8×8 block size is selected, each 8×8 block may further be decomposed into 4×8 or 8×4, or 4×4 sub-blocks. Hence, the number of MVs per macroblock ranges from 1 MV for the entire 16×16 macro-block, up to 16 MVs, each for a 4×4 sub-block. In all, 41 MVs need to be evaluated, from which a specific sub-block partition will be chosen as the optimal set of motion vectors by solving a rate-distortion optimization problem.

Instead of using a full search algorithm that may consume excessive CPU cycles without yielding significant performance benefit, we implemented a Modified Spiral Search (MSS) [9] algorithm, where the search starts from the center of the search region and moves outward in a spiral-like order. If the distortion of a point is greater than a threshold, the next few points are skipped. In all, the number of skipped search points is proportional to the distortion and displacement. A high-level implementation of the MSS algorithm is given in Code 2-1.

To implement the MSS motion estimation algorithm using PLX ISA, our strategy is to exploit the SIMD sub-word parallel instructions to reduce execution cycles of Steps d, e, and f in Code 2-1. Assume that both the 16×16 macroblocks in the current frame and in the reference frame respectively have been loaded into an on-chip data memory in a row-major ordering. As such, an entire 1×16 row of pixels of either macroblock can be loaded into the two 64-bit PLX register without incurring any overhead. Then the 4×4 SAD calculation (Sep d) can be performed efficiently using the sub-word parallel instructions.

Code 2-1. Modified spiral search algorithm.

```
Set 41 minimum SAD values to infinity;
Set 41 motion vectors to (0,0);
k=0;
while(k<(searchrange×searchrange)) begin
   a. get displacement (m,n) from lookup table;
   b. set current frame address pointer to (0,0);
   c. set reference frame address pointer to (m,n);
   d. calculate SAD of 16 4×4 partitions;
   e. calculate SAD of other 25 partitions;
   f. compare 41 SAD with the minimum SAD, and
     replace minimum SAD and motion vectors;
   g. k+=step(SAD,m,n);
end while
Select the set of motion vector(s) according to some
rate-distortion criterion.
```

While two 8-bit values are subtracted, the result precision is 9-bit, which becomes overflow on an 8-bit scalar operation unit. The loaded pixels should be extended into 16-bit by using double registers, where computation is doubled.

Motion estimation is a *losable* algorithm. We care about which MV has a minimum SAD, but the SAD value itself is not important. PLX saturation arithmetic instruction is useful to reduce SAD computation. Assume that the minimal SAD value is less than 255, then saturating the other non-minimal SADs into 255 will not change the MV search result. The MV search will miss only when the minimal SAD is greater than 255, which means no similar macroblock is found, and a bad macroblock is chosen for reference.

The modified SAD operation using saturation arithmetic becomes:

$$SAD\,(m,n) = \sum_{i=0}^{N-1} sat\,(\sum_{j=0}^{N-1} |\, sat\,(C(i,j) - R(i+m, j+n))\,|)$$

Figure 2-6 shows the process to compute 16 4×4 SADs. The basic routine is to compute 16 4×1 subresults in parallel. This routine takes 8 load, 4 saturation substract (psubs), 4 absolute (abs), and 3 saturation addition instructions, and the results are saved in a 64-bit register. This routine executes 8 times to compute all the 64 4×1 subresults. These subresults are arranged as a 4×16 matrix in 8 registers. Then the matrix is transposed for computing the 4×4 SADs. The matrix transposition takes 16 PLX permutation instructions. The 4×4 summation takes 6 instructions. Thus Step d of Code 2-1 takes 174 cycles. This result is 7.2 times faster than scalar operation, and 3.2 times faster than the process without using saturation arithmetic instruction.

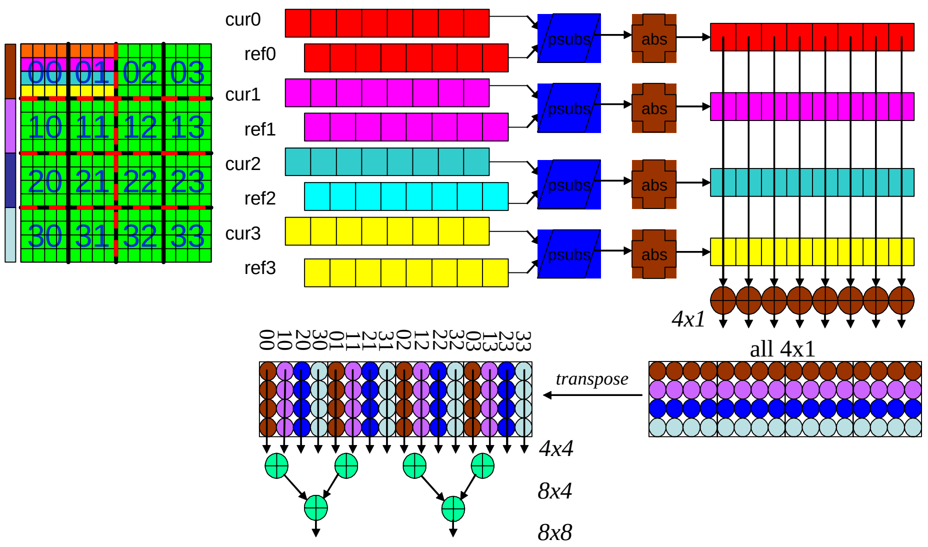


Figure 2-6. SAD computation using PLX ISA.

The 16 4×4 SADs are stored in two 64-bit registers. Step e of Code 2-1 computes the other 25 SADs, it totally takes 29 cycles by the following steps:

(1) Add every even and odd byte pair to get eight 8×4 SADs. Right shift 8-bit in a 16-bit subword mode can parallelly generate odd bytes. This step takes 6 cycles.
(2) Add the first to fourth bytes with the fifth to eighth bytes in every 64-bit register. The result is the eight 4×8 SADs. This step takes 8 cycles.
(3) Add every even and odd word pair from the results in Step 2 to get four 8×8 SADs, pack them into one register. This step takes 6 cycles.
(4) Add every even and odd 16-bit subword pair of from the results in Step 3 to get two 16×8 SADs. This step takes 2 cycles.
(5) Add first two 16-bit subwords with the last two 16-bit subwords to get two 8×16 SADs, pack the results in Step 4 into one register. This step takes 5 cycles..
(6) Add the two results in Step 4 to get a 16×16 SAD. This step takes 2 cycles.

The 41 SADs are stored in 9 registers. They are compared to the saved minimum SAD to decide MV. If any one SAD is lower than the saved minimum SAD, this new value will replace the minimum SAD and the MV be updated. A parallel-compare instruction is used for comparison. If the first operand is less than the second operand, the result subword is set to -1 (all bit be 1) , otherwise it is set to zero, as shown in the following definition:

`P[n]=(I1[n]<I2[n])?-1:0`, $0 \le n <$ subword number

Parallel-replace is performed by the following operation:

`O[n]=(P[n]&new[n])|(~P[n]&old[n])` , $0 \le n <$ subword number

Each SAD register needs one compare, one inverse, 4 logical AND, and 2 logical OR operations to update minimal SAD and MV. Step f of Code 2-1 takes 72 cycles.

Assume that the modified spiral search only searches 1/3 points, and Steps a, b, c, and g take 20 cycles, the total number of motion estimation execution cycles in a macroblock takes 256×(20+174+29+72)/3=25173 cycles. To process a CIF-size (352×288) video, the frame rate in a 260MHz processor can be 260000000/ (25173×352×288/16/16)=26 frames per second. To meet the 30-frames real-time constraint, we have to reduce the number of search points to 1/3.5, which will lower image quality. In next section, we will use other parallelization techniques to improve performance.

Table 2-1 lists the PSNR (Peak Signal-to-Noise Ratio) of H.264 reconstructed frames using 16-bit operation and 8-bit saturation operation on two sequences. The sequence *Stefan* has fast moving, and sequence *Weather* has little moving. From the results shown on the two sequences, the 8-bit saturation method is only a little bit worse than the 16-bit method, but 8-bit method is much faster than 16-bit method. That illustrates PLX ISA is useful to improve motion estimation performance.

Table 2-1. PSNRs of two SAD methods.

PSNR	*Stefan*		*Weather*	
Frame	16bit	8bit-sat	16bit	8bit-sat
1	37.35942	37.35031	37.52711	37.52356
2	37.29617	37.28674	37.71939	37.71754
3	37.25709	37.23658	37.80259	37.80131
4	37.07474	37.07342	37.81671	37.81664
5	36.99427	36.99151	37.81785	37.81560
6	36.91317	36.91058	37.79967	37.79784
7	36.92029	36.88299	37.76962	37.76885
8	36.88774	36.87031	37.74985	37.74801
9	36.81931	36.80784	37.73409	37.72860
Average	37.05802	37.04559	37.74854	37.74644

2.3 PLX2 Processor Design

On the implementation of video encoding in Section 2.2, we had encountered the following problems:

(1) Power is wasted on 32-bit scalar operations. Except SIMD operations, a multimedia application has so many 32-bit scalar operations to execute, such as Huffman decoding in H.264. Using a 64-bit processor to perform these 32-bit operations will waste power.

(2) Clock rate is restricted by a single-cycle ALU. To improve clock rate, ALU should be pipelined.

(3) Video size is restricted by processor performance. To perform video with a larger size, without increasing the clock rate, we need other parallelization techniques to improve performance and energy efficiency.

(4) Memory stall is large. The performance analysis in last section does not include memory stall, otherwise the actual frame rate is much lower.

We will describe how to utilize VLIW and SMT techniques to solve these problems in the following.

2.3.1 MAC on VLIW

In digital signal processing, *multiply-accumulation* (MAC) is the most often used operation. A finite-impulse response (FIR) filter equation is represented as

$$y_t = \sum_{i=0}^{N-1} c_i \times x_{t-i}$$

On software implementation, each pair of c_i and x_{t-i} are multiplied and accumulated into y_t. Some DSP processor such as TI TMS320C541 [10] implements MAC with automatic looping and index increase, which can process above equation as a single operation. Since MAC is composed of multiplication and addition operations, it is always the longest path in an ALU. Clock rate is restricted by MAC critical path, thus it is not chosen in PLX ISA.

MAC critical path can be reduced in a VLIW processor. VLIW is a type of instruction level parallelism (ILP) machine, which uses multiple ALUs to execute multiple instructions in one cycle. Instruction parallelism is determined by compiler.

On a 4-issue VLIW, FIR can be implemented as shown in Figure 2-7. When c_0 and x_t are multiplied in ALU2 at time 1, c_1 and x_{t-1} are loaded at the same time. The four ALUs perform as a 3-stage pipelined ALU at this example. Thus, an N-stage MAC operation just needs N+2 cycles to complete.

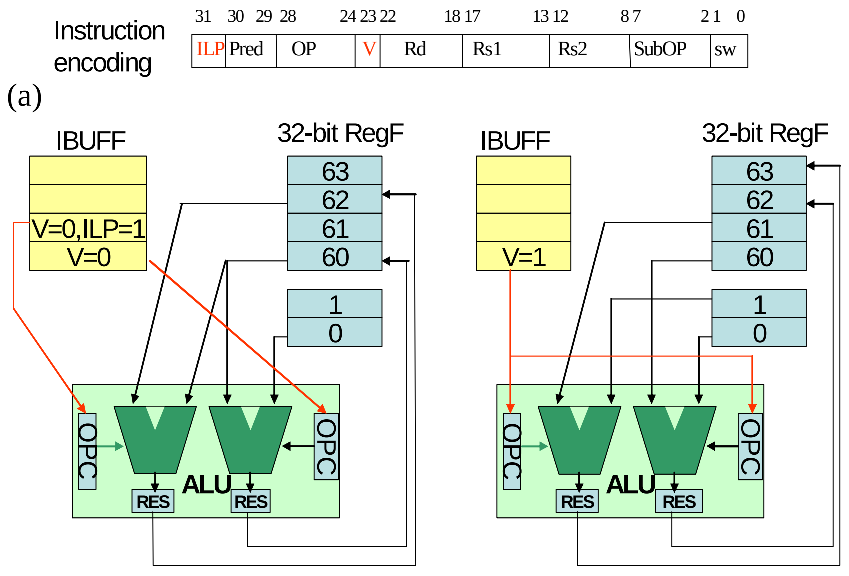
time	ALU0	ALU1	ALU2	ALU3
0	Load c_0	Load x_t		
1	Load c_1	Load x_{t-1}	MUL (0)	
2	Load c_2	Load x_{t-2}	MUL (1)	ADD (0)
3			MUL (2)	ADD (1)
4				ADD (2)

Figure 2-7. Execution of MAC on VLIW.

In a VLIW processor, all ALUs only perform the basic instruction, thus the ALU design can be as simple as a RISC. Complex special operations mus be handled in software. The power efficiency of a VLIW is worser than the processor with a MAC instruction, but it offers flexibility to implement special operations in software.

2.3.2 Reconfigurable VLIW/SIMD

In order to execute 32-bit operations on a 64-bit processor efficiently, we make a special two-issue design: an instruction-level reconfigurable VLIW/SIMD design. Figure 2-8 shows this design, where the 64-bit ALU is partitioned into two 32-bit ALUs, each controlled by a control unit OPC. In order to let this design function well, the instruction encoder and register file designs are modified.



(b) 32-bit VLIW configuration (c) 64-bit SIMD configuration

Figure 2-8. Reconfigurable VLIW/SIMD design.

An ISSUE stage is inserted between the DECODE and OPFETCH stages. It checks the *V* and *ILP* flags in an instruction word to decide how to dispatch control signals to the two OPCs. The *V* flag indicates that this instruction is a 64-bit vector operation; otherwise it is a 32-bit scalar operation. The *ILP* flag indicates that this instruction is independent to the previous instruction, thus both can be executed in parallel. This flag set by compiler helps the ISSUE stage to select instructions to form a VLIW. Without this flag, the ISSUE stage has to check the dependence of instructions in instruction buffers by itself.

Dealing with 32-bit operations, the register file should be 32-bit wide. The original 32-item 64-bit wide register file is reorganized into 64-item 32-bit wide, with four read ports and two write ports.

On the 32-bit two-issue VLIW configuration, two instructions are dispatched into two OPCs, as shown in Figure 2-8(b). The 5-bit operand field in an instruction word is mapped to register file address bit 1 to bit 5, and register file address bit 0 is set by V, the thread bank flag.

On the SIMD configuration as shown in Figure 2-8(c), the two ALUs are logically merged as a 64-bit element. The same instruction is dispatched to both OPCs such that the two ALUs will perform the same operation as an SIMD processor does. The even register port address bit 0 is set to 0, and odd register port address bit 0 is set to 1, thus two neighboring 32-bit registers are combined as a 64-bit register to serve the 64-bit operation.

All possible VLIW/SIMD configurations are listed in Table 2-2.

Table 2-2. VLIW/SIMD Configurations.

1st V	1st ILP	2nd V	2nd ILP	thread bank	ALU0 OPC	ALU1 OPC	EvenPort addr b0	OddPort addr b0
1	X	X	X	X	1st	1st	0	1
0	X	0	0	0	1st	NOP	0	No use
0	X	0	0	1	1st	NOP	1	No use
0	X	1	X	0	1st	NOP	0	No use
0	X	1	X	1	1st	NOP	1	No use
0	X	0	1	0	1st	2nd	0	0
0	X	0	1	1	1st	2nd	1	1

2.3.3 VLIW Limitation

VLIW processor implements dependence removal and operation scheduling by a compiler. The hardware cost of implementing these two techniques in a superscalar processor is high; it is not affordable for portable devices. Since operation dependences can be determined in a program code, it can be optimized by a compiler to save hardware cost. The disadvantage is that software needs re-compilation when processor micro-architecture is changed. It is not acceptable for a general-purpose desktop processor, but feasible for an embedded processor.

In designing VLIW processor, the overhead is on its register file size. As shown in Figure 2-8, the register file port number is doubled. When the number of access ports doubled, the routing area is squared as shown in Figure 2-9. That is, chip cost is increased in both area and speed. As shown in Figure 2-5, the curve `regf4r2w` depicts the delay-area relation of the 64-item 32-bit wide, 4-read 2-write register file used in PLX2. Its area is much larger than `regf2r1w`, the 32-item 64-bit wide PLX register file.

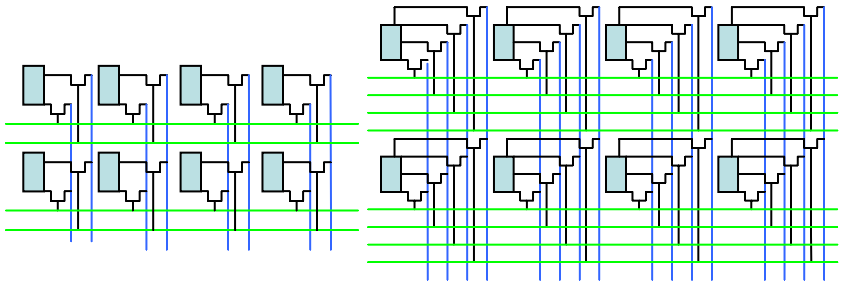


Figure 2-9. Register file.

The *bypass logic* causes another overhead. On a pipelined RISC architecture, ALU result is buffered in a temporal result register before it is written into the register file. To avoid blocking on continuous read-after-write dependent instructions, bypass logic is used to forward ALU result in a previous instruction to the ALU input port or to the operand register in a current instruction. Figure 2-10 shows the bypass logic on a 2-issue VLIW processor. Variables R2 used in S2 and R3 used in S3 were modified in previous cycle, they should be forwarded from ALU result to ALU operand input port. R1 used in S3 was modified two cycles before; it should be forwarded from ALU result to operand register. Each ALU result needs to be forwarded to all ALU input ports and operand registers. A large fan-out induces a long wire delay and thus slows down the clock rate.

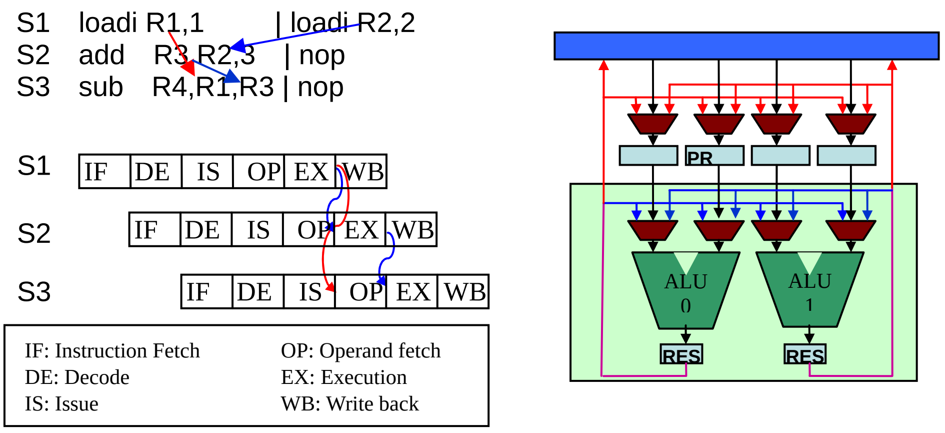


Figure 2-10. Bypass path in a 2-issue VLIW.

Control flow handling is more important for VLIW processor design. A control flow induces a conditional branch, which may cause instruction stream change. On a VLIW processor, when two ALUs generate different branches, which will be the next instruction to execute?

A simple way is to avoid packing multiple branch operations in one cycle, but it will degrade performance. The better solution is using *predication execution*, or so-called *if-conversion*. This technique changes control flow into data flow by introducing a condition expression to be the third operand. Then the instruction stream can be packed in one line. The implementation of predication requires extra flags to store the comparison result, which will be passed to ALU as the third operand.

On a pipelining architecture, branch induces pipeline re-fill that wastes computation power. On a VLIW processor, a pipeline stage contains many operations, thus the waste becomes higher. On a high-performance processor, accurate branch prediction hardware is necessary. On an embedded processor, simpler methods are used to reduce hardware complexity.

A solution is using an unbundled branch technique, which is introduced in HP PlayDoh architecture [11]. Unbundled means that the `compare` instruction and `branch` instruction are far away. It works in a way similar to delayed branch [12] but not the same. Figure 2-11 shows this technique. Figure 2-11(a) is a traditional RISC code. After the `branch` instruction S5 executed, S2 will also execute by delayed branch, then pipeline is cleared and refilled by S6. Figure 2-11(b) is a code using unbundled branch. At T=5, when the comparison operation in S1 is executed, and the compared result is true, instruction fetch stage (IF) changes to fetch from S6. At T=5, S5 is loaded in the IF stage, S6 address can be extracted from IF buffer by a simple decoder. But note that many other pipeline stages in S2 to S4 are not cleared. They should be independent instructions such that the pipeline can continue their executions without wasting time. Then, when S5 reaches T=9, the program counter (PC) is set for S6 execution without changing pipelines.

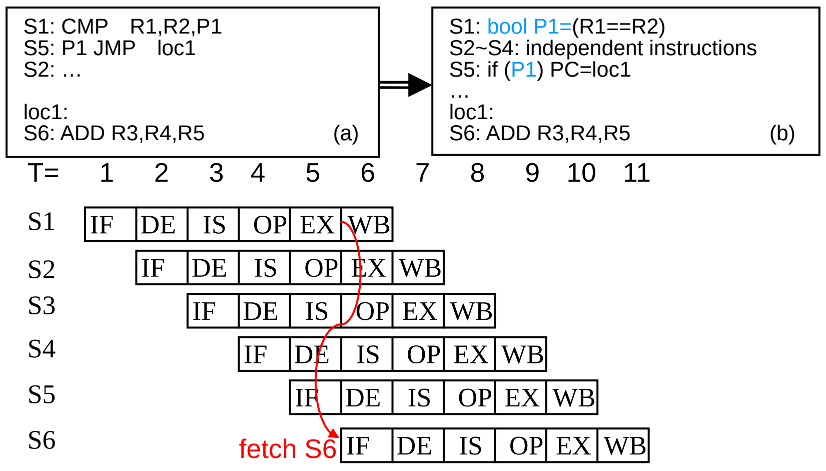


Figure 2-11. Unbundled branch

2.3.4 SMT

Threading is a way for a code to split itself into two or more concurrently (or pseudo-simultaneously) running tasks. In general, a task spends much time on waiting peripheral I/O response. Since peripheral I/O communication is much slower than the CPU speed, direct memory access (DMA) is often used to handle peripheral I/O communication. When a CPU wishes to send a message to peripheral I/O, it puts data in memory and calls DMA to transfer the data. After transmission, DMA will generate an interrupt to inform the CPU. During transmission, the CPU is idling.

Multitask OS improves CPU utilization by time-sharing. Assume that an OS has picked a task to execute once. Sometimes, when this task is waiting for I/O, or when it has been run for such a long time that a timer interrupt occurs, OS will pick another thread to execute. By multi-tasking, I/O latency is overlapped with other tasks under execution.

Context switch is an overhead for multi-tasking. When OS wishes to pick a task, it should save the current task context into memory and load the new task context from memory. The context contains register file, program status, and resource configurations. A context switch needs hundreds of cycles.

In real-time systems, some event requires fast response. For example on a communication system with a 1Gbps bandwidth and a packet size of 1Kbit, the data in buffer should be read in 1μs before next packet comes in. In time-sharing multitasks, the time to wait OS switch context may exceed real-time constraint. *Simultaneous multi-threading* (SMT) [13] can solve this problem by applying hardware-supported thread level parallelism.

The original SMT design was used to fill wide superscalar execution slots. Superscalar execution slots are often wasted by a long dependent instruction stream that exceeds its instruction buffer capacity. Instead of increasing instruction buffer size, a more efficient way is to fetch instructions from many independent threads. While a superscalar core can execute instructions out-of-order, mixed instructions from different threads will not change the execution unit design. Intel Hyper-Threading technique [14] is one example of commercial SMT implementation.

We implement SMT in another style. Figure 2-12 shows this concept, there are four threads running in time-sharing, this design can reduce ALU complexity. At the IFETCH stage, four instruction streams are kept alive simultaneously. Each instruction stream possesses its own program counter. The ISSUE stage dispatches VLIW instructions into ALU from one of available streams. If all streams are available, they are dispatched in a round-robin order.

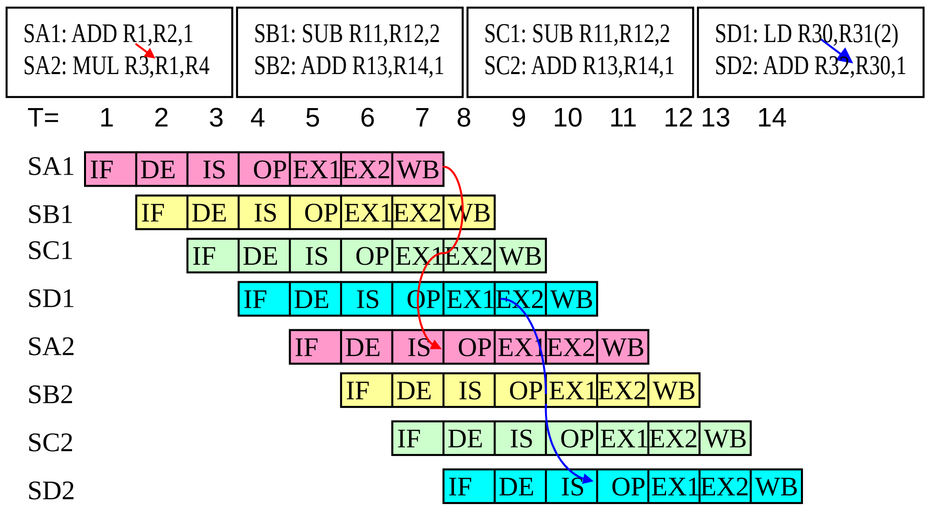

SA1: ADD R1,R2,1
SA2: MUL R3,R1,R4
SB1: SUB R11,R12,2
SB2: ADD R13,R14,1
SC1: SUB R11,R12,2
SC2: ADD R13,R14,1
SD1: LD R30,R31(2)
SD2: ADD R32,R30,1
T= 1 2 3 4 5 6 7 8 9 10 11 12 13 14
SA1 IF DE IS OP EX1 EX2 WB
SB1 IF DE IS OP EX1 EX2 WB
SC1 IF DE IS OP EX1 EX2 WB
SD1 IF DE IS OP EX1 EX2 WB
SA2 IF DE IS OP EX1 EX2 WB
SB2 IF DE IS OP EX1 EX2 WB
SC2 IF DE IS OP EX1 EX2 WB
SD2 IF DE IS OP EX1 EX2 WB

Figure 2-12. 4-thread time-sharing SMT execution.

The time-sharing SMT implementation has three major benefits: (1) fast real-time response, (2) higher clock rate, and (3) reduced memory access latency.

For an event which needs fast real-time response, a thread can be used to monitor the event status, while other threads are performing the main computation. On a 4-thread SMT processor, the CPU utilization is 75%. An aggressive design can let this monitor thread idle, and set it be woken up by a specified event without generating an interrupt. The ISSUE stage will not dispatch instruction from idle thread, thus CPU utilization can be 100% occupied by the other threads. When the event occurs, the instruction next to idle is dispatched. While this instruction is already in instruction buffer, the response time is only one cycle.

In Figure 2-12, instruction SA2 uses variable R1 which was modified in SA1. In single cycle RISC execution, SA1 result should be forwarded to SA2 by a bypass logic as described in Figure 2-9. But since SA2 will use R1 at 4 cycles after SA1's execution, the time is enough for R1 being written into register file, thus the bypass logic is not needed, which saves wire delay and area. Moreover, ALU can be divided into two pipeline stages. The 4-thread SMT now can work as four logical processors, each has a 4-cycle execution. The critical path from the operand fetch, the execution, to the write back stages has 4 cycles to wait. Thus ALU is given two cycles for execution, shown as the EX1 and EX2 stages in Figure 2-12. The multiplier can be pipelined into two stages, thus improving clock rate. When more physical threads are used, more complex operations such as MAC can be implemented without worrying about the critical path delay.

Multi-cycle execution simplifies function unit design. A typical double-precision floating-point function unit should be divided into four pipelining stages to balance its critical path delay the same as an integer function unit. When a program is mixed with integer and floating point instructions, integer function unit is often stalled to wait the floating point be ready. When integer and floating function units have the same pipeline stages, instruction scheduling becomes easier.

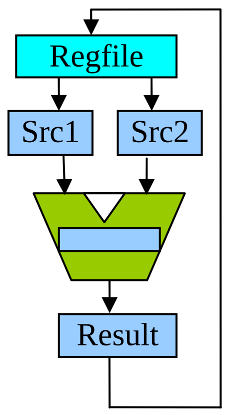
Memory access latency can be hidden under multi-threaded execution. When a cache-miss occurs, the thread enters idle. But the ISSUE stage can continue to dispatch instructions from other threads. On a computation dominant case, memory access latency can be fully overlapped. On the motion estimation example, the memory bandwidth required for a CIF with a rate of 30 frames per second is 6 MB/s. The bandwidth of a 32-bit 266-MHz, 3-cycle latency, and 8-burst double data rate (DDR) SDRAM is 304 MB/s. The required bandwidth is much lower than the SDRAM bandwidth, thus the memory access latency can be fully hidden.

2.3.5 Power Efficiency Consideration

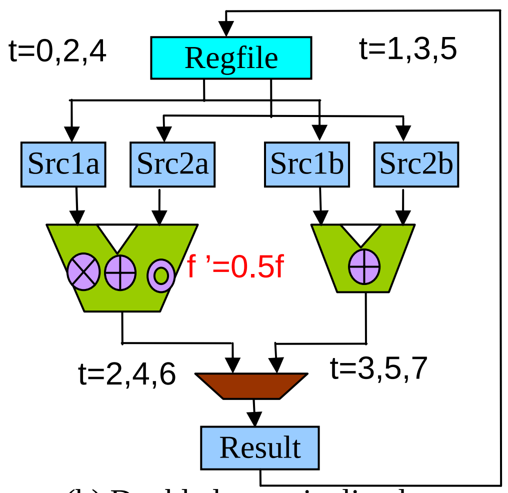
Combination of SMT and SWP-SIMD features, the ALU power can be reduced. In micro-architecture view, a pipelined ALU is equivalent to two non-pipelined ALUs working in an interleaving way. Figure 2-12 shows such two circuits.

Figure 2-13(a) shows a two-stage pipelined ALU. On designing a pipelined ALU, it is difficult to evenly dispatch a critical path delay into two parts, thus the clock rate is unable to be doubled. Figure 2-13(b) shows a two-ALU design, the left ALU works in even time and the right one works in odd time. Each ALU uses two cycles for execution, and the clock rate can be really doubled. On PLX, the ALU critical path is on the 16-bit multiplier which restricts clock rate to 260MHz. Using the circuit as shown in Figure 2-13(b), the clock rate can be speeded up to 520MHz.

Though the ALU area is doubled, it can be reduced. In multimedia applications, 80% of operations are 8- or 16-bit subword-parallel additions and comparisons. The path delay of a 16-bit addition is lower than half of a 16-bit multiplication. As shown in Figure 2-13(b), only low-precision addition/comparison operations are implemented on the right ALU, which area occupies only 1/8 of the original ALU. At the ISSUE stage, low-precision instructions are all dispatched to the right ALU, it works as single-cycle ALU run at 520MHz. The left ALU is idle without consuming power. This design is an alternate power-aware design compared to Figure 2-2.



(a) Pipelined



(b) Doubled non-pipelined

Figure 2-13 Equivalent ALU micro-architecture.

2.3.6 PLX2 Performance

Figure 2-14 shows the PLX2 chip architecture. It uses two processor cores, each is a 4-thread SMT, 2-issue VLIW design. In IFETCH, 4 threads of instructions are read from ICACHE and saved in their own IBUFF. Each core contains 4 ALUs, two smaller and two larger. The smaller ALU includes only 8- and 16-bit additions/comparisons that can be executed in a single cycle. The larger ALU contains multiplier that needs two cycles to execute. In DCACHE, tag search result is buffered in *row* register to reduce cache-hit critical path latency. The two cores share their ICACHE and DCACHE. Only the owner core can update the tag and RAM content, but another core can lookup the tag and read content from RAM. The shared-cache design reduces cache capacity requirement and simplifies cache coherence.

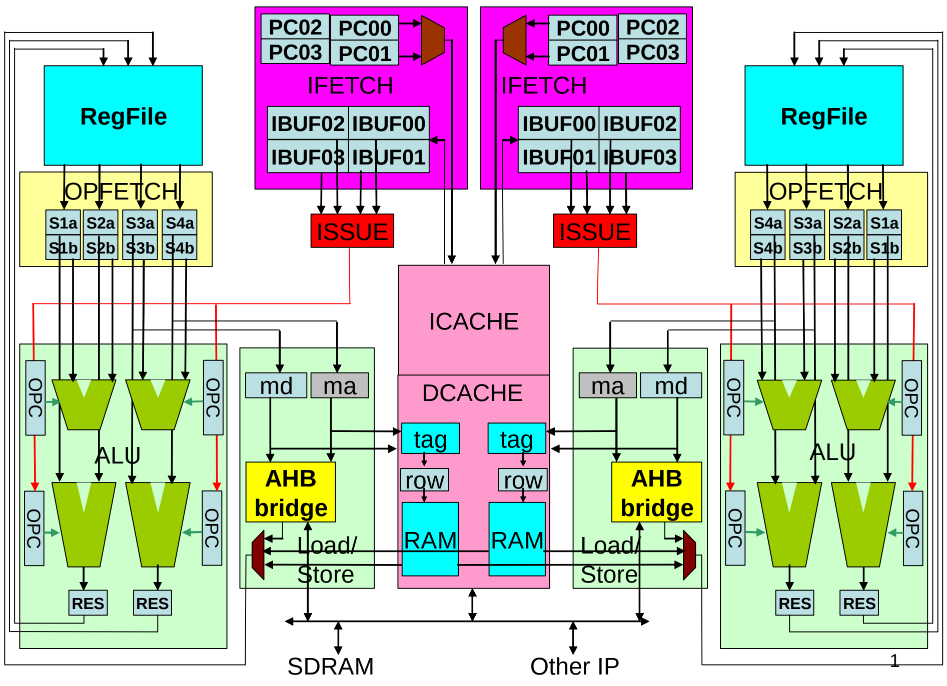


Figure 2-14 PLX2 chip architecture

Table 2-3 lists the area of PLX and PLX2 components for comparison. The designs use TSMC 0.18 μm standard cells. The DCACHE area is mainly occupied by the tag array, and the data RAM area is not counted. Each of the 4 threads in IFETCH has its own PC and a 4-item IBUFF. ALU is 32-bit in PLX2 and 64-bit in PLX. The ALU in PLX2 does not use bypass logic, so it is half smaller than the PLX ALU. The register file in PLX2 uses double access ports, and its working speed is double of PLX, and its area is larger than the PLX register file.

Table 2-3 Component areas of PLX and PLX2.

Component	PLX (mm^2)	N	PLX2 (mm^2)	N	Component	PLX (mm^2)	N	PLX2 (mm^2)	N
ICACHE	0.070	1	0.070	2	OPFETCH	0.007	1	0.014	2
DCACHE	0.140	1	0.140	2	LoadStore	0.003	1	0.003	2
IFETCH	0.013	1	0.054	2	SmallALU	-	0	0.009	4
Dec/Issue	0.003	1	0.005	2	LargeALU	0.146	1	0.063	4
OPC	0.002	1	0.002	8	RegFile	0.212	1	0.365	2

Table 2-4 lists comparisons on the execution cycle, the current of some algorithms, and the cost of PLX and PLX2. Current is obtained from FastSpice simulation results. The external SDRAM, cache RAM and I/O pad are excluded in current and area computation. PLX frequency is 260MHz and PLX2 is 520 MHz.

To compare efficiency, we use a cost function of energy-delay-area product. The lower the cost the better the efficiency. The equation is listed as follows:

Execution delay: $t = cycles/freq\ (MHz)$

Energy consumption: $E = current(A) \times 1.8(V) \times t$

Cost function $= E \times t \times area$

$= (cycles^2 \times area \times current \times 1.8)/freq^2$

The algorithm motion estimation (ME) on PLX2 implementation is partitioned into 8 threads to fully utilize PLX2 dual-core resource, where each thread is used to compute one search point.

The Huffman variable length decoding (VLD) is a 32-bit sequential algorithm, which can be improved by VLIW. Motion compensation (MC) is a memory access dominant algorithm with few computation. On PLX2, these two algorithms are implemented as two threads of a core. The memory access latency of MC can be overlapped with VLD computation.

The algorithm YUV reads an image in a YUV420 format from memory, converts it into an RGB format, and writes it back to memory. It can be divided into 8 threads, but the computation is not heavy enough to hide memory access latency.

The algorithm FIR2 is a 2-D spatial image filter. By the cache capacity limitation, many pixels should be loaded in groups many times. Using multi-threading can reduce the memory access redundancy thus improve efficiency.

The algorithm RSA computes modular multiplication using Montgomery algorithm [15] which only uses addition and shift operations. It is a sequential algorithm with heavy computation and no cache-miss. The 1024-bit addition uses 32 32-bit additions with carry propagation. On PLX2 without bypass logic, a dependent addition has to wait 3 cycles, thus the execution time is much longer than PLX.

On cost computation, the one-core *area* is used for ME, FIR2 and RSA. In general, for an algorithm that can be parallelized by VLIW and SMT, and which memory latency can be overlapped with computation, PLX2 has better efficiency. For a sequential algorithm or memory dominant algorithm, PLX is better.

Table 2-4. Cost comparison.

Algorithm	PLX cycles	Current (A)	Cost	PLX2 Cycles	Current (A)	Cost
ME	9990684	0.042	6.653×10^7	4984254	0.163	4.329×10^7
VLD	9624	0.035	5.255×10^1	6324	0.081	1.732×10^1
MC	1820	0.021				
YUV	382761	0.039	9.068×10^4	271341	0.131	1.031×10^5
FIR2	38812	0.075	1.793×10^3	27168	0.143	5.642×10^2
RSA	109568	0.038	7.240×10^3	404480	0.019	1.662×10^4

CHAPTER THREE
SYSTEM LEVEL DESIGN AND VERIFICATION

Dual-processor architecture is widely used in modern handheld embedded systems such as smart phone. It typically contains a DSP to handle baseband and image computation, and a 32-bit CPU to handle peripheral function blocks and general-purpose processing. On a smart phone, many function blocks need processors to handle: the Real-Time Clock (RTC) for scheduling; LCD Controllers for display; Keyboard, Digitizer and Touch Panel Controllers for human interface input; Voice Codec for speaker and microphone; Baseband Codec connected to a radio frequency (RF) front-end for wireless communication; GPS for navigation; SD card interface for storage; Smartcard Controller for authentication; USB/UART/I2C to communicate with other systems; and an H.264 hardware accelerator to assist video processing. All these function blocks are real-time functions, and are connected by a bus hierarchy. These function blocks are typically built as ASIC components for power or performance issue.

With the progress of technology, the ASIP power and performance are now competitive with ASIC, such that some of the above function blocks can now be replaced by programmable ASIP. Therefore, multi-processor system on chip (MPSoC) with ASIPs inside becomes feasible for handheld devices.

Baseband is a candidate to move into software. In near future, smart phone will be required to support many wireless communication standards, including GSM, GPRS, W-CDMA, 802.11 a/b/g, Bluetooth, WiMAX, GPS, DVB-T, and so on. To support so many standards, *Software Defined Radio* (SDR) brings programmable and dynamically reconfigurable method to implement the multiple-standard communication systems. A wide-band zero-IF receiver is used in modern wireless devices to down convert wireless signal directly to data frequency, and do all baseband processing in software.

3.1 Memory Sharing

In a multi-processor system, inter-processor communication bandwidth dominates system performance. Processors are connected as a network. In an off-chip interconnection, the number of links is determined by the pin-out limitation. Many interconnection topologies such as hyper-cube or token-ring have been introduced to deal with the interconnection cost and broadcast efficiency trade-off. In an on-chip interconnection, this limitation is not critical. The topology choice is determined by the memory sharing strategies.

Multi-processor has four memory sharing strategies to choose. Figure 3-1(a) shows a distributed system where memories in processors are not shared. Data transfer should pass through a message-passing channel. Figure 3-1(b) shows a shared-address local memory system where each memory is assigned a unique address and connected to a bus. Data can be directly transferred between memories through DMA. Figure 3-1(c) depicts a private cache system, where each processor has its private L1 cache. When a processor wishes to transfer data to another processor, the producer should write data into shared L2 cache for consumer to read. Figure 3-1(d) shows a distributed shared cache system, where L2 caches are distributed in processors. When an L1 cache miss occurs, cache coherence logic forwards the memory request to the L2 cache that owns the data.

The choice of memory sharing strategy is a trade-off between hardware cost, communication bandwidth and software design effort. A hardware cache is used to buffer recently used data to reduce external memory access latency. To implement a cache, except the storage memory, it needs large tag array and cache coherence logic. Whenever a processor writes data to shared memory, cache coherence logic should broadcast this information to all other processors to synchronize their cache content [16], which will occupy large communication bandwidth.

Without a hardware cache, the chip area and communication bandwidth can be reduced. Software should correctly buffer the frequently used data to avoid reloading it from external memory to reduce system performance. It is a heavy loading for software programmer, especially when a complex data structure is used. Some modern processor such as IBM Cell processor [17] performs data pre-fetch prediction in compiler to reduce software programmer effort.

Our PLX is a newly-designed processor, and its compiler is under development. We make an OpenMP to TLM tool to analyze data sharing and data reuse from OpenMP code, which will be discussed in Section 3.4. Currently this tool is only able to analyze array, and cannot correctly handle pointer and large data structure. As a tradeoff, we had implemented a hardware cache to buffer local variables which can reduce programmer's effort, and used the OpenMP to TLM tool to handle global shared-data. The cache coherence is determined in this tool, thus hardware cache coherence logic and bandwidth can be saved.

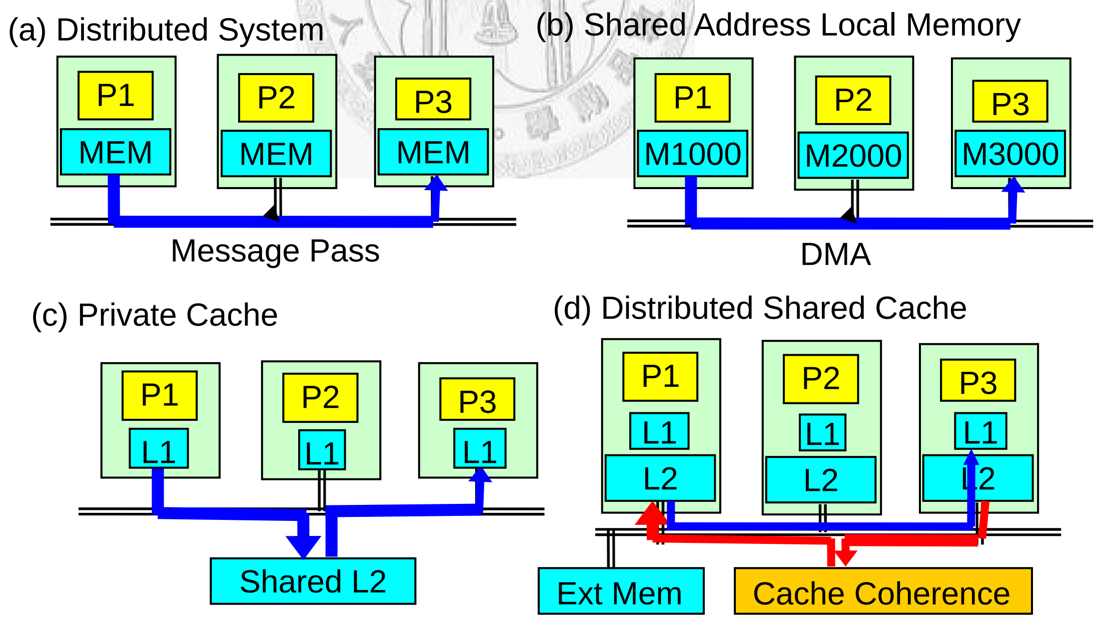


Figure 3-1. Memory sharing strategies.

3.2 Message Passing over Private Cache

The computation of multimedia application can be largely speeded up by PLX ISA as shown in Chapter 2. Thus, data transfer becomes the major part of a program. The inter-processor communication bandwidth always dominates system performance. As shown in Figure 3-1, data transfer on a hardware cache system needs two communications. In order to reduce the communication bandwidth, we modify the cache design to allow a message be directly transferred to another processor without going through the shared L2 cache.

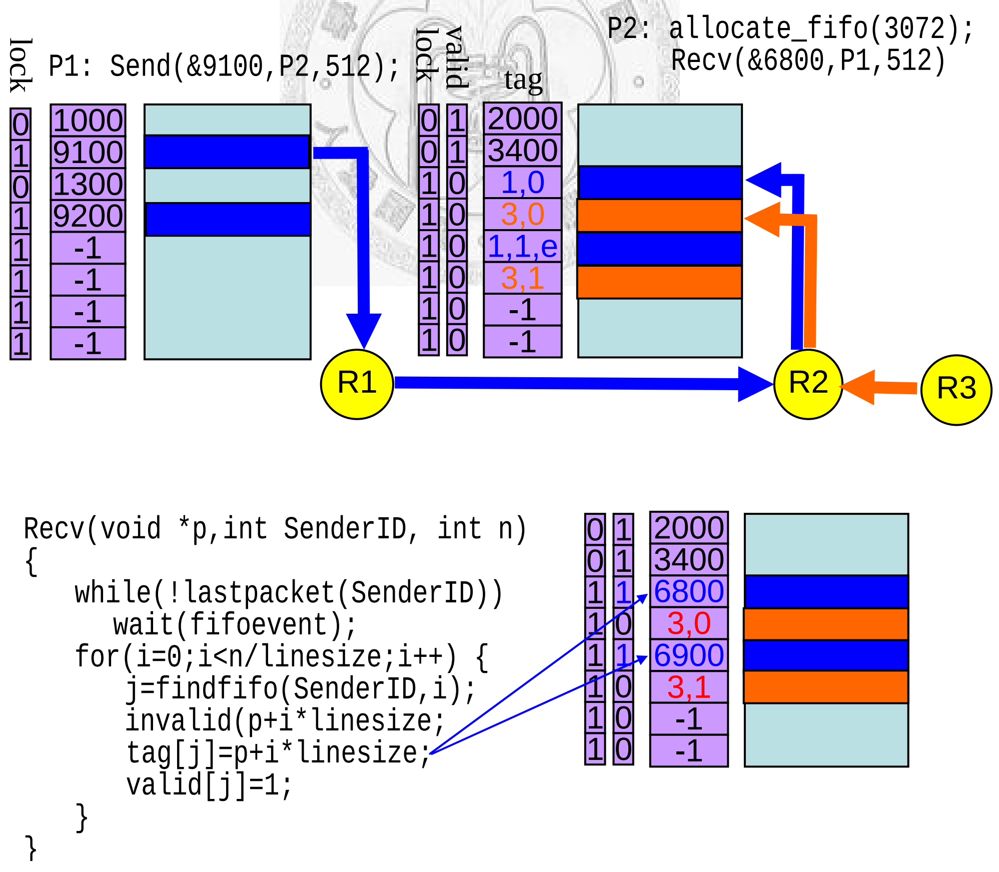
On a typical message-passing interface (MPI), the producer specifies a memory address and calls a `Send` function to perform the transaction. The data is packed into a packet with a sender ID and a receiver ID in the packet header. Router forwards a packet by these IDs. When the packet arrives at the consumer first-in first-out (FIFO) buffer, a `Recv` function moves it to the target memory.

On a private cache system, when the producer specifies a data in a memory address to `SEND`, the memory content may not be in the cache. Thus a cache-miss read occurs, and the producer has to load it from main memory into cache. The consumer should also allocate a space in cache to receive the packet. If all cache lines are dirty, a cache-miss write occurs. We need to flush a cache line into main memory to generate a space. Thus, the communication is triple, and the bandwidth used is larger than the method with a shared L2 cache. And the packet should be buffered in a router to wait for the cache-miss being over. It blocks other transactions from passing through the router.

For the MPI to work efficiently, at the producer side the data should be *locked* in a cache to guarantee no cache-miss read occurs, and at the consumer side the cache should serve as a FIFO to receive data in the packet.

Our implementation as shown in Figure 3-2 utilizes a cache tag array, where a cache is logically partitioned into many cache lines and each cache line has a tag to indicate its physical address. Two flags are used in each cache line. The *lock* flag disables the cache line to be swapped-out, thus the memory content can stay in cache.

The *valid* flag set to one indicates that the cache line is buffering a memory data; otherwise, the cache line can be used as a FIFO. That is, when the cache line is not valid (set to 0), its tag can be used to save the incoming packet header sequentially as a FIFO. As shown in Figure 3-2, processor P1 wishes to send 512 bytes from address 9100 to P2. It should lock the cache lines before sending. The 512-byte data is partitioned into two packets to fit the cache line size. In processor P2, 3072 bytes of cache spaces are allocated to form a FIFO before any communication. The tag in the FIFO cache line if set to -1 indicates that it is empty. When a packet is received, its header is sequentially saved in an empty cache line tag. If another processor sends a packet at the same time, these two packets will be saved in an interleaving way. When the last packet is received, an event is triggered to wakeup the `Recv` function. Instead of copying data from FIFO to the target memory, our design can directly set the target address point to a tag to save the memory copy time.



```
Recv(void *p,int SenderID, int n)
{
    while(!lastpacket(SenderID))
      wait(fifoevent);
    for(i=0;i<n/linesize;i++) {
        j=findfifo(SenderID,i);
        invalid(p+i*linesize;
        tag[j]=p+i*linesize;
        valid[j]=1;
    }
}
```

Figure 3-2. Message passing over private cache.

3.3 TLM

Network-on-Chip (NoC) [18] is becoming an important research topic for future large scale chips. NoC is connected by routers. A typical NoC router has 5 ports connected to its 4 (east, south, west, and north) neighboring routers and to an attached processor. Each processor is attached to a router.

By network propagation latency, router micro-architecture can be differentiated as static routing or dynamic routing. In static routing, resources on the routing path are allocated before transaction. Message is directly forwarded to destination with only one cycle of latency on each router. IBM Cell processor works in a static routing style. In dynamic routing, the router buffers a full packet instead of a word. Intel tera-flops processor [19] works in a dynamic routing way. For research purpose, MIT RAW processor [20] implements both static and dynamic routing channels on chip.

In dynamic routing, routing resource is not initially allocated, thus a packet may stay in a router for a long time when its outgoing port is occupied by other transaction. Deadlock possibly occurs when many transactions are waiting each other to release resource. Networking strategy becomes the key factor in system performance rather than processor core speed. Many routing algorithms had been introduced for supercomputer and NoC. A wormhole routing [21] requires less buffer, because a large packet is cut into many smaller FLITs (FLow control unIT). Router begins to transmit when the first flit is received instead of buffering flits to form a full packet. Deadlock free routing [22], congestion avoidance routing [23], and flow maximization routing [24][25] are more aggressive algorithms for specific applications. In a heavy communication system, the latency varies according to its routing algorithms and applications.

On a large chip, clock skew becomes so large that EDA tools cannot guarantee signal integrity all over the chip. Globally asynchronous locally synchronous (GALS) technique [26] is introduced to solve the problem. Each processor and router work on their own clock. Router communicates asynchronously to its neighbors. Many asynchronous connection protocols had been introduced for such GALS communication, including the two-phase, four-phase, dual-rail, and current-mode protocols. Dual-rail design uses redundant lines to encode data such that the receiver can check whether all bits are stable, thus the communication is delay-insensitive [27][28]. In a current-mode design [29], signal uses a lower voltage swing to save power. In a two-phase design, the sender changes a *req* signal and sends data at the same time; the receiver has to latch data whenever the *req* change is detected. It causes a risk when the *req* transmission is faster than the data. Using four-phase protocol is safer. Sender should set up data no later than the *req* rise. Receiver raises an *ack* after the *req* rise is detected. Receiver latches data when *req* falls. While sender is locally synchronous, data setup time needs at least one cycle. The maximum throughput is two cycles per data.

Network congestion is a main concern on multi-processor performance. A task will be stalled when its required data is blocked en route. Many static task scheduling algorithms [30] [31] had been introduced to maximize resource utilization. In these static task scheduling algorithms, all data are assumed to be received at task beginning, and be sent at task ending. Actually, this assumption is not correct, communication may occur during task execution. Thus, the real-time constraint should be verified after task scheduling.

The NoC transactions come from three ways:

(1) Synchronization by `pthread_create` and `pthread_join`.

(2) Core-to-core message passing.

(3) Cache misses.

To simulate communication between multi-processors, using a cycle-accurate model will spend too much time. *Transaction Level Modeling* (TLM) offers the ability to simulate C source code on an abstracted hardware description.

TLM is developed for architecture level design and exploration. Literally a transaction is the exchange of goods, services or funds; or a communicative action or activity involving two parties or things that reciprocally affect or influence each other. Both meanings have two ingredients, exchange/communication and goods/influence.

In an electronic system, the goods or influence can be considered as the computation or the effect of the computation. There are many discussions regarding TLM over the years, and the definitions, terminologies and libraries had been developed by the OSCI TLM Working Group (TLM WG) [32].

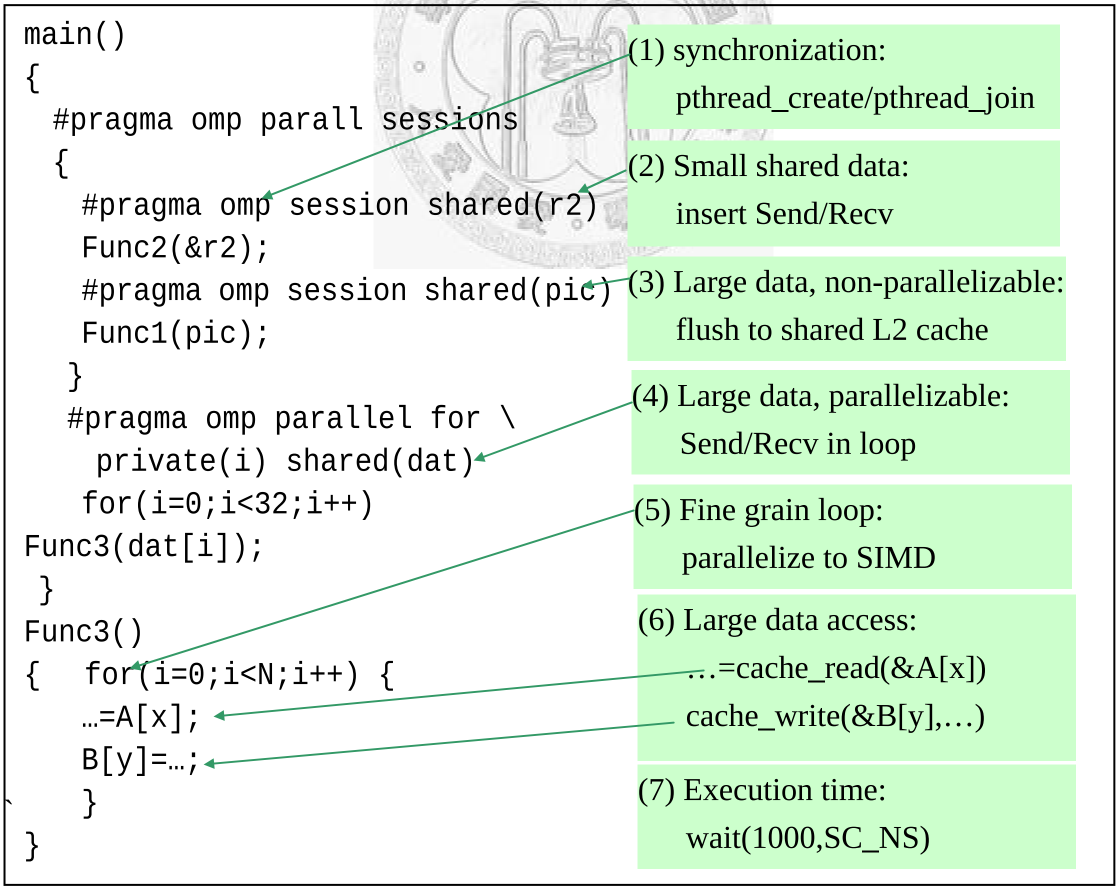
SystemC is an executable and integrating language for representing a design at abstraction levels above RTL. SystemC provides a number of datatypes that are useful for hardware design. These datatypes are implemented in C++ classes. To simulate a large design such as an MPSoC on RTL will spend days or months. TLM using SystemC [33] is becoming a standard for communication verification.

3.4 OpenMP to TLM

Many languages and tools had been introduced to simplify multi-processor programming, such as Message Passing Interface (MPI) [34], Portable Operating System Interface (POSIX) pthread [35], OpenMP [36], and StreamIt [37]. MPI is mostly used in a distributed system which describes communications between processors explicitly. OpenMP is suitable for shared-memory programming. Compiler often transforms OpenMP code into POSIX pthread, by inserting thread creation, synchronization, and memory management codes from OpenMP directives.

As described in Section 3.2, we wish to handle inter-processor communication explicitly in software. Thus the OpenMP shared-memory code should be converted into MPI-like code to describe transaction explicitly.

Figure 3-3 shows the works in a tool that converts an OpenMP code into a POSIX pthread code. Work (1) inserts `pthread_create` and `pthread_join` for processor synchronization from `#pragma omp` directives, and creates a thread body for each thread. There are three ways to perform data sharing. If the shared data size is small, we can directly pass it to the target processor using the method described in Section 3.2, Work (2) inserts a `Send` and a `Recv` function calls in the caller function and the created thread body respectively. If the shared data is large, but can be partitioned into smaller independent blocks, the `Send` and `Recv` are inserted in a loop to perform sequentially as in Work (4). A large non-parallelizable data, as shown in Work (3), can only pass through shared L2 cache, and a cache flush code is inserted. We should specify a processor ID for a thread to execute. A profiling with a given test pattern is performed to get the ID of each loaded function. Work (5) performs fine-grain parallelization on a loop to convert it into SIMD instructions to improve performance and get more correct function loading information. Parallelization will be described in Chapter 4 in more detail. The converted code is verified on a SystemC TLM platform. In a function with a large number of data accesses, cache-miss possibly dominates network bandwidth. A memory read is converted into a `cache_read` function and a memory write is converted into a `cache_write` function in Work (6) to integrate cache-miss transactions into verification. As shown in Work (7), the function execution time obtained by profiling is inserted at the end of every function to emulate the computation loading on our SystemC TLM platform.



```
main()
{
  #pragma omp parall sessions
  {
    #pragma omp session shared(r2)
    Func2(&r2);
    #pragma omp session shared(pic)
    Func1(pic);
   }
   #pragma omp parallel for \
     private(i) shared(dat)
    for(i=0;i<32;i++)
Func3(dat[i]);
 }
Func3()
{   for(i=0;i<N;i++) {
    …=A[x];
    B[y]=…;
`     }
}
```

Figure 3-3. OpenMP to TLM

In the following, we use an example to show above works and the SystemC TLM verification. Code 3-1 is an OpenMP code. Code 3-2 is its converted pthread code. Code 3-3 is the SystemC TLM platform.

The benefit of SystemC TLM implementation is that the code can run in its original style, and the timing information can obtain from this SystemC platform.

Each PLX2 contains 4 physical threads. In software view, the four physical threads work as four logical processors sharing a cache. Logical processor is the unit for thread creation and message passing. In Code 3-2, each physical thread is assigned a logical processor ID, it is combined of a processor ID and a physical thread ID. For example in Code 3-2, value LP4 means the thread is created at physical thread 0 (0=(4%4)) of processor 1 (1=(4/4)). Each logical processor can read its private status `LPID` to locate its position. In following example, the system has 4 processors, thus `LPID` value is from LP0 to LP15.

In Code 3-1, `Func3` occupies most computation. We allocate eight logical processors (`LP8` to `LP15`) for `Func3`. The other two functions `Func1` and `Func2` each uses one logical processor (LP4 and LP5).

In Code 3-2, thread body `ThreadA`, `ThreadB`, and `ThreadC` are created. Instructions under `#pragma omp` are moved into thread bodies. `Send`, `Recv`, `fifoalloc, cacheflush` and `cacheinvalid` functions are inserted in related thread body by the conversion of shared data.

At system startup, all logical processors fetch an instruction from memory address 0. All logical processors except `LP0` goes to `IDLE` soon by checking if its `LPID` is not 0, as shown by the first instruction in function `main`. Only `LP0` continues to execute the other `main` functions.

The function `pthread_create` sends a `FORK` packet to the target logical processor. The receiver hardware generates an interrupt whenever a control packet is received; target logical processor will save its current program counter (next to `IDLE`) and switches to an interrupt handler. When the function execution finishes, a `JOIN` packet is replied to the sender, then program returns to an `IDLE` status again.

In OpenMP coding, whether a variable is shared by or private to a thread should be carefully assigned in a clause list. A variable declared as private assumes that a thread initializes the variable and no other threads use it. A local variable will be added in the thread to replace the variable in the original code.

A main difference between an MPSoC and a multi-processor supercomputer is the limited local storage capacity. Cell processor has 256KB, RAW processor has 32KB, and Intel tera-flops has only 3KB memory in a core. A CIF picture size is 300KB, it exceeds all the MPSoC's local storage capacity. Thus large data cannot be directly transferred core-to-core, they should be stored in a shared memory and loaded into the target core by a cache-miss mechanism.

In Code 3-2, 32 `mydata` blocks need to be computed by `Func3`, each logical processor works on 4 blocks. Instead of transferring 4 blocks at the function beginning, `ThreadC` sequentially processes these 4 blocks, thus reduces the needed FIFO size.

`Func1` is a large data example. The array `pic` is too large for core-to-core transfer. Array `pic` is allocated by function `main`, `LP4` does not know its address at compile time. Thus its address should be sent to `LP4,` for `ThreadA` knows where to fetch data. Before `ThreadA` is created, `pic` should flush into a shared L2 cache for other core to use. While a partial data may be swapped in/out many times, the communication induced by cache-miss is dependent to the algorithm behavior. In `Func1` of Code 3-2, all memory access codes are replaced by functions `cacheread` and `cachewrite` to emulate the behavior in TLM.

Each logical processor has its own private resource, such as the `LPID` status. When two threads read their LPID, they should get a different value. We use C++ object to implement a private access. In Code 3-3, class `LogicProcessor` includes Code 3-2, thus the original C code can execute correctly without further manual modification.

Code 3-3 implements the MPSoC TLM platform. The processor and physical thread numbers are configurable by defining `ProcessorNum` and `ThreadNum`.

Processors are connected as a 2-D mesh by routers. An I/O controller and a shared-memory are included in the platform.

Routing algorithm is important to improve communication efficiency. Best routing algorithm is dependent on traffic style. On this example, we implement an X-Y routing algorithm. A flit has a one-word header and 8-word data. Transferring one word needs 2 cycles by using a 4-phase GALS channel as described in Section 3.3. Router buffers a flit before forwarding it to next router. Control packet has higher priority. Other packet priority is decided by its age, the time that it remains in router. Thus, the maximum latency of a flit in a router is 90 cycles, and the minimum latency of a packet is 18 cycles multiplied by the communication distance.

On SystemC modeling, components are implemented as module (SC_MODULE). Processors and routers are declared as SC_MODULE. The function `threadcall` in `Processor` module initializes the `main` function in every physical threads. In SystemC, all threads work in parallel, thus resource conflict such as two threads want to send their packets at a time will occur. In Code 3-3, variable `sendlock` in `Processor` module is used to perform mutual exclusion. If the `Sender` hardware is occupied by one thread, the other `Send` functions are stalled until a `sendevent` is issued when the transfer is complete. The `Receiver` function in `Processor` module checks the `cmd` field in a packet to decide what to do when the packet is received. A `FORK` packet will induce an interrupt to call the specified function. A `JOIN` packet will clear the `threadwait` flag in `LogicalProcessor` class. A `DCACHEACK` packet will refresh cache-miss content. And an `MSG` packet will update FIFO.

Delay is inserted in `Sender` and `Route` by wait function to emulate communication latency. Computation delay is inserted in Func3. The SystemC simulation result is at the approximate-time (AT) level.

Code 3-1. Example OpenMP code.

```
main()
{
  int r2;
  char *pic=new char[352*288];
  //run OS only on LP0
  if (LPID!=0) {
     while(1)  IDLE();
  }
  #pragma omp parall sessions
  {
    #pragma omp session shared(pic)
    Func1(pic);
    #pragma omp session shared(r2)
    Func2(&r2);
  }
  //sequential code
}
Func1(char *pic)
{
  //induce cache-miss
  …=pic[x];
  pic[y]=…
}
Func2(int *r2)
{
  int i;
  struct mydata dat[32];
  //setup, sequential
  for(i=0;i<32;i++) dat[i].a=…;
  #pragma omp parallel for \
   private(i) shared(dat)
   for(i=0;i<32;i++) func3(dat[i]);
}
Func3(struct mydata *blk)
{
  //heavy computations
  …
}
```

Code 3-2. Converted pthread code.

```
main()
{
  int i,r1,r2;
  char *pic=new char[352*288];
  pthread_t tid[2];
  if (LPID!=0) {
     while(1)  IDLE();
  }
  fifoalloc(3072);
  cacheflush(pic,352*288);
  cacheinvalid(pic,352*288);
  pthread_create(&tid[0], LP4, id_ThreadA, 0);
  pthread_create(&tid[1], LP5, id_ThreadB, 0);
  Send(&pic,LP4,4);
  Send(&r2,LP5,4);
  Recv(&r2,LP5,4);
  for(i=0;i<2;i++)    pthread_join(tid[i]);
  cacheinvalid(pic,352*288);
  //sequential code
}
void ThreadA(void *param)
{
  int *pic;
  fifoalloc(4);
  Recv(&pic,LP0,4);
  cacheinvalid(pic,352*288); //reload pic
  Func1(pic);
  cacheflush(pic,352*288); //write result to L2 cache
  fifofree();
}

Func1(char *pic)
{
  //induce cache-miss
   …=cacheread(&pic[0]);
   cachewrite(&pic[1],…)
}
```

```
void ThreadB(void *param)
{
  int r2;
  fifoalloc(datn*8);
  Recv(&r2,LP0,4);
  Func2(&r2);
  Send(&r2,LP0,4);
  fifofree();
}
Func2(int *r2)
{ int i,t,p,n[8];
  mydata dat[32];
  //setup, sequential
 for(i=0;i<32;i++) dat[i].a=…;
  pthread_t tid[8];
  for(t=0;t<8;t++) {
 pthread_create(&tid[t],LP8+t,id_ThreadC,0);
  }
  for(t=0;t<8;t++) {
    Send(&dat[t*4],
     LP8+t,datn);
    n[t]=0;
  }
  do {
   wait(recvevent);
   for(t=0;t<8;t++) {
   if (nb_Recv(
      &dat[t*4+n[t],
      LP8+t,datn)
   { n[t]++;
    if (n[t]<4))
    Send(&dat[t*4+n[t]]
     ,LP8+t,datn);
   }
  cacheinvalid(
   &dat[t*4+n[t]],datn);
  //calc recvdata,release
  cacheinvalid(
  &dat[t*4+n[t]-1],datn);
  } while(not_all_4(n));
  for(t=0;t<8;t++)
  pthread_join(tid[t]);
}
```

```
void ThreadC(void *param)
{
  int i;
  mydat *blk=new mydat;
  fifoalloc(datn);
  for(i=0;i<4;i++) {
    Recv(blk,LP5,datn);
    Func3(blk);
    Send(blk,LP5,datn);
    cacheinvalid(blk,datn);
  }
  fifofree();
}

void Func3(mydata *blk)
{
 heavy computation
 wait(tmFunc3,SC_NS);
}

void InterruptCall()
{switch(intrfunc) {
 case id_ThreadA:
  ThreadA(intrparam);
  break;
 case id_ThreadB:
  ThreadB(intrparam);
  break;
 case id_ThreadC:
  ThreadC(intrparam); break;
 }
 SendJoin(intrcaller);
}
```

Code 3-3. SystemC TLM platform.

```
SC_MODULE(CHIP)
{
 sc_out<bool >   poff;
 Processor  *P[ProcessorNum];
 Router  *R[ProcessorNum];
 MEM      *M;
 IOC      *C;
 sc_signal<bool >   poffx[ProcessorNum];
 sc_fifo<flit>   rto[ProcessorNum][5];
 sc_fifo<flit>   pro[ProcessorNum];
 sc_fifo<flit>   chx[ProcessorNum][4];
 sc_fifo<flit>   memo;
 sc_fifo<flit>   ioco;
 SC_CTOR(CHIP)
 {int i,j;
  for(i=0;i<ProcessorNum;i++) {
   P[i]=new Processor("Processor");
   R[i]=new Router("Router");
   for(j=0;j<ThreadNum;j++) {
    P[i]->T[j].LPID=i*ThreadNum+j;
   }
   P[i]->iport(rto[i][Center]);
   P[i]->oport(pro[i]);
   if (i==0) P[i]->poff(poff);
   else P[i]->poff(poffx[i]);
   R[i]->RID=i;
   R[i]->iport[Center](pro[i]);
   R[i]->oport[Center](rto[i][Center]);
   R[i]->oport[East](rto[i][East]);
   R[i]->oport[North](rto[i][North]);
   R[i]->oport[West](rto[i][West]);
   R[i]->oport[South](rto[i][South]);
  }
  M=new MEM("MEM");
  M->oport(memo);
  M->iport(rto[MPos][MPort]);
  C=new IOC("IOC");
  C->oport(ioco);
  C->iport(rto[CPos][CPort]);
  for(i=0;i<ProcessorNum;i+=MeshWidth) {
   for(j=0;j<MeshWidth;j++) {
    if (j!=MeshWidth-1) R[i+j]->
      iport[East](rto[i+j+1][West]);
    else if ((i+j==MPos)&&(MPort==East))
      R[i+j]->iport[East](memo);
    else if ((i+j==CPos)&&(CPort==East))
      R[i+j]->iport[East](ioco);
    else R[i+j]->iport[East](chx[i+j][East]);
    …//other directions similar
   }
  }
};
///////////////////////////////////////////////
SC_MODULE(Router)
{sc_port<sc_fifo_out_if<flit> >   oport[5];
 sc_port<sc_fifo_in_if<flit> >   iport[5];
 int      RID;
 flit   ibuff[5];
 char irdy[5],iage[5],odir[5],odecide[5];
 char   routedir(flit *f) //XY route
 { char target=f->head.dstid/ThreadNum;
  if (RID==target) {
   return((odecide[f->head.dport]==-1)?
    f->head.dport:-1);
  } else {
   if ((RID/MeshW)==(target/MeshW)) { //same row
    if (RID<target) return((odecide[East]==-1)?East:-1);
    else return((odecide[West]==-1)?West:-1);
   } else if (same col) {
    if (RID<target)
      return((odecide[North]==-1)?North:-1);
    else
      return((odecide[South]==-1)?South:-1);
   } else if ((RID%MeshW)>(target%MeshW))
   {//left side
     …
   }
  }
 void   Route()
 { int i,j;
  char k, prior[5];
  while(1) {
   for(i=0;i<5;i++) if (irdy[i]) iage[i]++;
   for(i=0;i<5;i++) {
    if (!irdy[i]) {
     if (iport[i]->nb_read(ibuff[i])) {
      printdebug('R','I',RID,0,0, ibuff[i]);
      irdy[i]=true; iage[i]=0;
     }
    }
   }
   decide_by(ibuff,iage);
   for(i=0;i<5;i++) {
    if (odecide[i]!=-1) {
     if (oport[i]->num_free()!=0) {
      printdebug('R','O',RID,odecide[i],i,
           ibuff[odecide[i]]);
      oport[i]->write(ibuff[odecide[i]]);
      irdy[odecide[i]]=0;
     }
    }
   }
   wait(transtime,SC_NS);
  }
 }
 SC_CTOR(Router) {
   SC_THREAD(Route);
 }
};

/////////////////////////////////////////////////////////
SC_MODULE(Processor)
{ sc_port<sc_fifo_out_if<flit> >  oport;
 sc_port<sc_fifo_in_if<flit> >   iport;
 LogicProcessor T[ThreadNum];
///////////////////////////SC_THREAD
 int threadcnt;
 void threadcall()
 { int ord = threadcnt ++ ;
  T[ord].main();
 }
///////////////////////////Send
 bool  sendlock;
 sc_event sendevent;
 bool issending() {  return(sendlock); }
 void Sender(char *ptr, char cmd, char ctrl,
   char src, char dst, char dport, int len)
{ int p,j;
 flit f;
 sendlock=1;
 f.head.srcid=src;  f.head.dstid=dst;
 f.head.cmd=cmd;  f.head.headflit=1;
 f.head.dport=dport;
 f.head.tailflit =(len<=FlitSize) ;
 if (cmd==pktcmd_ctrl) f.head.ord=ctrl;
 else if (f.head.cmd==pktcmd_dcachew) f.head.ord=len;
 else f.head.ord=0;
 if (len<FlitSize) j=len; else j=FlitSize;
 p=0;
 if (j!=0) memcpy(&f.dat,ptr+p,j);
 do {
  if (oport->num_free()!=0) {
   oport->write(f);
   f.head.headflit=0;
   f.head.ord++;
   p+=j;
   if ((len-p)>FlitSize) j=FlitSize;
   else {j=len-p; f.head.tailflit=1;}
   if (j!=0) memcpy(f.dat,ptr+p,j);
   wait(transtime, SC_NS);
  } else {
   wait(clockcycle, SC_NS);
  }
 } while(j!=0);
 sendevent.notify();
 sendlock=0;
}
void fork(char dthread, char srcid,
      int intrfunc, void *param)
{
 T[dthread].intrcaller=srcid;
 T[dthread].intrfunc=intrfunc;
 T[dthread].intrparam=param;
 T[dthread].intrevent.notify();
}
void join(char dthread,char srcid)
{
 T[dthread].threadwait[srcid]=0;
 T[dthread].joinevent.notify();
 }

bool nb_Recv(void *addr,unsigned char srcid,
           unsigned char dstid, int len)
 { int r,s,a,n,rown;
  flithead m;
  for(r=0;r<cacherows;r++) {
  if (cachelock[r]) {
    m.w=cachetag[r][0];
    if ((m.w!=-1)&&(m.srcid==srcid)&&
      (m.dstid==dstid)&&m.tailflit) break;
   }
  }
  if (r==cacherows) return(false);
  for(all locked rows & sets) {
    cachetag[r][s]=addr+(s<<cachecdb);
    cachevalids[r][s]=1;
    memcpy(a,&cacheram[r][0][0][0],n);
  }
  return(true);
 }
///////////////////////Receiver HW
 void Receiver()
 { flit f;  int dthread;
  while(1) {
   if (iport->nb_read(f)) {
    dthread=f.head.dstid&(ThreadNum-1);
    switch(f.head.cmd) {
    case pktcmd_ctrl:
     switch(f.head.ord) {
     case ctrlpkt_fork:
      fork(dthread,f.head.srcid,f.wdat[0]);
      break;
     case ctrlpkt_join:
      join(dthread,f.head.srcid);
      break;
     }
     break;
    case pktcmd_dcacheack:
     memcpy(&cacheram[rdr][rds][rdc][0],
        f.dat,FlitSize);
     rdc++;
     dcacheackevent.notify();
```

```
      break;
     case pktcmd_msg:
      fifowrite(&f);
      if (f.head.tailflit)
        T[dthread].recvevent.notify();
      break;
     }
    }
   wait(clockcycle,SC_NS);
  }
 }
 void readmem(int t,int s,int c)
 { int dat[2];
  dat[0]= cachetag[r][s]; dat[1]=FlitSize*Cols;
  if (dat[0] in IOC range) {
    dcore=CPos; dport=CPort;
  } else {dcore=MPos; dport=MPort;}
  Send(dat,pktcmd_dcachereq,dcore,dport);
  wait(dcacheackevent);
 }
 int cacheread(void *addr)
 { cachebusy=1;
  c=(addr>>cachedatb)&cachecolmask;
  s=(addr>>(cachecdb))&cachesetmask;
  m=addr&~cachescdmask;
  r=find((cachetag[r][s]==m)&&cachevalid);
  if (r not found) { //read miss
   r=findlru(s);
   cacheflushcol(r,s);
   cachetag[r][s]=m;
   readmem(r,s,c);
   cachevalid[r][s][c]=1;
  }
  cachebusy=0;
  cacheevent.notify();
  return(cacheram[r][s][c][addr&bytemask]);
 }
 void fifowrite(flit *f)
 { int r,s,c;
  flithead m; m.w=f->head.w;
  if ((m.ord&cachescmask)==0) {
   c=0; s=0;
   do {
    for(r=0;r<cacherows;r++)
     if (cachelock[r]&&(cachetag[r][0]==-1)
     && (fifothread[r]==(m.dstid&3)))
      break;
    if (r==cacherows) //Receiver hangup
      wait(fifoevent);
   } while(r==cacherows);
   m.headflit=0;  cachetag[r][0]=m.w;
  } else {
   c=(addr>>cachedatb)&cachecolmask;
   s=(addr>>(cachecdb))&cachesetmask;
   r=find_tag_with_same_id(m);
  }
  memcpy(cacheram[r][s][c],f->dat,FlitSize);
  cachevalid[r][s][c]=1;
 }
 SC_CTOR(Processor)
 {
  for(int j=0;j<ThreadNum;j++) {
   T[j].parant=this;
   sc_thread_handle handle[j] =
simcontext()->register_thread_process( "",
   SC_MAKE_FUNC_PTR(
     Processor, threadcall ), this );
     …
   }
 }
///////////////////////////////////////////////
class LogicProcessor
{
public:
 int LPID,intrfunc,intrcaller;
 Processor *parant;
 sc_event intrevent, joinevent, recvevent,
 void *intrparam;
 bool threadwait[ProcessorNum*ThreadNum];
#include "Code3-2"

void IDLE()
{
  while(1) { wait(intrevent); InterruptCall(); }
}
void pthread_create(pthread_t *tid, int attrib, int pfunc,
void *param)
{ int dat[3];
 threadwait[attrib]=true;
 if (((attrib^LPID)>>ThreadNumb) == 0)) {
  parant->fork((attrib&(ThreadNum-1)),
   LPID,pfunc,param);
 } else {
  dat[0]=pfunc;  dat[1]=(int)param;
  while (parant->issending())
   wait(parant->sendevent);
  parant->Send((char *)dat,pktcmd_ctrl,
    ctrlpkt_fork,LPID,attrib,CENTER,8);
 }
 *tid=&threadwait[attrib];
}
void pthread_join(pthread_t tid)
{while(*tid) { wait(joinevent); }
}
void Send(void *ptr, int dst, int num)
{while (parant->issending()) wait(parant->sendevent);
 parant->Sender((char
*)ptr,pktcmd_msg,0,LPID,dst,CENTER,num);
}
void Recv(void *ptr, int src, int num)
{ while (!parant->nb_Recv(ptr, src, LPID, num))
{wait(recvevent);  }
}
int  cacheread(void *ma)
{while (parant->iscachebusy() )
   wait(  parant->cacheevent);
 return(parant->cacheread(ma));
}
void cachewrite(void *ma,int v, int bytes)
{while (parant->iscachebusy() )
   wait(  parant->cacheevent);
 parant->cachewrite(ma,v,bytes);
}
void fifoalloc(int bytes)
{while (parant->iscachebusy() )
   wait(  parant->cacheevent);
 parant->fifoalloc(bytes);
}
};
```

CHAPTER FOUR
PARALLELIZATION

Parallel processing had been developed in 1960s on some high-speed vector processors such as ILLIAC-IV and Cray-1 to increase the scientific computation speed. Since scientific codes contain many one-dimension vector and two-dimension matrix operations, a vector processor is often used to perform these operations simultaneously on its processing elements. Since then, many parallelization techniques have been developed.

4.1 Vectorization

A basic way to explore code parallelism is to transform operations in a loop into as many vector operations as possible. In most cases, parallelism is destroyed by bad code structure. Vectorization techniques try to improve parallelism by dependence reduction and loop transformation. The techniques described in this section are from the background of our parallelization tool development.

A *vector* is represented as *A*[`begin: end: stride`]. The array index is extended to 3 literals to represent the vector operation performing on element `begin, begin+stride, begin+2*stride`, …, `end`. When `stride` is 1, it can be omitted. For example, the following code:

```
for(I=1;I<=64;I++) C[I]=A[I+1]+B[2*I-1]
```

can be represented as a vector addition

```
C[1:64]=A[2:65]+B[1:127:2]
```

In addition to scientific computation, people wish to utilize the vector computation in more fields. To optimize a general algorithm into vector needs in-depth analysis. *Vectorization* technique for sequential code had been widely studied in the 1970s. In general, vectorization technique transforms nested loops into vector by dependence analysis, dependence removal, and loop transformation as decribed in the following subsections.

4.1.1 Dependence Analysis

If there is no semantic difference between executing a loop in a sequential order and executing it as a vector operation, this loop is able to parallelize. A counter example is shown in the following code

```
for(I=1;I<=64;I++) A[I+1]=A[I]+B[I];
```

On executing the code as a sequential loop, we have the following result: $A[3]_{new}=A[2]_{new}+B[2]_{old}= A[1]_{old}+B[1]_{old}+B[2]_{old}$. On executing it as a vector operation: `A[2:65:1]=A[1:64:1]+B[1:64:1]`, the result will become $A[3]_{new}=A[2]_{old}+B[2]_{old}$, which is different to the result obtained by executing it as a sequential loop; thus the loop is unable to parallelize.

In above example, the operand of the second iteration uses the result of the first iteration $A[2]_{new}$. In other words, the execution of the second iteration is dependent on the first iteration. Two statements can be executed in parallel only when there is no dependence between them. The statements of the first iteration and the second iteration are dependent, so they cannot be executed in parallel as a vector.

Dependence can be classified into the following four types [38]:

(1) *Flow dependence*, or *Read after Write* (RAW) dependence.

If one operand of the second statement is the result of the first statement, the second operation should wait until the first statement finishes.

(2) *Anti dependence*, or *Write after Read* (WAR) dependence.

If the second statement overwrites one operand of the first statement, the second statement cannot be executed earlier than the first statement to avoid change of operand value.

(3) *Output dependence*, or *Write after Write* (WAW) dependence.

If two statements write to the same destination, they cannot be executed simultaneously to avoid having an ambiguous result.

(4) *Input dependence*, or *Read after Read* (RAR) dependence.

When two statements use the same operand, they are said having input dependence.

Input dependence is not an actual dependence because the statement execution is not dependent on each other. Input dependence is used to group the statements closer such that we can reuse the same operand from the register to save memory load time.

The anti and output dependences can be removed by the variable rename technique, thus they are also called *false dependences*, and only the flow dependence is called a *true dependence*.

Therefore, the *loop-carried flow dependence* actually limits vectorization. To precisely determine the loop-carried flow dependence, we can analyze the array index relationship of the statements in a loop [39]. Consider the generalized expression:

```
for(I=1;I<=N;I++) A[a+b*I]=f(A[c+d*I])+g(I);
```

where g(I) does not use array A. Relative to the above example, `a=1, b=1, c=0, d=1, g(I)=B[I]` and `f(A[x])=A[x].` To analyze a loop-carried flow dependence is to check whether the result `A[a+b*x]` is used as an operand `A[c+d*y]` at a later iteration. The loop-carried flow dependence exists if and only if there exist integers *x* and *y*, $1 \le x < y \le N$, such that $a+b*x=c+d*y$.

By the number theorem, the equation $a+b*x=c+d*y$ has integer solution *x*, *y* if and only if *a*-*c* is multiple of `GCD(b, d)`, or `GCD(b, d)|(a-c)`, where GCD is the Greatest Common Divisor. From the above example, `GCD(b, d) = GCD(1,1) = 1, a-c = 1-0 = 1, GCD(b,d)|(a-c)` is true.

For a loop that contains many statements, we should check whether the statement result is used as operand by any other statement at the later iteration or not, that is, we should check the `GCD(b,d)|(a-c)` for all statement pairs.

The single loop dependence check can be extended to nested loop. For example, given the following sample code:

```
for(K=1;K<=L;K++)
    for(J=1;J<=M;J++)
      for(I=1;I<=N;I++) A[a0+a1*I+a2*J+a3*K]=
                              f(A[b0+b1*I+b2*J+b3*K]);
```

The dependence checking is performed from the innermost loop to the outmost loop. At a specific outer loop J $=x_2$ and K$=x_3$, the innermost loop contains loop-carried dependence if and only if there exist $1\le x_1\le N$, $1\le x_2\le M$, and $1\le x_3<y_3\le L$, such that $a_0+a_1*x_1+a_2*x_2+a_3*x_3 = b_0+b_1*x_1+b_2*x_2+b_3*y_3$, or $(a_1-b_1)*x_1+(a_2-b_2)*x_2 +a_3*x_3-b_3*y_3 = b_0-a_0$. The integer solution exists when $\mathrm{GCD}(a_1-b_1,a_2-b_2,a_3,b_3)|(b_0-a_0)$. Similarly, the dependence check equation for the second loop is $\mathrm{GCD}(a_1-b_1,a_2,b_2,a_3,b_3)|(b_0-a_0)$, and $\mathrm{GCD}(a_1,b_1,a_2,b_2,a_3,b_3)|(b_0-a_0)$ for the outmost loop.

4.1.2 Loop Normalization

By the number theorem, the above GCD test for dependence analysis works only when the loop index begins from 1, ends at a number N, and increases by 1. General loop that does not satisfy this constraint needs to *normalize*.

Given the following code:

```
P=10;
 for(J=0;J<100;J=J+2) A[P++]=A[2*J]+J;
```

By performing analysis on the loop argument, we can replace the loop index *J* to a new index *K*, that is, *J*=2**K*-2, and change the loop index dependent variable *P* to *P*=*K*+9. The loop is transformed into:

```
for(K=1;K<=50;K++) A[K+9]=A[4*K-4]+(2*K-2);
```

Now this loop is normalized and dependence check can be performed.

4.1.3 Loop Transformation

Consider the following code:

```
for(J=1;J<=M;J++)
     for(I=1;I<=N;I++) A[I][J]=A[I-1][J];
```

This code fills the whole array with row 0 in an order of column by column, and the inner loop contains loop-carried dependence. If the code is transformed into:

```
for(I=1;I<=N;I++)
     for(J=1;J<=M;J++) A[I][J]=A[I-1][J];
```

By exchanging the two loops, the new code works row by row. The two results are the same but the later row-wise order can work more efficiently in a vector machine.

Loop transformation procedure sequentially selects a pair of loops in which it is legal to apply a transformation, and checks its dependence by GCD test as described above. If more than one solution is available, the performance or data locality gain is used to help decision making.

Loop transformation is the key technology to improve parallelism and data locality. Many transformations had been introduced [40][41]. For example, loop skewing helps systolic array algorithms to utilize memory; loop interchange and reversal helps linear algebra that contains dense matrices.

4.1.4 Dependence Removal

Instruction Level Parallelism can be improved by removing false dependence. The techniques include variable rename, scalar expansion, node splitting and control flow conversion. The following code is used to explain.

```
      for(I=1;I<=N;I++) {
S1:   v=A[I]+B[I];
S2:   v=v*C[I];
S3:   C[I+1]=v+I
S4:   D[I]=D[I-1]+D[I+1]+2;
S5:   if (E[I]>F[I])
S6:      R[I]=E[I]-F[I];
S7:   else R[I]=E[I]+F[I];
     }
```

S1 and S2 are outputs dependent on variable *v* that restricts S1 vectorization. If variable *v* from the result in S2 and the operand in S3 is renamed to *w*, the output dependence is removed. This technique is called *variable rename*. The lifetime of a local variable starts from its value settled, thus renaming it as a new variable will not cause semantic differences.

S1 contains loop-carried output dependence to itself on variable *v*; this dependence disables S1 to vectorize. Renaming variable `v` to `v[I]` removes this dependence. This technique is called *scalar expansion* for it expands a scalar variable into an array. The disadvantage is that it needs to allocate more memory.

S4 contains a loop-carried flow-dependence, where `D[1]` is modified at the second iteration and then loaded at the second iteration by `D[I-1]`. S4 contains 2 additions, one is vectorizable. The non flow-dependent part `D[I+1]+2` can be lifted to a new statement, and store its result on a new variable `T[I-1]`, then use `T[I-1]` to replace the non flow-dependent part in the original statement. After that, the new statement becomes vectorizable. This technique is called *node splitting*. The index I-1 of the new variable *T* is aligned to the flow dependence part `D[I-1]` such that the data shift which is required for ILLIAC-IV array architecture can be performed in parallel.

S6 and S7 have control dependence on S5. The program counter (PC) value set by S5 conditional branch operation is the address of S6 or S7, which will depend on the S5 comparison operation result. It causes the program counter value to become ambiguous when all iterations of S5 are executed simultaneously. In other words, S5 has loop-carried output dependence on program counter. To avoid program counter ambiguity, conditional branch operations should be removed. In ILLIAC-IV, each PE contains a *mode* register that can disable the current instruction execution. When a PE is disabled, the relative result keeps no change. The conditional branch execution can be changed to execute all statements with a proper vector mask. A new Boolean array is added to store the S5 comparison result. The Boolean array (1-bit for each element) is sent to the *mode* register or *vector mask* register when the S6 vector is executing, and its complement is sent when the S7 vector is executing. The execution with mask expression works as a three-operand one-result operation, the three operands are the original two ALU operands plus the vector mask; such *control flow conversion* [42] technique changes control dependence into data dependence.

The result after dependence removal is shown in the following code:

```
         for(I=1;I<=N;I++) {
S1:      v[I]=A[I]+B[I];
S2:      w=v[I]*C[I];
S3:      C[I+1]=w+I;
S4a:    T[I-1]= D[I+1]+2;
S4b:    D[I]=D[I-1]+T[I-1];
S5:      VM[I]=(E[I]>F[I]);
S6:      R[I]=(VM[I])?(E[I]-F[I]);
S7:      R[I]=(~VM[I])?(E[I]+F[I]);
        }
```

4.1.5 Strongly Connected Component

The above transformation can be performed more efficiently by applying the graph theorem on the dependence graph.

As defined, a dependence graph [43] is a directed graph, whose nodes represent code statements, and arcs are dependences. Figure 4-1 shows the dependence graph of the example in Section 4.1.4.

On a directed graph, a *strongly connected components* (SCC) is defined as: for every pair of nodes *u* and *v* if there is a path from *u* to *v* and a path from *v* to *u*. An SCC can be found by using depth-first search technique [44].

In geometric view, an SCC contains nodes that form a circle. A circle in a dependence graph means that there are loop-carried dependences on these statements, which are not vectorizable. A *single-tone SCC* is defined as an SCC having only one node and no arc to itself; thus, it is vectorizable.

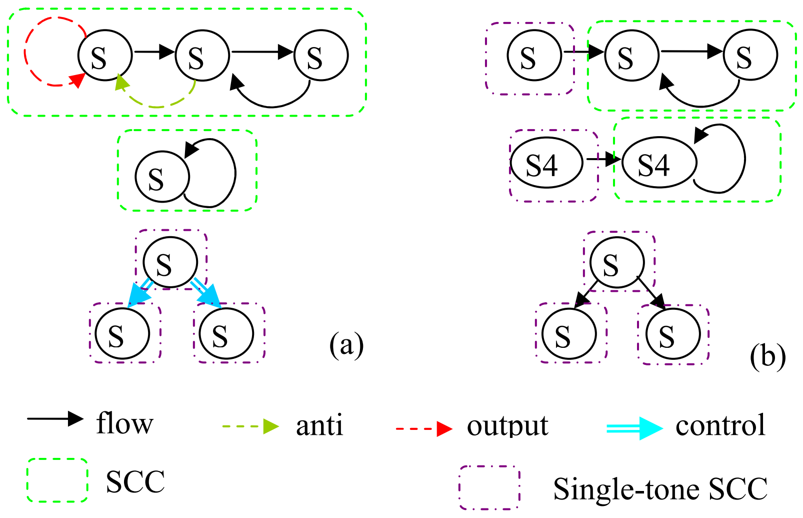


Figure 4-1. Dependence graphs: (a) original and (b) after dependence removal.

4.1.6 Loop Distribution

If we treat an SCC as a single supernode, all the arcs in a dependence graph will have a forward direction. The loop can be partitioned into many sub-loops on a forward-only path that will not make a semantic difference. All SCCs have to be changed into independent loops, and the original loop headers have to be distributed to these new loops. A single-tone SCC can be directly transformed into a vector. The result of the above example then becomes:

```
S1:  v[1:N]=A[1:N]+B[1:N];
       for(I=1;I<=N;I++) {
S2:     w=v[I]*C[I];
S3:     C[I+1]=w+I;
       }
S4a:  T[0:N-1:1]= D[2:N+1:1]+2;
S4b:  for(I=1;I<=N;I++) D[I]=D[I-1]+T[I-1];
S5:    VM[1:N]=(E[1:N]>F[1:N]);
S6:    R[1:N]=(VM[1:N])?(E[1:N]-F[1:N]);
S7:    R[1:N]=(~VM[1:N])?(E[1:N]+F[1:N]);
```

4.2 SIMDization

A subword-parallel SIMD processor has more restrictions than a vector machine. For example, a subword-parallel SIMD core is restricted on memory access. Given the following code:

```
for(I=1;I<=64;I++) C[I][I]=A[I][I]+1;
```

The memory items are discontinuous. To process the above example, PLX has to load the discontinuous memory items `A[0][0]` and `A[1][1]` by different load instructions, and pack them into one register to process addition instruction together. While memory access latency is very long, the addition of `A[0][0]` can finish when waiting for `A[1][1]` to be loaded in a sequential execution scalar processor. Packing operations would not improve performance, but would increase the pack/unpack overhead.

When vector items are continuous, subwords can be loaded together by one load instruction, and memory access count can be reduced. It induces extra effort to handle neighboring data in a subword-parallel SIMD mode.

4.2.1 Control Flow Conversion

Control flow conversion that converts if-else control into execution with mask is introduced in Section 4.1.4. Implementing an execution with mask needs three read ports on a register file and three operand ports on an ALU for the extra mask operand, and the register file write port needs to be byte writable; thus, the hardware cost is increased. While most of the time, control flow will not become the performance bottleneck, increasing hardware cost is not worthy.

A multiplexer behavior, such as `R=X?A:B`, can be implemented using an AND-OR logic as `R=(X&A)|(~X&B)`. The S6 and S7 statements in the example of Section 4.1.4 can be changed to:

```
S6:  R[I]=(VM[I]&(E[I]-F[I]))|(~VM[I]&R[I]);
S7:  R[I]=(~VM[I]&(E[I]+F[I]))|(VM[I]&R[I]);
```

The two statements can be further optimized as:

```
R[I]=(VM[I]&(E[I]-F[I]))|(~VM[I]&(E[I]+F[I]));
```

The new statement only contains basic logic operations that can execute on a 2-operand ALU. But `VM[I]` is one-bit length and `E[I]-F[I]` are subwords. Before its execution, `VM[I]` has to expand into a subword size for the bitwise AND/OR operation. This expansion is performed by the subword-parallel comparison operation in S5. Subword-parallel ALU sets every bit in the related subword to 1 (as an integer value -1) when the comparison result is true, and sets to 0 when the comparison is false.

4.2.2 Memory Alignment

For cost and power consideration, most RISC processors require all memory accesses to be aligned, that is, the data loaded from memory cannot cross the 64-bit boundary if the processor is of 64-bit length. An across-boundary access should be split by a compiler.

Memory alignment becomes more critical when using an SWP-SIMD processor on multimedia applications, where we are asked to pack the memory components into one superword to be accessed together. Although each component does not cross the boundary, the packed one may cross. For example on processing an RGB24 format picture, the packed element is 24-bit (8-bit for each of the Red, Green and Blue components), the third element will cross the 64-bit boundary. Another example is Motion Estimation. This algorithm shifts a search window one pixel at each iteration, making most reference frame accesses misaligned.

Current technology handles misaligned vectors as a stream [45]. Registers are used for each vector as a stream buffer. Vector elements are collected in the registers and shifted to the proper aligned position. Figure 4-2 shows the concept. Assume that the data precision is 16-bit, a vector begins from w1, and the vector length is 4. Loading 64-bit from w1 will cross the 64-bit boundary. To avoid the misalignment problem, `Vload` instruction loads two words from `w0` and `w4`, and use `Vpermute` instruction to combine the loaded words. The second word is kept in a stream register for the following vector.

Operands of a vector equation may have different stream shifts. As the following example shows:

```
C[2:66]=A[1:65]+B[3:67];
```

Using the above policy, both streams *A* and *B* have to left shift 1 and 3 positions respectively, and the addition result has to right shift two positions. If they have to be aligned to stream *C*, stream *A* has to right shift one position and stream *B* left shift one position, saving one shift operation.

By above discussion, many policies are possible to handle the stream shift.

(1) Zero Shift: It is the same as Figure 4-2. This policy shifts each misaligned load stream with an offset of zero, and shifts the store stream from offset zero to the alignment of the store address.

(2) Eager Shift: This policy shifts each load stream directly to the alignment of the store stream.

(3) Lazy Shift: This policy pushes the shift towards the root of the expression tree as close as possible. And

(4) Dominant Shift: This policy shifts to the most frequent alignment position in equation.

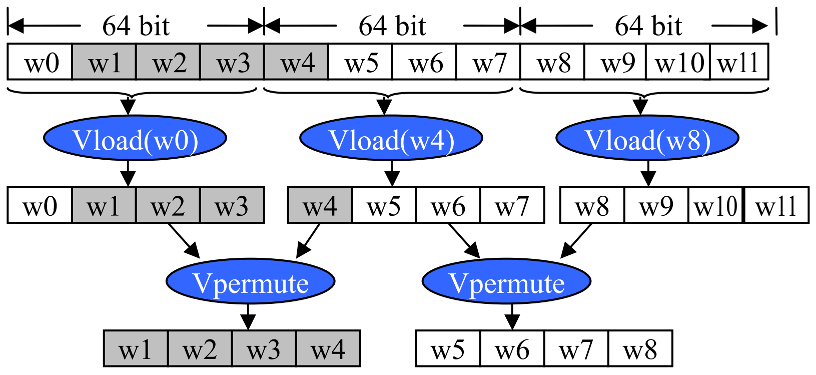


Figure 4-2. Streaming vector loading.

4.2.3 Permutation Optimization

Data length conversion can also be handled in streams [46]. When the variables in a statement have different precisions, the load streams have to be unpacked with the largest precision, and the result has to be packed with the same precision as the store variable.

While the subwords of an SIMD instruction are packed into a register, each subword cannot be easily moved alone. In order to unpack four 16-bit subwords in a register with a 32-bit precision, the first subword should right shift 16 bits and the second subword right shift 32 bits to combine into a new register; the third subword is left shifted 16-bit and combined with the forth subword. In total, 3 shift and 2 combine operations are needed, not including the sign extension.

Many multimedia algorithms themselves contain *permutation*, such as the butterfly-order on FFT, or the average/difference of two audio channels on MP3. Efficiently handling permutations is not easy. Figure 4-3 shows two implementations of a simple example in MP3 encoder, which calculates the average and difference of the left and right channel samples.

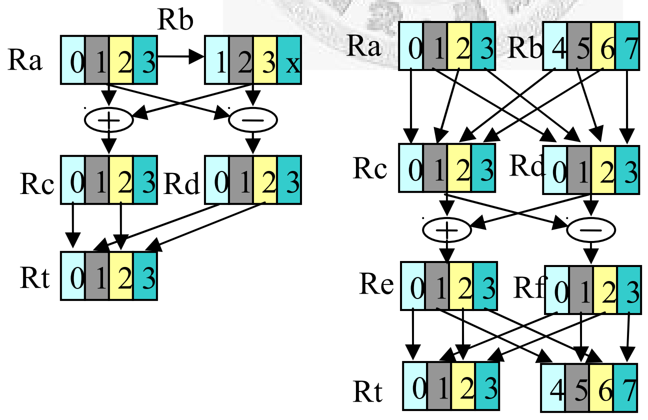


Figure 4-3. Interleaved average/difference implementations.

The left channel is the even parts of the audio sample array, and the right channel is its odd parts. The results should also be interleaved into a one-dimension array. Figure 4-3(a) first left shifts samples to a stream aligned on the right channel, then calculates its average and difference, and packs them into the result register. Figure 4-3(b) loads double samples into two registers, packs their even and odd parts, calculates their average and difference, and packs the even and odd results into the result register. The first method uses 5 registers and 5 operations to get 4 result samples; the second implementation uses 7 registers and 8 operations to get 8 result samples. The throughput of the second implementation is higher, but it needs more registers, a tradeoff in optimization. Optimizing a code with the fewest permutation instructions can be formulated as a multi-cut problem which is NP-hard [47].

4.2.4 Subword Fusion

General software contains many non-vector operations. Packing them into a *superword* to process together may increase performance. When addition and subtraction operations are adjacent, negating the subtraction operand and adding them in subword-parallel can improve performance.

MIT University first introduced the concept of *fusioning* these operations [48]. They used a heuristic-based two-cluster partitioning algorithm. Instructions are partitioned into scalar and vector parts. One instruction is moved from the scalar part to the vector part once, and the vector part is re-packed to find the minimum cost. The cost contains pack/unpack overhead.

Vienna University extended the fusion on addition/subtraction pair to increase SIMD utilization [49]. They use depth-first search based sorting method with chronological backtracking to discover SIMD style parallelism in a scalar code block, aiming to reduce the overall instruction count. The addition/subtraction pair finding is considered as reducing number of source operands.

4.2.5 Matrix Transposition

Most processors store array elements in row-wise. Column vector items are not continuous in memory. They should be loaded independently and packed together. Packing four column elements into a superword needs four non-sequential memory `load` and three `pack` operations, which is a large overhead relative to the small code size. To speedup, we can load a 4×4 array from memory into 4 registers, which only needs 4 memory loads. Then transposing the array to get 4 column vectors. Transposition can be efficiently obtained by eight PLX permutation instructions as shown in Figure 4-4. The first stage exchanges the odd subwords in an even row and the even subwords in an odd row by using two permutation instructions. The second stage exchanges double words of row0/row2 and row1/row3, each takes two permutation instructions.

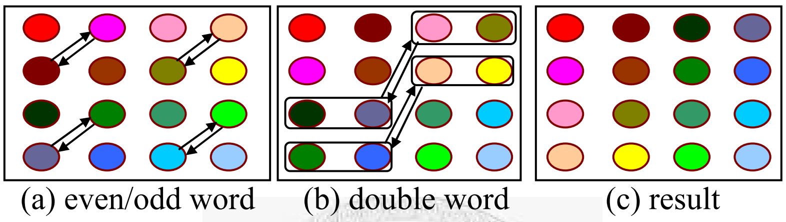


Figure 4-4. Matrix transposition in SWP-SIMD.

4.2.6 Reduction

An extra effort to parallelize multimedia application is to convert summation. Considering the following code:

```
for(I=1;I<=N;I++) s=s+f(A[I]),
```

which contains loop-carried dependence on variable *s*, vectorization can do nothing. While summation is often used in multimedia applications, it greatly affects the performance. While the SWP-SIMD vector length is short, the loop can be partitioned into partial summations such that we can sequentially summarize these partial results at the final stage [50], as shown in the following code:

```
psum[0:VL-1]=0;
for(I=1;I<=N/VL;I++)
    psum[0:VL-1]+=f(A[I:I+VL-1];
for(I=0;I<VL;I++) s+=psum[I];
```

4.2.7 Loop Unrolling

The vector length of a subword-parallel SIMD processor is short; it can be only 4 or 8 depending on data precision. While a loop iteration count is usually larger than the short vector length, the loop has to be unrolled to fit the short vector length. The example in Section 4.1.4 can be implemented in either one of the following two ways (only the first S1, S3 and S3 statements are shown here):

```
S1:     for(I=1;I<=N;I+=VL)
           v[I:I+VL-1]=A[I:I+VL-1]+B[I:I+VL-1];
        for(I=1;I<=N;I++) {
 S2:        w=v[I]*C[I];
 S3:        C[I+1]=w+I;
        }
```

Or

```
        for(I=1;I<=N;I+=VL) {
S1:         v[0:VL-1]=A[I:I+VL-1]+B[I:I+VL-1];
            for(J=0;J<VL;J++) {
 S2:            w=v[J]*C[I+J];
 S3:            C[I+1+J]=w+I+J;
            }
        }
```

The first implementation unrolls the loop after loop distribution, and the second implementation unrolls the loop before loop distribution. The second implementation allocates *v* in a register file, which saves the memory access time for *v*. As semi-conductor technology progresses, the register access time is much faster than the memory one; thus, the performance difference of the two implementations becomes significant.

The second implementation is not always better than the first implementation when it causes data cache swap. If a loop contains many array variables that cannot all fit in a data cache, the partial data of array A that were preloaded in cache (which amount is larger than the vector length) at S1 will be replaced, so it will waste more time to reload array *A* from the main memory at each iteration. This will overcome the gain of register reuse.

The optimal solution for loop distribution is to group all SCCs that are connected by dependence in one loop, but cannot be too large to cause cache swap. The optimization has to compromise between cache strategy and register allocation.

Memory access latency usually affects performance greatly. Software pipelining [51] can be applied to further improve memory access efficiency. With software pipelining, we can reschedule ALU instructions to fill the time when memory load is waiting. By considering memory sequential accesses and hardware pipeline architecture, and using software pipelining, the performance can improve 34% [52].

4.3 ILP Scheduling

Instruction level parallelism (ILP) scheduling assigns operations into a 2-D slot of spatial and time dimension. ILP scheduling can be divided into cyclic and acyclic scheduling methods. *Cyclic scheduling* works on loop and *acyclic scheduling* works on a basic block code region.

4.3.1 Software Pipelining

Figure 4-5 shows one cyclic scheduling method called *software pipelining*. Assume that this machine is a 3-issue VLIW. A loop of iterations 0 to n-1 contains 6 operations from A to F. Operation A is loop-carried dependent to B, so A_1 can be executed in parallel with C_0 at the earliest time slot. There are two schedules, as shown in Figure 4-5(a), where the first ALU executes iterations 0, 3, *etc*; the second ALU executes iterations 1, 4, *etc*; and the third ALU executes iterations 2, 5, *etc*. In Figure 4-5(b), the first ALU executes all A and B operations; the second ALU executes all C and D operations; and the third ALU executes all E and F operations. Schedule (a) has better data locality which is necessary for clustering architecture, but schedule (b) optimizes different functions on the 3-issue ALU.

In general, data dependences exist in various types. A data may be referenced k iterations later where k is not 1. Then the software pipelining cannot be as simple as the above example. Sometimes it requires using a heuristic method, such as *modulo scheduling*, to schedule.

A_0		
B_0		
C_0	A_1	
D_0	B_1	
E_0	C_1	A_2
F_0	D_1	B_2
A_3	E_1	C_2
B_3	F_1	D_2
C_3	A_4	E_2
D_3	B_4	F_2
E_3	C_4	A_5

(a)

i=0 to n-3

A_0		
B_0		
A_1	C_0	
B_1	D_0	
A_{i+2}	C_{i+1}	E_i
B_{i+2}	D_{i+1}	F_i
	C_{n-1}	E_{n-2}
	D_{n-1}	F_{n-2}
		E_{n-1}
		F_{n-1}

(b)

Figure 4-5. Software pipelining.

4.3.2 Basic Block Extension

Acyclic scheduling works on a basic block code region. A basic block has a single entrance at its head and an exit at its tail in a control flow graph. No backward arc is inside a basic block. The code formation is a heuristic process, it selects instructions with data dependence constraint and resource usage conflict, to target optimization of code size or power consumption.

A larger code region has more instructions to select and gets better efficiency. The key technique of acyclic scheduling is to enlarge the code region. A basic technique is *loop unrolling*. It removes the backward arc in the control flow graph, thus the basic block is extended to contain *n* times of operations.

Tail duplication is another technique to extend basic block. As shown in Figure 4-6, the control flow graph is partitioned into 4 basic blocks by an `if-else` decision. Duplicating BB4 and moving them into the `if-else` region will reduce the basic block number to 3, but BB2 and BB3 are enlarged.

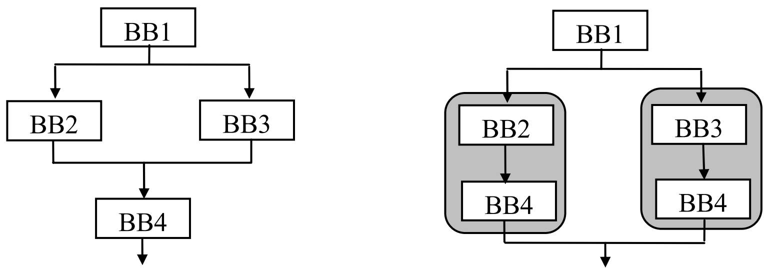


Figure 4-6. Basic block tail duplication.

Control flow always limits the instruction stream to fill ILP wide spatial slot. Sometimes if we know by profiling that a branch has a higher probability to execute, it must be executed in parallel with a current basic block. The speculation technique will bring this branch ahead before the check point to improve parallelism, and the execution will be recovered if the speculation result is negative.

Preload is a frequently used speculation method. A memory load is able to execute whenever the `load/store` unit is not in use. If the control flow branches to another path, the loaded data is just waste and will not affect the result. In moving more instructions before the branch, more registers are required to store these temporary results, which reduce the number of available registers in the basic block.

4.4 TLP Scheduling

Multitask OS scheduling is maintained on two levels: process and thread. A *process* is a standalone program. Killing a process during scheduling will not affect other processes. Process has its own heap and stacks memory, file handler, and so on. Synchronization between processes is seldom. *Thread* is a piece of process execution stream. Threads are not independent, killing a single thread may cause process execution to fail. Each thread has its own stack, but the heap memory and file handler are shared with others.

Threaded programming offers software portability for parallelization. On a serial machine, threads can work concurrently by time sharing; on a simultaneous multi-threading or multi-processor machine, threads can work in parallel simultaneously. The difference of scheduling is taken care by the OS. To achieve portability, a standard to handle threads is necessary.

4.4.1 Profiling

In a general code, 90% of the execution time is spent on 10% of the code. *Profiling* is used to tell the programmer where the performance bottleneck is. The result of profiling is statistical information on a code, such as execution time, subroutine call statistics, operations used, and memory access time.

Static profiling works by analyzing the representation of a program code without executing it. The non runtime environment gives the possibility of going into greater detail in the analysis but also places restrictions on it. Non deterministic properties, such as recursion, dynamic data structures, and non bound loops in a code region cannot be estimated without running data from the input, which in turn requires dynamic profiling.

Dynamic profiling, on the other hand, executes a code with a given testbench instead of analyzing it. During execution the profiler gathers a code which is being executed and properties of the execution that are deemed interesting. The dynamic profiling cannot give the engineer as profound information on the code as the static profiling does, but it can report in detail what happens during the execution of the code with a well-defined set of inputs.

In order to discover parallelism, data dependence is one of the most important characteristics in a code. A code can often be clustered by its spurious dependences; for example, the accesses of two memory objects may be conflicting, if the objects cannot be proved independent. A single spurious dependence can prevent multiple opportunities for parallel execution. Analysis clarifies the picture either by finding precise data dependences or by removing spurious ones to improve parallelism [53].

The chief obstacle to discovering opportunities to parallelize a multimedia application is identifying dependences between pointer references. A high-quality pointer analysis is essential in determining the relationship between pointer references. However, there are many coding constructs and programming practices that veil the true picture of memory usage from pointer analysis. For some of these cases, like recursive data structures and arrays, more specialized analyses such as shape analysis and array analysis will be very helpful in clarifying the picture.

Pointer analysis determines what objects a memory reference can possibly access. Heap-sensitive pointer analysis finds whether the allocation function for a particular type of dynamically-allocated memory object is frequently reused to allocate multiple objects. Such kind of code reuse is a must to distinguish objects that share a static allocation site. Field-sensitive pointer analysis will group together all of the objects pointed to by a structure. This prevents the compiler from distinguishing objects through those pointers. This case appears regularly since multimedia programs commonly manipulate multiple data channels, and programmers use structures to organize data hierarchically.

Array analysis can indicate whether or not the pointers really refer to the same memory location, when two pointers are known to refer to the same object. This form of analysis conveys information about which loop iteration carries a data dependency. Array analysis can also determine whether different loops access the disjoint subsets of a given object. Finally, array analysis can be used to derive the data correlation between iterations of separate loops.

One important aspect of multimedia applications is that they often have a range of supported sample rates, sizes, or resolutions and use many symbolic variables in the interest of code reuse. Dimensions determined at runtime create non-affine expressions and variable loop bounds, which stymie many simple array disambiguation tests. In these cases, value constraints analysis can be obtained or computed to assist the array disambiguation.

Value constraint analysis finds the information about the possible range or other constraints on a value, it can be critical in evaluating symbolic tests. Many variables in a code have a relatively small set of values during the majority of code execution, restricted by control flow tests or written constants.

Value relationship inference helps to find out the relationship between the values of different variables. Often, one variable is used to compute the value of several other variables. When related variables appear in an index expression, symbolic analyses typically lose precision unless they know the relationship between the variables. These relationships are found by tracking values back through def-use relationships to find common terms. This requires inter-procedural expression computation through memory objects, often dynamically-allocated, to find the relationships between values [54].

4.4.2 Structuring

S*tructuring* the threads of a task helps to maximize concurrency and minimize synchronization effort. Some structure patterns [55] used to parallelize a code are presented in the following.

The basic structure is *parallel threading*. Typically parallel threads are decomposed from an independent loop. When each iteration of a loop depends on different data, they can be separated into threads and executed in parallel via loop distribution as shown in Figure 4-7(a). When one loop produces a data that will be consumed by the following loop, and each iteration of the following loop only depends upon a limited and known number of iterations of the previous loop and does not overwrite the first loop's input data, it is possible to execute part of the two loops in parallel as long as the real data dependences are respected. Figure 4-7(b) shows such an example.

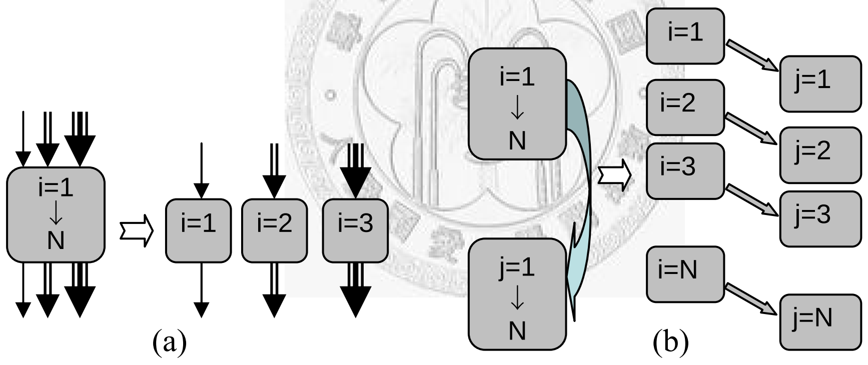
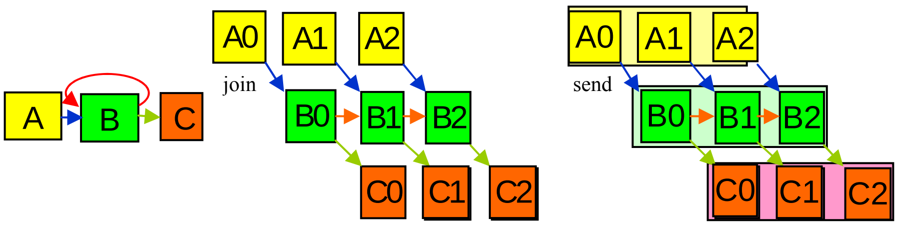


Figure 4-7. Loop parallelism.

The second structure is *pipeline threading*. This kind of structure can be derived by using the *software pipelining* technique as depicted in Figure 4-5. Load balance is a challenge in pipelining structure; it is restricted by loop-carried dependence. Figure 4-8 demonstrates an example. Figure 4-8(a) shows the data dependency graph of a loop body, where a loop-carried dependence is represented by a backward arc. Due to the existence of this arc, the graph has to be partitioned into three partial functions: A, B and C. Figure 4-8(b) shows that the iterations of all functions are partitioned into threads and schedules as pipelining, each thread should wait for its dependent thread to finish by `pthread_join`. Figure 4-8(b) is a message-passing implementation. Three threads are partitioned into three processors. Send/receive is used for synchronization, each work waits its requested data before it can run.



(a) DDG (b) Pipelined threads (c) 3 threads with communication

Figure 4-8. Pipeline thread structure.

The third structure is *task/data decomposition*. Assume that an algorithm has to compute the average and difference of RGB components in an image. The two jobs are independent; they can be parallelized by task decomposition. The processing on RGB components is independent, they can be parallelized by data decomposition.

Data decomposition structure is often used to handle large input data such that all threads can perform the same computation on different data areas. The data is often a multi-dimension array that can be decomposed into multi-dimension grids. *Single Program Multiple Data* (SPMD) coding style in which multiple threads run the same code is a way to save instruction space. Each thread needs a mechanism to distinguish its data grid. On thread creating, a parameter is put on its stack header as an argument; thread can use this parameter to identify its grid location.

Task decomposition is often applied by a *divide and conquer* approach. A complex task can usually be partitioned into many independent simpler sub-tasks. Recursive algorithms such as binary search is an example of divide and conquer, which main thread creates two child threads to search the two parts of an input database. Each child thread also creates two child threads until the search range is small enough.

4.5 SIMDization for Memory Access Redundancy Optimization

In memory access dominated applications, good memory access organization will greatly improve the performance. To reduce memory bandwidth, the memory reuse concept is often used in an optimized compiler [56]. When a compiler works on an array-based program, it will analyze reference patterns in the program to derive a linear transformation of the data, and reorganize the computation to reuse the data from memory hierarchy.

Memory misalignment is a critical problem when running multimedia applications on an SWP-SIMD core. Most RISC-based processor requires all memory accesses to be aligned, that is, the data loaded from memory cannot cross the 64-bit boundary if the processor is of 64-bit width. In an SWP-SIMD core, many memory components are packed as one superword to be accessed together. Although each component is under boundary, the packed one may cross the boundary. For example on processing RGB24 format picture, the packed element is 24-bit (8-bit for each of the Red, Green and Blue components), the third element has to cross the 64-bit boundary.

To adapt memory reuse concept for an SWP-SIMD core, avoiding misalignment is very important. This will increase the complexity of the linear transformation method. We introduce a graphical method to simplify the data organization analysis for an SWP-SIMD core. Our method sequentially selects an aligned basic block, and then parallelizes the operations bound to this block to avoid misalignment.

The first stage is to find a parallelizable memory load operation and its maximum parallelizable code range. From the memory load operation, the loop-carried data dependence is checked from the innermost loop to the outer loop. A parallelizable operation might be hidden under bad coding style. To find the largest parallelizable loop, all false dependences should be removed by the techniques introduced in Section 4.1. After dependence removal, an SCC in the data dependence graph as shown in Fig. 3 represents the largest parallelizable part of the algorithm. The memory load operation in a single-tone SCC is chosen to begin our procedure. A single-tone SCC memory load operation represents that all memory items covered by this operation are able to perform simultaneously. Thus we can reorganize the computation sequence to fit our SWP-SIMD PLX platform constraint.

After a parallelizable memory load operation in the largest loop is determined, we have to examine its cover area and redundancy. The memory cover area is obtained by exploring the memory index in the loop. For a multimedia application, the cover area usually forms a 2-D rectangle. The redundancy is obtained by dividing the load operation count by the cover area word count. For example, 8 load operations work on a 16-byte (two 64-bit words) area, the redundancy is 8/2=4. It means we can reduce memory access time to 1/4 if all reuses can be applied.

The second stage is to group operations from the store operation on the unrolled data flow graph. The group is used to select proper operations to pack into an SWP-SIMD core. From each store operation, operations can be grouped by backward tracing till the leaf of load operations. These operations are necessary to generate the store result. A data should be kept in register until it is written into memory. Grouping store operations as soon as possible can reduce the number of registers used for temporary data. Thus the operations in the same group have higher priority to pack together.

With these groups, we can examine the redundancy information. For a regular array-based code, each group may cover the same size of *load block* by their load operations. If a memory item can be reused for two blocks, their load blocks are overlapped on this item. The load block offset can be transformed into a linear form to help the block reorganization. For the image filter example in Section 4.1, the load block is 3×3, and the next block is one item right or down shift.

The third stage is to allocate an aligned basic block, which is the smallest memory area to preserve in register for reuse, and to be merged with the loaded blocks. The left-top load block is selected first. When its right edge is not aligned to the (64-bit) word boundary, the right load block, which is not overlapped, is merged into the basic block. With this basic block size, we can immediately know how many registers are necessary to buffer the basic block. If we have enough registers, the basic block can be extended right or down to increase register reuse.

The fourth stage is to analyze which groups need to reuse from this basic block. The groups whose load operations are binding to the first row of the basic block are chosen. These blocks may extend the load area to the right. The extra load result can be reused for the next basic block. These groups become the unit to generate an SWP-SIMD code.

The other groups can be generated by repeatedly applying the above four stages. The outer loop can first go either rightward (in the x-axis direction) or downward (in the y-axis direction). The direction which has a better reuse rate is chosen first.

We use three examples to explain this strategy. The first spatial-image filter example containing much redundant memory access will be described in detail. The second SAD example shows that its load block is not overlapped. And the third matrix multiplication shows the application of a simple loop-unrolling parallelization method.

4.5.1 Spatial Image Filter

A spatial image filter is a 2-D FIR (finite impulse response) filter, defined as:

$$g(x,y)=\sum_{i=0}^{K-1}\sum_{j=0}^{L-1}h(i,j)*f(x+i-\frac{K}{2},y+j-\frac{L}{2}),$$

where *f*(*x*,*y*) is the image value at position (*x*,*y*), *g*(*x*,*y*) is the result image value, and *h* is the filter impulse response matrix of size K×L. For example, a 3×3 sharpness filter which can emphasize object boarder is given as:

$$h=\begin{bmatrix}1 & 1 & 1\\ 1 & -7 & 1\\ 1 & 1 & 1\end{bmatrix}$$

The C code for a 96×96 image and 3×3 impulse response is listed as follows. The border of array *f* is extended and filled with zero to simplify FIR boundary management.

```
Code 4-1. Spatial image filter.
short f[98][98], g[98][98];
const short h[3][3]={{1,1,1},{1,-7,1},{1,1,1}};
register short R1,R2,R3,R4;
     for(y=1;y<=96;y++) {
       for(x=1;x<=96;x++) {
S0:      R4=0;
         for(j=0;j<3;j++) {
           for(i=0;i<3;i++) {
S1:          R1=f[y+j-1][x+i-1];
S2:          R2=h[j][i];
S3:          R3=R1*R2;
S4:          R4+=R3;
           }
         }
S5:      g[y][x]=R4;
       }
     }
```

The first stage is to find a parallelizable memory load operation, which is S1 in this example. The largest parallelizable loop in S1 is the loop indexed by y. The S1 operation count is 96×96×3×3=82944, and the size of f array is 98×98×2=19208 bytes. This will derive a redundancy of 82944/(19208×8/64)=34. Since the redundancy value is large, we can expect to get good performance improvement by memory reuse.

The second stage is to group operations from each store instruction. The store operations in S5 are within the loop indexed by y and x. This group is shown as the blue area in Figure 4-9. The left-top load block `f[0:2][0:2]` covered by the group of index `{y,x}={1,1}` is the purple area. The binding of load operations and memory items are also shown on Figure 4-9.

The third stage is to allocate an aligned basic block. The left-top load block `f[0:2][0:2]` is not aligned to the 64-bit boundary. The right non-overlapped block `f[0:2][3:5]` belongs to the group indexed by `{y,x}={1,4}`, which is not aligned either. The next two blocks `f[0:2][6:8]` and `f[0:2][9:11]` are added to reach the alignment boundary. The aligned basic block is `f[0:2][0:11]`, and it belongs to groups indexed by `{y,x}={1,1},{1,4},{1,7},{1,10}`. Nine registers are used to buffer this basic block.

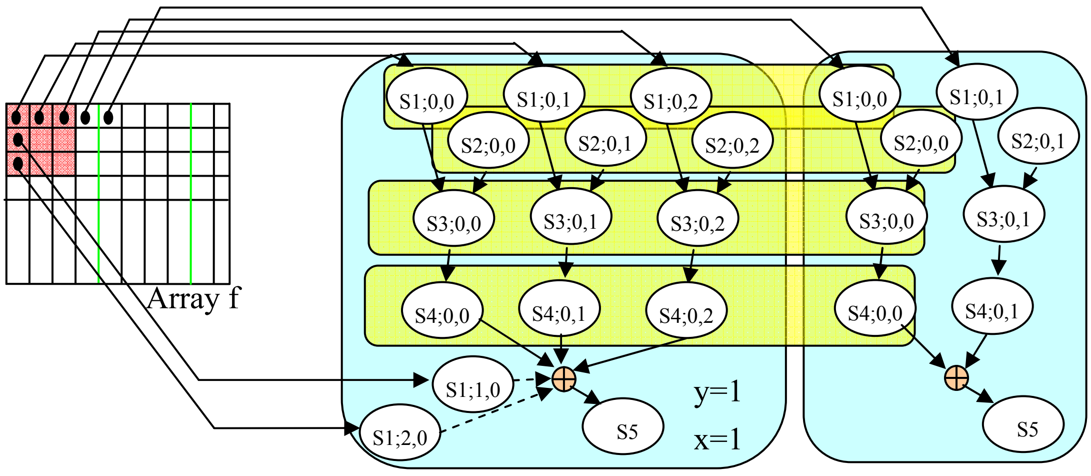


Figure 4-9. SWP-SIMD for Code 4-1.

The load operations bound to the first word of the basic block are `y+j-1=0` and `0 ≤ x+i-1 ≤ 3`, and they contain S1 in iterations `{y,x,j,i}={1,1,0,0:2},{1,2,0,0:2},{1,3,0,0:1}` and `{1,4,0,0}`, having 9 operations in total. `S1{1,1,0,0:2}` and `S1{1,4,0,0}` are within the basic block groups, they are packed to generate first vector load instruction. By similar analysis, `S1{1,2,0,0:2}` will induce a block `f[0:2][1:12]` by groups of `{y,x}={1,2},{1,5},{1,8},{1,11}`. The block is overlapped on basic block with 1 item offset. Thus the left 11 columns can be shifted out from the 9 registers of the basic block, but the rightmost column needs to be loaded. The extended load block `f[0:2][0:15]` uses 12 registers to buffer. `S1{1,3,0,0:1}` induces a block `f[0:2][2:13]` which is able to be reused from the extended load block. Thus, groups of `{y,x}={1,3},{1,6},{1,9},{1,12}` are selected.

The SWP-SIMD instructions can be generated by processing the remaining data flow graph. It begins from the first generated vector load instruction, shown as the yellow area in Figure 4-9. The constant assignment S2 is expanded to fit the SWP-SIMD size. The summation operation S4 is converted into partial summation. The next step can move to the right (x-index) or down (y-index) in a similar way. The right basic block `f[0:2][13:24]` targets groups `{y,x}={1,13:24}`. The overlapped area with the extended block is f[0:2][12:15], where 27 load operations of the target groups are bound to it. The down basic block f[1:3][0:11] targets groups {y,x}={2,1:12}. The overlapped area is f[1:2][0:12], where 72 load operations of the selected groups are bound to it. The better reuse rate is y-axis.

The result pseudo code is built as follows. The outmost loop is x, increased by 12. The inner loop is y, increased by 1. The basic block is loaded into 12 RL registers at the beginning of y loop. At y=1, the 12 registers are loaded from memory, as listed in SLa. The following y loop only needs to load 4 registers, the other 8 registers are able to be reused from the previous iteration, as listed in SLb. The loop k represents the three reuse phases. The basic block is the one obtained when k=0. The other k loops are the two reused phases indexed by S1{1,2,0,0:2} and S1{1,3,0,0:1} as discussed above. The innermost loop j contains the three rows of a block. In loop j, R1 is shifted out from RL by an index of k. Finally, partial summation should be summarized as S4b.

Code 4-2. SWP-SIMD code of spatial image filter.

```
  short f[98][98], g[98][98];
  const short h[3][3]={{1,1,1},{1,-7,1},{1,1,1}};
  register packed_short RL[3][0:15];
  register packed_short R1[0:11],R2[0:11],R3[0:11],
                        R4[0:11],Rs[0:11];
       for(y=1;y<=96;y+=12) {
          for(x=1;x<=96;x++) {
S0:        R4[j][0:11]=0;
           if (y==1) { //first y, load from memory
SLa:          for(j=0;j<2;j++) {
                for(i=0;i<4;i++) RL[y+j-1][i*4:i*4+3]=
                                     *(&f[j][x+i*4]);
             }
           } else { //reuse 2 rows from last y
SLb:          for(j=0;j<2;j++) {
                 for(i=0;i<4;i++) RL[j][i*4:i*4+3]=
                                  RL[j+1][i*4:i*4+3];
                for(i=0;i<4;i++) RL[2][i*4:i*4+3]=
                                 *(&f[y+1][x+i*4]);
          }
          for(k=0;k<3;k++) { //k=0:basic block,
                               //others: overlapped block
             for(j=0;j<3;j++) {
S1:            R1[0:11]=RL[j][0:15] << (k*16);
S2:            R2[0:11]={h[j][0:2],h[j][0:2],
                         h[j][0:2],h[j][0:2]};
S3:            R3[0:11]=R1[0:11]*R2[0:11];
S4a:           R4[0:11]+=R3[0][0:11];
             }
             //final sequential summation
S4b:          Rs[k]=R4[0][0]+ R4[0][1]+ R4[0][2];
              Rs[k+3]=R4[0][3]+ R4[0][4]+ R4[0][5];
              Rs[k+6]=R4[0][6]+ R4[0][7]+ R4[0][8];
              Rs[k+9]=R4[0][9]+ R4[0][10]+ R4[0][11];
          } //k
S5:       for(j=0;j<12;j+=4) g[y][x+j:x+j+3]=Rs[j:j+3];
        }//y
      }//x
```

The concept of memory access redundancy is represented in Figure 4-10. Figure 4-10(a) is the aligned basic block decided at the third stage. On the second x iteration, the load block moves right as shown in Figure 4-10(b). The purple area is overlapped with Figure 4-10(a) that can move from the load registers. Only the red area needs to load. The red area is larger than the needed block, and it is used at the third x iteration as shown in Figure 4-10(c). Thus the third x iteration does not load any memory. Figure 4-10(d) shows the load block of the second y iteration.

While there is no loop-carried dependence in Code 4-2, it is parallelizable by multi-thread, and loop x and loop y are exchangeable. By careful code scheduling, the memory access redundancy of x-axis and y-axis can be combined to get more performance improvement. Table 4-1 shows the results of 4 configurations. The first two configurations are single thread by putting x or y at the outer loop. The last two configurations uses two threads, their performances are better than the single thread configuration.

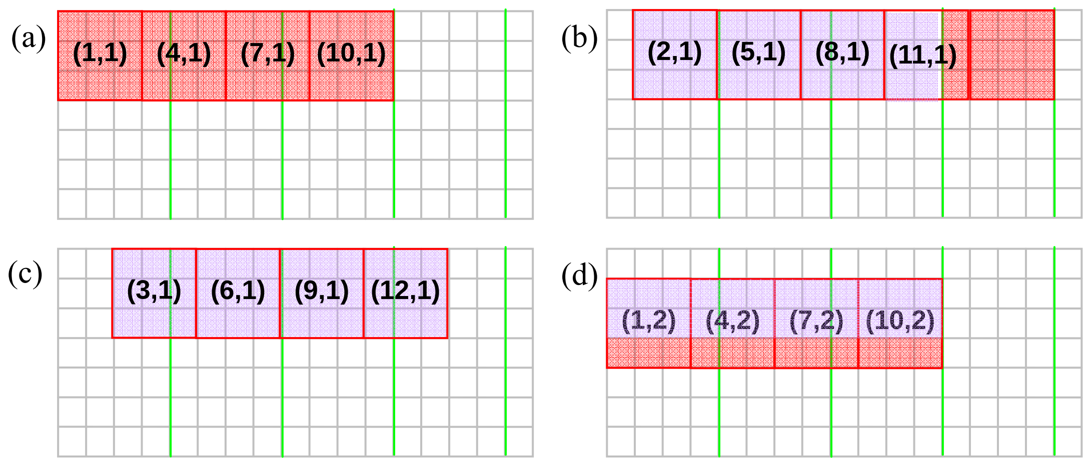


Figure 4-10. Memory access redundancy of Spatial Image Filter.

Table 4-1. Performance on four configurations of Spatial Image Filters.

	Thread num	Outmost loop	Second loop	Cycles
A	1	y	x	51448
B	1	x	y	38812
C	2	y	x	30613
D	2	x	y	27168

4.5.2 SAD

H.264/AVC is one of the newest video encoding standards. The basic processing unit in H.264 is a 16×16 macro-block, and a picture has to be partitioned into macro-blocks before processing. In video encoding, motion estimation is the major computational part, which occupies 78% of the computation power [57]. In video encoding, a prediction block is formed based on previously encoded and reconstructed blocks with *motion vectors* (MV). A motion vector is defined as the displacement of an encoding (current frame) block and a reconstructed (reference frame) block that yields the minimum *sum-of-absolute-difference* (SAD) value in a search range. In summary, SAD is a criterion used to gauge the similarity between two blocks.

For a K×L block, one has

$$SAD(m,n) = \sum_{i=0}^{K-1} \sum_{j=0}^{L-1} | C(i,j) - R(m+i, n+j) | ,$$

where $C(i, j)$ is the luminance value of a current frame pixel and $R(i, j)$ is the luminance value of a reference frame pixel. Argument (m,n) is the displacement of two blocks, and K×L is the block size. A block can be a 16×16 macro-block, or one of 40 sub-blocks including sixteen 4×4, eight 8×4, eight 4×8, four 8×8, two 16×8, and two 8×16 blocks. The SAD of a larger sub-block can be derived from the smallest sixteen 4×4 SADs. Thus, the basic operation is to calculate the sixteen 4×4 SADs.

Figure 2-6 shows the sixteen 4×4 sub-blocks ordering in a 16×16 macro-block. The algorithm used to calculate the sixteen 4×4 SADs on displacement (m,n) is shown in the following Code 4-3. PICW and PICH are picture width and height, BX and BY are macroblock location, sb is the sub-block number, and row and col are row and column numbers in a sub-block.

There are two memory load operations: S1 and S2. The largest parallelizable loop in S1 is the loop indexed by sb. The S1 operation count is 16×4×4=256, and the C array size used is 16×16=256 bytes. Thus, the redundancy is 256/(256×8/64)=8. This redundancy value is the same as the SWP-SIMD vector length, which means no memory can be reused. The group of store operations in S5 is shown as the blue area in Figure 4-11.

Code 4-3. Calculation of sixteen 4×4 SADs.

```
unsigned char C[PICH][PICW]; //current_frame;
unsigned char R[PICH][PICW]; //reference_frame;
register unsigned char R1,R2;
register short R3,R4, SAD[16];
     for(sb=0;sb<16;sb++) {
  S0:   R4=0;
         for(row=0;row<4;row++) {
           for(col=0;col<4;col++) {
  S1:        R1=C[BY+sb/4+row][BX+(sb%4)*4+col];
  S2:        R2=R[BY+n+sb/4+row][BX+m+(sb%4)*4+col];
S3:         R3= R1-R2;
S4:         R3=abs(R3);
S5:         R4+=R3;
           }
       }
S6:    SAD[sb]=R4;
  }
```

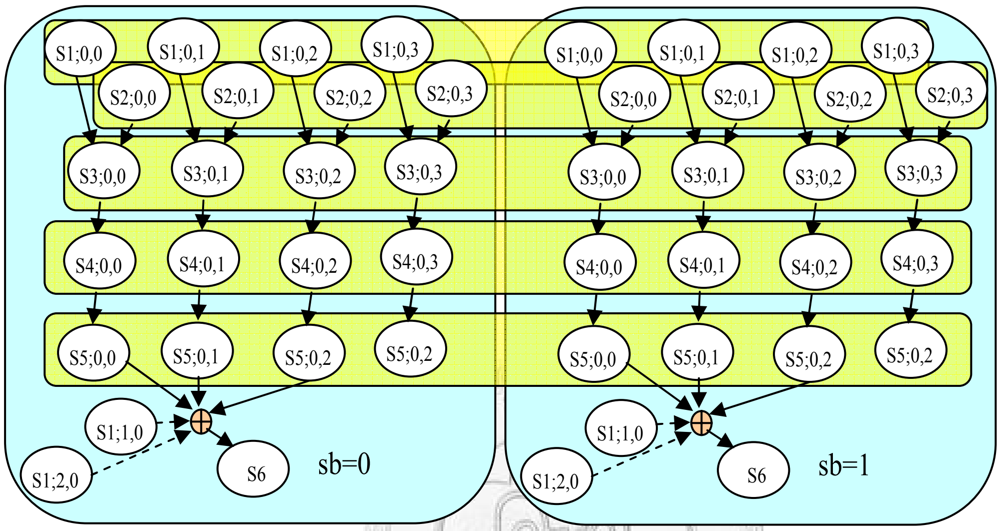


Figure 4-11. SWP-SIMD for Code 4-3.

The left-top load block `C[BY:BY+3][BX:BX+3]` belongs to the group indexed by sb=0, which is not aligned to the 64-bit boundary. Its right non-overlapped block `C[BY:BY+3][BX+4:BX+7]` belongs to the group indexed by sb=1, which is aligned. Only four registers are used as the buffer memory. It has room to expand the basic block, which is decided as `C[BY:BY+3][BX:BX+15]` that uses eight registers.

To deal with the outer loop, where no load block can leave rightward (in the x-axis direction), we can only go downward (in the y-axis direction). And we observe that all load operations bound to the basic block were transformed already, the next block `C[BY+4:BY+7][BX:BX+15]` to deal with is not overlapped with the basic block. The result pseudo code is built as follows. The outmost loop is indexed by sb. While the basic block does not need to buffer for reuse, and the remaining four rows behave the same, they are rolled into a loop indexed row-wise. While alignment check is based on array C, alignment may occur at array R. This can be checked by offset m. While S2b contains misaligned access, we can use one additional load and one shift to maintain the alignment. While S3 works on 16-bit precision, the 8-bit precision R1b and R2b should be expanded into 16-bit precision by using a permutation instruction, as listed in S1w and S2w.

Code 4-4. SWP-SIMD result of 16 4×4 SADs.

```
unsigned char C[PICH][PICW]; //current_frame;
unsigned char R[PICH][PICW]; //reference_frame;
register packed_byte R1b[0:15], R2b[0:23];
 register packed_short R1w[0:15], R2w[0:15], R3[0:15],
                        R4[0:15], Rs[0:3];
   for(sb=0;sb<16;sb+=4) {
 S0:    R4[0:15]=0;
       for(row=0;row<4;row++)
 S1:      for(j=0;j<2;j++) R1[j*8:j*8+7]=
                            *(&C[BY+sb/4+row][BX]);
         If ((m%16)==0) {
 S2a:       for(j=0;j<2;j++) R2[j*8:j*8+7]=
                        *(&R[BY+n+sb/4+row][BX+m]);
         } else {
 S2b:       for(j=0;j<3;j++) R2[j*8:j*8+7]=
                    *(&R[BY+n+sb/4+row][BX+m-(m%15)]);
           R2[0:15]=R2[0:23]<<((m%16)*8);
         }
 S1w:     R1w[0:15]=byte2short(R1b[0:15]);
 S2w:     R2w[0:15]=byte2short(R2b[0:15]);
 S3:      R3[0:15]=R1w[0:15]-R2w[0:15];
 S4:      R3[0:15]=abs(R3[0:15]);
 S5a:     R4[0:15]+=R3[0:15];
       }
 S5b:  Rs[sb]=R4[0]+R4[1]+R4[2]+R4[3];
       Rs[sb+1]=R4[4]+R4[5]+R4[6]+R4[7];
       Rs[sb+2]=R4[8]+R4[9]+R4[10]+R4[11];
       Rs[sb+3]=R4[12]+R4[13]+R4[14]+R4[15];
 S6:    SAD[sb:sb+3]=Rs[0:3];
   }
```

4.5.3 Matrix Multiplication

Matrix multiplication is a basic function module in many linear algebra programs. The equation is represented as $C=A\times B$ where C is an $M\times N$ matrix, A is an $M\times K$ matrix, and B is a $K\times N$ matrix. Sometimes B needs to be transposed to have $C=A\times B^T$, if B is an $N\times K$ matrix. The code for matrix multiplication is listed as follows.

```
Code 4-5. Matrix multiplication.
int A[M][K], B[K][N], C[M][N];
register int R1,R2,R3,R4;
       for(m=0;m<M;m++) {
         for(n=0;n<N;n++) {
  S0:      R4=0;
           for(k=0;k<K;k++) {
  S1:          R1=B[k][n];
  S2:          R2=A[m][k];
  S3:          R3=R2*R1;
  S4:          R4+=R3;
           }
  S5:      C[m][n]=R4;
         }
       }
```

In the above code, the entire matrix B is loaded at every m iterations. If the cache is large enough, matrix B will be loaded with data in cache at the first m iterations, and reuse data in cache at other m iterations. If the cache is not large enough, only part of matrix B's data could be loaded from main memory, which increases the execution cycle estimation complexity. In the following discussion, we assume that the cache is large enough to hold a whole matrix. The case of a large matrix multiplication will be discussed later.

Assume that a register has P subwords (register width=4P bytes for a float data type). All S1 and S2 are for loading a new block data. S1 has $M\times N\times K$ counts, each of the $M\times K/P$ loads take β cycles to load data from main memory; each of the others takes α cycle to reload data from cache. S2 has $M\times N\times K$ count, each of the $N\times K/P$ loads take β cycles to load from main memory; each of others takes α cycle to reload data from cache. The total execution count is then $(N\times K/P)\times\beta+(M\times N\times K-N\times K/P)\times$

α+(M×K/P)×β+(M×N×K-M×K/P)×α+M×N×(1+K×(1+1)+1)=(2MNK-NK/P-MK/P)α+(NK/P+ MK/P)β +2MNK+2MN.

The following Code 4-6 shows how to unroll the loop of index k in factor P to use the subword-parallel feature in a SIMD core.

```
Code 4-6. Second implementation of matrix multiplication.
float A[M][K], B[K][N], C[M][N];
register packed_float R1[0:P-1],R2[0:P-1],
                  R3[0:P-1],R4[0:P-1];
       for(m=0;m<M;m++) {
         for(n=0;n<N;n++) {
  S0:        R4[0:P-1]=0;
           for(k=0;k<K;k+=P) {
  S1:          for(j=0;j<P;j++) R1[j]=B[k+j][n];
  S2:          R2[0:P-1]=*(&A[m][k]);
  S3:          R3[0:P-1]=R2[0:P-1]*R1[0:P-1];
  S4:          R4[0:P-1]+=R3[0:P-1];
           }
  S5:         C[m][n]=R4[0]+R4[1]+… +R4[P-1];
         }
       }
```

P times of loads in S2 are merged into one load operation. S1 loads a matrix B in the vertical direction, which cannot be merged. Instead, P load and P-1 pack operations are used. S3 and S4 can be combined into subword-parallel. The final S5 summation takes 2P cycles. If K is a multiple of P, the total execution count is then (N×K/P)×β+(M×N×K-N×K/P)×α+(M×K/P)×β+(M×N×K/P-M×K/P)×α+M×N×(1+K/P×(1+1)+2P)=(MNK+MNK/P-NK/P-MK/P)α+(NK/P+MK/P)β+2MNK/P+2MNP+MN.

The inefficiency of S1 load wastes too much time. We try to unroll the loop of index n and keep the loop of index k.

```
Code 4-7. Third implementation of matrix multiplication.
float A[M][K], B[K][N], C[M][N];
register packed_float R1[0:P-1],R2[0:P-1],
                  R3[0:P-1],R4[0:P-1];
```

```
       for(m=0;m<M;m++) {
         for(n=0;n<N;n+=P) {
S0:        R4[0:P-1]=0;
           for(k=0;k<K;k++) {
S1:            R1[0:P-1]=*(&B[k][n]);
S2:            R2[0]=*(&A[m][k]);
S6:            R2[0:P-1]={R2[0],…,R2[0]};
S3:            R3[0:P-1]=R2[0:P-1]*R1[0:P-1];
S4:            R4[0:P-1]+=R3[0:P-1];
           }
S5:          C[m][n:n+P-1]=R4[0:P-1];
         }
       }
```

Loop n is unrolled in factor P. While elements in matrix A are the same for all n iterations, they are loaded in S2 and duplicated to fill the register in S6. P loads of S1 are merged into one load operation. The total execution count is $(N\times K/P)\times\beta+(M\times N/P\times K-N\times K/P)\times\alpha+(M\times K/P)\times\beta+(M\times N/P\times K-M\times K/P)\times\alpha+M\times N/P\times(1+K\times(1+1+1)+1)=(2MNK/P-NK/P-MK/P)\alpha+(NK/P+MK/P)\beta+3MNK/P+2MN/P$.

Compared to Code 4-6, the load counts of S1 and final sequential summation are both much reduced; thus the performance of Code 4-7 is better than Code 4-6. The main reason is that applying subword-parallel on row-major operations can improve them. Code 4-7 works by partitioning a matrix C into many 1×P sub-matrices, and the P multiplications can be calculated in parallel.

An M×N matrix multiplication can be partitioned into sub-matrix multiplications as follows.

$$C_{pq}=\sum_{r=0}^{\frac{K}{J}-1}A_{pr}B_{rq},0\le p<\frac{M}{H},0\le q<\frac{N}{G}$$

Matrix C is partitioned into $\frac{M}{H}\times\frac{N}{G}$ sub-matrices, each sub-matrix is of H×G size. The size of sub-matrix A is H×J, and the size of sub-matrix B is J×G. Code 4-7 is the special case of H=1, J=1 and G=P. Generally for H, G and J, the sub-matrix multiplication code is as follows.

Code 4-8. Fourth implementation of matrix multiplication

```
float A[M][K], B[K][N], C[M][N];
register packed_float R1[J][G/P][0:P-1],R5[0:P-1],
        R2[H][J/P][0:P-1],R3[0:P-1],R4[H][G/P][0:P-1];
      for(m=0;m<M;m+=H) {
        if (J==K) {
          for(h=0;h<H;h++)
             for(k=0;k<K;k+=P)
  S2a:          R2[h][k][0:P-1]=*(&A[m+h][k]);
        }
        for(n=0;n<N;n+=G) {
          for(i=0;i<H;i++)
            for(j=0;j<G;j+=P)
     S0:      R4[i][j/P][0:P-1]=0;
          for(k=0;k<K;k+=J) {
            for(j=0;j<J;j+)
              for(g=0;g<G;g+=P)
  S1:            R1[j][g/P][0:P-1]=*(&B[k+j][n+g]);
            if (J!=K) {
             for(h=0;h<H;h++)
               for(j=0;j<J;j+=P)
  S2b:           R2[h][j/P][0:P-1]=*(&A[m+h][k+j]);
            }
            for(h=0;h<H;h++) {
             for(g=0;g<G;g+=P) {
               for(j=0;j<J;j+=P) {
                 for(i=0;i<P;i++) {
  S6:               R5[0:P-1]=
                       {R2[h][j/P][i],…,R2[h][j/P][i]};
  S3:               R3[0:P-1]=
                        R5[0:P-1]*R1[j+i][g/P][0:P-1];
  S4:               R4[h][g/P][0:P-1]+=R3[0:P-1];
                 } //i
               } //j
             } //g
           } //h
          } //k
          for(i=0;i<H;i++)
          for(j=0;j<G;j+=P)
  S5:       C[m+i][n+j:n+j+P-1]=R4[i][j/P][0:P-1];
        } //n
      } //m
```

In every k loop, the H×J elements of sub-matrix A are loaded into R2 by S2b, and the J×G elements of sub-matrix B are loaded into R1 by S1. When J is equal to K, H×J sub-matrix occupies a full row of matrix A, R2 remains unchanged for all n iterations, thus it can be moved out of loop n to S2a. This kind of load redundancy removing can deliver better performance.

For J=K, the total execution count is then:

$$(N\times K/P)\times\beta+(M/H\times N/G\times K\times G/P-N\times K/P)\times\alpha+(M\times K/P)\times\beta+$$
$$(M/H\times H\times K/P-M\times K/P)\times\alpha+M/H\times N/G$$
$$\times(H\times G/P+K/K\times(H\times G/P\times K\times(1+1+1))+H\times G/P)$$
$$=(MNK/HP-NK/P)\alpha+(NK/P+MK/P)\beta+2MN/P+3MNK/P.$$

This equation is not related to parameter G, it decreases when H increases. The reason is that the count of S2a becomes constant to load a matrix A, redundancy only exists on S1 that has relation to H. The minimum execution count occurs on the largest H count derived from the register used.

Code 4-8 uses $JH/P+JG/P+HG/P+2$ registers, it should be less than the available register number Q, as shown in the following inequation:

$$JH/P+JG/P+HG/P+2 \leq Q$$

Maximum H is represented as:

$$H \leq (PQ-JG-2P)/(J+G)$$

H increases when G decreases. To use subword-parallel feature, G should be a multiple of P, and the minimum is P. The maximum H for $Q=32,\ J=K=16$ and $G=P=4$ is $H=[(4\times32-8\times4-2\times4)/(8+4)]=2.$

The inequation of J then becomes:

$$J \leq (PQ-HG-2P)/(H+G) \tag{4-1}$$

When maximum J occurs at minimum H=1 and G=P, the above inequation becomes

$$J \leq (PQ-P-2P)/(1+P)$$

When K is greater than the maximum J, a register is not large enough to hold a full row of matrix A, so R1 should be reloaded at each n iteration. The total execution count of Code 4-8 with J!=K is

$$\begin{aligned}\text{count} &= (N\times K/P)\times\beta+(M/H\times N/G\times K/J\times J\times G/P-N\times K/P)\times\alpha+(M\times K/P)\times\beta \\ &\quad +(M/H\times N/G\times K/J\times H\times J/P-M\times K/P)\times\alpha+M/H\times N/G\times \\ &\quad (H\times G/P+K/J\times(H\times G/P\times J\times(1+1+1))+H\times G/P) \\ &= (MNK/HP+MNK/GP-NK/P-MK/P)\alpha+ \\ &\quad (NK/P+MK/P)\beta+2MN/P+ 3MNK/P. \qquad (4\text{-}2)\end{aligned}$$

The above equation is not related to J, that is, the partition of dimension K does not affect the execution cycle. We can select a minimum J (=P) to reserve register for other dimensions.

By the basic arithmetic average theory:

$$\frac{H+G}{2} \geq \sqrt{HG}$$

The lower bound of Eq (4-2) is

$$\begin{aligned}\text{count} &\geq ((MNK/P)(2/\sqrt{HG}))\alpha+(-NK/P-MK/P)\alpha+ \\ &\quad (NK/P+MK/P)\beta+ 2MN/P+ 3MNK/P.\end{aligned}$$

The minimum execution count will occur at HG which is maximum and satisfies Eq (4-1). This is an integer programming problem to be solved. Since the available register number Q is not large, if the solution space is small, it can be directly counted. For example, if Q=32, J=P=4, the maximum HG is 56. While G should be a multiple of P to utilize subword-parallel feature, we can select G=8 and H=7.

In geometric view, the larger the sub-matrix of C, the fewer iteration needed. The maximum HG is the maximum size of sub-matrix C.

Large matrix multiplication can be partitioned with similar thinking. The register-cache relation is extended to cache-memory relation. When a matrix is too large to fit in the cache, it can be partitioned into smaller sub-matrices that can fit in the cache, and perform sub-matrix multiplication. If the sub-matrix of C is H’ ×G’, sub-matrix of A is H’ ×J’, sub-matrix of B is J’ ×G’, and cache size is Q’, to fit the 3 sub-matrices into a cache, the inequation is”

$$J'H'+J'G'+H'G'\le Q' \tag{4-3}$$

Similar to the discussion on Code 4-8, if $J'=K$, the sub-matrix $H'\times J'$ can fill a full row of matrix A, and remains in cache with an N dimension moving, the performance is the best. If the cache size is not large enough to fit $J'=K$, assign J' to a minimum of P, and select a maximum $H'\times G'$, where G' must be a multiple of P and satisfy Eq. (4-3).

The above discussion requires that the matrix dimensions N and K are a multiple of the subword capacity P to fully utilize the subword-parallel feature. If they are not a multiple of P, the rightmost or bottommost sub-matrix multiplication should work under lower parallelism.

When $M=1$, the matrix multiplication changes to a vector-matrix multiplication. Eq. (4-2) is able to use for M=1 and H=1, the minimum execution counts occur at maximum G that satisfies Eq. (4-1).

When N=1, the matrix multiplication changes to a matrix-vector multiplication. A one-dimension vector is stored as a row in memory. Elements in a row can be loaded together, the P load operations in S1 of Code 4-6 can be merged into one load operation, thus Code 4-6 can be used for $N=1$.

When $K=1$, matrix A is a column vector and matrix B is a row vector, and the resultant matrix C is an M×N array built of vector scalars. The first resultant row is a vector with B scalars of A[0], and the second resultant row is a vector with B scalars of A[1]. The result is the same as Code 4-7 with $K=1$.

4.5.4 Performance Analysis

To calculate the performance, some platform parameters, such as cache block size and memory access latency, are needed.

The latency of loading data from main memory is α cycles, and the burst length is 4 for a 32-bit SDRAM. The cache word line is 32 bytes, and the latency of cache hit load is β cycles. The cache contains a line buffer; sequentially loading from the line buffer needs γ cycles of latency. On a portable device running at 100MHz clock, the typical value of β is 7, which includes address calculation, bus issue, SRAM address assert, SRAM data load, send to store stage, write into register buffer, and fetch into operand. The average value of α is 20, which contains cache miss, external bus request, and SDRAM refresh wait. Cache size is assumed to be large enough to buffer all data.

To focus on comparing load reuse, memory store latency and all arithmetic instructions are assumed to be 1-cycle. All codes have been unrolled to remove the jump invalidation penalty and memory address calculation overhead. The execution cycles of above 3 codes are listed in Table 4-2, assuming α=20, β=7, γ=3, M=N=K=32, and P=8. Without loss of generality, we assume that data in the current frame exist in cache, and those in the reference frame are loaded from main memory for the SAD example.

We can observe from Table 4-2 that memory access latency of the Spatial Image Filter example is largely reduced, but ALU execution has only two times speedup. It is because the additional shift operation SLb and the final sequential summation S4b occupy a large part of the execution time. In the SAD and Matrix Multiplication examples, loaded data are not reused; their speedup is contributed to SWP-SIMD parallelization.

Table 4-2. Performance on SIMDization of the three Examples.

	Original		SWP-SIMD		Speedup	
	Memory	Total	Memory	Total	M	T
Spatial Image Filter	588α+27550β+54806γ=369028	MT+267264=636292	588α+686β+1862γ=22148	MT+124320=146468	16.60	4.34
SAD	16α+496γ=1808	MT+800=2608	16α+48γ=464	MT+136=600	3.89	4.34
Matrix Multiplication	(MK/8+NK/8)α+(2MNK-MK/8-NK/8-3MNK/4)β+(3MNK/4)γ=363776	MT+2MNK+2MN=431360	(MK/8+NK/8)α+(MNK/4P+2MNK/P-MK/8-NK/8)β+(3MNK/4P+(P-4)MNK/2P)γ=101632	MT+3MNK/4+MN/2=126720	3.57	3.40

CHAPTER FIVE
CONCLUSION

We had designed a PLX-based multi-processor system-on-chip. The system design was started with knowledge obtained from many multimedia applications. By analyzing multimedia applications, we decided that the processor needs an SWP-SIMD instruction set to process low-resolution pictures in a data level parallelism way. The first version PLX chip can run at 260MHz.

Multimedia applications can be parallelized on thread level parallelism. The simultaneous multi-threading technique improves processor performance/power efficiency by increasing ALU pipeline stages and removing bypass logics. A VLIW/SIMD instruction-level configurable issue-logic design enables 32-bit scalar operations to work more efficiently in a 64-bit core. The improved PLX2 chip can run at 520MHz.

To enable messages be sent directly core-to-core to reduce communication traffic, a message-passing over private cache design is introduced by configuring the cache to perform as FIFO. To reduce the programmer effort, an OpenMP to TLM tool is made to transform OpenMP code into an MPI code. A SystemC TLM platform was introduced for system level hardware/software co-design and co-verification.

A parallelization tool is made to transform fine grain loop tailored for a PLX SWP-SIMD feature. It focuses on reducing memory access redundancy by aligning memory boundary and reusing the memory content loaded in the register and cache.

For the cost/power efficiency and programmability purposes, a system-on-chip embedded with an application-specific instruction set processor is necessary at the nano-meter era. A complicated hardware-in-the-loop design flow induces heavy work on engineers who have to develop system level models, a multi-level parallelized processor, a parallel compiler, and a real-time OS for the multi-core SoC. This Dissertation gives an overall introduction to materials on all these knowledge domains. The experiences gained in the implementation of a PLX processor by system-level design and verification tools can shorten the design cycle. We hope that our success will encourage more multi-core system research and implementation.

REFERENCES

[1] R. Bergamaschi, L. Benini, K. Flautner, W. Kruijtzer, A. Sangiovanni-Vincentelli, and K. Wakabayashi, "The State of ESL Design," *IEEE Design & Test of Computers*, vol. 25, no. 6, pp. 510-519, Nov. 2008.

[2] R. Gupta, Arvind, G. Berry, and F. Brewer, "Advances in ESL Design," *IEEE Design & Test of Computers*, vol. 25, no. 6, pp. 510-519, Nov. 2008.

[3] R. B. Lee and A. M. Fiskiran, "PLX: a Fully Subword-Parallel Instruction Set Architecture for Fast Scalable Multimedia Processing," *Proceedings of IEEE International Conference on Multimedia and Expo*, pp.117-120, Aug. 2002.

[4] R. B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22-32, Apr. 1995.

[5] R. B. Lee and A. M. Fiskiran, "PLX: An Instruction Set Architecture and Testbed for Multimedia Information Processing," *Journal of VLSI Signal Processing*, vol. 40, no. 1, pp. 85-108, May 2005.

[6] Y. Cao and H. Yasuura, "A System-Level Energy Minimization using Datapath Optimization," *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 231-236, Aug. 2001.

[7] T. Ishihara and H. Yasuura, "Programmable Power Management Architecture for Power Reduction," *IEICE Transactions on Electronics*, vol. E81-C, no. 9, pp.1473-1480, Sep. 1998.

[8] A. Sinha, A. Wang, and A. P. Chandrakasan, "Algorithmic Transforms for Efficient Energy Scalable Computation," *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 31-36, Jul. 2000.

[9] N. Kroupis, M. Dasygenis, K. Markou, D. Soudris and A. Thanailakis, "A Modified Spiral Search Motion Estimation Algorithm and its Embedded System Implementation," *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 347-350, May 2005.

[10] http://focus.ti.com/lit/ug/spru538/spru538.pdf

[11] V. Kathail, M. Schlansker and B. Rau, "HPL-PD Architecture Specification: Version 1.1," *Technical Report HPL-93-80*, http://www.hpl.hp.com/techreports/93/HPL-93-80R1.html, Hewlett-Packard Laboratories, Feb. 2000.

[12] T. R. Gross and J. L. Hennessy, "Optimizing Delayed Branches," *Proceedings of the 15th Annual Workshop on Microprogramming*, ACM SIGMICRO, pp. 114-120, Oct. 1982.

[13] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392-403, Jun. 1995.

[14] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 36-46, Feb. 2002.

[15] P. L. Montgomery, “Modular Multiplication without Trial Division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519-521, Apr. 1985.

[16] A. Agarwal, R. Simon. M. Horowitz, and J. Hennessy, "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of the 15th Annul International Symposium on Computer Architecture*, pp. 280-289, Jun. 1988.

[17] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, Ke. O'Brien, and Ka. O'Brien, "A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor," *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, pp. 125-140, Oct. 2007.

[18] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," *Proceedings of the 38th Conference on Design Automation*, pp. 684-689, Jul. 2001.

[19] S. Vangali, J. Howard, G. Ruhi, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskotel, and N. Borkarl, "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS," *Proceedings of IEEE International Solid-State Circuits Conference*, pp. 98-589, Feb. 2007.

[20] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: Raw Machines," *IEEE Computer*, vol. 30, no. 9, pp. 86-93, Sep. 1997.

[21] C. J. Glass and L. M. Ni, "The Turn Model for Adaptive Routing," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 278-287, May 1992.

[22] G. M. Chiu, "The Odd-even Turn Model for Adaptive Routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 7, pp. 729-738, Jul. 2000.

[23] Y. C. Lan, M. C. Chen, A. P. Su, Y. H. Hu, and S. J. Chen, "Fluidity Concept for NoC: A Congestion Avoidance and Relief Routing Scheme," *Proceedings of IEEE International SOC Conference (SOCC)*, pp. 65-70, Sep. 2008.

[24] A. Mekkittikul and N. McKeown, "A Practical Scheduling Algorithm to Achieve 100% Throughput in Input-Queued Switches," *Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 2, pp. 792-799, Mar.-Apr. 1998.

[25] Y. C. Lan, M. Chen, A. Su, Y. H. Hu, and S. J. Chen, “Flow Maximization for NoC Routing Algorithms,” *Proceedings of IEEE Computer Society Annul Symposium on VLSI*, pp. 335-340, Apr. 2008.

[26] D. M. Chapiro, "Globally-Asynchronous Locally-Synchronous systems," *Ph.D. dissertation*, Stanford University, Stanford, California, USA, Oct. 1984.

[27] E. Nigussie, J. Plosila, and J. Isoaho, "Delay-Insensitive On-chip Communication Link using Low-swing Simultaneous Bidirectional Signaling," *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, pp. 217-222, Mar. 2006.

[28] T. Verhoeff, "Delay Insensitive Codes--an Overview," *Distributed Computing*, vol. 3, no. 1, pp. 1-8, Mar. 1988.

[29] R. Bashirullah, W. Liu, and R. K. Cavin III, "Current-Mode Signaling in Deep Submicrometer Global Interconnects," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 11, no. 3, pp. 406-417, Jun. 2003.

[30] Y. K Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406 - 471, Dec. 1999.

[31] M. K. Dhodhi, I. Ahmad, A. Yatama and I. Ahmad, "An Integrated Technique for Task Matching and Scheduling onto Distributed Heterogeneous Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1338-1361, Sep. 2002.

[32] http://www.systemc.org/groups

[33] A. Clouard, K. Jain, F. Ghenassia, L. Maillet-Contoz, and J. P. Strassen, "Using Transactional Level Models in a SoC Design Flow," in *SystemC Methodologies and Applications*, Chapter 2, pp. 29-63, Ed. W. Müller, W. Rosentiel, and J. Ruf, Kluwer Academic Publishers, 2003.

[34] http://www.mcs.anl.gov/mpi/standard.html

[35] http://standards.ieee.org/regauth/posix/

[36] http://openmp.org/wp

[37] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," *Proceedings of the International Conference on Compiler Construction*, pp.179-196, Apr. 2002.

[38] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, " Dependence Graphs and Compiler Optimizations," *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 207-218, Jan. 1981.

[39] K. Kennedy and R. Allen, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491-554, Oct. 1987.

[40] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452-471, Oct. 1991.

[41] A. Darte and F. Vivien, "A Classification of Nested Loops Parallelization Algorithms," *Proceedings of IEEE Symposium on Emerging Technologies and Factory Automation*, vol. 1, pp. 217-234, Oct. 1995.

[42] J. R. Allen, K. Kennedy, C. Porterfield and J. Warren, "Conversion of Control Dependence to Data Dependence," *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 177-189, Jan. 1983.

[43] R. Kramer, R. Gupta and M. L. Soffa, "The Combining DAG: a Technique for Parallel Data Flow Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 805-813, Aug. 1994.

[44] R. Tarjan, "Depth-first Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146-160, 1972.

[45] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD Architectures with Alignment Constraints," *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pp. 82-93, Jun. 2004.

[46] P. Wu, A. E. Eichenberger, and A. Wang, "Efficient SIMD Code Generation for Runtime Alignment and Length Conversion," *Proceedings of International Symposium on Code Generation and Optimization*, pp. 153-164, Mar. 2005.

[47] G. Ren, P. Wu, and D. Padua, "Optimizing Data Permutations for SIMD Devices," *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 118-131, Jun. 2006.

[48] S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 145-156, Jun. 2000.

[49] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient Utilization of SIMD Extensions," *Proceedings of IEEE Special Issue on Program Generation, Optimization, and Platform Adaptation*, vol. 93, no. 2, pp. 409-425, Feb. 2005.

[50] S. Larsen, R. Rabbah, and S. Amarasinghe. "Exploiting Vector Parallelism in Software Pipelined Loops," *Proceedings of the 38th International Symposium on Microarchitecture*, pp.119-129, Nov. 2005.

[51] D. M. Lavery and W. M. Hwu, "Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs," *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 126-137, Dec. 1996.

[52] G. H. Lin, S. J. Chen, R. B. Lee, and Y. H. Hu, "Memory Access Optimization of Motion Estimation Algorithms on a Native SIMD PLX Processor," *Proceedings of IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 567–570, Dec. 2006.

[53] S. Ryoo, S-Z Ueng, C. I. Rodrigues, R. E. Kidd, M. I. Frank, and W. M. Hwu, "Automatic Discovery of Coarse-Grained Parallelism in Media Applications," *Transactions on High-Performance Embedded Architectures and Compilers*, Springer, vol. 4050, pp. 194-213, Jan. 2007.

[54] P. Tu and D. Padua, "Gated SSA-based Demand-Driven Symbolic Analysis for Parallelizing Compilers," *Proceedings of International Conference on Supercomputing*, pp. 414-423, Jul. 1995.

[55] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*, Addison Wesley, 2005.

[56] 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 Transactions on Computers*, vol. 29, no. 12, pp.84-89, Dec. 1996.

[57] T.-C. Chen, S.-Y. Chien, Y.-W. Huang, C.-H. Tsai, C.-Y. Chen, T.-W. Chen, and L.-G. Chen, "Analysis and Architecture Design of an HDTV 720p 30 Frames/s H.264/AVC Encoder," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 6, pp. 673-688, Jun. 2006.

BIOGRAPHY

Education:
1991 B.S. Mechanical Engineering, National Chiao Tung University
1998 M.S. Graduate Institute of Electrical Engineering, National Taiwan University
2009 Ph.D. Graduate Institute of Electrical Engineering, National Taiwan University.

Published List:

International Journal:

1. Y. N. Wen, G. H. Lin, S. J. Chen, and Yu-Hen Hu, "Optimal Multiple-Bit Huffman Decoding," *IEEE Transactions on Circuits and Systems for Video Technology IEEE Circuits and Systems*, 2009.

Local Journal:

2. G. H. Lin, Y. N. Wen, X. L. Wu, S. J. Chen, and A. P. Su, “SIMD Code Generation for Multimedia Application,” *International Journal of Electrical Engineering*, vol.16, no. 1, Feb. 2009, pp. 1-12.

International Conferences:

3. P. H. Cheng, G. H. Lin, Y. P. Chen, T. N. Chien, J. S. Lai, and S. J. Chen, “Design of a Healthcare Standard Chip,” *The 8th World Multiconference on Systemics, Cybernetics and Informatics* (SCI), Orlando, Florida, USA, July 2004, vol. 12, pp. 128-132.
4. P. H. Cheng, F. M. Shyu, G. H. Lin, S. J. Chen, and J. S. Lai, “Moving Toward Healthcare Data Exchange Chip,” *Medinfo*, San Francisco, California, USA, pp. 1551, Sep. 2004.
5. P. H. Cheng, T. H. Yang, C. H. Yang, G. H. Lin, F. Lai, C. L. Chen, H. H. Lee, Y. S. Sun, J. S. Lai, S. J. Chen, “A Collaborative Knowledge Management Process for Implementing Healthcare Enterprise Information Systems,” *IEEE/IEE International Engineering Management Conference (IEMC)*, Newfoundland, Canada, pp. 604-607, Sep. 2005.
6. G. H. Lin, S. J. Chen, R. B. Lee, and Y. H. Hu, “Memory Access Optimization of Motion Estimation Algorithms on a Native SIMD PLX Processor,” *IEEE Asia-Pacific Conference on Circuits and System (APCCAS)*, Singapore, pp. 567-570, Dec. 2006.

7. G. H. Lin, Y. N. Wen, X. L. Wu, S. J. Chen, and Y. H. Hu, “Design of a SIMD Multimedia SoC Platform,” *IEEE International SOC Conference (SOCC)*, Hsinchu, Taiwan, ROC, pp. 51-54, Sep. 2007.
8. C. J. Wei, G. H. Lin, Y. N. Wen, S. J. Chen, and Y. H. Hu, “Symbolic Verification and Error Prediction Methodology,” *IEEE International SOC Conference (SOCC)*, Hsinchu, Taiwan, ROC, pp. 201-204, Sep. 2007.

Local Conferences:

9. Y. H. Hsieh, G. H. Lin, and S. J. Chen, " Design and Implementation of an RSA Encryption/Decryption Processor on IC Smart Card," *The 10th VLSI Design/CAD Symposium*, NanTou, Taiwan, ROC, pp. 343-346, Aug. 1999.
10. T. W. Chung, C. Yu, G. H. Lin, and S. J. Chen, “Design and Implementation of 2-D Discrete Wavelet Transform VLSI Architecture for JPEG2000,” *The 13th VLSI Design/CAD Symposium*, TaiTung, Taiwan, ROC, pp.363-366, Aug. 2002.
11. C. F. Yang, G. H. Lin, Y. H. Hsieh, and S. J. Chen, “A Dual-Mode Channel-Select Filter for WLAN and Bluetooth,” *The 15th VLSI Design/CAD Symposium*, Kenting, Taiwan, ROC, pp. 2-11, Aug. 2004.
12. G. H. Lin, Y. N. Wen, S. J. Chen, and Y. H. Hu, “Multimedia SoC System Level Virtual Platform Design,” *The 17th VLSI Design/CAD Symposium*, HuaLien, Taiwan, ROC, pp. 329-332, Aug. 2006.
13. G. H. Lin, C. C. Jean, S. J. Chen, and A. P. Su, “SIMD Code Generation for Multimedia,” *The 18th VLSI Design/CAD Symposium*, HuaLien, Taiwan, ROC, pp. 519-522, Aug. 2007.
14. C. J. Wei, G. H. Lin, Y. N. Wen, S. J. Chen, and Y. H. Hu, “Error Estimation for Saturation Arithmetic Functions,” *The 19th VLSI Design/CAD Symposium*, Kenting, Taiwan, ROC, pp. 110-113, Aug. 2008.

Book:

15. S. J. Chen, G. H. Lin, P. A. Hsiung, and Y. H. Hu, *Hardware Software Co-design of a Multimedia SoC Platform*, Springer, 2009.