

中国科学技术大学

University of Science and Technology of China

硕士学位论文



论文题目 轻量级自动机处理器的设计与实现

作者姓名 夏昊珺

学科专业 计算机系统结构

导师姓名 周学海教授 宫磊博士

完成时间 二〇二一年五月

中国科学技术大学

硕士学位论文



轻量级自动机处理器的设计与实现

作者姓名： 夏昊珺

学科专业： 计算机系统结构

导师姓名： 周学海 教授 宫磊 博士

完成时间： 二〇二一年五月二十九日

University of Science and Technology of China
A dissertation for master's degree



The design and implementation of a lightweight automata processor

Author: Xia Haojun

Speciality: Computer Architecture

Supervisors: Prof. Xuehai Zhou, Dr. Lei Gong

Finished time: May 29, 2021

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名： 夏昊谔

签字日期： 2021.5.31

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

公开 保密 (____年)

作者签名： 夏昊谔

导师签名： 何杰

签字日期： 2021.5.31

签字日期： 2021.5.31

摘 要

随着信息时代的到来，我们每天都在产生大量的数据。这其中，有大量非结构化的文本数据，例如在因特网上高速传送的网络数据包、日常收发的邮件、基因测序得到的生物基因序列和计算机系统记录的各种日志文件。一方面，我们可能需要出于安全的目的监控某个系统中的文本数据流。比如，我们可以对接收的 TCP 网络包进行深度的检测，从而保护计算机系统不被入侵和损坏。另一方面，我们可以从这些文本数据中挖掘很多有价值的信息。例如，通过分析分布式系统的日志文件，我们可以诊断出该系统近期出现的错误。在这些应用中，我们需要在大规模的输入字符流中同时查询和匹配众多指定的模式。这被称为多模式匹配。然而，CPU 和 GPU 不适用于高性能的多模式匹配计算。CPU 和 GPU 在进行多模式匹配计算时将产生大量不可预测的分支跳转和内存访问，这将引发频繁分支预测失败和 Cache Miss，从而大幅降低处理速度和能效。为了实现更高效的多模式匹配计算，本文基于领域专用架构设计和实现了一款轻量级自动机处理器 LAP。该处理器可使用较少的硬件资源实现高吞吐的多模式匹配计算。

本文的第一项主要工作是：提供高效的自动机处理器设计方案，支持快速的 ADFA 执行。基于自动机处理器进行多模式匹配计算时，我们需要将正则表达式集合编译为对应的有限自动机模型，并随后根据该自动机模型生成该专用处理器能够识别的二进制程序。在给定正则表达式集合的情况下，如果我们可以生成存储器空间占用更小的二进制程序，这将意味着我们的自动机处理器可以同时查找更多的模式串。为了实现该目的，本文实现了对 ADFA 模型（一种增强的 DFA 模型）的支持。ADFA 模型基于高效的增量存储算法，实现了有效的自动机模型压缩。因此，基于 ADFA 模型生成的二进制程序也非常小巧。然而，已有硬件解决方案在运行 ADFA 模型时处理速度却并不理想。在本文，我们提出的两条专用于 ADFA 模型的硬件指令。为了支持这两条新的指令，我们提出了新的自动机处理器硬件架构，同时修改了前人提出的编译器算法。实验表明当运行 ADFA 模型时，LAP 相比于另一款轻量级自动机处理器 UAP 实现了 32% 到 91% 的性能提升。不仅如此，我们设计并实现了一个 4 阶段的无停顿流水线，可以高效支持 LAP 指令集。我们使用 Verilog 语言实现了 LAP 处理器核心，并将其以 263 MHz 的主频部署到了 Xilinx Artix-7 FPGA 上。实验也表明，由于使用了 ADFA 模型和高效的编译算法，LAP 的二进制程序存储效率比 IBM 公司的 RegX 加速器和 Micron 公司的 AP 处理器高了 8 倍。

本文的第二项主要工作是：提供完整的设计方案，用于完整地构建基于 LAP 和 ARM CPU 的异构计算片上系统 LAP_SoC。该硬件系统主要包含三个部分：以

通用处理器为核心的通用计算系统；由 LAP 核心组成的面向多模式匹配任务的专用计算系统；以及负责大批量数据传输的 DMA 硬件控制器组。该系统可以实际运行在 FPGA 板卡上，并被用来解决生活中实际的多模式匹配问题。不仅如此，LAP_SoC 具有良好的用户接口。我们在 ARM CPU 系统上部署了 Linux 操作系统，并用 C 语言编写了针对 LAP 和 DMA 控制器的驱动程序。用户只需要具有编写 C 语言的能力，便可以驱动 LAP 核心进行高效的多模式匹配计算。最终，我们将该片上系统部署到了 Zedboard 开发板上。我们发现，在 Zedboard 的双核 Cortex-A9 处理器系统上添加一个 LAP 核心，可以在仅增加 5% 功耗的情况下提升 40 多倍面向多模式匹配任务的处理吞吐。

关键词：模式匹配；有限自动机；处理器；现场可编程逻辑门阵列

ABSTRACT

With the advent of the information age, we are generating large amounts of data every day. Among them, there are a large amount of unstructured text data, such as network data packets transmitted on the Internet, daily emails sent and received, biological gene sequences, and various log files recorded by computer systems. On the one hand, we may need to monitor the text data in some systems for security purposes. For example, we can perform deep inspections on the received TCP data packets to protect the computer system from intrusion and damage. On the other hand, we can mine valuable information from these text data. For example, by analyzing the log files of a distributed system, we can diagnose recent errors in the system. In these applications, we need to simultaneously search and match many specified patterns in a large-scale character stream, which is called multi-pattern matching. However, CPU and GPU are not suitable for high-performance multi-pattern matching computations. CPU and GPU will generate a large number of unpredictable conditional branches and memory accesses when performing multi-pattern matching computations, which will frequently cause wrong branch predictions and Cache Misses, thereby greatly reducing processing speed and energy efficiency. In order to achieve more efficient multi-pattern matching computations, we designed and implemented a lightweight automata processor LAP based on a domain-specific architecture. This processor can use few hardware resources to achieve high-throughput multi-pattern matching computations.

The first work of this thesis is to provide an efficient automata processor design solution supporting fast ADFA execution. When performing multi-pattern matching computations based on an automata processor, we need to compile the set of regular expressions into a corresponding finite automata model, and then generate a binary program that can be recognized by the dedicated processor. Given a set of regular expressions to be matched, if we can generate a binary program with a smaller memory footprint, our automata processor will be able to search for more patterns simultaneously. In order to achieve this goal, we enabled the support for ADFA model (an enhanced DFA model) in LAP. Incremental storage is enabled in ADFA model, which results in effective compression of the automata model. Therefore, the binary program generated based on the ADFA model is also very small. However, the processing speed of existing hardware solutions is not satisfactory when running ADFA models. To improve this, we propose two new hardware instructions dedicated to the ADFA model. In order to support these

two instructions, we proposed a new hardware architecture for automata processors, and at the same time modified the existing compilation algorithms. Experiments show that compared to another lightweight automata processor UAP, LAP has improved its processing speed by 32% to 91% when they both running ADFA models. What's more, we designed and implemented a 4-stage stall-free pipeline that can efficiently support the LAP instruction set. We implemented the LAP processor core using Verilog language, and deployed it to Xilinx Artix-7 FPGA at 263 MHz. Experiments also show that due to the utilization of ADFA model and the efficient compilation algorithm, the storage efficiency of LAP's binary program is 8× higher than that of IBM's RegX accelerator and Micron's AP.

The second work of this thesis is to provide a complete design of a heterogeneous computing system (LAP_SoC) based on the LAP core and ARM CPUs. The hardware system mainly includes three parts: a general-purpose computing system; a dedicated computing system for multi-pattern matching tasks composed of LAP cores; and DMA hardware controllers responsible for mass data transmission. The system on chip can actually run on the FPGA board and be used to solve the actual multi-pattern matching problem in real life. Not only that, LAP_SoC has a good user interface. We deployed the Linux operating system on the ARM CPU system, and wrote drivers for LAP and DMA controllers in C language. Users only need to have the ability to write C language to drive the LAP core to perform efficient multi-pattern matching computations. Finally, we deployed the system-on-chip to Zedboard evaluation board. We found that adding a LAP core to Zedboard's dual-core Cortex-A9 processor system can increase the processing throughput for multi-pattern matching tasks by more than 40× while only increasing power consumption by 5%.

Key Words: Pattern Matching; Finite Automata; Processor; Field Programmable Gate Array

目 录

第 1 章 绪论	1
1.1 课题背景及意义	1
1.2 相关工作	3
1.2.1 基于 DFA 模型的模式匹配引擎	3
1.2.2 基于 NFA 模型的自动机处理器	4
1.2.3 专用流水线架构实现并行的模式匹配	6
1.2.4 广泛支持各类模型的统一自动机处理器	7
1.3 本文主要工作	8
1.4 论文组织安排	9
第 2 章 基础理论与部署平台	10
2.1 自动机理论与各类自动机模型	10
2.1.1 DFA 与 NFA 模型	10
2.1.2 正则表达式到 NFA 的转化	11
2.1.3 NFA 模型到 DFA 模型的转化	12
2.1.4 Delayed Input DFA 模型	15
2.1.5 A DFA 模型	19
2.1.6 其他的衍生自动机模型	22
2.2 ZYNQ 的异构计算平台	23
2.2.1 CPU+FPGA 的架构	24
2.2.2 AXI 总线系统	24
2.2.3 AXI4 直接内存访问	26
第 3 章 LAP: 高效的轻量级自动机处理器核心	27
3.1 研究动机	27
3.2 LAP 的指令集设计	29
3.2.1 LAP 的硬件执行模型	29
3.2.2 LAP 指令集	30
3.3 LAP 指令集优化	35
3.3.1 A DFA 对自动机模型的高效压缩	35
3.3.2 A DFA 降低模式匹配速度的根本原因	36
3.3.3 A DFA 降低模式匹配速度的优化思路	37
3.3.4 并行化的 A DFA 执行算法与新的机器指令	39

3.4 LAP 的微体系结构	41
3.4.1 四级流水线架构	41
3.4.2 细粒度多线程	43
3.5 编译算法设计与实现	44
3.5.1 将正则表达式集合转化为等价自动机模型	45
3.5.2 自动机模型的 Split EffCLiP 二进制表示	45
3.6 实验与评估	47
3.6.1 LAP 处理速度总览	48
3.6.2 对 LAP 处理速度的详细评估	49
3.6.3 LAP 存储效率的分析和讨论	50
第 4 章 LAP_SoC: 基于 LAP 核心的异构片上系统	52
4.1 构建 LAP_SoC 的动机	52
4.2 LAP 的标准化数据接口	53
4.2.1 基于 AXI-Stream 的输入接口	53
4.2.2 基于 AXI-4 的程序写入接口	54
4.2.3 基于 AXI-Stream 的输出接口	54
4.2.4 基于 AXI-Lite 的配置接口	55
4.3 LAP_SoC 硬件系统	55
4.4 LAP_SoC 软件系统	57
4.4.1 操作系统配置与部署	58
4.4.2 驱动程序设计与实现	58
4.5 LAP_SoC 部署与评估	60
4.5.1 实现平台与部署结果	60
4.5.2 模式匹配性能的评估方法	62
4.5.3 实验结果与性能分析	64
第 5 章 总结与展望	67
5.1 本文总结	67
5.2 未来展望	68
参考文献	69
致谢	74
在读期间发表的学术论文与取得的研究成果	76

插图清单

图 1.1	本文相关工作的总结	3
图 1.2	RegX 硬件加速器处理流程	4
图 1.3	基于 RegX 的片上系统结构	4
图 1.4	Automata Processor 的空间体系结构	5
图 1.5	并行化自动机的执行过程：枚举所有可能的路径	7
图 1.6	HARE 的硬件流水线结构	7
图 1.7	UAP 的微体系结构	8
图 1.8	基于 UAP 的多核系统	8
图 2.1	正则表达式转化为 NFA 的三条规则	12
图 2.2	示例：正则表达式转化为 NFA	12
图 2.3	需要被转化为 DFA 的 NFA 模型	14
图 2.4	子集构建法生成的 DFA 模型	15
图 2.5	DFA 和 D^2FA 模型，接受三个模式： $a+, b+c$ 和 $c*d+$	16
图 2.6	接受 $a+, b+c$ 和 $c*d+$ 的 DFA 的空间缩减图	18
图 2.7	空间缩减图可生成的不同的最大权重生成树	18
图 2.8	ZYNQ 体系结构总览	25
图 2.9	AXI DMA 模块框图	26
图 3.1	LAP 的硬件执行模型	29
图 3.2	LAP 的指令格式	30
图 3.3	DFA 与等价的 ADFA 模型的状态转换图	36
图 3.4	ADFA 模型对状态转换表的压缩	36
图 3.5	ADFA 模型的典型拓扑结构	38
图 3.6	第一类状态回退的优化方案	40
图 3.7	第二类状态回退的优化方案	41
图 3.8	LAP 的四级流水线架构	43
图 3.9	细粒度多线程技术在 LAP 中的应用	44
图 3.10	ADFA 模型的层次化结构	45
图 3.11	ADFA 模型的 Split EffCLiP 表示	46
图 3.12	LAP 在 Artix-7 FPGA 上的 Schematic	47
图 3.13	LAP 在不同工作负载下的总体处理速度	49
图 3.14	详细的 LAP 性能分析	50

图 3.15	LAP 的存储效率对比和分析	51
图 4.1	基于 LAP 的 Xilinx IP 核	53
图 4.2	LAP 的流式输入接口设计	54
图 4.3	LAP 计算结果的返回包格式	55
图 4.4	LAP 控制状态寄存器格式	55
图 4.5	LAP 片上系统整体架构设计	56
图 4.6	ZYNQ 系统的 Block Design	57
图 4.7	ZYNQ 系统的地址分配	57
图 4.8	ZYNQ 系统 DDR 空间分配	58
图 4.9	LAP_SoC 的部署平台: Zedboard 开发板	60
图 4.10	LAP 片上系统的层次结构	61
图 4.11	LAP 片上系统功耗分布	62
图 4.12	桌面 Intel CPU 基于 grep 命令进行多模式匹配	63
图 4.13	嵌入式 ARM CPU 基于 grep 命令进行多模式匹配	63
图 4.14	使用 LAP 进行多模式匹配	64
图 4.15	LAP 与 Intel 处理器进行模式匹配计算 (不同输入规模下) 的时间统计	65
图 4.16	三种硬件平台下进行模式匹配计算 (不同输入规模下) 的加速比	66

表格清单

表 2.1	有 n 个状态的 NFA 和等价 DFA 的存储复杂度和计算复杂度对比	11
表 2.2	子集构建法生成的 DFA 模型状态转换表	15
表 3.1	LAP 支持的 7 条指令和它们的功能描述	31
表 3.2	LAP 在不同工作负载下的总体处理速度原始数据	49
表 3.3	详细的 LAP 性能分析原始数据	50
表 3.4	LAP 的存储效率原始数据	50
表 4.1	驱动 LAP 的 C 函数	59
表 4.2	LAP 片上系统总体资源占用	61
表 4.3	评估性能所使用的三种硬件平台	62
表 4.4	LAP 在不同规模下运行 <code>grep</code> 命令耗时统计	65
表 4.5	Intel 处理器在不同规模下运行 <code>grep</code> 命令耗时统计	65
表 4.6	Cortex-A9 处理器在不同规模下运行 <code>grep</code> 命令耗时统计	66
表 4.7	三种硬件平台下进行模式匹配计算的加速比原始数据	66

第1章 绪 论

1.1 课题背景及意义

随着信息时代的到来,我们每天都在产生大量的数据。自从2011年开始,“big data”这个词开始广为流传^[1]。Laney于2001年提出3Vs (Volume, Variety, 和 Velocity) 是大数据管理三个维度的挑战^[2]。随后,该定义作为通用框架被用来描述 big data。Volume 指的是数据规模之大。大数据一般以 TB 和 PB 作为单位进行衡量,而 1TB 可以存储相当于 1500 张 CD 或 220 张 DVD。Velocity 指的是数据生成的速率和它应当被分析和处理的速度。传感器和数字设备(比如智能手机)的广泛使用使得数据以前所未有的速度在产生,被实时处理的需求也带来了极大地挑战。Variety 指的是数据集中数据结构的异构性。结构化数据(关系型数据库或者平展的表格中的数据)仅仅占据现有数据量的 5%^[3]。事实上,我们每天都在产生大量非结构化的文本数据,比如各类文章、聊天记录、计算机日志、网络流量和基因序列。处理和分析这些文本数据可以帮助我们挖掘一些信息或者出于安全的目的监控某个系统。本文主要研究如何使用专用的硬件处理器进行高效的文本数据处理。

模式匹配被广泛的应用于多种应用领域中,例如网络安全^[7], 计算生物学^[8], 文本数据挖掘^{[4][5][6]} 和计算物理学^[9] 等应用。这些应用有一个共同的特征,他们都需要在大规模的输入字符序列上同时查询和匹配多个指定的子串/模式。例如,在进行度网络包检测/网络入侵检测时,我们可以通过指定一组恶意代码的指纹集合,并检测接受的网络包的包头和负载中是否包含这样的恶意代码。基于这样的目的,很多基于指纹进行深度网络包检测的软件工具应运而生,包括 Snort、Bro 和 ClamAV。然而,在进行深度网络包检测时,针对网络包的扫描速度需要匹配网络传输速度,也就是要达到 Gbps 的量级。不仅如此,我们需要同时匹配由几百到上千个正则表达式组成的庞大模式串集合。这些正则表达式可以是简单的字符串匹配,也可能包括正则表达式支持的通配符(.)和 Kleene 星号(*)等运算符。除此以外,生物的基因序列也是一种大数据。在生物学的研究中,我们需要从海量的生物基因序列中查找特定的基因串。例如,基于 CRISPR/Cas9 免疫系统进行基因编辑是当下非常流行的一种技术。然而在使用该技术时,我们需要使用 gRNA 来引导 Cas9 核酸酶到达预定的靶点(具有特定的 DNA 序列的位置)进行基因编辑。值得注意的是,gRNA 可以与众多和目标 DNA 序列存在少量差异的 DNA 序列结合。因此,同一款 gRNA 可以结合的潜在目标靶点非常多。这导致从庞大的 DNA 链中找出所有的潜在靶点成为了很复杂的计算问题。总的来说,多模式匹配被广泛的应用于文本数据的处理,而且有着很高的性能需求。

有限自动机 (Finite Automata^[13]) 是一个重要的数学概念, 也作为一种计算模型被广泛地用于模式匹配计算。更具体来说, 有限自动机可以自然地支持多模式匹配: 同时从主串中查找众多不同的模式。我们可以将要匹配的所有模式串转化为单个有着特定拓扑结构的有限自动机 (该自动机的状态数目和状态间的连接关系由待匹配的所有模式共同决定)。随后, 我们通过运行单个自动机模型, 就可以并行地在大规模输入上同时查找众多的模式。我们熟知的非确定性有限状态机 (Non-deterministic Finite Automata, NFA) 和确定性有限状态机 (Deterministic Finite Automata, DFA) 都属于有限自动机这个范畴。除此以外, 还有很多新颖的自动机模型被提出和应用于多个应用场景, 包括 AC 自动机和 ADFA 模型^[32]。其中, Aho-Corasick 算法是一款经典的多模式匹配算法。该算法主要靠构造一个 AC 自动机来实现。AC 自动机本质上是一个拓展的确定性有限自动机 (DFA), 它支持“失配指针”。通过在一个 trie 树中添加“失配指针”, 我们便可以构造出对应的 AC 自动机。这些额外的“失配指针”允许在查找字符串失败时进行回退 (例如设 Trie 树的单词 cat 匹配失败, 但是在 Trie 树中存在另一个单词 cart, 失配指针就会指向前缀 ca), 转向某前缀的其他分支, 免于重复匹配前缀, 提高算法效率。本文关注的 ADFA 模型吸收了 AC 自动机的思想, 实现了更通用的多模式匹配计算。回顾模式匹配算法的历史, 我们可以发现 KMP 算法^[10] 和 BM 算法^[11] 是面向单模式串匹配 (同时查找和匹配单个模式) 的经典算法。然而在上文提出的应用中, 我们则需要同时查找和匹配多个不同模式。对于这样的多模式匹配任务, 我们当然也可以将它分解为多个单模式匹配任务并串行的执行这些子任务。但是, 这将会非常的低效。由此我们可以看出, 基于有限自动机的多模式匹配算法在本文的目标应用领域中有着不可替代的地位。

通用处理器不适用于高性能的多模式匹配计算。首先, 如果我们运行在通用处理器 (CPUs 或 GPUs) 上模拟自动机模型的执行过程, 我们需要重复地根据当前的输入切换当前状态。这会引发大量的不可预测的分支跳转或大量的不规则的内存访问。不可预测的分支预测会导致通用处理器中的分支预测机制失效, 引发大量的分支预测失败。紧接着, 处理器流水线需要被冲刷和重新执行, 从而极大地降低处理器的性能和效率。类似的, 不规则的内存访问会引发频繁的缓存缺失 (Cache Miss)。这会使得处理器流水线被阻塞, 从而降低处理器性能。在过去的十几年中, 算法层面的优化被不断提出。这些优化后的模式匹配算法也在 CPU 和 GPU 上取得了更高的计算吞吐和能效。然而, 随着待处理的数据量越来越大, 我们对于多模式匹配的计算性能也有了越来越高的需求。但受限于通用处理器自身体系结构的限制, 基于传统处理器的多模式匹配计算在性能和能效方面已经无法获得进一步的显著提升。因此近些年, 学术界和工业界投入了越来越多的精力去设计和实现专门用于多模式匹配的硬件加速器/自动机处理器。这些

专用硬件相比于传统处理器在性能和能效方面有数量级上的提升。

1.2 相关工作

本小节将总结近十年内学术界与工业界提出的面向多模式匹配任务的专用硬件。根据它们所使用的基础架构，这些硬件可以被分类成4类（如图1.1所示）。

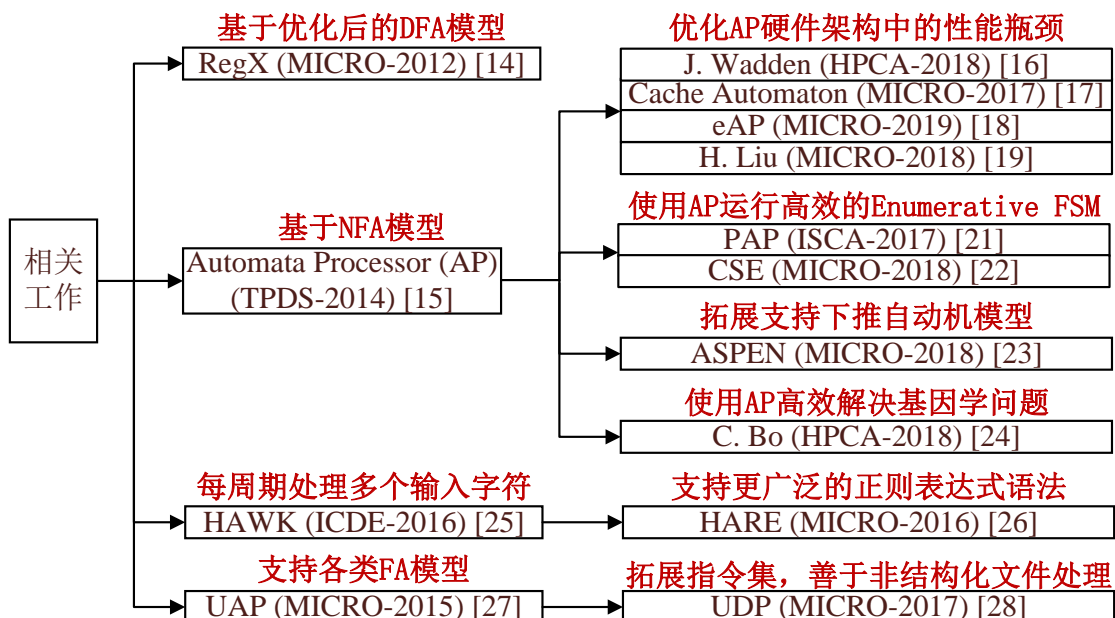


图 1.1 本文相关工作的总结

1.2.1 基于 DFA 模型的模式匹配引擎

RegX 加速器^[14]是由 IBM 公司设计实现的可编程正则表达式匹配引擎，用于进行网络入侵检测。该成果不仅仅发表于体系结构顶级会议上，同时也实际部署在了 IBM 公司的 PowerEN 处理器中。该加速器可以同时匹配上千条模式并实现足以匹配以太网链路带宽的扫描速度（理论扫描速度为 73.6 Gbit/s）。对于典型的网络入侵检测任务，RegX 的实际处理速度达到了 15 到 40 Gbit/s。RegX 的执行流程如图1.2所示，分四步处理每一个输入字符。首先，根据当前状态和当前输入，计算出地址；然后，根据第一步计算出的地址从存储器中取出相应的规则；第三步，从取出的规则中选择需要的部分；第四步，根据取出的规则更新当前状态，实现状态机的一次状态切换。值得一提的是，RegX 需要使用外部 DRAM 来存储其自动机模型（如图1.3），而且内部使用了复杂的 Cache 缓存部分自动机模型。总的来说，该硬件加速器基于拓展的 DFA 模型（名为 B-FSM 模型），实现了较为紧致的存储和非常高速的执行。

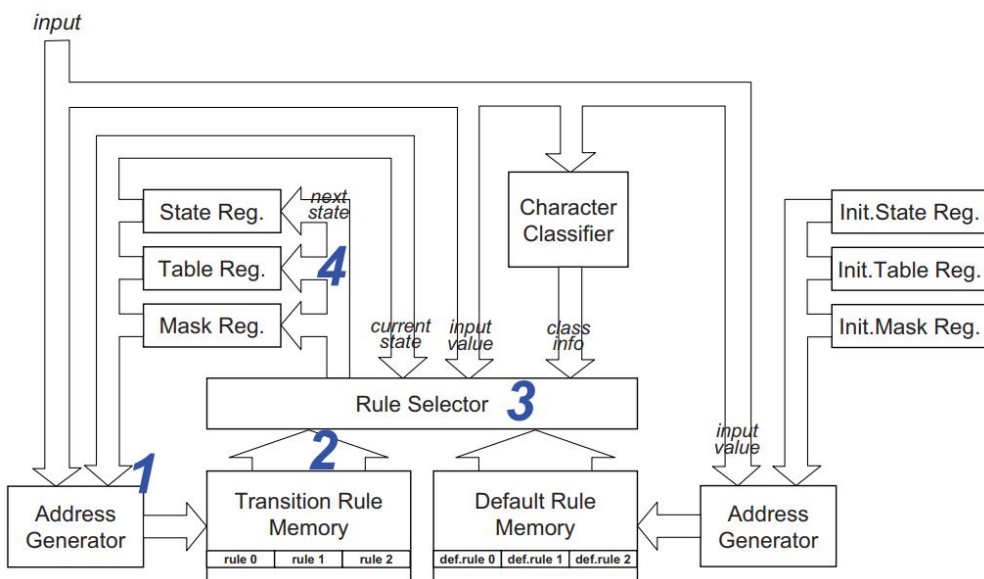


图 1.2 RegX 硬件加速器处理流程

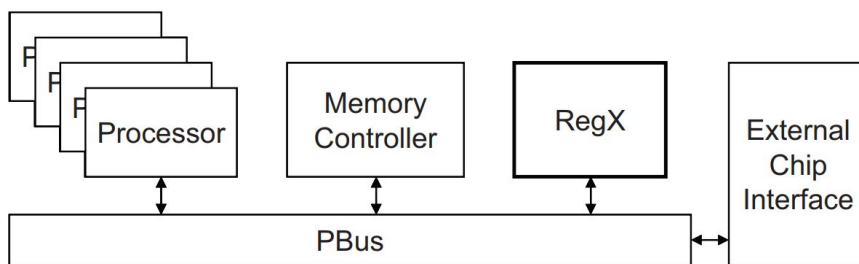


图 1.3 基于 RegX 的片上系统结构

1.2.2 基于 NFA 模型的自动机处理器

Automata Processor^[15] (本文简称 AP) 是由 Micron 公司设计并生产的基于 NFA 模型的自动机处理器。该处理器可以直接将非确定性有限状态机直接映射到其空间体系结构上, 从而实现大规模的、非常复杂的正则表达式匹配。SDRAM 存储器阵列是当代计算机主存储器的核心结构, 而 AP 就是建立在这样的空间架构下。如图1.4所示, AP 由众多的状态转换元件 (STE) 构成。这些元件可以并发的运行从而提供非常高的并行度。通过这个方法, AP 可以提供 NFA 模型所需要的巨大的算力。另一方面, AP 内部使用独热码的方式编码状态机的每个状态。这使得 AP 需要使用 256 个比特去编码一个状态。然而, 状态机的状态数会随着待匹配的正则表达式的数量的增加而增加。当正则表达式总数达到上千条时, 状态机状态数也会变得非常巨大。因此, AP 需要占用较大的存储空间。为了解决这个问题, AP 直接使用 SDRAM 阵列存储 NFA 模型, 而不是使用 SRAM。通过这个方法, AP 可以存储非常多且非常复杂的正则表达式。相应的, SDRAM 存储阵列也占据了非常多的片上空间和功耗。除了存储阵列, AP 内部还包含一个非常复杂的硬件互联。该硬件互联在图1.4中被称为 Routing Matrix Structure。这

是一种可配置的互联结构，根据要匹配的正则表达式使用不同的方式将 STE 两两连接起来。总的来说，Automata Processor 是一款非常有名且非常高性能的自动机处理器。随后，学术界问世了大量基于 AP 架构的自动机处理器。

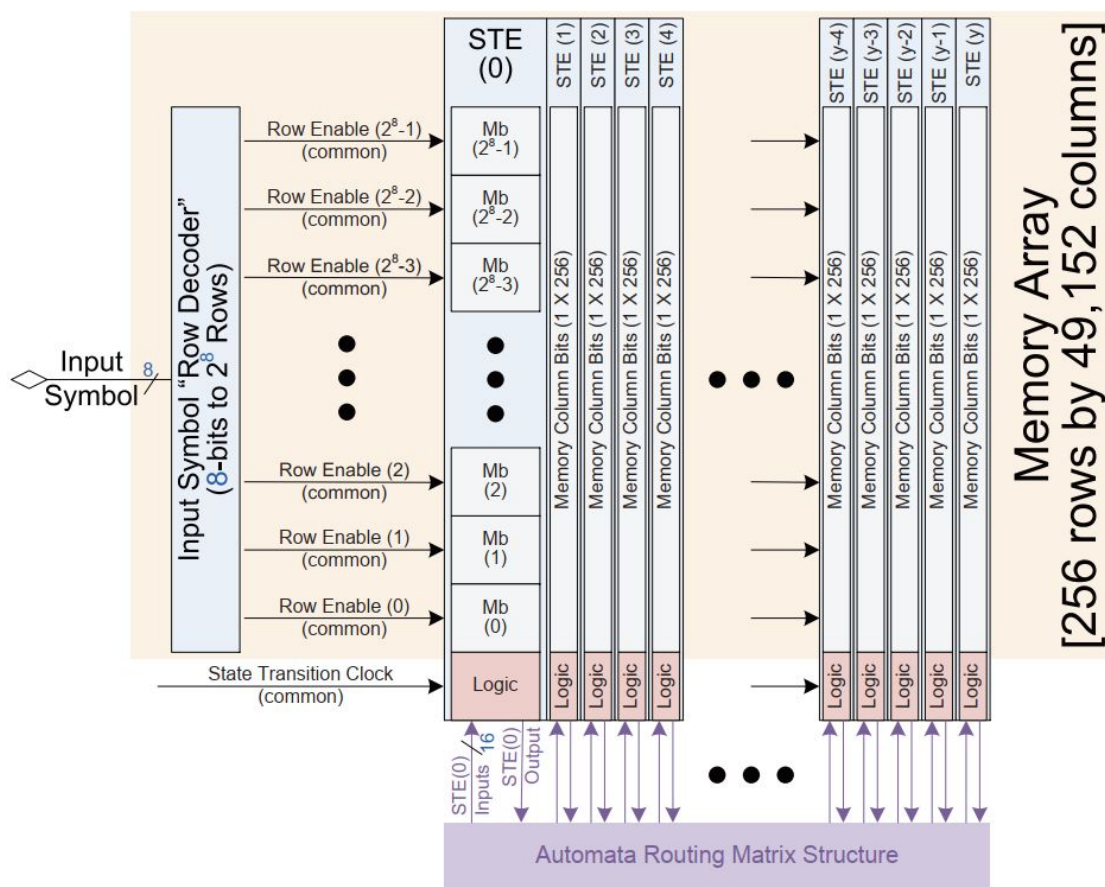


图 1.4 Automata Processor 的空间体系结构

有一部分研究者试图去分析和优化 AP 存在的性能瓶颈，并在顶级会议上发表了若干篇代表性论文。J. Wadden^[16] 通过分析和实验证实，AP 在向宿主处理器汇报匹配结果时存在瓶颈。J. Wadden 等人发现，AP 在运行常见基准测试程序时，会非常频繁的汇报匹配结果；然而，每次汇报的结果数目却很少。因此，他们设计了新的输出压缩策略和新的体系结构专门负责高效的结果汇报。最终，他们相比 AP 实现了 5.1 倍的加速效果。Cache Automaton^[17] 修改了 Last-level Cache 的结构，在 Cache 上实现了类似 AP 的架构。通过将实现介质从 SDRAM 更换为 SRAM，Cache Automaton 相比 AP 实现了显著的功耗降低和性能提升。eAP^[18] 优化了 AP 的路由体系结构。具体来说，eAP 优化了 AP 内部的可配置互联结构。通过这种方式，eAP 分别实现了相比 Cache Automaton 和 AP 5.1 倍和 207 倍的计算吞吐的提升。H. Liu^[19] 则给 AP 增加了体系结构上的支持，使得新的架构可以支持更大规模的自动机处理。他们发现在运行实际的模式匹配任务时，生成的自动机模型中有大量的状态是永远不会被激活的。然而，这样的状态却会被配

置在 AP 中，同时占据大量的资源。因此，H. Liu 等人预测了这些“无用的”状态，并不将它们配置到处理器中。相应的，他们提出了新的执行机制来处理错误的预测。在经过大量的基准测试后，他们的方案实现了平均 2.1 倍的性能提升。

Micron 公司的 AP 被众多研究者应用于并行化单个有限自动机的执行。有限自动机的执行有着与生俱来的串行依赖：当前状态 a 根据当前输入的状态会切换到下一个状态 b，而下一个状态 b 又会在接受另一个输入后切换到第三个状态 c。在这个过程中，状态 b 与状态 a 有关，而状态 c 又取决于状态 b。因此，这样一个计算过程难以被并行化。如图 1.5 所示，T. Mytkowicz^[20] 提出可以将状态转换过程切分。除了 segment 1 以外，其他过程都没有确定的开始状态。这是因为这些过程的开始状态是它们前面一个过程的结束状态。因此，我们需要计算 segment 2 可能存在的每一条状态转换路径。这意味着，segment 2 有多少可能的开始状态我们就需要计算多少条路径。最终，对于每一个 segment，我们将从预先计算的路径中挑选一条路径，并将这些路径拼接起来得到最终的路径。但是，枚举所有路径带来了额外的计算开销。T. Mytkowicz^[20] 等人使用了通用处理器支持的 SIMD 指令进行这些枚举路径的并行计算。PAP^[21] 首先提出 AP 可以用于并行化非确定性有限自动机的执行。AP 提供的并行度被利用以实现并行的路径枚举。同时，各种优化策略被提出用于降低枚举路径所需的计算量。进一步的，CSE^[22] 提出新的计算原语用于进行更高效的自动机可能路径的枚举。在之前的工作中，枚举路径的结果是计算出每条路径输入状态到输出状态得映射。在 CSE 中，则是计算一个输入状态集合到输出状态集合的映射。这样的计算原语更适合 AP 的计算架构，因此取得了更高的计算吞吐。总的来说，枚举的方法可以实现自动机执行过程的串行化，但是也带来了非常高的枚举开销。因此，该项技术主要面向有计算延迟约束的场景，而不适用于面向高吞吐的模式匹配任务。

AP 也被广泛的应用于解决工业界和学术界的实际问题。ASPEN^[23] 吸取了先前自动机处理器的设计思想，并增加了对栈进行操作的原语，从而实现了向下推自动机的支持。基于该自动机模型，ASPEN 可以高效执行对树形结构的递归嵌套的语言（例如 Cool, DOT,JSON 和 XML）的解析。C. Bo^[24] 等人使用 AP 搜索 CRISPR/Cas9 系统下的的潜在 gRNA 脱靶位点。发现这些潜在的位点对进行有效的基因编辑非常重要，但是这一个非常复杂的计算过程。C. Bo 等人提出的基于 AP 的方案，相比 CPU 和 GPU 分别实现了 900 倍和 120 倍的加速比。可以看到，自动机处理器有着日益广泛的应用场景。

1.2.3 专用流水线架构实现并行的模式匹配

HARE^[25] 是一款比较特殊的模式匹配处理器，它为文本和非结构化数据的处理提供了强大而灵活的工具。HARE 可以实现超高的模式匹配吞吐量。它的

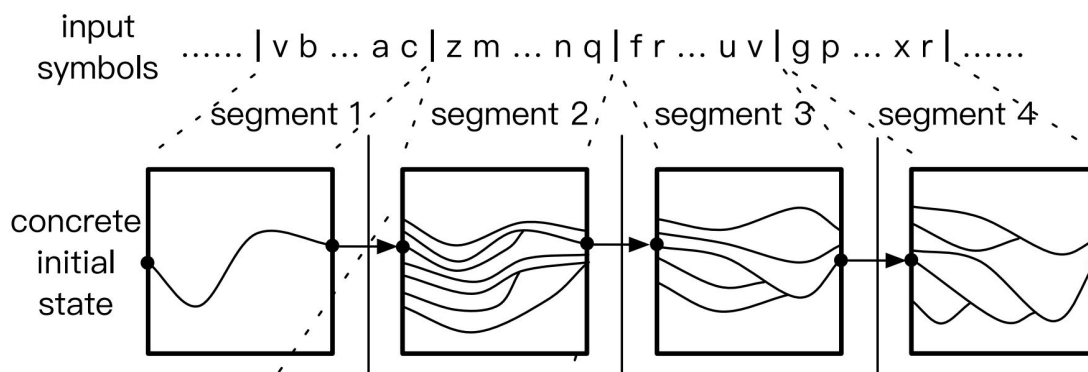


图 1.5 并行化自动机的执行过程：枚举所有可能的路径

目标是匹配当下计算机系统的主存带宽（非常大），而不是匹配计算机系统的输入输出带宽。HARE 最特殊的地方在于，它可以在一个时钟周期内同时处理 32 个输入字符，而之前我们提及的各种自动机专用硬件最快只能一个周期处理一个字符。HARE 和它的前身 HAWK^[26] 一样，拥有不会停顿的流水线结构，使用 bit-split 自动机模型实现模型压缩，还可以同时处理一个窗口内的全部输入字符。HARE 的流水线结构如图1.6所示。通过增加 Counter-based Reduction Unit, HARE 提供了对正则表达式中 Kleene Star (*) 的支持。总的来说，HAWK 和 HARE 是一类高吞吐的模式匹配加速器;HARE 相比 HAWK 可以支持更加广泛的正则表达式语法，因此拥有更广泛的应用场景。

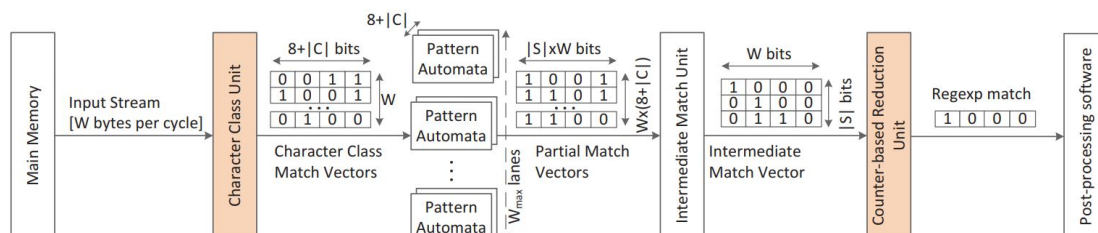


图 1.6 HARE 的硬件流水线结构

1.2.4 广泛支持各类模型的统一自动机处理器

除了经典的 DFA 和 NFA 之外，研究人员提出了越来越多新的自动机模型。这些新颖的自动机模型往往对于某一类的计算负载有着显著地裨益。但是，却没有哪一个自动机模型适用于所有的正则表达式集合。因此，UAP^[27] 提出了统一的自动机处理器，广泛地支持各类自动机模型（包括 DFA,NFA,ADFA,JFA 等）。

为了实现通用自动机模型所需要的各种底层机制，UAP 实现了对应的状态转换原语和对寄存器的操作原语。其中，每条原语对应于 UAP 的一条硬件指令。UAP 的微体系结构如图1.7所示。对于每种自动机模型，Combining Queue 负责存储所有被激活的自动机状态。UAP 每个周期从 Combining Queue 中取出一个激活状态，并从 Prefetcher 中取出当前要处理的字符。随后，UAP 计算出下一条

机器指令的地址。该条机器指令将用于更新当前自动机状态。同时，Action Unit 可以实现对 UAP 内部寄存器的修改，从而实现更复杂的自动机模型。UAP 系统中包含了一个 RISC 的通用处理器核心和 64 个 UAP 核心（如图1.8所示）。每个 UAP 核心平均每周期可以处理一条状态转换原语和寄存器操作。同时，每个 UAP 核心拥有一个片上存储器 Bank，用来存储各自要运行的自动机模型。

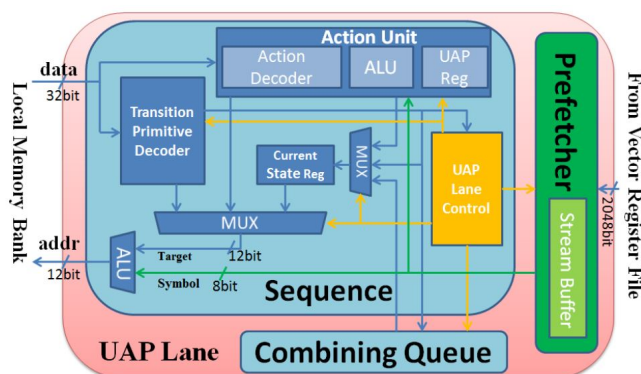


图 1.7 UAP 的微体系结构

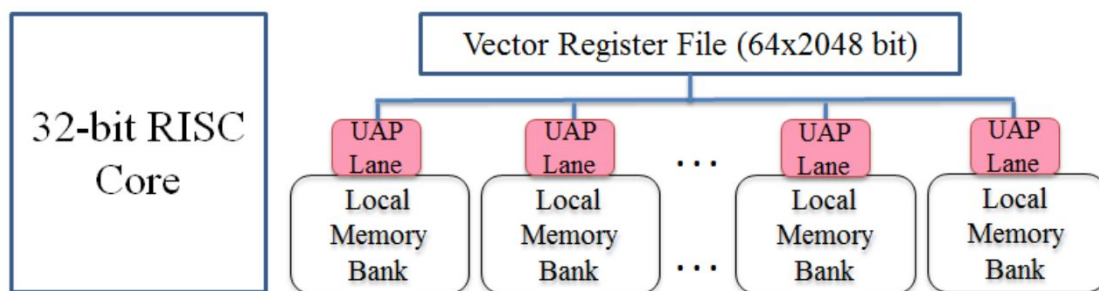


图 1.8 基于 UAP 的多核系统

UDP^[28] 建立在 UAP 的架构上，同时增加了对更多指令的支持。通过这种方法，UDP 可以高效的进行分支频繁的计算任务，例如 CSV 文件解析、霍夫曼编解码、模式匹配和 Snappy 压缩解压。

1.3 本文主要工作

本文致力于提出并实现一种轻量级、高性能的自动机处理器。该轻量级处理器可以通过 FPGA 或者 ASIC 平台实现，并与通用 CPU 形成多核异构计算系统，从而实现高吞吐高效的多模式匹配计算。本文的工作主要包含两个部分：

(1) LAP: 更快的面向多模式匹配任务的轻量级自动机处理器

- 通过利用新颖的 ADFA 模型和高效的模型存储算法，LAP 实现了非常高的存储效率，LAP 可以将整个自动机模型压缩存储到有限大小的 SRAM 中，从而不依赖于外部 DRAM 独立的运行。另外，借助于软硬件协同设计的方

法，我们并行了 ADFA 模型的执行算法，从而极大的提高了 LAP 的执行速度。在本文中，我们通过 C++ 和 Python 代码实现了相应的编译算法。

- 我们设计了一个 4 阶段流水线，用于在 FPGA 和 ASIC 上部署高主频的 LAP 核心。为了实现一个无停顿的流水线，我们使用细粒度多线程技术化解了流水线内部的相关。
- 我们使用 Verilog 语言实现了 LAP 核心，并将其部署到 Xilinx 公司的 Artix-7 FPGA 芯片上。最终，LAP 可以运行在 263 MHz 的主频下。实验结果表明 LAP 相比于 IBM 公司的 RegX 加速器和 Micron 公司的 AP 处理器的存储效率高了 8 倍。同时，相比其他的轻量级自动机处理器，我们的处理器实现了 32% 到 91% 的性能提升。

(2) LAP_SoC: 基于 LAP 核心的高能效异构计算片上系统实现

- 为了便于将 LAP 核心与 Xilinx FPGA 内的 ARM 处理器集成在一起，我们首先为 LAP 设计并实现了基于 AXI 标准数据接口。然后，我们将 LAP 封装成独立的 IP 核。之后，我们使用 Vivado 的 Block Design 功能，设计并实现了 ARM CPU 和单 LAP 核心的异构计算系统。
- 我们为 LAP 核心和 DMA 控制器编写了相应的 CPU 端驱动程序，驱动 LAP 核心进行模式匹配计算。在多种计算负载下，LAP_SoC 取得了相比于 Cortex-A9 处理器 40+ 倍和相比于 Intel Core-I5 处理器 3+ 倍的性能提升。

1.4 论文组织安排

本文的第一章已经介绍了自动机处理器的应用场景和相关工作。第二章将详细介绍与 LAP 的设计和实现密切相关的基础知识，包括自动机理论和 LAP_SoC 的部署平台（ZYNQ FPGA）。第三章作为本文的第一项核心工作，将主要讲述 LAP 高能效核心的设计与实现，从指令集设计，到指令集优化，再到流水线设计和编译器实现。通过运行标准的 benchmark 并与其他自动机加速器/处理器对比，我们细致地评估与分析了 LAP 的各项性能。第四章是本文的第二项核心工作，主要讲述了基于 LAP 核心和通用处理器的异构计算片上系统的设计、实现与评估。第五章将对本文的工作进行总结，并展望未来可开展的工作。

第 2 章 基础理论与部署平台

本章将首先介绍自动机的基础理论和相关学术研究，最后重点阐释 ADFA 模型（第 3 章中介绍的自动机处理器 LAP 所使用的计算模型）。同时，本章将介绍 ZYNQ 系列 FPGA 的基础架构和特征。通过了解 ZYNQ，读者将可以更好地理解本文是如何依托现有硬件框架构建的 LAP_SoC 片上系统（见第 4 章）。

2.1 自动机理论与各类自动机模型

有限自动机^[13] (finite automaton, FA), 又称有限状态机 (finite-state machine, FSM), 是表示有限个状态以及在这些状态间进行转移和动作等行为的数学计算模型。同时，有限自动机也是非常有效的可用于进行模式匹配的算法工具。我们可以把数以千计的正则表达式编译成单个自动机，然后将我们需要处理的字符流输入该自动机进行模式匹配计算。通过这样的方式，我们可以同时匹配大量的模式而不是每次只匹配一个模式。自动机处理器正是基于这些数学模型所设计的。因此，本节将介绍两类基本的自动机模型及其衍生模型。

2.1.1 DFA 与 NFA 模型

确定性有限状态机 (Deterministic Finite Automata, DFA) 和非确定性有限状态机 (Non-deterministic Finite Automata, NFA) 是最经典的两类自动机模型。

非确定性有限自动机 (NFA) 可表示为一个五元组 $(Q, \Sigma, \delta, q_0, F)$:

- 有限的状态集合 Q ;
- 有限的输入字符集合 Σ ;
- 状态转换函数 $\delta(q, \alpha)$;
- 自动机的初始状态 $q_0 (q_0 \in Q)$;
- 接受态状态集合 $F (F \subset Q)$ 。

其中，状态转换函数 $\delta(q, \alpha)$ 定义了当前状态 q 在接受输入字符 α 后会激活的一组状态集合。这意味着，单个初始状态可能会接受一个输入字符而激活多个目标状态。更进一步的，这些目标状态又可能在接受同一个输入后各自激活更多的状态。因此，NFA 模型中，同一时间可能有多个活跃状态。

确定性有限自动机 (DFA) 是 NFA 的特例，由类似五元组 $(Q, \Sigma, \Delta, q_0, F)$ 构成。

- 有限的状态集合 Q ;
- 有限的输入字符集合 Σ ;
- 状态转换函数 $\Delta(q, \alpha)$;

- 自动机的初始状态 q_0 ($q_0 \in Q$);
- 接受态状态集合 F ($F \subset Q$)。

不同之处在于，DFA 模型的状态转换函数 $\Delta(q, \alpha)$ 每次只会激活一个状态，而不会激活一组状态集合。DFA 起初只有一个活跃的初始状态，每次该状态只会激活另一个状态。因此，DFA 模型在执行过程中，只会同时有一个活跃状态。

每一个 NFA 模型都可以转化为等价的 DFA 模型。但是如表格 2.1 所示，DFA 和 NFA 模型在时间复杂度和空间复杂度之间各自做了极端的权衡，因此各有利弊。NFA 模型有着有限的体积，不过对于每个输入字符需要进行更多的操作；等价 DFA 模型的计算操作得到了简化，却需要占用更多的存储空间。

表 2.1 有 n 个状态的 NFA 和等价 DFA 的存储复杂度和计算复杂度对比

状态机类型	计算复杂度	存储复杂度
非确定性有限状态机 (NFA)	$O(n^2)$	$O(n)$
确定性有限状态机 (DFA)	$O(1)$	$O(\Sigma^n)$

2.1.2 正则表达式到 NFA 的转化

正则表达式是计算机科学的一个概念。它使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串。在很多文本编辑器里，正则表达式就通常被用来检索和替代匹配某个模式的文本。不仅如此，正则表达式被广泛的应用在各种模式匹配的应用中，比如深度网络包检测和非结构化数据挖掘。接下来我们将展示一个关键的过程：我们是如何将正则表达式集合转化为单个自动机模型的。此处，我们主要展示转化为 NFA 和 DFA 的过程。在后面的章节，我们会进一步展示正则表达式向其他衍生模型的转化。

为了讨论方便，自动机模型可以通过带标签的有向图来直观的进行表示。因此，此处将使用这一表示方法来形象的表示具体的 DFA 和 NFA 模型。具体来说，有向图的一个节点对应自动机中的一个状态。同时，每条边代表两个状态间的转换关系。每条边上的标签代表了转换触发所需要的条件，也就是当前输入字符应该满足的条件。通常来说，非接受态的状态使用单个圆圈表示，而接受态状态使用同心双圆圈表示。

正则表达式可以很自然地转化为 NFA 模型。图 2.1 展示了将正则表达式转化为对应的 NFA 的三条规则。

- **规则 1:** 如果待匹配的正则表达式子串是连续的字符，这表示我们想按顺序不间断的匹配这些字符。这意味着，我们可以将这些字符拆解为多次状态转换，并将这些过程顺序地连接起来。
- **规则 2:** 如果待匹配的正则表达式两个子串间是“或者”（符号表示为 $|$ ）的

关系，我们可以将这两部分拆解成拥有同一源头和同一目标的两个“并联的”转换过程。

- **规则 3:** 如果待匹配的正则表达式子串是包含在克莱尼（Kleene, 符号表示为 $*$ ）星号里的，这表示该子串可以出现 0 次或者无数次。这样的子串应该转化为如图 2.1 最下方所示的结构。（该子串两端引入了 ϵ 边。）

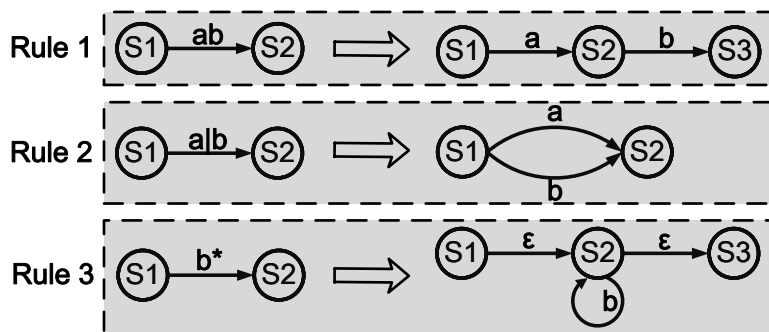


图 2.1 正则表达式转化为 NFA 的三条规则

如图 2.2 所示，我们可以利用上述三条规则，将正则表达式 $(a|b)(c|d)^*(e|f)$ 转化为对应的 NFA 模型。第一步，我们将该正则表达式拆解为串行连接的三个子串部分，并按照规则 1 进行分解。第二步，我们使用规则 2 和规则 3 将 “|” 和 “ $*$ ” 进行分解。最终，我们便得到了如图 2.2 所示的 NFA 模型。

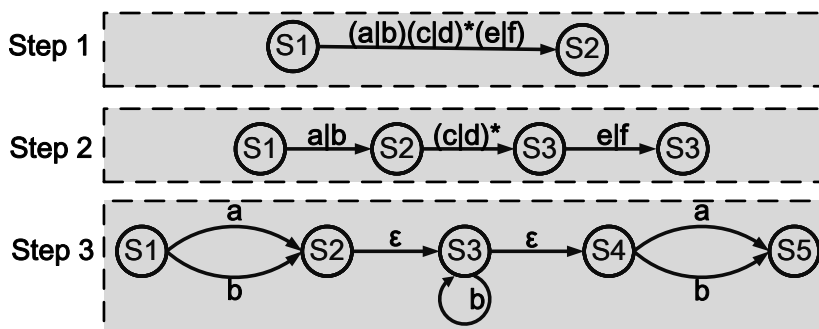


图 2.2 示例：正则表达式转化为 NFA

2.1.3 NFA 模型到 DFA 模型的转化

如果需要进一步得到 DFA 模型，我们可以首先将正则表达式转化为 NFA 模型，再将该 NFA 模型转化为对应的 DFA 模型。在将 NFA 转化为 DFA 的过程中，我们需要使用子集构造法。子集构造法^[29]可以将输入的 NFA 模型转化为对应的 DFA 模型。利用该算法生成的 DFA 模型有这样一个特点：输出 DFA 模型中每一个状态对应着输入 NFA 模型的一个状态集合。更具体来说，子集构造法构造的 DFA 在接受输入序列 $a_1, a_2, a_3, \dots, a_n$ 之后，能到达的状态应当对应于输入 NFA 模型的一个状态集合。该状态的集合包含着输入 NFA 模型沿着 $a_1, a_2, a_3, \dots, a_n$ 序列

将会激活的所有状态的集合。

在完整介绍子集构造算法前，我们先定义三个函数和两个变量。其中， s 表示 NFA 的单个状态，而 T 表示 NFA 的一个状态集合。

- $\epsilon - closure(s)$: 该函数定义从 NFA 模型的状态 s 开始，只通过 ϵ 边能够到达的 NFA 状态的集合（状态 s 的 ϵ 状态闭包）；
- $\epsilon - closure(T)$: 该函数定义从 T 中某个 NFA 状态 s 开始，只通过 ϵ 边能够到达的 NFA 状态的集合 ($\cup_{s \in T} \epsilon - closure(s)$)；
- $move(T, a)$: 该函数定义从 T 中的某个状态 s 出发，通过标号 a 的路径能够到达的 NFA 状态的集合。
- $Dstates$ 是要我们要构建的 DFA 模型的状态集合；类似的，每个 $Dstate$ 就是该 DFA 模型的一个状态。值得注意的是，每个 $Dstate$ 对应着输入的 NFA 模型的一个状态集合。
- $Dtran[T, a]$ 表示输出 DFA 中，状态 T (T 是一个 $Dstate$) 在接收到字符 a 后会跳转到的 $Dstate$ 。

子集构造算法的伪代码如算法2.1所示。算法2.1的第一行中， s_0 是 NFA 的开始状态。在读入第一个输入字符之前，输入 NFA 模型能够到达的状态集合是 $\epsilon - closure(s_0)$ 。因此， $Dstate$ 以该值初始化。随后，我们观察第二行开始的 `while` 循环。当不再有新的状态集合可以被构建出来，该 `while` 循环结束。每个 NFA 所对应的 DFA 模型，其状态一定是有限的。因此，该 `while` 循环一定会结束。在第三行代码中，对于需要处理（也就是未加标记）的 $Dstate$ ，我们要先给他加上标记，用来代表该状态已经被处理过了。随后，我们进入第四行的 `for` 循环。对于每一个可能的输入字符 a ，我们获取 $Dstate T$ 可以到达的状态闭包 ($\epsilon - closure(move(T, a))$)。对于每一个输入字符 a ，我们都可能会构造出一个新的状态集合 U 。第 6-8 行代码表明，如果 U 不在 $Dstates$ 中（也就说这是一个新的状态集合），我们需要把 U 加入到 $Dstates$ 中，用于被以后的 `while` 循环处理。如果 U 在 $Dstate$ 中，说明这个状态集合我们已经探索和处理过了，无需重复构造。第 9 行代码意味着，我们将在 DFA 模型的状态转换表中添加一个新的项：DFA 状态 T 在输入字符为 a 时跳转到 DFA 的另一个状态 U 。最终，我们获得了要构建的 DFA 模型的状态集合 $Dstates$ 和其状态转换表 $Dtran$ 。

图2.3中展示了一个示例 NFA 模型，我们将使用子集构建算法构造其对应的 DFA 模型。其构造过程如下：

- 输入 NFA 模型的初始状态为 1，因此我们将 $Dstates$ 初始化为 $\epsilon - closure(1)$ ，也就是状态 1 的 ϵ 状态闭包 $\{1,2\}$ 。
- 随后进入 `while` 循环，我们处理 $Dstates$ 中唯一一个未被标记的状态 $T=\{1,2\}$ ，并给这个状态加上标记。

算法 2.1 子集构造法：NFA 模型转化成 DFA 模型**Data:** 输入的 NFA 模型**Result:** 要构造的 DFA 模型的状态集合 $Dstates$ 和其状态转换表 $Dtran$

```

1 初始化时,  $\epsilon - closure(s_0)$  是  $Dstate$  中的唯一状态, 而且该状态未被标记;
2 while 在  $Dstates$  中存在一个未标记的状态  $T$  do
3     给  $T$  加上标记;
4     for 每个输入字符  $a$  do
5          $U = \epsilon - closure(move(T, a));$ 
6         if  $U$  不在  $Dstate$  中 then
7             将  $U$  加入到  $Dstates$  中, 且不加标记;
8         end
9          $Dtran[T,a] = U;$ 
10    end
11 end

```

- 进入 for 循环, 对于输入字符 a , 我们得到的新的 ϵ 状态闭包 $U = \epsilon - closure(move(T, a))$, 也就是 $\{2,4,5,6,7\}$ 。新构建的状态 U 不属于 $Dstates$ 。因此, 我们将 $U = \{2,4,5,6,7\}$ 不加标记的加入 $Dstates$ 中, 同时在 DFA 状态转换表中增加一项 $Dtran[\{1,2\},a] = \{2,4,5,6,7\}$ 。
- 随后, 进入 for 循环的第二个循环, 并处理输入字符 b 。我们得到一个新的 ϵ 状态闭包, $U = \epsilon - closure(move(T, b))$, 也就是 $\{3,8\}$; 状态 U 不属于 $Dstates$ 。因此, 我们将 U 不加标记的加入 $Dstates$ 中, 同时在 DFA 的状态转换表中增加一项 $Dtran[\{1,2\},b] = \{3,8\}$ 。
- 回到外层 while 循环, 此时 $Dstate$ 中有两个未标记状态 $\{2,4,5,6,7\}$ 和 $\{3,8\}$ 。
- 后续循环同理可得.....

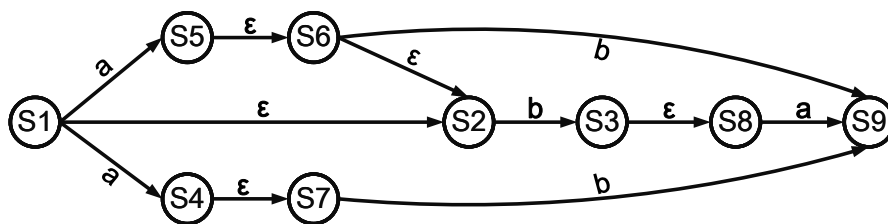


图 2.3 需要被转化为 DFA 的 NFA 模型

最终, 我们获得了要构建的 DFA 模型的状态集合和其状态转换表 (如表格 2.2 所示)。状态转换表在后面的章节还会用到, 我们在此处稍加解释。如表格 2.2 所示, 表格第一列列举了 DFA 模型中全部的状态, 而第二、第三列则分别表示每个 DFA 状态在接受不同输入字符时会转换到的目标状态。如果输入字符集合 Σ 不是 $\{a,b\}$, 那么表格的列数和列标题将做相应的调整。值得注意的是, NFA 模型也可以用状态转换表表示。在 NFA 模型的状态转换表中, 每个状态的目标

状态不再是一个状态，而可能是一个状态集合。

表 2.2 子集构建法生成的 DFA 模型状态转换表

DFA 状态 \ 输入字符	a	b
{1,2}	{2,4,5,6,7}	{3,8}
{2,4,5,6,7}	{}	{3,8,9}
{3,8}	{9}	{}
{3,8,9}	{9}	{}
{9}	{}	{}

表格2.2对应的确定性有限状态机模型如图2.4所示。在新生成的 DFA 模型中，我们对每个状态进行了重新的编号：NFA 中的状态集合 {1,2} 对应 DFA 中的状态 1；NFA 中的状态集合 {2,4,5,6,7} 对应 DFA 中的状态 2；NFA 中的状态集合 {3,8} 对应 DFA 中的状态 3；NFA 中的状态集合 {3,8,9} 对应 DFA 中的状态 4；NFA 中的状态集合 {9} 对应 DFA 中的状态 5。

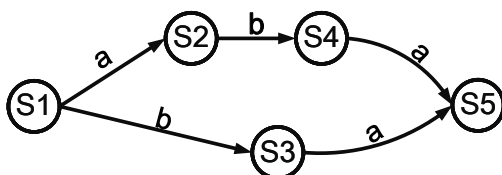


图 2.4 子集构建法生成的 DFA 模型

2.1.4 Delayed Input DFA 模型

1. D^2FA 模型简介

Delayed Input DFA (D^2FA)^[30] 是一种衍生 DFA 模型。 D^2FA 模型同样可以用 DFA 的五元组 $(Q, \Sigma, \Delta, q_0, F)$ 来描述。不同之处在于， D^2FA 的状态转移函数 Δ 在接受一个输入字符后，所需要进行的操作更为复杂。DFA 模型在进行状态跳转时，只需要查询当前状态的跳转表，并根据输入字符从表中选择一项，即可得知跳转的目标状态 (Next State)。 D^2FA 模型对每个状态的状态转移表进行了“增量式”的存储。因此， D^2FA 模型的状态转移函数需要连续对一到多个状态的转移表进行查询，才可以确定跳转的目标状态。在进行连续访存的过程中，当前输入字符被 delay (延迟处理) 了；直到该过程结束，当前输入字符才被真正地被 D^2FA 模型消耗掉。因此，我们称之为 Delayed Input DFA。在存储同样的正则表达式时， D^2FA 模型相比于传统的 DFA 模型可以减少 95% 的存储开销。

D^2FA 模型的初衷是为了实现存储更紧致的 DFA 模型，也就是说用更少的存储空间存储相同功能的 DFA。对于任意的正则表达式集合，都存在一个拥有最少状态的 DFA 模型与之对应。然而，传统的 DFA 模型需要非常大的存储空间，

用于存储其状态间的跳转关系。最坏情况下，用来表示这个 DFA 模型所需要的存储器容量为：DFA 状态数和每个状态的出边上限数的乘积。如果输入字符使用 ASCII 码编码，那么每个状态最多有 256 条出边；同时，用于网络入侵检测的正则表达式集合通常包含上百条模式。对应的，该 DFA 模型将包含上千个状态，并需要几百兆的存储空间。更不利的是，每个状态的跳转表自身很难用一般的压缩技术进行压缩。这是因为，每个状态在不同输入下都有着不一样的目标状态。因此，在每个状态内部，这些边无法被有效压缩。

D^2FA 模型的状态转移函数比 DFA 模型的转移函数更复杂：其状态转移函数支持 D^2FA 模型中引入的新类型的边——“默认边” (Default Edge)。虽然对于每个 DFA 状态，它的 256 条出边我们很难去压缩。但是，基于 DFA 模型中状态之间的相似性， D^2FA 提出了新的压缩思路。假设 DFA 模型中有两个状态 s_1 和 s_2 ，并且它们接受相同的输入字符 $c \in C$ 时，都会跳转到同一个目标状态 $s \in S$ 。我们称之为：状态间的相似性。利用该相似性，我们可以消除 s_1 中的这些出边，并用一条从 s_1 指向 s_2 的 Default Edge 来替代它们。通过这种方式， s_1 中的边将仅仅包含和 s_2 中行为不一样（接受相同输入字符，目标状态不同）的边。

图 2.5 左半边展示了一个示例 DFA 模型。该 DFA 接受三个模式： $a^+, b+c$ 和 c^*d^+ （* 表示匹配 0 或者无穷个 * 前方的字符，+ 表示匹配 1 或者无穷个 + 前方的字符），且其输入字符集合为 $\{a,b,c,d\}$ 。该 DFA 的初始状态（initial state）是状态 1，而状态 2,4,5 则是其接受态（对应于要匹配的三个模式）。图 2.5 的右半边展示了对应的 D^2FA 模型。其中，没有标签的边指代 D^2FA 模型中引入的 Default Edge。在处理当前输入字符时，如果当前状态没有针对该字符的有效边，default edge 就会被用来确定下一个状态。假设当前输入字符流是 $aabdbc$ ，那么左边的自动机会依次访问状态 $1\underline{2}2\underline{3}4\underline{3}5$ ，而右边的自动机会依次访问状态 $1\underline{2}1\underline{2}314\underline{1}3\underline{5}$ 。此处，有下划线的状态代表该自动机中的接受状态。可以看到，DFA 和对应的 D^2FA 在接受相同的输入字符流后会进入的接收状态。也就是说，这两个自动机的匹配结果是一致的。事实上，我们可以证明这两个自动机的功能完全等价。

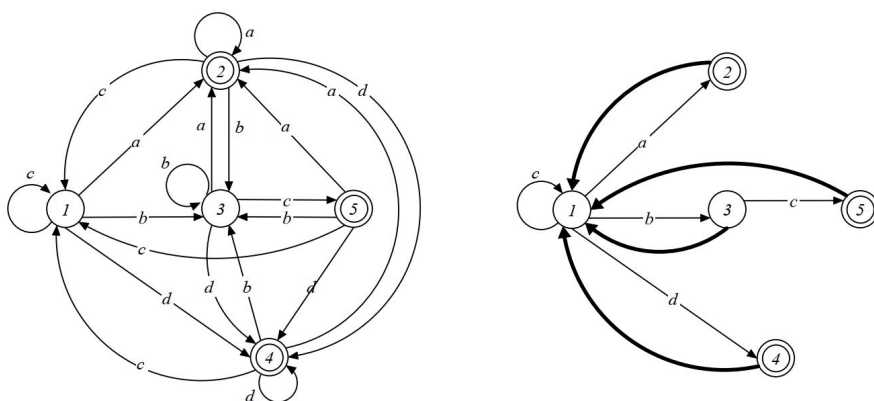


图 2.5 DFA 和 D^2FA 模型，接受三个模式： $a^+, b+c$ 和 c^*d^+

从图2.5中可以发现, DFA 模型有 20 条边, 而其对应的 D^2FA 模型只有 9 条边。在实际的应用中, DFA 的规模要大的多, 使用 D^2FA 模型进行压缩的收益可以提升到 95%。然而, D^2FA 模型也是有缺点的。我们可以注意到, 在 Default Edge 被触发时, 当前输入字符不会被处理掉, 而是会被“Delay”(延迟处理)。因此, 这样的压缩方式会使得 D^2FA 的字符处理速度比对应的 DFA 模型慢。

2. 构建 D^2FA 模型的基本原理

我们没有通用的算法直接生成一个 D^2FA 模型。事实上, 我们需要基于已有的 DFA 模型, 构建其对应的 D^2FA 模型。具体来说, 我们在需要在原本 DFA 模型上, 系统地使用 default edge 替换原本的边。在这个过程中, DFA 模型和 D^2FA 模型的等价性必须一直被保持。

定理 1: 在 D^2FA 模型中, 如果两个不同的状态 u 和 v 满足以下条件, 我们就可以引进一条从 u 指向 v 的 default edge, 同时去掉 u 中从 u 到 $\delta(a,u)$ 的边。我们可以证明, 这个过程前后, 该 D^2FA 模型是等价的。具体条件如下:

- u 有一条标签为字符 a 的出边;
- u 没有向外的 default edge;
- $\delta(a,u) = \delta(a,v)$ 。

基于定理 1, 我们可以得到更进一步的结论。如果存在多个输入字符 a 满足 $\delta(a,u) = \delta(a,v)$, 那么在 u 和 v 之间引入 default edge 将可以消除多条边。构建 D^2FA 的过程, 就是在原本 DFA 上不断使用定理 1 进行边的替换。由于每一步都维持了模型的等价性, 最初 DFA 模型和最终得到的 D^2FA 模型是等价的。图2.5右边的 D^2FA 模型就是在图左边 DFA 模型的基础上, 依次在以下节点间增加 default edge 构建出来的: (2,1), (3,1), (5,1) 和 (4,1)。

3. 如何构建边最少的 D^2FA 模型

对于每个状态, 只能有一条向外的 default edge。因此, 我们需要很仔细的选择每个状态的默认状态, 从而最大化冗余边的消除。同时, 有一点需要一直被满足: 不能形成由默认边组成的环。在这样的限制下, 默认边将共同组成树形结构。并且, 这些边将成为指向树的根节点的有向边。因此, 我们可以把选择默认边集合的过程转化为带权重的最大生成树问题。同时, 我们把每个 DFA 用来产生 D^2FA 生成树的图称为 space reduction graph (空间缩小图)。对于任何一个给定的 DFA 模型来说, 它的空间缩小图是一个无向的完全图, 而且拥有和原本 DFA 完全一样的顶点集合。空间缩小图中, 任意两个节点间会被赋予一个权值 $w(u,v)$ 。其中, $w(u,v)$ 的值是节点 u 和节点 v 间满足 $\delta(a,u) = \delta(a,v)$ 的边的数目总和减去 1。如果 u 和 v 之间没有任何这样的边, 那么 u 和 v 之间则不存在边。图2.5左边 DFA 对应的空间缩小图如图2.6所示。可以看到2.5右边的 D^2FA 模型的默认边组成了一棵生成树。而且, 这棵树正是从图2.6中选择的。该生成树边

的权重总和为 $3+3+3+2=11$ ，这也正是图2.5中的 DFA 模型和 D^2FA 模型的边的总数的差。

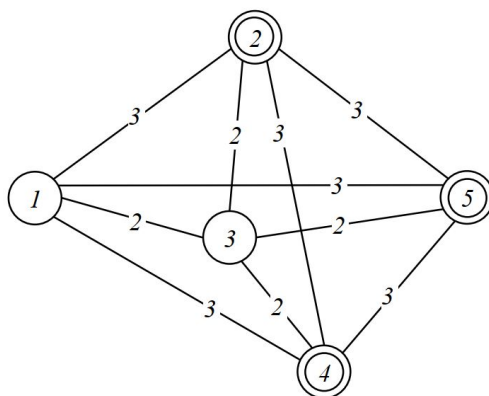


图 2.6 接受 $a+, b+c$ 和 $c*d+$ 的 DFA 的空间缩减图

4. 约束默认路径的最大长度

值得注意的是，从空间缩小图中选择不同的生成树就会生成不同的 D^2FA 模型。而且，空间缩小图往往有着多棵最大生成树。如图2.7所示，图2.6可以有有着两种不同结构的最大权重生成树。这两棵树边的总权重是一样的，这意味着对应的 D^2FA 模型的边的总数是一样的。但是，我们还需要考虑他们的**最大默认路径**（由默认边组成的最长路径）。例如，图2.7左边的最大默认路径为 3，而右图的最大默认路径为 2。当自动机沿着默认路径进行状态切换时，是不会处理任何输入字符的。因此，最大默认路径越大， D^2FA 模型的最坏情况下的字符处理速度就越低。

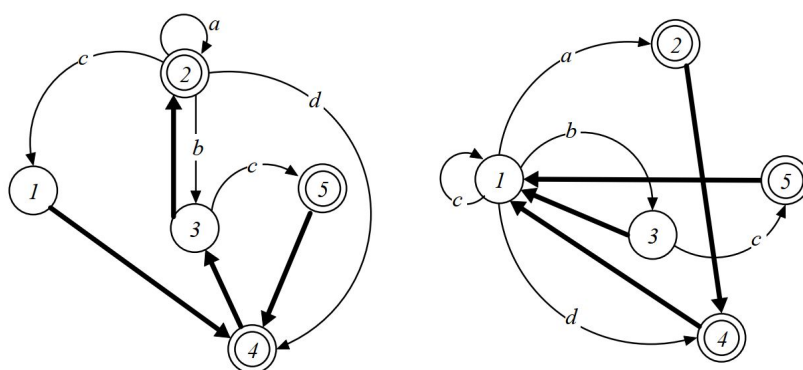


图 2.7 空间缩减图可生成的不同的最大权重生成树

如果我们的目标仅仅是最小化 D^2FA 模型的存储空间，那么我们只需要在空间缩减图中运行最大生成树的搜索算法即可。事实上，我们还需要解决另一个问题：每条默认边的方向如何选择。我们可以随机的在树中选择几个根节点（root），并让所有默认边都指向根节点的方向。但是，该方法构造的生成树往往会有很长的最大默认路径。这意味着，对应的 D^2FA 模型需要经过很多次默认

切换 (default transition) 才能处理掉一个输入字符。 D^2FA ^[30] 中采用的办法是: 在构造最大生成树时, 保证树的直径不会大于一个固定的阈值, 从而约束整棵树的默认路径。

采用上述方法构建最大生成树的过程分为两步。

- 基于 Kruskal 算法^[31], 按照空间缩减图中边权重 d 的降序依次检验每条边, 对于每条被检验的边 u,v :
 - 如果他不属于同一棵树, 而且增加这条边也不会创建一棵直径超过固定阈值的树, 则这条边可以被选择;
 - 否则, 继续检验下一条边。

一旦所有的边都被检验过了, 我们就会得到一棵或者多棵生成树。其中, 树上的边就是我们要添加在 D^2FA 模型中的默认边。

- 我们为每棵生成树选择一个根节点, 使得其叶子结点到根节点最短路径的长度总和最小。值得一提的是, 检测树的直径是一个比较复杂的计算过程。此处我们不展示该算法的伪代码, 我们将在下一节给出其优化版本的伪代码。

2.1.5 A DFA 模型

相比 DFA 模型, Delayed Input DFA (D^2FA) 模型实现了非常紧凑的存储。但是, D^2FA 模型却有着两点明显的不足。第一, D^2FA 模型缺少最坏情况下执行时间的保障, 坏的输入序列可能会使其频繁触发默认边, 从而极大的降低字符处理速度; 第二, 该构建过程的计算复杂度很高。为了解决 D^2FA 模型中存在的两个问题, M. Becchi 等人提出了相应的优化。

1. A DFA 模型简介

A DFA^[32] (Amortized time/bandwidth overhead DFA) 模型是 D^2FA 模型的优化版本。它可以直接用 D^2FA (见第1小节) 的五元组 $(Q, \Sigma, \Delta, q_0, F)$ 来描述。A DFA 模型与 D^2FA 模型在数学上的表示和实际执行时的操作完全一致。两者唯一的不同在于, 他们的构建过程和构建完成后状态间的连接关系不同。具体来说, A DFA 模型与 D^2FA 模型都是用 “Default Edge” 来替代 DFA 模型中的一些冗余边; 然而, A DFA 模型在选择 “Default Edge” 时和 D^2FA 模型采取了不一样的算法。因此, 构建 D^2FA 模型和 A DFA 模型所需要的算法复杂度是不同的。更重要的是, D^2FA 模型和 A DFA 模型最终需要的存储空间和运行时的访存次数是不同的。

相比于 D^2FA 模型, A DFA 模型有着两个显著的优势。

- A DFA 模型可以保证, 对于任意长度为 N 的输入字符流, A DFA 模型最多产生 $N(k+1)/k$ 次状态转换 (包括沿着普通边和沿着默认边进行状态转换)。因此, A DFA 模型可以提供最坏情况下执行时间的保障。此处, k 是一个正

整数。随着 k 值的增大, ADFA 模型的状态转换次数将逐渐接近 DFA 模型。同时, ADFA 模型的性能也会不断接近 DFA 模型。不过与此同时, ADFA 模型的压缩效果也会不断下降。特别的, $k=1$ 时 ADFA 模型可以取得最大化的压缩效果。此时, ADFA 模型可以保证: 最坏情况下, 处理长度为 N 的输入字符流只需要 $2N$ 次状态转换。相对应的, D^2FA 模型的最坏情况状态转换次数则是取决于其生成树的最大默认路径 (往往远大于 2)。总的来说, ADFA 模型可以让用户灵活的地在存储和执行速度之间做权衡。通过详细的评估可以发现:

- 在实现相同程度的压缩效果时, ADFA 模型可以提供比 D^2FA 更好的执行速度;
 - 在提供相同的最坏执行速度保证时, ADFA 模型可以取得 10 倍于 D^2FA 模型的压缩效果。
- 在基于正则表达式集合构造对应的 ADFA 模型时, 我们不需要先构造出对应的 DFA 模型, 而是可以根据 NFA 模型直接构造出对应的 ADFA 模型。这是因为, ADFA 的构造算法可以嵌入在 NFA 到 DFA 的转换算法 (即算法 2.1) 之中。在构建 ADFA 模型之前, 我们不需要先构造出一个可能规模大到不可行的 DFA 模型。这使得 ADFA 模型可以支持更大规模的正则表达式输入。而且在构造 ADFA 模型时, 我们不需要维护一个规模非常大的图结构。总的来说, ADFA 模型的构造算法的时间、空间复杂度相比 D^2FA 的构造算法要低得多。

2. ADFA 模型的核心思想

ADFA 模型的提出基于如下观察: 正则表达式匹配的计算过程都开始于单个初始状态 (initial state); 自动机中存在大量“往回指的边”, 它们要么指向自动机的初始状态, 要么指向在抵达当前状态之前途径过的一些邻居。事实上, 在 D^2FA 模型中, 被压缩的边主要就是这些“往回指的边”。

定义 1: 自动机中的每个状态都有一个固定的属性: **state depth(状态深度)**。每个状态的状态深度是一个非负整数, 等于在状态转移图中该状态到自动机初始状态的最短距离。

也就是说, 自动机初始状态 s_0 的深度为 0; s_0 可以直接到达的一组状态 S_1 的深度都是 1; S_1 中的状态可以直接到达的另一组状态 S_2 的深度都是 2; 以此类推。借助于这个概念, 我们可以有如下两条定理:

定理 2: 在 DFA 中, 如果任何一条默认边都满足: 其出发状态的深度 d_i 大于它的目标状态的深度 d_j , 那么处理任何输入到该自动机的长度为 N 的字符流所需要的状态切换总数 (包括沿着默认边的切换) 不会超过 $2N$ 。换句话说, 一个只有回向默认边的 DFA 可以保证: 最多需要 $2N$ 的时间进行长度为 N 的字符流处

理。

定理 2 的证明过程如下。对于 ADFA 模型，每一个输入字符会导致恰好一次常规状态转换（labeled transition）和零或多次默认转换（default transition）。设想某个时间节点，一连串 d 个默认转换将从状态 s 开始被触发。根据定理 2 的假设，我们知道默认转换总是朝着深度更浅的状态。因此，状态 s 的深度一定是大于等于 d 。也就是说，在此之前至少有 d 次常规转换发生了。因此，默认转换的次数总是至少比常规转换的次数小 1。不仅如此，长度为 N 的输入字符流只会引发 N 次常规转换。因此，该过程中总的状态转换数不会大于 $2N-1$ 。

定理 3: 在 DFA 中，如果任何一条默认边都满足：其出发状态的深度 d_i 和其目标状态的深度 d_j 满足： $d_j \leq d_i - k$ (k 是任意正整数)。那么，处理任何输入到该自动机的长度为 N 的字符流所需要的状态切换总数（包括沿着默认边的切换）不会超过 $N(k+1)/k$ 。

定理 3 是定理 2 的推广。定理 3 中，默认边的出发状态和目标状态的深度差不再要求是大于等于 1，而是进一步要求大于等于 k 。定理 3 的证明如下。最坏情况下，在发生最少数量的常规转换后就会发生一次默认转换。而默认转换的始终状态的深度的差值大于等于 k ，因此在发生每次默认转换前至少要发生 k 次常规转换。也就是说，最坏情况下，每 k 次常规转换后发生一次默认转换。这意味着，为了处理长度为 k 的输入字符流，最多发生 $k+1$ 次状态转换。也就是说，长度为 N 的输入字符流最多需要 $N(k+1)/k$ 次状态转换。

3. ADFA 模型的构建算法

ADFA 模型构建算法的目标是为了实现一个有 default transition（默认转换）的 DFA。而且该 DFA 应当可以实现最大程度的模型压缩，并保障最坏情况下的状态转换开销。基于定理 2 和定理 3，我们可以利用有向图上最大生成树问题来形式化地描述 ADFA 的构建过程。同 D^2FA 一样，我们也把这样的有向图称之为 space reduction graph（空间缩减图）。

ADFA 模型的空间缩减图可按以下四步构建：

- 为 DFA 模型中的每一个状态在 ADFA 的空间缩减图中创建一个副本；
- 对于空间缩减图中的每一个点对，如果点 s_1 的深度和点 s_2 的深度差不小于 k ，在 s_1 和 s_2 之间添加一条边；
- 为第二步中添加的边设置方向（从深度更深的指向深度更浅的）；
- n_X 和 n_Y 之间边的权值等于 s_X 和 s_Y 之间有着相同行为的边的总数。其中， s_X 和 s_Y 是 DFA 中的两个状态，而 n_X 和 n_Y 是状态缩减图中对应的两个点，在空间缩减图中，每一条边意味着一条可以被选择的默认边（default edge）。同时，这些边的权重代表着：如果这条默认边被选用，可以移除多少带标签的常规边。每个 DFA 状态最多只能有一条向外的默认边。因此，挑选一组最佳的默

认边（实现最大程度的模型压缩）等价于在这个有向图中搜索一个最大生成树（或者森林）。不同于 D^2FA 模型，ADFA 模型在找到最大生成树（或者森林）后无需再计算每条边的方向，也无需确定根节点。这是因为，ADFA 的空间缩减图是有向图。另一点不同于 D^2FA 的地方是，ADFA 在搜索最大生成树时，不需要化解可能在图中产生的环状结构。这是因为，ADFA 的状态缩减图从一开始就不包含任何环：每条边都是从深度更深的节点指向深度更浅的节点，因此不会产生环状结构。因此，ADFA 模型在构建时可以省掉这一计算复杂的过程。

ADFA 模型中引入空间缩减图的概念是为了和 D^2FA 的算法做对比。事实上，我们在构建 ADFA 模型的最大生成树时，不需要构建这样一个庞大的图结构作为辅助。具体来说，这个图问题可以降维成：每个 DFA 状态，从比自己深度浅（深度差大于等于 k ）的节点中，找一个和自己有着最多相同行为的边的节点作为自己的默认状态。当有多个节点满足条件时，优先选择深度最小的节点。（这样可以减少最大默认路径，同时优化访存的局部性。）

如果我们进行 DFA 模型的广度优先遍历，那么节点深度的计算和默认边的选择可以在一次遍历中做完（伪代码如算法 2.2 所示）。在该算法中，DFA 可以被描述为状态总数 n 和状态转移函数 $\delta(states, \Sigma) \rightarrow states$ 。当一个状态 s 从 queue 中被抽取出来时，所有深度比他小的节点都已经被处理过了。因此，这些节点的 **depth** 已经被赋予了一个正确的值。虽然比状态 s 深度更深的节点还未被处理，但是他们的深度都已经被初始化 n 。因此，这些节点不会引发错误的操作。事实上，去除冗余边的操作也可以在这一次算法遍历中完成。类似的，该算法还可以和先前提到的子集构造法（算法 2.1）组合在一起。因为在子集构造法中，DFA 状态节点正是按照与本算法相同的访问顺序被构造的。

2.1.6 其他的衍生自动机模型

除了 D^2FA 和 ADFA 模型外，还有很多衍生的自动机模型。这些衍生模型通常为了解决以下两个目标中的一个：

- **通过压缩机制实现更小的 DFA 内存占用。** 该目标并不包括解决状态爆炸的问题。相反的，这一类自动机通常是可为可行的 DFA 提供一个有效的紧凑的存储方式。它们往往利用边的冗余性进行自动机模型的压缩。前文提到的 D^2FA 模型（1）和 ADFA 模型（2.1.5）就是属于这一类自动机模型。该类别的类似模型还有 δFA ^[33] 模型。
- **在 DFA 模型变得不可行时（发生状态爆炸时），用来替代 DFA 模型。** 这一类自动机包括 XFA^{[34][35]}、HistoryFA^[36]、HybridFA^[37]、MultipleDFA^[38] 和 BFSM^[39]。值得一提的是，这些模型往往是基于 DFA 模型的拓展，它们需要借助于一个高效的 DFA 模型作为核心模块。

算法 2.2 ADFa 构建算法

```

Data: DFA dfa=(n, $\delta$ (states,  $\Sigma$ )); 整数 k
Result: 数组 default[n]
1 初始化空队列 queue 和初始化空数组 depth[n];
2 for state s  $\in$  states do
3     | depth[s]=n;
4     | default[s]=s;
5 end
6 depth[0]=0;
7 queue.push(0);
8 while !queue.empty() do
9     | state s = queue.pop();
10    | int saving=1;
11    | for char c  $\in$   $\Sigma$  do
12        | if depth[ $\delta$ (s, c)] $\neq$ n then
13            | depth[ $\delta$ (s, c)]=depth[s]+1;
14            | queue.push( $\delta$ (s, c));
15        | end
16    | end
17    | for state t  $\in$  states && depth[t]  $\leq$  depth[s]-k do
18        | int common=common_transitions(s,t);
19        | if common>saving || (common==saving && depth[t]<depth[default[s]]) then
20            | default[s]=t;
21            | saving=common;
22        | end
23    | end
24 end

```

2.2 ZYNQ 的异构计算平台

ZYNQ-7000^[40] 代表了一系列基于 Xilinx SoC 架构 FPGA 芯片。这些产品将拥有丰富特性的通用处理器系统 (PS) 和 28 纳米工艺的 Xilinx 可编程逻辑 (PL) 集成到了同一个芯片中。其通用处理器系统通常基于双核心或单核心 ARM Cortex-A9。不仅如此, 该系统还包括片上存储器、外部 DDR 接口以及丰富的外设接口。通过这样的方式, ZYNQ 的单个芯片可以同时提供通用处理器的普适计算能力和 FPGA 的定制计算能力。

2.2.1 CPU+FPGA 的架构

ZYNQ-7000 的架构框图如图2.8所示。图2.8上半部分是基于 ARM CPU 的通用处理系统，其核心是两枚最高主频达 1 GHz 的 Cortex-A9 处理器。除此以外，PS 部分还有丰富的外设控制器，包括以太网控制器、USB/SPI/I2C/CAN 等接口、UART 串行接口、8 通道 DMA 控制器、多功能 DRAM 控制器和 Flash 控制器。这些部件和 ARM CPU 通过 AMBA^[41] 总线系统实现互联。

图2.8中黄色部分包括了现场可编程逻辑门阵列（FPGA）以及常用的逻辑硬核。可编程逻辑不同于 ASIC 电路，它是一种基于查找表的可编程电路。用户可以通过使用 Verilog 语言或者 VHDL 语言进行编程，设计他们想要的定制电路。然后，用户可通过 Vivado 集成开发工具，将 Verilog 代码编译成对应的逻辑结构和查找表。将生成的查找表写入 FPGA，就实现了对 FPGA 的烧写和逻辑定制。随着 FPGA 技术的发展，当代的 FPGA 逻辑中都包含了常用的硬核。这些硬核有着固定的内部逻辑和外部接口，灵活性要低于可编程逻辑。但是，他们往往拥有更高的能效、更快的速度和更低的面积占用。ZYNQ-7000 的内置硬核资源包括：

- 36Kb 容量的块随机访问存储器（BRAM），最大支持 72bits 的宽度；
- 数字信号处理器（DSP），可以支持 18 比特 *25 比特的有符号乘法、48 比特的加法或者累加或者 25 比特的预加器；
- 可编程输入输出模块，可以支持 1.2V/3.3V 的 LVCMOS、LVDS 或 SSTL；
- 高速串行收发器，数据速率最高可达 12.5 Gb/s；
- 2 代 PCIE 接口模块，最多支持 8 通道。

图2.8中 PS 部分和 PL 部分之间可以通过 EMIO GPIO 接口进行通讯。ARM 处理器可以通过该接口对 FPGA 进行引脚级的控制和读取。除此以外，PS 的 AMBA 总线系统也预留了 AXI 接口给 PL 部分。如图2.8所示，PS 可以通过标准的高性能 AXI 端口或者慢速的通用 AXI 接口对 PL 进行访问和控制。通过 AXI 接口的方式进行 PS+PL 的互联，可进行便捷的系统集成，也可以保证了通讯的速度和效率。因此，本文正是基于 AXI 总线进行的系统集成。

2.2.2 AXI 总线系统

Xilinx 采用先进可拓展接口^[42]（Advanced eXtensible Interface, AXI）协议进行知识产权核心（Intellectual Property, IP）的系统级集成。AXI 是 ARM 公司 1996 年提出的 AMBA 总线协议的一部分，包含了一系列微控制器总线协议。第一个版本的 AXI 协议在 2003 年被提出。7 年后，AMBA 4.0 被提出。同时，AXI 的第二个主要版本 AXI4 被提出。总的来说，一共有三种 AXI4 接口：

- **AXI4:** 面向高性能的内存映射式访问；
- **AXI4-Lite:** 低吞吐的内存映射式访问（比如对控制状态寄存器的访问）。

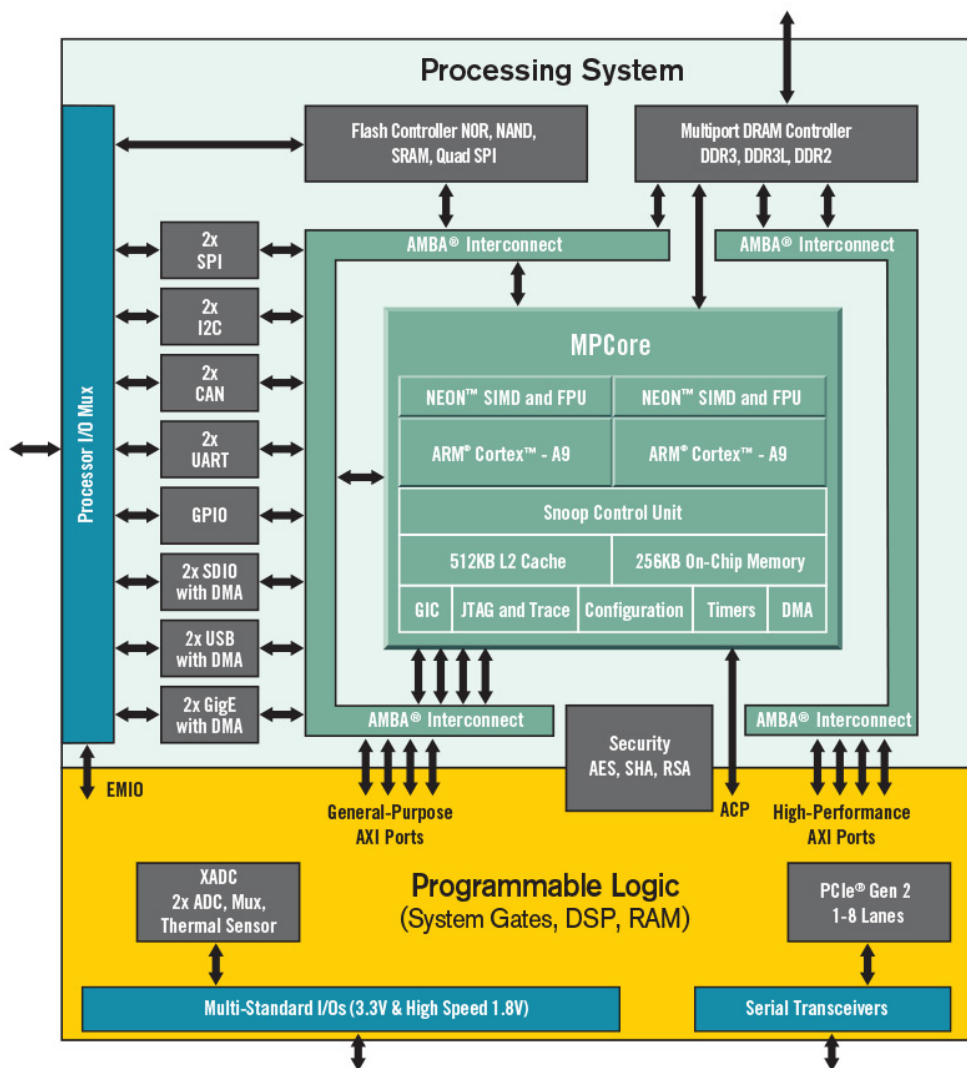


图 2.8 ZYNQ 体系结构总览

- **AXI4-Stream:** 面向高速的流数据的访问。
Xilinx 产品广泛采用 AXI4 总线，获得了很高的生产率、灵活性和可用性。
- 由于只使用 AXI4 规范，IP 开发者只需要仅仅学习这一种协议。
- AXI4 提供 3 种上述协议，可以为不同的应用场景提供适用的协议。
 - AXI4 是一种内存映射式接口，允许高吞吐的数据传输。AXI4 的单个地址发送阶段后可以附带最大 256 个数据周期的突发传输。
 - AXI4-Lite 是一种轻量化的内存映射接口，不支持突发数据传输。它有很小的资源占用，而且是一个很小很简单的接口，便于开发者去实现和使用。
 - AXI4-Stream 接口无需地址传输，而且可以有无限大小的突发数据传输。因此，AXI4-Stream 有着非常高的传输速率和简单的控制接口。
- 使用 AXI4 总线规范还极大地提高了产品的可用性。用户不仅仅可以使用

Xilinx 公司开发的 IP 核，还可以使用 ARM 社区提供的第三方 IP 核。

2.2.3 AXI4 直接内存访问

如图2.8所示，PS 和 PL 部分可以通过 AXI 总线进行数据传输。为了实现系统内存和 AXI4-Stream 类型的目标设备间的高吞吐数据传输，我们需要借助于 AXI4 DMA^[43] 控制器进行高效的直接内存访问（Direct Memory Access, DMA）。Xilinx 公司提供的 AXI DMA 核心是一种软 IP 核。该 IP 不是固化在 Xilinx 的 FPGA 中的硬件核心，而是需要用可编程逻辑搭建。AXI DMA 可以提供介于内存映射设备和 AXI4-Stream 接口设备之间的高带宽数据传输。它支持可以选的 scatter/gather 能力，可以代替中央处理器负责数据传输任务的控制和执行。图2.9展示了 AXI DMA 的功能组件。其中，最主要的数据搬运是通过两个内置的 DataMover 模块进行的。上方的 DataMover 可以从内存映射的设备（比如系统 DDR）中读取大量数据，并将数据写入到 AXI Stream 类型的目标设备中。而下方的 DataMover 则是从 AXI Stream 的设备中读取流数据，并将这些数据写入到内存映射设备中。这两个 DataMover 可以独立的工作。用户可以通过 AXI4-Lite 从接口读写 AXI DMA 的寄存器，从而控制该核心的配置、启动和停止。

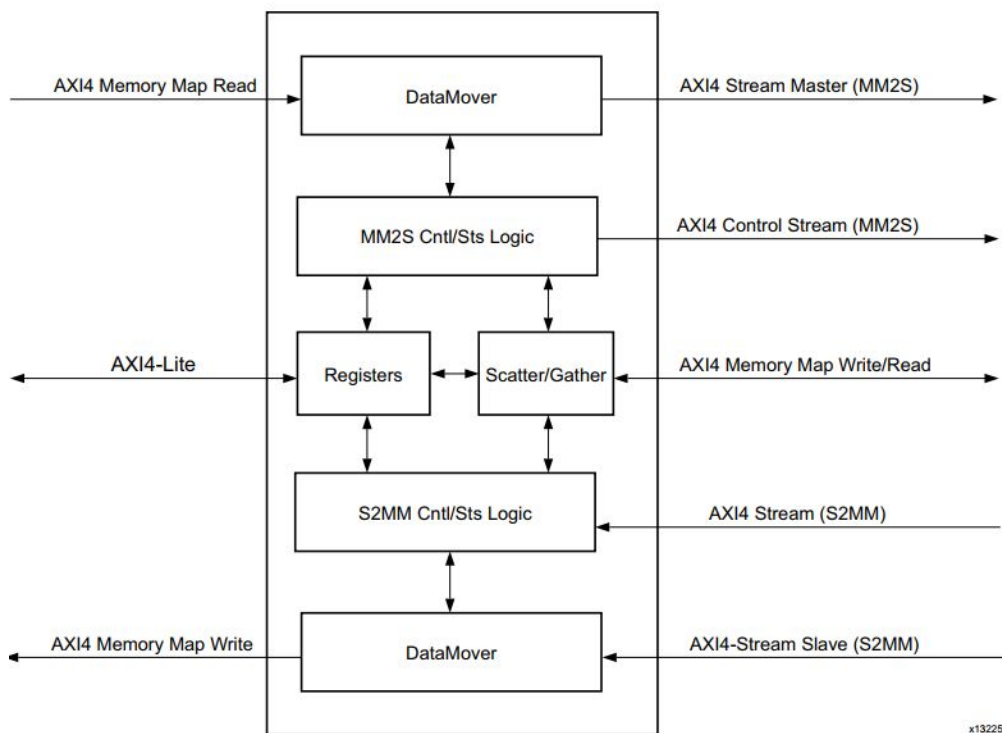


图 2.9 AXI DMA 模块框图

第3章 LAP: 高效的轻量级自动机处理器核心

本文核心目标是实现更快的基于 ADFA 模型（见第2.1.5小节）的轻量级自动机处理器。一方面，ADFA 模型搭配上字母压缩策略实现了已知的最高效的 DFA 模型压缩^[32]。尽管在编译复杂的正则表达式集合时可能会发生状态爆炸的问题，这使得 ADFA 模型不适用于所有场景下。但是，用来解决状态爆炸问题的有效方法^{[38][37][34][35][14]} 却又往往使用 DFA 模型作为核心模块。因此，无论是否会发生状态爆炸，高效的 DFA 处理能力是必不可缺的。另一方面，本文提出的用于实现高效的 ADFA 执行的策略与解决状态爆炸问题的方法是完全正交的。因此，本文要解决的核心问题和提出的方案都是有着重要的意义的。

本章将主要介绍本论文的核心工作：轻量级的自动机处理器核心 LAP 的设计、实现与评估。（我们已经在国际会议上向国内外学者介绍过这一工作，对应的论文见第5.2小节。）我们将首先介绍 LAP 的研究动机。随后，我们将介绍 LAP 的硬件模型和 LAP 指令集。在第三节，我们将进一步拓展 LAP 的硬件模型，并优化 LAP 指令集。相应的，我们提出了两条新的硬件指令，它们将有助于实现更快的 ADFA 计算。在第四节，我们将阐释 LAP 的微体系结构，包括其流水线结构和细粒度多线程技术。第五节中，我们将讲述 LAP 编译器的实现。最后，我们将评估 LAP 的资源占用、性能和功耗。

3.1 研究动机

我们首先分析下已有的模式匹配专用硬件的不足。Micron 公司的 Automata Processor^[15] 基于空间架构实现了高并发的基于 NFA 模型的执行，然而这样的架构需要基于大规模的 DRAM 存储器阵列，也需要复杂的硬件互联。另一方面，基于 DFA 模型的 RegX^[14] 处理器需要借助于外部的大容量 DRAM 存储其自动机模型代码。为了掩盖 DRAM 访存延迟，它的内部还使用了较为复杂的层次化硬件 Cache。除此以外，HAWK^[26] 和 HARE^[25] 拥有非常复杂的硬件流水线。他们都有着共同的不足：需要借助于大容量的存储器而且拥有复杂且庞大的硬件架构。因此，这些专用硬件往往需要占用较大的芯片面积，同时产生较大的功耗。

我们试图设计并实现一款更轻量化的自动机处理核心，LAP。我们之所以称之为“轻量级”的处理器核心，是因为这款处理器满足两个特征：

- 在实现相同的模式匹配任务时，LAP 相比于已有的解决方案需要更小容量的存储器。这使得 LAP 可以将整个自动机模型存储进有限大小的 SRAM 中，并在不借助外部 DRAM 的情况下独立的运行。

- LAP 的硬件流水线所使用的硬件资源很少。在已有的 CPU 系统中添加 LAP 核心既不会显著提升芯片面积，也不会显著提高系统的总体功耗。

为了使得 LAP 的硬件流水线结构简单、资源占用少，我们选择了基于确定性有限自动机 (DFA) 模型实现模式匹配计算。非确定性有限自动机 (NFA) 有着非常高的计算复杂度。对于每一个输入字符，我们需要检查每一个 NFA 中活跃的状态，并实现这些状态的切换。因此，基于 NFA 模型的自动机专用硬件要么处理速度慢（串行进行状态切换），要么需要支持大规模的并行访存和计算。确定性有限自动机 (DFA) 最多只能有一个活跃状态。因此，对于每一个输入字符，DFA 只需要进行一次状态切换。事实上，在将 NFA 转化为 DFA 的过程中 (见算法 2.1)，我们已经充分挖掘了 NFA 中计算的冗余性并予以消除。总的来说，DFA 模型虽然相比 NFA 模型需要更大的存储空间，但是 DFA 模型极大地降低了所需要的计算量。而且随着很多优化技术被应用到 DFA 模型中，DFA 模型的存储需求也被极大地降低了。这使得 DFA 模型可以非常高效的执行在定制化的硬件体系结构上。

为了使 LAP 需要的硬件存储器容量大幅降低，我们需要高度压缩由正则表达式集合生成的自动机模型。真实的模式匹配任务（深度网络包检测、基因串匹配）往往需要同时匹配大量的正则表达式。因此，根据这些正则表达式产生的 DFA 模型往往也会体积较大。这使得我们需要消耗很多的片上资源用于存储模型的二进制代码。在 LAP 中，我们使用 ADFA 模型（一种优化后的 DFA 模型，详情见第 2.1.5 小节）代替 DFA 模型，作为主要的自动机模型。我们可以借助 ADFA 模型强大的压缩算法实现自动机模型的大幅压缩，从而降低存储器容量的占用。

尽管 UAP^[27] 使用 ADFA 模型实现了较为轻量级的模式匹配计算，然而该处理器基于 ADFA 模型的处理速度却并不让人满意。ADFA 模型引进的 **default transition**（默认状态转换）可以实现高效的模型压缩，但同时也带来了更复杂的计算流程。这使得 UAP 在基于 ADFA 模型进行模式匹配计算时字符处理速度较慢。为了享受 **default transition** 带来的存储效益，又不过度损害自动机的处理速度，我们需要提出新的自动机执行算法。同时，为了支持新的执行算法，我们需要有新的硬件支持。本文正是按照这样的思路，采取软硬件协同优化的方式来设计和实现 LAP。相比于其他轻量级的自动机处理专用硬件，LAP 可以实现更快的处理速度。总的来说，我们的目标是实现一款更高效的轻量级自动机处理器。

3.2 LAP 的指令集设计

本文所设计实现的轻量级自动机处理器 LAP 不同于传统的冯诺依曼结构，它使用一种专门为执行自动机模型所设计的硬件抽象。同时，LAP 是一类可编程处理器，它拥有自己的指令集。本小节将先介绍 LAP 的硬件抽象模型，然后介绍其指令集。

3.2.1 LAP 的硬件执行模型

图3.1展示的是LAP所执行的硬件执行模型。该执行模型与UAP^[27]和UDP^{[28][44]}系统的执行模型类似，可以支持多种多样的自动机模型，包括NFA、DFA、ADFA以及其他已知的所有自动机模型。该硬件模型的功能是顺序处理一个包含N个输入字符的字符流。此处，我们把该自动机处理器硬件模型的执行过程划分成相邻的N阶段。每当处理完字符流中的一个字符后，该自动机处理器会紧接着读取该字符的后一个相邻字符。同时，当前输入字符被处理和消耗掉，意味着当前阶段的结束和下阶段的开始。

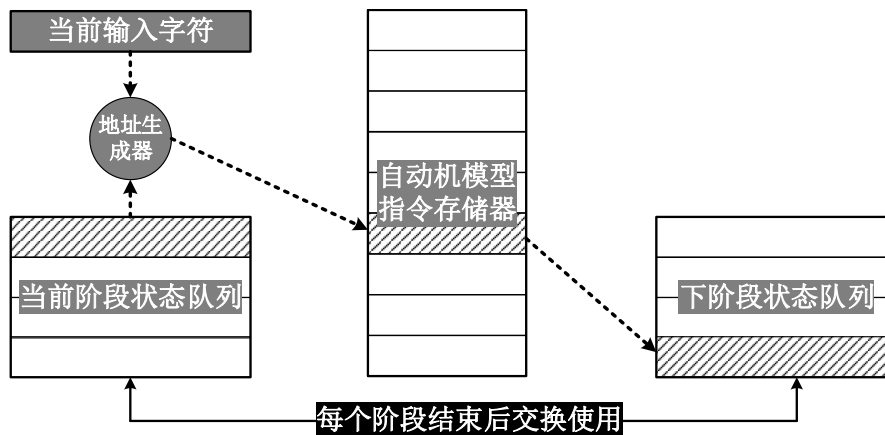


图 3.1 LAP 的硬件执行模型

“当前阶段状态队列”是使用寄存器组实现的硬件队列（Queue），存储当前阶段被激活的所有自动机状态。之所以使用队列，是为了支持非确定性有限自动机：非确定性有限自动机可能有多个状态同时处于活跃状态。如果不需要支持NFA模型，那么该执行模型中的两个队列可以换成单个寄存器。每个执行阶段，我们需要按顺序从当前阶段状态队列中依次取出所有的活跃状态，并顺序的处理他们的状态切换。最简单的情况下，对于取出的每一个活跃状态，LAP架构将根据取出的当前状态和当前输入字符计算出相应的指令地址。然后，LAP从自动机模型指令存储器中取出一条指令。该条指令编码了当前状态在接受当前输入的情况下应该切换到的目标状态。于是，我们将该目标状态写回“下阶段状态队列”。等到“当前状态队列”为空时，也就是说所有的活跃状态都处理完后，

LAP 将交换“当前状态队列”与“下阶段状态队列”。通过这样的方式，在处理完当前输入字符后，“当前阶段队列”会包含所有上个阶段被激活的自动机状态，而“下阶段状态队列”又会变成空队列。如此便可以循环往复。

值得一提的是，为了支持各种各样的自动机模型，从“当前状态队列”中取出的每个状态的处理过程并不总是像上述过程那么直接和简单。在 LAP 中，每一个自动机状态拥有特定的一种类型。上述过程讲述的恰好是最简单的一类自动机状态：BASIC 类型。对于有着复杂类型的自动机状态，在处理这一个状态的切换时，LAP 可能需要串行地从自动机模型指令存储器中连续访问多次，也可能向“下阶段状态队列中”写回多个状态。这一部分内容将在3.2.2中详细介绍。

3.2.2 LAP 指令集

事实上，每一条 LAP 指令不仅编码了一个自动机状态的 ID(身份编码)，也编码了该状态对应的类型和其他附加信息。LAP 的机器指令编码如图3.2所示。每条 LAP 机器指令包含“Signature”、“Target”、“Type”、“Accepted”和“Attach”五个域，共计 32 个比特。其中，“Signature”域是每条指令的指纹，用于检验该条指令是否是一条有效指令。之所以需要设立这一个域，是因为 LAP 会重叠的存储每个状态的状态切换表，每个状态可能会取到一条不属于该状态的指令。设立这样一个域，可以在每个状态取到不属于自己的指令时采取特殊的应对措施。“Target”域则是编码了目标状态的身份编码。在 LAP 中，每个自动机状态被编码成 12 个比特。因此，在本文的实现中，每个 LAP 核心支持最多 4096 个不同的自动机状态。“Type”域则是定义了目标状态的类型。在本文中，总共支持 7 种自动机类型。事实上，我们正是按照“Type”域将 LAP 指令划分成 7 类。LAP 的“Type”域就类似于 RISC 指令中的 OpCode（指令类型）。“Accepted”域只有 1 比特，用来指示该状态是否是接受态。如果当前状态的“Accepted”域值为 1，LAP 将向宿主 CPU 汇报该接受态。最后，8 比特的“Attach”域是附加信息。对于不同类型的状态类型，“Attach 域”的值有着不同的解释和用法。

指令格式				
Transition Primitive Instruction Format				
Signature	Target	Type	Accepted	Attach
8bit	12bit	3bit	1bit	8bit

图 3.2 LAP 的指令格式

我们参考 UAP^[27] 和 UDP^[28] 这两款处理器的指令集，设计了 LAP 指令集。表格3.1展示了 LAP 支持的指令类型和功能描述。为了支持 NFA、DFA 和 ADFA 模型，LAP 总共支持 7 条指令，也就是支持 7 种类型的自动机状态。事实上，在

后续章节3.3.3还会提出对这些指令的进一步优化。接下来，本文将会分类介绍这些指令所对应的自动机状态的执行细节（从该状态被 LAP 从“当前阶段状态队列”中取出到该状态执行结束）。我们将使用伪代码的形式展现这些过程，此处做一些约定：

- 参考图3.1, 我们用 CSQ(Current Stage Queue) 代指当前阶段状态队列，并使用 NSQ(Next Stage Queue) 代指下阶段状态队列。对于这两个队列，我们可以从队列头取出一个状态（pop()）或者向队列尾存入一个状态（push()）。
- 所有涉及到对指令存储器进行访问的操作都使用红色字体标识出来。Mem[Addr] 代表读取指令存储器地址为 Addr 处的指令。
- signature_check(input_char, Instruction1.signature) 函数的作用是对取出来的指令 Instruction1 的 signature 域和当前输入字符进行比较。如果检验通过，说明 Instruction1 是一条有效指令。否则，取出来的指令是无效的。准确来说，这条指令不属于当前状态。详情请参看3.5.2。
- Execute(Instruction2) 函数是指：不将 Instruction2 写入 NSQ，而直接将该指令转化为对应的自动机状态，并立即执行该状态。被执行的状态可以属于以下介绍的任何类型。该函数被调用时就相当于嵌套的立即调用以下介绍的6个算法之一。这也意味着，这条指令的执行过程需要多个连续的周期。
- assert() 是用来终止当前算法的执行的，如果断言不被满足。使用此函数是想说明，以下介绍的6个算法会针对特定的状态类型被选择性的调用。

表 3.1 LAP 支持的 7 条指令和它们的功能描述

指令类型	功能描述
NULL	空指令，在接受任何输入字符时不产生任何操作
BASIC	基本指令，实现 DFA 模型中带标签的状态切换
EPSILON	支持 NFA 模型，执行时可能会写回多个下阶段状态
PERSIST	优化 NFA 执行，将当前状态立即写回下阶段状态
DEFAULT_BASIC	支持 ADFA 模型，实现基础的 default transition
DEFAULT_CASCADE	支持 ADFA 模型，实现级联的 default transition
MAJORITY	压缩自动机状态的出边数目，实现模型压缩

1. 支持 DFA 模型

DFA 模型中的每个状态都会被实现为 BASIC 类型。同时，BASIC 状态也被用在 NFA 和各类衍生的动机模型中。这类状态只包含若干条向外的 labeled edge。如算法3.1所示，当前状态首先从当前阶段状态队列中被取出。如果经检测，该状态是 BASIC 类型状态，则继续处理。首先，我们需要从指令存储器中取出一条指令 Instruction1。指令的地址是通过当前状态的 Target 域也就是其状态编号

和当前输入计算出来的。此处的地址计算非常简单，这得益于编译算法3.5.2的巧妙安排。如果当前 `Instruction1` 的指纹检测通过，则说明这是一条有效指令。随后，该指令将被转化为对应的状态并写回 NSQ。否则，当前状态在当前输入下没有对应的有效出边。此时，LAP 不写回任何状态直接结束当前状态的处理。

算法 3.1 BASIC 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==BASIC);
3 Instruction1 = Mem[current_state.Target + input_char];
4 if signature_check(input_char,Instruction1.signature)==True then
5   |   NSQ.push(Instruction1);
6 end

```

2. 支持 NFA 模型

为了支持 NFA 模型，需要增加一种新状态类型：EPSILON 类型。DFA 模型的状态转移函数 $\delta(q, a)$ 在接受输入 a 时只会最多一个状态。此过程，可以使用 BASIC 状态实现。而 NFA 模型的 $\delta(q, a)$ 在接受输入 a 时可能会连续激活多个状态。在本文，我们实现 NFA 模型的方式是实现 NFA 模型中特有的一种边： ϵ 边（见第2.1.2小节）。EPSILON 类型状态，除了进行类似于 BASIC 类型的状态转换外，还会额外执行另一个状态（ ϵ 边指向的状态）。如算法3.2所示，代码 1-5 行的操作和3.1是一样的。然而，在此之后，EPSILON 状态会进行第二次对指令存储器的读取。使用的地址，是编码在 LAP 指令的“Attach”域的值。随后，立即解析出该指令包含的状态并立即执行该状态。值得注意的是，如果 ϵ 边指向的状态仍然是 EPSILON 状态，则可能会嵌套执行多个 EPSILON 状态。通过这样的方式，一个状态可以连续地激活 1 到无穷个状态，从而实现 NFA 的状态转移特性。

算法 3.2 EPSILON 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==EPSILON);
3 Instruction1 = Mem[current_state.Target + input_char];
4 if signature_check(input_char,Instruction1.signature)==True then
5   |   NSQ.push(Instruction1);
6   |   Instruction2 = Mem[current_state.Attach];
7   |   Execute(Instruction2);
8 end

```

除此以外，PERSIST 类型状态是为了优化 NFA 的处理性能。NFA 模型中有一类状态一旦激活就永远不会失活。这类状态在 NFA 的状态转移图中有一个典型的特征：有一条 ϵ 出边指向自己。NFA 自动机的初始状态往往就是这样的

PERSIST 类型。我们也可以使用 EPSILON 类型实现这样的结构。然而，这将使得我们需要嵌套调用两次 EPSILON 执行算法。事实上，如果我们已经知道某 ϵ 边的目的地正是当前状态自身，我们可以避免第二次存储器访问，直接将当前状态写回下阶段状态队列。如算法3.3所示，伪代码 1-5 行的操作类似于 EPSILON 状态的操作。然而，在此之后，PERSIST 状态在执行时，不需要进行第二次指令存储器访问，也无需嵌套的执行 EPSILON 状态。它直接将当前状态写回 NSQ。

算法 3.3 PERSIST 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==PERSIST);
3 Instruction1 = Mem[current_state.Target + input_char];
4 if signature_check(input_char,Instruction1.signature)==True then
5   |   NSQ.push(Instruction1);
6 end
7 NSQ.push(current_state);

```

3. 支持 ADFA 模型

为了支持 ADFA 模型，我们需要提供对 default transition（默认状态转换）的支持。为了优化性能，我们在 LAP 中实现了两类对 default transition 的支持。

我们为此引入的第一类新状态为 DEFAULT_BASIC。DEFAULT_BASIC 状态在自动机状态转移图中的特征为：它有一个默认状态，但是它的默认状态是 BASIC 类型（不包含默认状态）。该类状态的执行过程如算法3.4所示。伪代码的 1-3 行是常规操作。如果 DEFAULT_BASIC 状态取出的指令 Instruction1 通过了指纹检测，那么该类型状态的后续操作和 BASIC 类型一致，直接将 Instruction1 编码的状态写回 NSQ。如果指纹检测失败，LAP 将从 DEFAULT_BASIC 状态的默认状态中查找相应的边（关于 ADFA 模型，请参考第2.1.5小节）。由于 DEFAULT_BASIC 状态的默认状态一定是 BASIC 类型，我们可以直接利用该 BASIC 状态的 Target 域和当前输入字符计算出新地址重新进行新指令的读取。如代码的第 7 行所示，DEFAULT_BASIC 类型在被编码时，它的 Attach 域存储的就是其默认状态的 Target 域。因此，第 7 行代码可以在 DEFAULT_BASIC 类型状态取得拥有错误指纹的指令后，进行第二次指令获取。最后，我们将第二次访存获得的有效状态写回 NSQ。

第二类新状态为 DEFAULT_CASCADE 类型。DEFAULT_CASCADE 状态是 DEFAULT_BASIC 状态的推广。理论上，我么可以用 DEFAULT_CASCADE 实现 DEFAULT_BASIC，但是性能上将有所降低。DEFAULT_CASCADE 状态和 DEFAULT_BASIC 状态在自动机状态转换图上直观的区别是，DEFAULT_CASCADE 状态的默认状态本身也一定拥有一个默认状态。也就是说，假设 S_0 属于 DE-

算法 3.4 DEFAULT_BASIC 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==DEFAULT_BASIC);
3 Intruction1 = Mem[current_state.Target + input_char];
4 if signature_check(input_char,Intruction1.signature)==True then
5   |   NSQ.push(Intruction1);
6 else
7   |   Intruction2 = Mem[current_state.Attach + input_char];
8   |   NSQ.push(Intruction2);
9 end

```

FAULT_CASCADE 类型，那么一定存在一个 S_1 是 S_0 的默认状态，而且还存在一个 S_2 是 S_1 的默认状态。DEFAULT_CASCADE 类型状态的执行过程如算法3.5所示。该算法和3.4的执行过程类似，二者区别主要在 7-8 行伪代码。由于在执行 DEFAULT_CASCADE 状态时，我们无法得知其默认状态的类型。我们需要首先读取一条指令，该指令编码了其默认状态的信息。随后，我们直接执行该默认状态。如果该默认状态还是 DEFAULT_CASCADE 状态，那么重新调用该算法。通过这种方式，我们可以实现 ADFA 模型中的连续的 default transition。

算法 3.5 DEFAULT_CASCADE 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==DEFAULT_CASCADE);
3 Intruction1 = Mem[current_state.Target + input_char];
4 if signature_check(input_char,Intruction1.signature)==True then
5   |   NSQ.push(Intruction1);
6 else
7   |   Intruction2 = Mem[current_state.Attach];
8   |   Execute(Intruction2);
9 end

```

4. 其他指令

ADFA 模型中的 default transition 可以消除状态间的冗余边，而 majority transition 则是为了消除每个状态内部的冗余边。正则表达式支持字符范围的识别，比如 [a-z] 可以接受“a”到“z”之间 26 个字符中的任意一个，而 [^a] 接受除了“a”以外的任意字符。因此，如果实现为 BASIC 类型转态，那么该状态有 26 条出边 ([a-z]) 或者 255 条出边 ([^a])。但是这些边都有一个相同的目标状态，被称为 MAJORITY 状态。因此，我们引入 MAJORITY 类型的状态。该类状态拥有一个 majority 边，指向他们的 MAJORITY 状态，也就是大部分情况下会切换到的

目标状态。如算法3.6所示，代码 1-5 行和 BASIC 类型一致。当 MAJORITY 状态根据当前输入字符取得的是有效状态（指纹检测通过），则将该状态写入 NSQ。但是，如果取得的是无效指令时，LAP 则将 MAJORITY 状态写入 NSQ（如代码 7-8 行所示）。通过支持 MAJORITY 状态，可以实现更紧致的自动机模型，进一步减少自动机模型对存储器的空间占用。

算法 3.6 MAJORITY 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==MAJORITY);
3 Instruction1 = Mem[current_state.Target + input_char];
4 if signature_check(input_char,Instruction1.signature)==True then
5   | NSQ.push(Instruction1);
6 else
7   | Instruction2 = Mem[current_state.Attach];
8   | NSQ.push(Instruction2);
9 end

```

3.3 LAP 指令集优化

在本节，我们首先分析 ADFA 模型在模型压缩方面展现出的显著效果。随后，我们分析使用 ADFA 模型后模式匹配的性能下降的根本原因。最终，我们结合 ADFA 模型的特点，提出了两条新的机器指令。使用这两条指令，我们可以实现更高效的 ADFA 的执行。在这一过程中，我们改变了原本的 ADFA 的执行算法，同时也改变了底层硬件的设计。采用这样一种软硬件协同的设计方式，我们将可以实现更快的基于 ADFA 模型的模式匹配计算。

3.3.1 ADFA 对自动机模型的高效压缩

第2.1.5小节详细介绍了 ADFA 模型。图3.3则展示了接受相同正则表达式集合的 DFA 模型与 ADFA 模型。ADFA 模型通过增量存储的方式消除了大量的边，从而极大地压缩了模型的体积。从图3.3(a)中的红线和蓝线我们可以发现，S3 和 S1 在接受相同的输入“a”，“b”或者“d”时，会跳转到相同的状态。这就是 ADFA 模型试图利用的“状态间的冗余性”。在图3.3(b)中，S3 中添加了一条新的 default edge 指向 S1（通过虚线表示），同时删除了 S3 中的冗余边。

在 S3 中引入 default edge，这对于 DFA 模型的大小的影响还可以从图3.4中看出。该图清晰地展示了在引入 default edge 前后 Transition Table（状态转移表）的变化。此处先解释下，对于模式匹配任务，待匹配的模式通常是以正则表达式

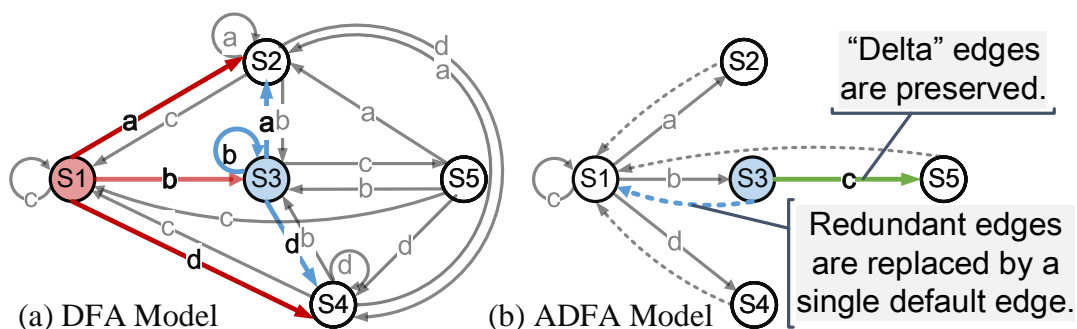


图 3.3 DFA 与等价的 ADFA 模型的状态转换图

的形式作为输入。这些正则表达式会被转换为对应的自动机模型：不同的正则表达式会产生不同状态数目和连接关系的自动机模型。无论是基于通用处理器还是专用处理器，我们都需要存储该自动机模型。本文使用类似状态转移表的形式，存储自动机状态和状态间的连接关系。因此，状态转移表的大小就是我们实际在存储自动机模型时需要的存储器大小。可以从图3.4中看到，假设输入字符集合为 $\Sigma = \{a, b, c, d\}$ ，那么为了存储状态 S1 和 S3 我们总共需要存储 8 个表项。S3 和 S1 之间边的冗余性可以从他们的状态转移表中直观的看出：S3 和 S1 的状态转移表的第 1、第 2 和第 4 行的内容是一样的。在引入 ADFA 模型的 default edge 后，S3 中 3/4 的表项可以被删除。事实上，实际的输入字符集合通常是 ASCII 码，也就是说 $|\Sigma| = 256$ 。此时每个状态有 256 个表项，default edge 的引入可以获得更大的收益。

	S1's Transition Table	Default Edge	S3's Transition Table
'a'	S2	←	S2
'b'	S3		S3
'c'	S1		S5
'd'	S4		S4

图 3.4 ADFA 模型对状态转换表的压缩

3.3.2 ADFA 降低模式匹配速度的根本原因

ADFA 模型也引入了更复杂的运行时操作（状态回退），从而导致了性能的损失。在 DFA 模型中，为了处理一个输入字符，我们只需要对状态跳转表进行一次内存访问。每当处理一个新的输入字符时，我们从当前状态的跳转表中读取一个表项，该表项编码了下一个状态的信息。然而对于 ADFA 模型来说，情况却有所不同。ADFA 模型使用了增量存储，每个状态的冗余边被去除了。这引进了一个新的计算需求：我们需要在运行时进行“fall-back”（状态回退）。一旦当前

状态需要得知一条已经被去除的冗余边的目的地，我们需要从当前状态的默认状态处获得这样的信息。因此，当前状态需要首先回退到它的默认状态，并且基于这个新状态（其默认状态）重新处理当前输入字符。比如，我们假设图3.3中的 S3 是当前状态，而当前输入字符是“b”。我们可以发现，(b) 图中 S3 并没有一条标签为“b”的有效边。这是因为，这条边作为冗余的边被删除了。因此，在实际执行时，S3 需要首先回退到它的默认状态 S1。随后，我们试图从 S1 的状态转移表中查找我们需要的边。这需要引入额外的对指令存储器的访问，因为我们需要额外访问 S1 的状态转移表。如果出现了多次连续的状态回退，我们可能需要更多的指令存储器访问。因此，额外的存储器访问被“fall-backs”引发。这降低了已有硬件方案基于 ADFA 模型的模式匹配速度。

3.3.3 ADFA 降低模式匹配速度的优化思路

基于以上分析和对 ADFA 模型的拓扑关系的分析，我们得出以下的核心优化思路。既然状态回退会降低 ADFA 的处理速度，那么我们如何更快的实现状态回退呢？图3.5展示了 ADFA 模型在实际的工作负载下的典型拓扑结构。相比于之前展示 ADFA 模型（如图3.3），图3.5中的 ADFA 结构更符合实际的工作负载。（该图为矢量图，可能需要一定时间加载。加载完成之前，图片可能显得比较模糊；加载完成后，图3.5会很清晰。）该图展示的 ADFA 模型接受以下的正则表达式：

- Vm6qZ
- rpdP(T0)?
- rgtPL
- vbLf
- 2w(HRQ6|UX7j|S9v3)
- tWzwOL

图3.5需要从两个层次来观察。首先，我们观察由黑色的边（labeled transition）和各状态节点组成的类似树状结构。该树状结构的根节点是 ADFA 模型的初始节点。值得一提的是，如果按照本文的参数（见第3.5小节）进行 ADFA 模型构建，则有且只有一个根节点。从根节点出发，如果当前输入字符序列符合某个我们预期的正则表达式，那么当前状态会沿着该树状结构一直往下走（随着字符的不断输入，当前状态会不断的切换）。如果走到了最底部的双圆圈节点（接受态），则说明发现了符合一到多个正则表达式的输入子序列。同时，图3.5中绿色和蓝色的边代表着 ADFA 模型中的 default transition。这些边和状态节点共同组成了 ADFA 模型的第二个层次。在当前状态从初始节点沿着某个路径一直往下走时，可能在某个地方突然不再匹配正在匹配的某个模式。那么此时，当前状态

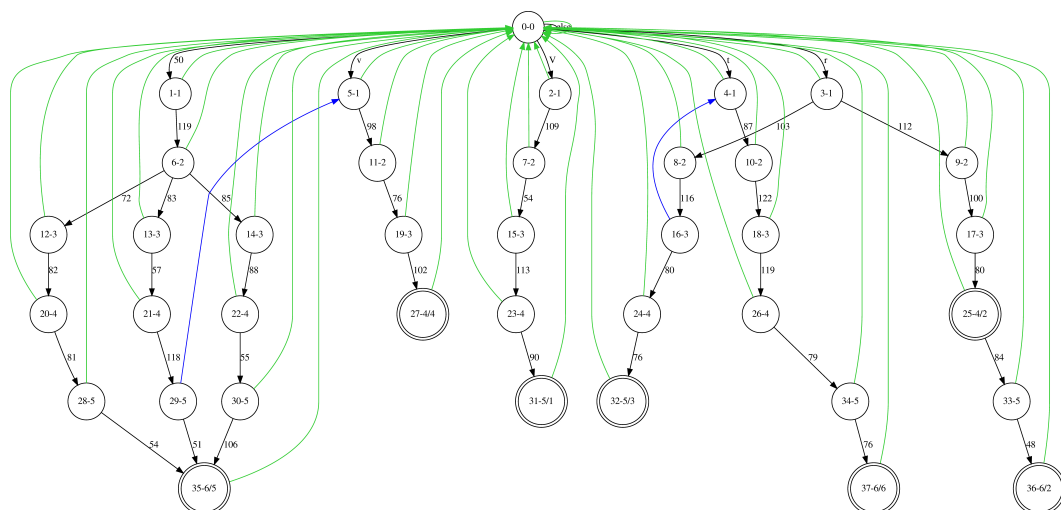


图 3.5 ADFA 模型的典型拓扑结构

在当前输入下不再有有效的出边。随后，该状态将会沿着绿色或者蓝色的边进行状态回退。值得注意的是，沿着这种边进行回退时，当前输入字符是不被消耗的。这也正是，状态回退会导致字符处理速度下降的根本原因。

此时，我们可以有一个直观感受：ADFA 模型在运行时，如果当前输入字符符合当前正在匹配的若干模式，当前状态会向下走，会走的更深。严谨来说，就是当前状态的深度（距离初始状态的最短距离）会随着匹配的过程不断增加。如果当前字符导致匹配到一半的模式发生了不匹配，那么当前状态会回退到深度更浅的地方。之所以回退过程中当前状态的深度一定会降低，是因为第 2.1.5 小节介绍过 ADFA 模型的一个约定：ADFA 模型中的 default edge 只会从深度更深的节点指向深度更浅的节点。

在图 3.5 中，我们将 default edge 划分成两类。第一类 default edge 我们使用绿色的边表示，这些边直接指向 ADFA 的初始状态。第二类 default edge 我们使用蓝色的边表示，这些边指向非初始节点。值得一提的是，更大规模的更贴近现实的正则表达式集合对应的 ADFA 模型的基本结构和图 3.5 类似，但是第二类 default edge 的占比会相应提高。通过分析 ADFA 模型的状态间的拓扑结构，我们可以发现每个状态很容易发生状态回退。这是因为在当前状态下，当前输入必须等于当前状态下存在的某条出边的 label。而一旦发生第一类回退，当前状态会立即回退到初始状态。而如果发生第二类回退，也有可能最终连续回退到初始状态。因此，优化针对 ADFA 模型中初始状态的回退（第一类回退）成为最关键的优化方向。同时，第二类回退随着正则表达式规模的增加比例也会增加。因此，如何高效的进行第二类状态回退也是一个值得关注的优化方向。

3.3.4 并行化的 ADFA 执行算法与新的机器指令

第3.3.2小节已经分析了 ADFA 模型性能下降的根本原因：运行时要进行状态回退。同时，第3.3.3小节也分析了我们应该关注的两个优化方向：优化第一类状态回退和第二类状态回退。本节将从指令集和硬件执行模型的角度提出具体的优化方案。

按照旧的硬件执行模型（见第3.1小节），一个硬件周期最多可以进行一次对指令存储器的读取。同时，ADFA 模型的第一类回退是通过前文介绍过的 `DEFAULT_BASIC`（算法3.4）状态类型实现，而第二类回退是通过前文介绍过的 `DEFAULT_CASCADE` 状态类型（算法3.5）实现。通过分析这两个算法我们可以发现，他们都需要进行多次指令存储器访问。因此，为了实现更快的状态回退，我们需要尽可能并行化这些对指令存储器的访问。为了实现这个目的，我们需要做两件事：第一，拓展原本的硬件执行模型，使得指令存储器支持并行的读取（一个周期读取多条指令）；第二，优化原本的 ADFA 执行算法，使得其内部的存储器访问和其他操作尽可能并行化。

1. 针对第一类状态回退的优化指令：`DEFAULT_BASIC_OPT`

我们已经观察到，ADFA 的初始状态的状态转换表会被频繁地访问，无论是因为正常的状态切换还是因为状态回退。因此，我们此处试图通过并行访问的方式掩盖对初始状态的状态转换表的访问开销。图3.6左侧展示了未优化的 `DEFAULT_BASIC` 状态类型的执行流程。前文中算法3.4描述了该过程。如蓝色线条所示，一条 LAP 指令在第一个周期被读取。如果这条指令通过了指纹检测，它将会被用于更新当前状态。但如果指纹检测失败了，则需要额外进行一次针对 ADFA 初始状态的状态转换表的内存访问。比如，假设当前状态是图3.3中的 S3 而当前输入字符是“b”。那么 LAP 会首先将当前状态回退到状态 S1，然后在第二个硬件周期从 S1 的状态转换表中读取我们需要的那条被标为“b”的边。总的来说，对第二条指令的读取是可选的，这取决于第一条指令的指纹检测是否通过。这一依赖使得我们只能串行处理这两次内存访问。

幸运的是，这样的依赖是可以被打破的。如图3.6右侧所示，我们拓展了原本的硬件执行模型。在原本的主要的指令存储器之外，我们增加了一个很小的辅助存储器（Auxiliary Memory）。从而，我们每个硬件周期可以从指令存储器和辅助存储器中各自读取一条指令。`DEFAULT_BASIC_OPT` 状态并不会可选性的启动第二次存储器访问。相反的，优化后的指令并行的读取这两条指令，而不考虑第二条指令是否会用得上。由于该存储器很小，因此带来的额外开销很小。随后，LAP 根据对第一条指令的指纹检测结果选择这两条指令中的一条进行后续的处理。通过这种方式，这两条指令的读取被并行化，原本需要两个硬件周期才能完

成的第一类状态回退被优化为了 1 个硬件周期。

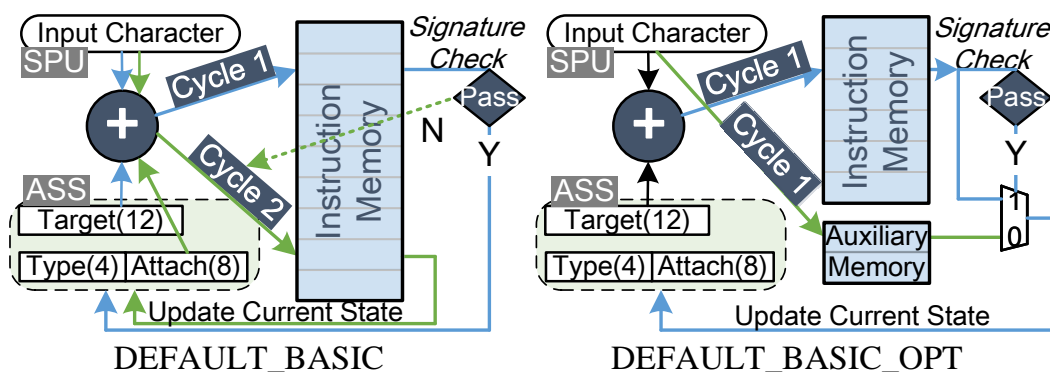


图 3.6 第一类状态回退的优化方案

对应的，DEFAULT_BASIC_OPT 的执行算法如算法 3.7 所示。

算法 3.7 DEFAULT_BASIC_OPT 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==DEFAULT_BASIC_OPT);
3 ///////////////Parallel Memory Access Begin//////////////////;
4 Intruction1 = TransitionMem[current_state.Target + input_char];
5 Intruction2 = AuxiliaryMem[256 + input_char];
6 ///////////////Parallel Memory Access End//////////////////;
7 if signature_check(input_char,Intruction1.signature)==True then
8 |   NSQ.push(Intruction1);
9 else
10 |  NSQ.push(Intruction2);
11 end
    
```

2. 针对第二类状态回退的优化指令：DEFAULT_CASCADE_OPT

第二类回退的执行流程与第一类回退截然不同。第二类回退的目标状态都有其自己的 default state。因此，这些状态的冗余边已经被 default edge 替代而且他们的状态转移表是不完整的。如果我们仍然使用第一类状态回退的优化方式，并行的访问他们状态转移表是没有意义的。因为，从它们的状态转移表中读取的指令可能仍然是无效指令。

但是，DEFAULT_CASCADE 状态类型也是可以优化的。不同于第一类状态回退，第二类状态回退的目标状态的状态类型是未知的。目标状态的类型可以是仍然是 DEFAULT_CASCADE 状态类型也可以是 DEFAULT_BASIC。因此，在我们回退到目标状态前，我们需要获得目标状态的完整信息（Target、Type 和 Attach）。这么多的信息是无法存储在当前状态的 attach 域的。因此，我们使用一条完整的 LAP 指令（本文称之为“附属指令”）去存储第二类回退的目标状态

的完整信息，而当前状态的 `attach` 域则被用于去寻址这一条指令。因此，如果读取的第一条指令的指纹是错误的，我们则需要读取第二条指令。也就是说，第二条指令得读取是可选的（取决于第一条指令的 `signature`），而且第二条指令的读取发生在第二个机器周期。图3.7左侧展示了未优化前第二类状态回退的执行流程，该过程在算法3.5中也有所描述。我们观察到，附属指令通常在前一条指令（被附属的指令）执行结束后立即被读取。因此，我们把附属指令与其他指令分离开，并存放在辅助存储器中。通过这样的方式，我们可以在前一条指令还在访问主指令存储器的同时从辅助存储器中读取该条指令的附属指令。如图3.7右侧所示，这项优化通过 LAP 指令 `DEFAULT_CASCADE_OPT` 实现。通过利用辅助存储器的空闲带宽，我们掩盖了读取附属指令的存储器访问开销。通过这样的方式，我们将原本两个周期才能完成的第二类状态回退优化为 1 个周期完成。值得一提的是，虽然第二类状态回退只需要一个周期即可完成，但是这个周期是不会消耗任何输入字符的。因此，第二类状态回退仍然会使得 ADFA 模型的执行速度比不上 DFA 模型。

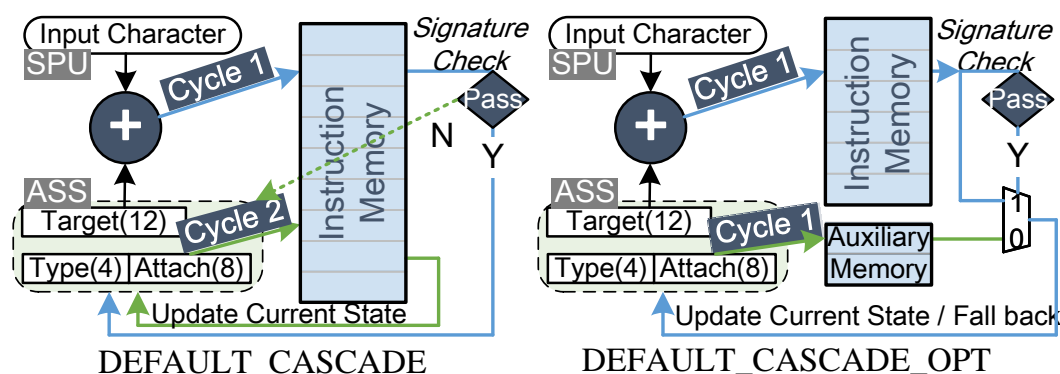


图 3.7 第二类状态回退的优化方案

对应的，`DEFAULT_CASCADE_OPT` 的执行算法如算法3.8所示。

3.4 LAP 的微体系结构

基于第3.2.1小节介绍的硬件执行模型和优化 AFDA 时进行的硬件执行模型的拓展（3.3.3），本节将详细介绍 LAP 的微体系结构。我们将首先介绍 LAP 的核心部件和流水线设计。随后，我们将介绍细粒度多线程技术在 LAP 中的应用。

3.4.1 四级流水线架构

正如前文2.1所介绍的，自动机的执行开始于单个活跃状态 q_0 。在我们的实现方案中，每一个状态 $q \in Q$ 被赋予了一个 12 比特的状态编码。同时，每一个输入字符 $ain\Sigma$ 是一个 8 比特的 ASCII 码字符。随着第一个字符输入自动机， q_0

算法 3.8 DEFAULT_CASCADE_OPT 状态的执行算法

```

1 current_state = CSQ.pop();
2 assert(current_state.Type==DEFAULT_CASCADE);
3 ///////////////Parallel Memory Access Begin//////////////////;
4 Intruction1 = TransitionMem[current_state.Target + input_char];
5 Intruction2 = AuxiliaryMem[current_state.Attach];
6 ///////////////Parallel Memory Access End//////////////////;
7 if signature_check(input_char,Intruction1.signature)==True then
8   |   NSQ.push(Intruction1);
9 else
10  |   Execute(Intruction2);
11 end

```

根据状态转移函数 $\delta(q, \alpha)$ 切换为 1 或者多个活跃状态。自此之后，每当有一个字符被输入自动机，每一个活跃状态根据状态转移函数 $\delta(q, \alpha)$ 独立地切换到另外一组新的状态。为了在硬件上运行自动机模型，我们需要用硬件记录在整个自动机执行过程中每一时刻的所有活跃状态，同时周期性的进行状态的切换。

LAP 包含以下核心部件：

- 在 LAP 中，我们把所有的自动机活跃状态存储在**活跃状态栈 (Active State Stack, ASS)** 中，并且动态的更新 ASS 中存储的内容。为了支持 ADFA、DFA 和 NFA 模型，ASS 可以同时存储多个活跃状态并且串行的输出这些状态。因此，NFA 中每一个活跃状态的状态切换过程可以轮流执行。
- 输入流预取部件 (Stream Prefetch Unit, SPU) 的作用是从输入字符缓存 (Input Buffer) 中加载输入数据。LAP 的输入数据是一串字符流，其中每个字符都定义在 Σ 中。SPU 串行的从字符流中顺序的输出每个字符；而且 SPU 只会在当前字符已经被彻底被自动机处理掉后才会输出新的字符。输出一个字符后，SPU 内部的指向数据流的指针会自增。
- 主指令存储器和辅助存储器 (Instruction Memory and Auxiliary Memory) 存储 LAP 的二进制程序。LAP 的二进制程序由前文提到的 LAP 指令集 (3.2.2) 构成。这两个基于 SRAM 的存储器存储着整个自动机模型的状态节点和节点间的拓扑关系，从而描述了自动机模型的整体行为。值得一提的是，LAP 的二进制程序是根据我们要匹配的正则表达式集合和要使用的自动机模型生成的。当要匹配的模式发生改变时，我们需要重新编译 LAP 的二进制程序并写入这两个存储器。
- 指令译码器 (Instruction Decoder) 是 LAP 的主控制器。该控制器是纯粹的组合逻辑，负责收集其他部件的输出信息去更新 ASS 的内容或者去汇报被激活的接受态 $q \in F$ 。

为了在 FPGA 和 ASIC 平台上实现较高的硬件主频，我们为 LAP 设计了四级的流水线架构。图3.8展示了 LAP 简化的硬件结构。如果所示，LAP 的数据通路被分割成了 4 个阶段。

- 第一阶段，ASS 输出一个活跃状态（当前状态）；同时，SPU 输出一个字符（当前输入字符）。
- 第二阶段，LAP 根据当前状态和当前输入计算出两个物理地址。通过编译器的精心布局，要读取的指令的地址可以通过很简单的操作（图3.8中所示的整数加法）计算出来。
- 第三阶段，两条指令被并行的读取。
- 第四阶段，指令译码器更新 ASS 并且向外部汇报当前状态的编码（如果当前状态 q 是接受态，即 $q \in F$ ）。

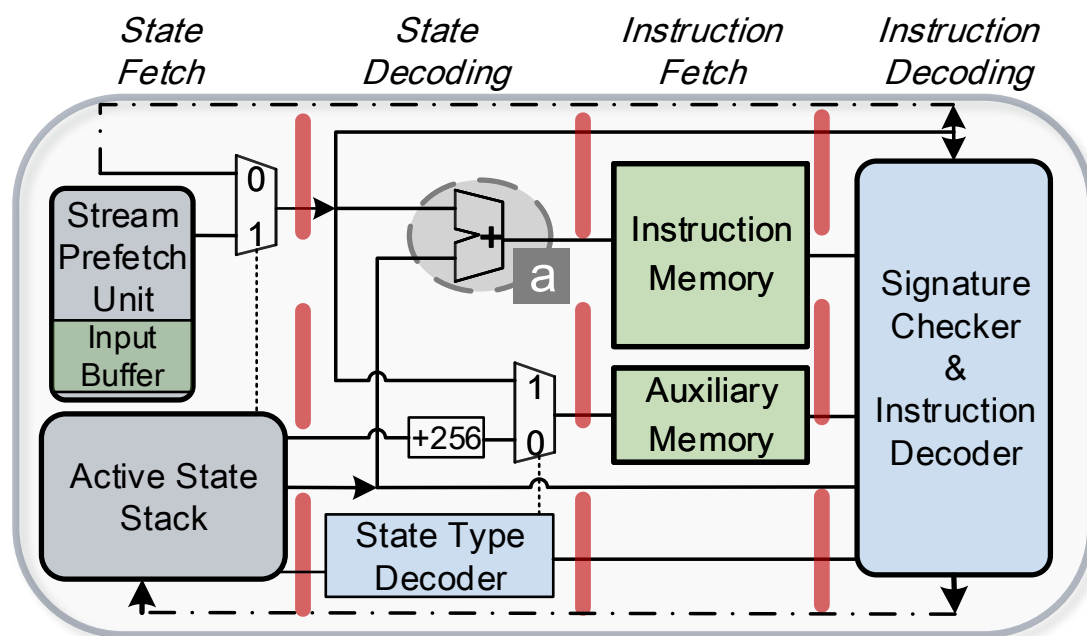


图 3.8 LAP 的四级流水线架构

通过实验证明，该流水线的设计和流水级的划分是合理的。我们在 FPGA 平台上综合实现 LAP 的时候，LAP 的每个阶段的电路延迟是均衡的。

3.4.2 细粒度多线程

我们将细粒度多线程技术^[45]运用到了 LAP 的微架构中，从而消除了 LAP 流水线中的相关性。通过在 LAP 的流水线中填充四个完全不相关的线程，LAP 的流水线不会出现任何停顿，也实现了最大化的流水线吞吐。图3.9是 LAP 在多线程模式下工作时的硬件部件时空图。其中，纵轴是 LAP 硬件流水线的四个阶段，横轴是当前所处的机器周期编号。从图中可以看出，LAP 的四个流水级可以填充 4 个不同的线程。这四条线程的指令轮流从 SF 阶段被发射。比如，第 0

个周期我们发射了线程 1 的指令，第 1 个周期则发射线程 2 的指令。如图 3.9 中虚线所示，每个线程内部的指令都有数据和控制相关。然而，由于相同线程中的指令每 4 个周期才能发射一次。相同线程的指令都相隔了 4 个周期发射，因此他们的数据和控制相关不会引起任何问题（后一条指令会在前一条指令完成后再发射）。因此，通过细粒度多线程技术，我们避免了复杂的流水线冲突处理。同时，不同线程的指令之间不存在任何数据和控制相关。因此，LAP 的流水线是 Stall-Free（无停顿）的。值得注意的是，在 LAP 中每一个线程都有自己独享的输入字符流。这意味着，LAP 每个线程独立的处理一个输入字符流。另一方面，LAP 所有的线程需要共享同一个 LAP 二进制程序。因此，LAP 将同时从四个输入流中查找相同的一组预定义模式（正则表达式集合）。

为了使得 LAP 支持细粒度多线程，我们在 SF 阶段实现了一个 2 比特的硬件计数器。该计数器每个硬件周期自动加一。并且，该计数器的值（我们称之为 Context ID, CID）随着硬件流水线一级级向后传递。那么，LAP 的每一个流水级都有他们自己的 CID。在第一个流水级中，我们使用 CID 来选择要读取的输入字符流。同时，我们在 ASS 中存储了 4 个线程各自的机器状态，使用 CID 选择其中一组状态输出。总的来说，细粒度多线程技术可以显著地提高流水线的利用率，减少甚至避免流水线停顿。它带来的弊端则是，使得处理器单线程处理速度下降。然而，LAP 是一个面向高吞吐的计算负载而并不追求单线程计算能力。同时，多模式匹配的任务可以很容易的切分成多个部分并分配给不同的线程。因此，细粒度多线程成为了 LAP 用于解决流水线冲突的极佳选择。

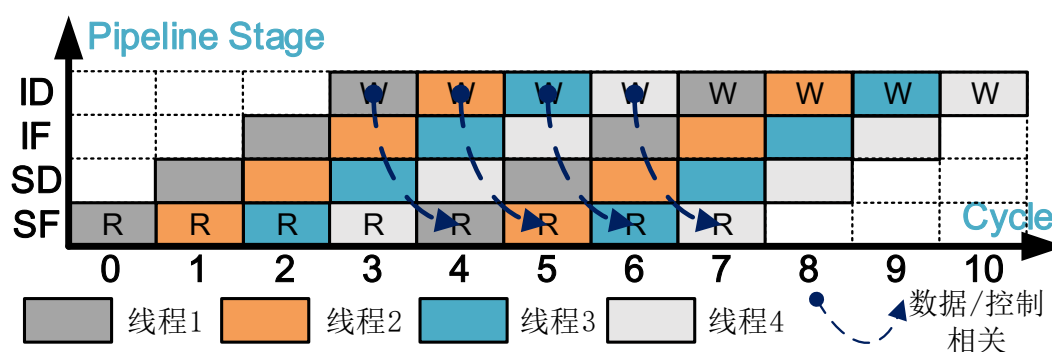


图 3.9 细粒度多线程技术在 LAP 中的应用

3.5 编译算法设计与实现

为了在 LAP 上运行模式匹配任务，我们设计并实现了相应的编译软件。我们改编了开源的 C++ 软件^[46]用于将正则表达式编译成特定的自动机模型（一阶段编译）。同时，我们用 Python 语言开发了二阶段编译器。二阶段编译器可以根

据第一阶段生成的自动机模型，进一步生成 LAP 二进制代码。

3.5.1 将正则表达式集合转化为等价自动机模型

在本文，正则表达式可以被转换为 ADFA 模型或者 NFA 模型。图3.10就为这样的转换过程举了一个例子，并且层次化的展示了生成的 ADFA 模型。图中 *labeled edges* 用于表示指纹检测通过时会进行的状态转换，而 *default edges* 则是在指纹检测失败时触发。值得一提的是，我们在生成 ADFA 模型时，将开源软件^[46]中的一个重要参数“深度约束”（Depth Bound, DB）设置为2。”DB”这个参数是用于在构建 ADFA 模型的时候约束最长的 *default* 路径，也就是图3.10中 *hierarchy-2* 的最长路径不能大于“DB”的值。通过这样的方式，我们可以保证每个状态最多经历两次状态回退就可以到达初始状态。更重要的是，这可以将“附属指令”（见第3.3.3小节）的数目约束在一个较小的值，从而保证“附属指令”只会占用辅助存储器很少的存储空间。

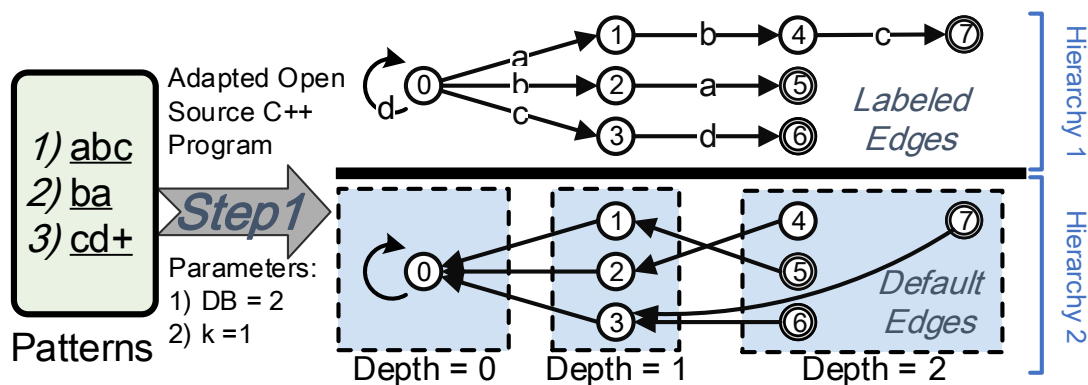


图 3.10 ADFA 模型的层次化结构

3.5.2 自动机模型的 Split EffCLiP 二进制表示

在 LAP 中，所有状态各自的状态转换表都利用高效的线性打包算法 (*Efficient Coupled-Linear Packing algorithm, EffCLiP*^[47]) 重叠的存储在 SRAM 中。通过这种重叠存储的方式，可以实现非常紧凑的自动机模型的存储。同时，为了支持第3.3.3小节提出的优化方案，本文提出 *Split EffCLiP* 的表示方式，用于将自动机模型紧凑的存储在两个分离的 SRAM 存储器中。如图3.11所示，图3.10中的 *labeled edge* 都被 *EffCLiP* 算法非常紧凑的存储在线性的地址空间 (Instruction Memory) 中。同时，为了支持对初始状态的查找表的并行访问，我们将初始状态的查找表 (图中称为“Initial Table”) 存储在辅助存储器中。不仅如此，为了支持对“附属指令”的并行访问，我们也将这些附属指令分离出来并存储在辅助存储器中。

此处我们介绍下 *Split EffCLiP* 的二进制表示下，LAP 是如何进行访存和计算的。如图3.11所示，图的左边是使用 *EffCLiP* 重叠存储的各个自动状态的跳转

表；而图的右边则是平铺存储的 Initial State 的跳转表和用于编码 default state 完整信息的指令。该图存储了图3.10中所示的自动机模型。值得注意的是，我们为图3.10中的每个状态 S0-S7 赋予了一个新的状态编号。比如图中，S0 在 Split EffCLiP 中的编号为 0，S3 的编号为 1。

假设当前状态是图3.10中的 S3，那么当前状态的 Target 域值则为其编号 1。如果当前输入是 d，那么我们可以计算出下一条指令的地址为 1+'d'（使用'd'的 ASCII 码值来做整数加法）。图中为了简化（否则我们需要画出有几十个表项的 Instruction Memory），我们把 S3 的下一个状态存放在地址为 (1+'d')-'a' 的位置，也就是 1+3 的位置处。据此，我们取出的指令为“Target:4, Type:Default_CASCADE_OPT,Attach:0”。事实上，该指令还包含了 Signature 域。同时，该指令的 Signature 一定是'd'，也就是当前输入字符，因此可以通过 signature check。因此，以上过程中，我们完成了从 S3 跳转到 S6（Target: 4 等价于 S6）。

在 S6 状态（“Target:4, Type:Default_CASCADE_OPT,Attach:0”）下，假设当前输入字符是'a'。那么，下一条指令地址则为 4（4+'a'-'a'）。而事实上，S6 状态在输入为'a'时没有出边。因此，尽管我们可以根据该地址取出一条指令。该指令的 signature 域却是'd'，不等于当前输入字符'a'。因此，指纹检测失败。随后，检测当前状态 S6 的属性。由于 S6 是 Default_CASCADE_OPT 类型，因此我们启动第二类状态回退（如3.3.4所介绍的）。根据当前状态的 Attach 域，我们从 Auxiliary Memory 中取出当前状态的 Default State 的完整编码（也就是 Target: 1, Type: Default_BASIC_Opt, 对应了状态 S3）。在这样的过程中，我们实现了从状态 S6 到状态 S3 的第二类状态回退。在 LAP 中，这两次内存访问是并行的。

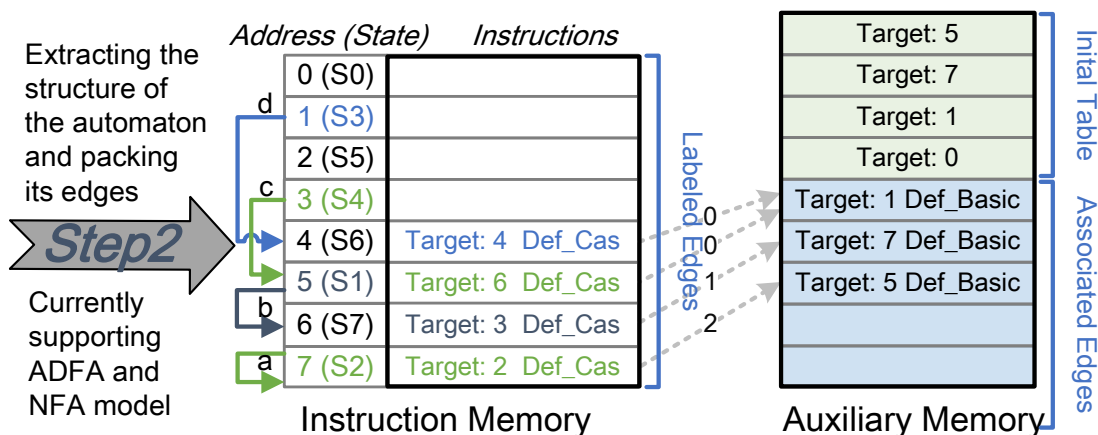


图 3.11 ADFA 模型的 Split EffCLiP 表示

3.6 实验与评估

我们使用 Verilog 硬件描述语言实现了 LAP，并使用 Vivado 设计套件 2017 将 LAP 的原型以 263MHz 的主频部署（综合和实现）到了 Xilinx Artix-7 FPGA 上。图3.12展示了由 Vivado 集成开发工具自动生成的 LAP 原理图（Schematic）。图中显示了 LAP 的各个核心模块和模块接口，以及这些模块之间的电路连线。值得注意的是，图中不仅仅包括前文介绍的 ASS、SPU、Instruction Memory 和 Auxiliary Memory 等部件，还包括为了实现硬件流水线而设置的四级段寄存器。其中，主指令存储器（16KB）和辅助存储器（0.6KB）是通过 FPGA 上的 BRAM 资源实现的。总的来说，每一个 LAP 核心占用了 724 个 LUT（7%）、2199 个 FF（11%）和 5 个 BRAM（20%），总体的动态功耗为 0.12W。

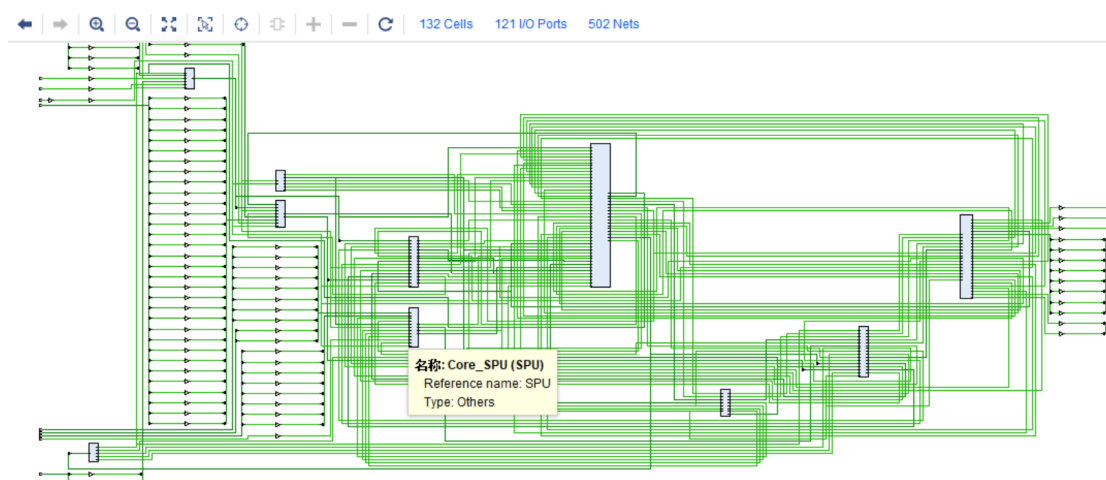


图 3.12 LAP 在 Artix-7 FPGA 上的 Schematic

本文使用两类标准的工作负载用于 LAP 的评估。PowerEN simple^[14] 是一系列标准数据集。它最早被 IBM 公司用于评估 PowerEN 处理器内部的正则表达式硬件加速器的性能。另外，Exact-Match(EM) 和 range1 是通过软件工具^[46] 合成的工作负载。我们使用 line rate 这一指标来衡量 LAP 的处理速度，并使用 pattern density 去衡量 LAP 的存储效率。Line rate (symbol/cycle) 指示了每个 LAP 核心平均每个硬件周期可以处理的输入字符的数目。另一方面，pattern density(#patterns/KB) 是将待匹配的正则表达式的数目除以对应的自动机占用的存储器空间计算得来的。我们使用自己实现的编译软件将这些 workload 编译为对应的 LAP 二进制程序。随后，我们使用 Vivado Simulator 仿真这些程序在 LAP 上的执行过程。我们此处使用的 Vivado Simulator 是 Xilinx 官方提供的精确到每个硬件周期的仿真器，因此实验结果非常可信。总的来说，处理每个 workload 所用的硬件周期总数是通过观察 Vivado Simulator 的仿真波形图得出的数据，而 pattern density 则是由我们设计实现的编译器在编译过程中计算出来的。

在展示 LAP 的各项性能指标时，我们对比了比较知名的相关工作。这些工作包括：芝加哥大学的 UAP^[27]、IBM 的 RegX^[14] 和 Micron 公司的 Automata Processor(AP)^[15]。在评估 LAP 的处理速度时，我们比较的是 LAP、UAP 和 RegX 的单核心性能。LAP 单核心的硬件复杂度和 UAP 单核心类似，而且要比 RegX 单核心更轻量化。因此，比较它们的单核心性能是公平的。同时，我们选择了 UAP^[27] 一文中使用过的评价指标 (line rate) 来评价三款处理器的性能。我们使用了 UAP^[27] 一文中的实验结果和 RegX 处理器的理论最佳性能 (无 Cache Miss) 作为 LAP 的对照数据。值得注意的是，line rate 这一评价指标评估的是一个硬件时钟周期内的平均吞吐，与具体的硬件平台无关。这使得本文提出的基于 FPGA 的 LAP 可以与基于 ASIC 平台的 UAP 和 RegX 进行公平的平台无关的比较。在评估 LAP 的二进制代码存储效率时，我们使用了 UAP^[27] 一文中**相同的数据集和评价指标**进行评估。我们也从这篇文章中得知了 RegX 和 UAP 的存储效率。Micron 公司的 AP 采用的是独热码的编码方式：每个 NFA 状态需要占用 256 比特。因此，我们将 simple 400 正则表达式集合编译为对应的 NFA，并获取了生成的 NFA 的状态总数。随后，我们通过二者之间的乘法 (256 bits * NFA 状态数) 计算出了 AP 的存储器空间占用。

3.6.1 LAP 处理速度总览

图3.13展示了 LAP 在不同的 workload 上整体的处理速度 (line rate)。虽然我们的工作主要是为了实现更快的基于 ADFA 的计算，为了实现更为全面的性能评估，我们此处也评估了 LAP 基于 NFA 模型的性能。除此以外，UAP^[27] 的性能也被包含在图中。LAP 和 UAP 使用类似的指令集，因此与 UAP 做直接的性能对比可以公平的展示本文提出的软硬件协同设计的有效性。如图3.13所示，LAP 使用 NFA 模型时可以实现和 UAP 差不多的处理速度。(LAP 平均每个机器周期可以处理 0.49 到 0.66 个输入字符。) 当使用 ADFA 模型时，LAP 在不同的工作负载上实现了 0.91 到 0.97 的 line rate。与之对应的，“ADFA 模型是一个更复杂的模型，因此每次状态转化需要更多顺序的存储器访问，因此产生了较低的 line rate，平均每个周期处理 0.47 到 0.75 个字符” (UAP^[27],2015)。总的来说，LAP 在基于 ADFA 模型的模式匹配方面取得了显著的性能提升：相比于 UAP，在各个数据集上实现了 32% 到 91% 不等的速度提升。理论上，装备 5 个 LAP 核心的 FPGA 原型系统基于 ADFA 模型可以实现 9.5Gbps 的处理吞吐。

图3.13中的原始数据如表格3.2所示。

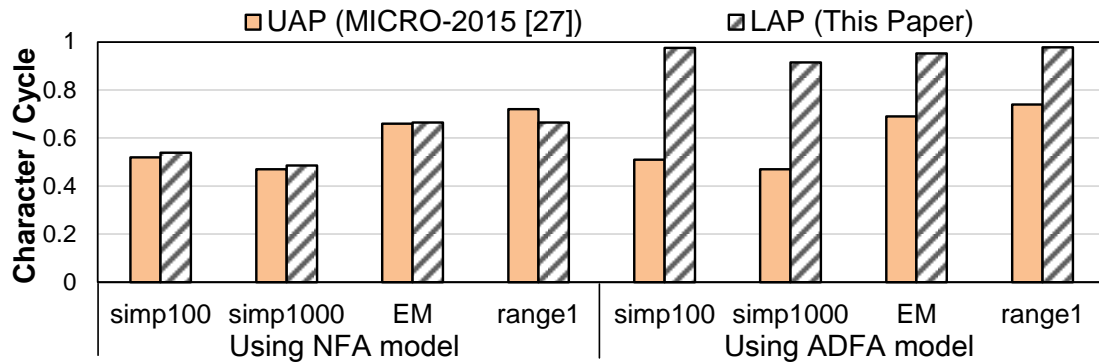


图 3.13 LAP 在不同工作负载下的总体处理速度

表 3.2 LAP 在不同工作负载下的总体处理速度原始数据

	simple100	simple1000	EM	range1
UAP @ NFA	0.52	0.47	0.66	0.72
UAP @ ADFA	0.51	0.47	0.69	0.74
LAP @ NFA	0.539	0.485	0.665	0.665
LAP @ ADFA	0.975	0.914	0.952	0.977

注：表格中数据的单位是 Character/Cycle

3.6.2 对 LAP 处理速度的详细评估

如第3.3.2小节所述，ADFA 模型所引发的额外的存储器访问导致了更慢的字符处理速度。因此，UAP^[27] 系统基于 ADFA 模型进行模式匹配时处理速度较慢。相反的，RegX^[14] 则实现了 DFA 模型下理想的字符处理速度。因此，我们在图3.14中将 LAP 的处理速度和这两款经典的基于 DFA 的硬件方案做对比。在图3.14中我们把 UAP 作为 baseline，而把 RegX 作为性能的天板。通过这样的方式，读者可以直观的知道已有的基于 ADFA 的硬件方案已经达到了怎样的处理速度，同时也可以清晰地知道我们距离实现最理想的处理速度还有多远距离。同时，我们也评估了 LAP 在应用本文提出的优化 (见第3.3.3小节) 前后的性能提升，从而可以清晰地定位性能提升的源头。从图3.14中我们可以发现，LAP 在不使用任何优化的情况下和 UAP 的处理性能类似。然而，LAP 在使用第一项优化 (优化第一类状态回退) 后实现了显著地性能提升。这说明了限制 ADFA 处理速度的主要因素是第一类状态回退导致的额外存储器访问。更进一步，第二项优化 (优化第二类状态回退) 利用辅助存储器的空闲带宽进一步提升了 LAP 的字符处理速度。

图3.14中的原始数据如表格3.3所示。

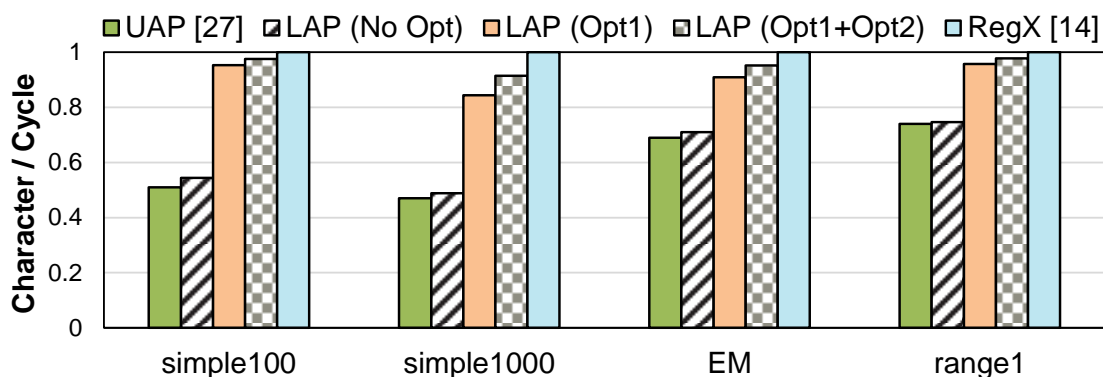


图 3.14 详细的 LAP 性能分析

表 3.3 详细的 LAP 性能分析原始数据

	simple100	simple1000	EM	range1
UAP	0.51	0.47	0.69	0.74
LAP @NoOpt	0.54	0.49	0.71	0.74
LAP @Opt1	0.95	0.84	0.91	0.96
LAP @Opt1+Opt2	0.97	0.91	0.95	0.98
RegX	1	1	1	1

注：表格中数据的单位是 Character/Cycle

3.6.3 LAP 存储效率的分析和讨论

图3.15(a)展示了在 simple 400^[14]数据集下 LAP 和其他已有硬件方案实现的存储效率。AP 代表了一类基于 Micron 公司的 Automata Processor^[15]的硬件实现。这一类实现都基于 NFA 模型而且使用相同的独热码编码，因此他们的存储效率是类似的。此处，我们使用 AP 的存储效率代表这一类方案的存储效率。RegX^[14]使用 BFSM 模型^[39]（一种 DFA 模型的增强版本）实现了和 AP 类似的存储效率。除此以外，UAP^[27]支持各种自动机模型。基于 DFA 模型时，UAP 实现了比较差的存储效率。然而，UAP 在使用 NFA 模型和 ADFA 模型时实现了最先进的存储效率（尽管使用这两种模型时 UAP 的处理速度不令人满意）。最重要的是，LAP 实现了很好的存储效率，相比于 RegX 和 AP 存储效率提升了 8 倍。

图3.15中的原始数据如表格3.4所示。

表 3.4 LAP 的存储效率原始数据

AP @NFA	RegX @BFSM	UAP @DFA	UAP @ADFA	UAP @NFA	LAP @ADFA
3.12	3	0.9	25	27	24.7

注：表格中数据的单位是 #Pattern/KB

接下来我们分析下，以上的硬件方案为什么在存储效率方面有着这么大的

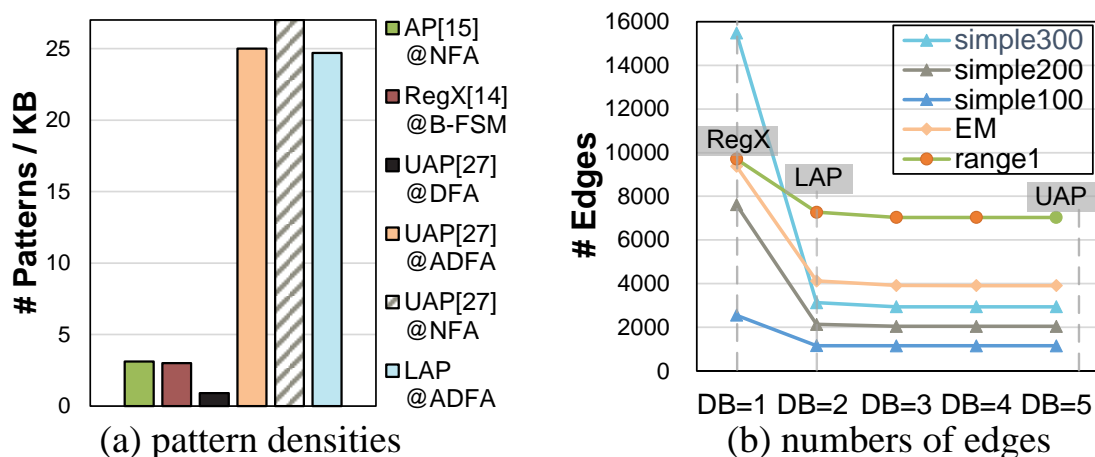


图 3.15 LAP 的存储效率对比和分析

差异。一般来说，NFA 模型相比于等价的 DFA 模型有着更少的状态数和更少的边数。因此，UAP 在使用 NFA 模型并搭配高效的模型打包算法后取得了最高的存储效率。虽然 AP 系列的硬件方案也是基于 NFA 模型，但是它们采用的空间架构要求其内部 NFA 模型必须采取独热码编码每个 NFA 状态。因此，AP 系列的存储效率并不高。除此以外，UAP 在基于 DFA 模型时实现的存储效率并不高，这是因为 DFA 模型中有大量的冗余边。相反，在 RegX、UAP 和 LAP 都依托于 default edge 实现了高效的增量存储。因此，这三个方案实现了很高的存储效率。值得一提的是，这三个实现方案之间最核心的不同是：每一个状态最多经过多少次连续的状态回退可以到达 DFA 的初始状态。也就是，在这三个方案中允许的最长的由 default edge 组成的路径长度是不同的。在 ADFA 模型构建时，这是一个重要的参数叫“深度约束”（Depth Bound, DB）。从图3.15可以发现，随着 DB 这个参数值得增加，DFA 模型中边的数目是不断降低的。RegX 仅仅支持最长为 1 的由 default edge 组成的路径，也就是 RegX 的 DB 等于 1。在本文，我们将 DB 设置为 2。我们可以从图3.15(b) 中发现，DB 从 1 变成 2 的时候 DFA 中边的数目发生了一次剧烈的下降。因此，LAP 相比 RegX 实现了高得多的存储效率。值得一提的是，本文中使用的高效的模型打包算法和高效的编码方式也是 LAP 可以实现最先进存储效率的重要原因。除此以外，UAP 支持任意长度的 default edge 组成的路径。因此对于 UAP 而言, DB 的值等于无穷大。所以，UAP 只需要存储比 LAP 还少的边，从而实现了比 LAP 略微高一些的存储效率。然而，LAP 却实现了相比 UAP 高得多的处理速度。因此，对于 LAP 而言，这是一个很划算的在存储效率和处理速度间的权衡。

第 4 章 LAP_SoC: 基于 LAP 核心的异构片上系统

在第3章，我们介绍了轻量级的 LAP 处理器核心。经过理论和实验验证，我们证实了 LAP 核心相比于已有的自动机处理器展现出了更高的存储效率和计算速度。然而，LAP 作为一个轻量级的专用处理器核心，不具备图灵完全的计算能力。LAP 需要通用 CPU 为其提供数据输入、控制信号和结果的输出汇报。因此，LAP 需要与通用处理器集成在一起协同地完成多模式匹配计算。本章将进一步拓展 LAP 处理器核心，将其封装为带有标准 AXI 数据接口的 Xilinx IP 核。通过这种方式，LAP 可以很方便地与 CPU 进行系统集成。基于 LAP IP 核，我们在 ZYNQ FPGA 硬件平台上部署了一个 LAP 处理器核心 + 低功耗 ARM CPU 的异构计算系统。实验结果表明，在只配备了一个 LAP 核心的情况下，该异构计算系统相比于原本的 CPU 系统在多模式匹配计算方面实现了 40+ 倍的速度提升。

4.1 构建 LAP_SoC 的动机

在本章，我们基于 LAP 专用处理器核心和 CPU 通用处理器核心构建了一个完整异构计算片上系统。通过在这个异构平台上运行多模式匹配计算，我们可以实现两个目的。第一，我们可以展示 LAP 可以按照什么方式与通用计算系统集成在一起，以及展示通用计算系统如何驱动该专用处理器进行高效的多模式匹配计算。第二，我们可以进一步得知在通用计算系统中添加 LAP 核心可以获得多少倍的性能提升，同时需要增加多少功耗。通过在 ZYNQ FPGA 上实际的运行 linux 操作系统，并驱动通过可编程逻辑构建的 LAP 核心，我们可以实际地测试该 SoC 系统进行多模式匹配计算时的端到端延迟和计算吞吐。

LAP_SoC 应当可以作为一个完整的异构计算系统，独立的进行多模式匹配计算。首先，我们的片上系统需要 CPU 的通用计算能力。LAP 不具备图灵完全的计算能力，也不支持操作系统。而且，LAP 需要外部控制器为其提供数据输入、输出和状态控制。低功耗的 CPU 系统恰好可以胜任这些工作。不仅如此，我们还需要为 LAP 提供友好的用户接口。在本文中，我们在 LAP_SoC 的通用处理器系统上运行了 Linux 系统，并为了 LAP 编写了可运行在 Linux 操作系统之上的 C 语言驱动程序。第二，我们需要为该 SoC 添加一到多个 LAP 核心。在本章，我们只添加了一个 LAP 核心，但是足以说明添加了 LAP 的异构系统的优越性（相比于原本的同构 CPU 系统）。如果需要更大的计算吞吐，我们可以进一步增加 LAP 的核心数。第三，我们需要为 LAP 核心配置直接内存访问控制器组 (DMAC)。这些 DMA 控制器可以负责将大批量的待处理的文本数据传输到 LAP

核心的输入缓存中，并在LAP完成计算后将计算结果传输到CPU系统中。LAP要处理的文本数据可以直接来自于网络控制器或者CPU系统的DDR。这些数据的量非常大，因此需要借助于DMAC进行高效的传输，而无需CPU的参与。综上，该SoC系统主要包含三个部分：

- 由通用处理器组成的多核计算系统，负责运行Linux操作系统；
- 由一到多颗LAP核心组成的面向多模式匹配任务的专用计算系统；
- 负责向LAP核心搬运数据和从LAP收集计算结果的DMA硬件控制器组。

4.2 LAP的标准化数据接口

为了将LAP与ZYNQ系统中的ARM CPU集成在一起，我们需要为LAP提供标准化的接口。Xilinx公司推出的产品内部都采用AXI总线系统，因此我们为LAP增加了一些外围逻辑使得LAP支持AXI协议。Xilinx IP核可以很便捷的通过Xilinx Vivado集成开发环境提供的Block Design^[48]功能与其他模块（比如CPU）进行系统集成。因此，我们在本节将LAP拓展为带AXI标准化接口的模块，并进一步将其导出为Xilinx IP核。

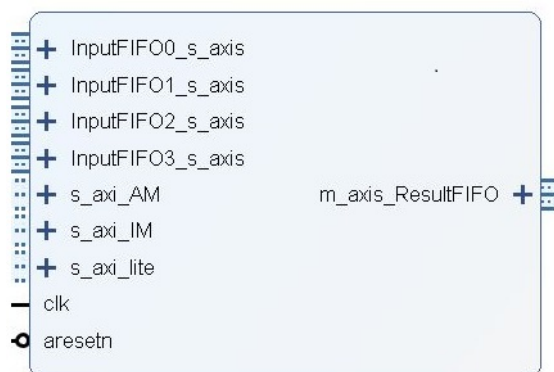


图 4.1 基于LAP的Xilinx IP核

基于LAP核心的IP核的接口情况如图4.1所示。该IP核包括：

- 4个AXI-Stream接口，用于同时接收四个待处理的输入字符流；
- 2个AXI-4接口，用于将LAP的二进制程序写入LAP；
- 1个AXI-Lite接口，用于ARM CPU对LAP进行控制和状态监控；
- 1个AXI-Stream接口，用于将LAP的匹配结果返回给ARM CPU。

4.2.1 基于AXI-Stream的输入接口

在第3章中，我们使用BRAM来当做LAP核心的输入字符流的Input Buffer并使用SPU部件从该buffer中按顺序的读取输入字符（如图3.8）。这实际上是一种流式的数据访问，因此在实际部署LAP核心时，我们将使用AXI-Stream接口的

FIFO来替代原本的BRAM式设计。由于LAP使用了细粒度多线程（见3.4.2节）技术，因此一个LAP核心可以独立运行4个线程，而每个线程处理独立的字符流。因此，LAP的流式输入接口我们使用4个AXI FIFO IP核来实现，每个周期选择其中的一个FIFO的输出值来进行当前线程（处于第一个流水级的线程）的处理。

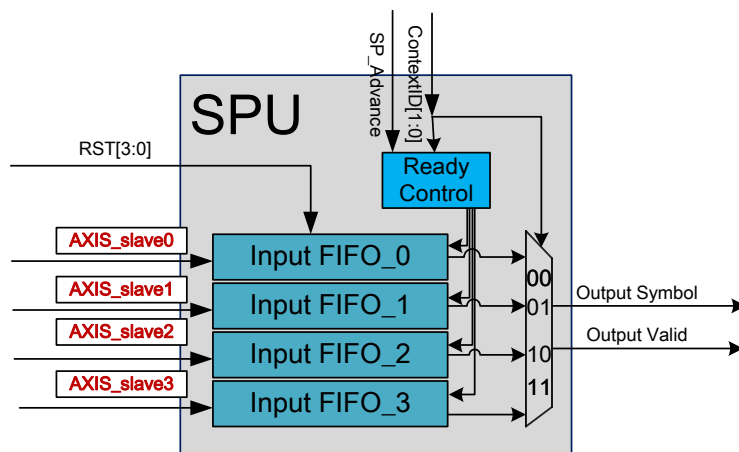


图 4.2 LAP 的流式输入接口设计

4.2.2 基于 AXI-4 的程序写入接口

与第3章中的设计相同，LAP 仍然使用 BRAM 来存储自动机模型编译成的二进制程序。该 BRAM 采用双端口模式，端口 A 用于给 LAP 进行指令读取，端口 B 用于外界 CPU 将二进制程序写入 BRAM。在实际实现时，我们采用了 AXI BRAM Controller^[49] 这个 IP 核，从而很便捷的将端口 B 的 naive 接口转换为 AXI-4 接口。从而，ARM CPU 可以通过 AXI 接口与之相连，并采用内存映射的方式对其进行读写。

4.2.3 基于 AXI-Stream 的输出接口

作为一个专用于模式匹配的处理器，LAP 需要输出所有发现匹配模式的位置（字符流中的位置）以及输出每个位置匹配的是哪几个模式。在 LAP 中，我们不仅需要输出每次匹配发生的位置 (Location)，我们还需要输出匹配发生在哪一个线程 (ThreadID)。同时，我们选择在 LAP 中输出汇报匹配的自动机状态 (StateID)。我们可以很简单的根据状态编号得到具体匹配的模式编号（匹配的是第几个正则表达式）。在实际运行模式匹配任务时，匹配的结果相比输入流要小得多。因此，CPU 将 StateID 翻译成具体的正则表达式编号将不会引起显著地开销。每当发生匹配时，我们把 ThreadID、Location 和 StateID 组合在一起作为一个整体 (entry)，并将这样一个组合的 entry（如图4.3所示）写入输出 FIFO

中。最终，外部 CPU 可以顺序地读取所有匹配结果。

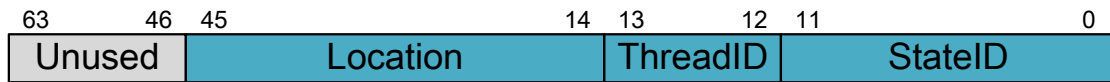


图 4.3 LAP 计算结果的返回包格式

4.2.4 基于 AXI-Lite 的配置接口

我们使用 Verilog 实现了 AXI-Lite 协议的接口电路，向外界提供一个 AXI-Lite 接口用于控制 LAP 的启动停止和监控 LAP 的运行状态。如图4.4所示，我们为每个 LAP IP 核配备了 4 个 32 比特的控制状态寄存器（CSR）。图中填充为水绿色的寄存器位是已经被使用的，其他寄存器位则是暂且不被使用的。这些状态寄存器可以由 LAP 核心读写或者通过 AXI 总线由外部设备读写。CSR0 中的 4 个比特可以独立地控制 LAP 四个线程的复位（也就是停止运行）或者运行。比如，CSR0[0] 等于 1 时，意味着 LAP 的 0 号线程将会被复位，也就是停止。CSR1 的 32 个比特则记载了 LAP 发生了多少次匹配。该信息可以指导 CPU 对 LAP 的 OutputFIFO 的读取。CSR2 则用于监测 LAP 执行过程中遇到的各种异常，此处我们暂且只检测两种异常。如果 CSR2[0] 被 LAP 核心置高，这意味着 LAP 的状态队列发生了下溢，没有有效的状态可以读取了。如果 CSR2[1] 被 LAP 核心置高，这意味着 LAP 的状态队列发生了上溢，这表示下阶段状态队列的状态发生了上溢。这种情况只会在 LAP 运行 NFA 时发生，此时 NFA 模型中同时处于激活状态的状态总数超出了 ASS 的硬件队列允许的上限。

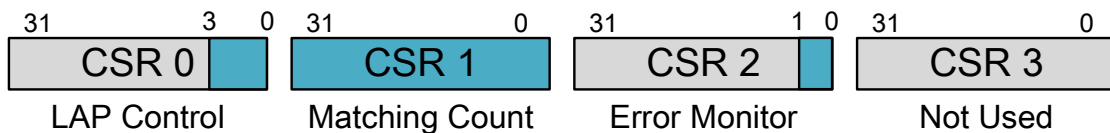


图 4.4 LAP 控制状态寄存器格式

4.3 LAP_SoC 硬件系统

本章的目的是设计实现一个面向 ZYNQ FPGA 硬件平台（见2.2节）的 LAP 异构计算系统，因此在进行片上系统设计时需要充分考虑 ZYNQ 的基础架构。图4.5是 LAP 异构片上系统的完整架构图。图4.5的上半部分展示了 LAP_SoC 的总体架构，而图4.5的下半部分则是 LAP Core 的详细架构。可以看到，LAP_SoC 由两部分组成。第一部分是 ZYNQ 硬件平台上的通用处理系统（Processing System）。该处理系统包括 512MB 的 DDR 芯片和双核心 Cortex A9 的 ARM 架构中央处理器系统，可以进行低功耗的通用计算。在 LAP 的片上系统中，该处理器系统作为

宿主机负责控制 LAP 核心的计算。LAP_SoC 的第二部分是可编程的硬件逻辑。我们正是使用这些可编程逻辑搭建了我们的 LAP 核心。可以看到，我们在 LAP Core 的基础上添加新的外围电路，将其封装成了带有标准 AXI 接口的 IP 核。进一步的，我们将输入输出的 AXI Stream 接口和 5 个 AXI DMA 控制器连接起来。DMA 控制器的另一端则是连接着 PS 系统中的 DDR。通过这种方式，我们可以使用直接内存访问技术在 LAP 和 PS DDR 间进行大规模的数据搬运。值得一提的是，这些 DMA 控制器的配置是通过修改他们的 AXI-Lite 的控制状态寄存器实现的。这样的任务也是由 ARM CPU 处理的。最终，我们将所有 LAP 的 AXI-4 接口和 AXI-Lite 接口连接到 AXI Interconnect 上。CPU 可以通过该总线互联控制全部的组件。

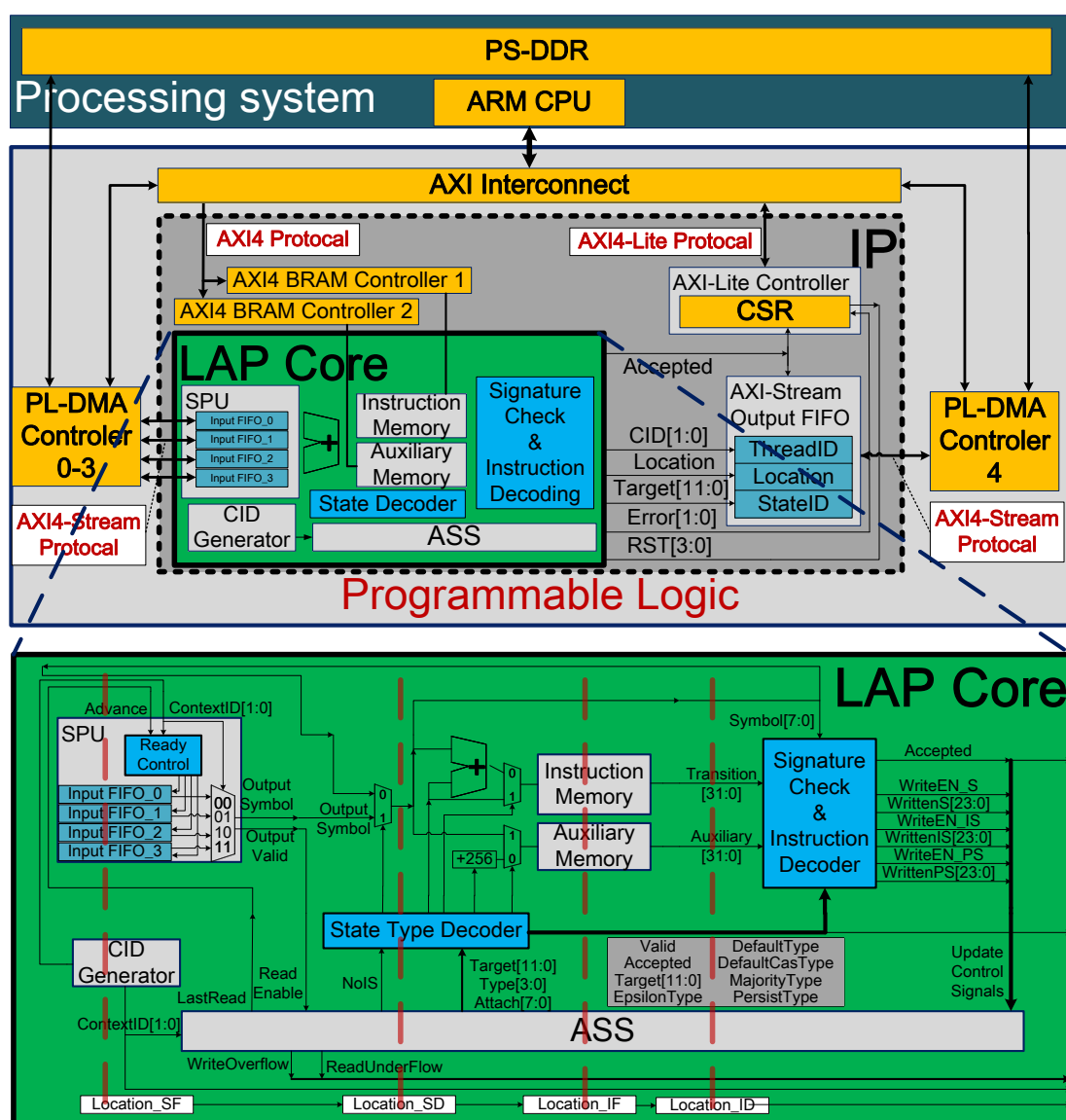


图 4.5 LAP 片上系统整体架构设计

根据上述架构图，我们使用 Vivado 的 Block Design^[48] 功能，将 LAP 的 IP

核与 ARM CPU 集成为如图4.6所示的异构系统。图中有 4 个 DMA 控制器负责给 LAP 输入要处理的输入字符流，而第 5 个 DMA 控制器则负责将 LAP 产生的匹配结果写回 PS 系统的 DDR 中。图右上角的 ARM CPU 则自始至终担任着调度和控制的角色，无需进行高性能的计算。除此以外的其他模块则是由集成开发工具自动生成的必须模块，包括 AXI 总线互联模块和全系统的复位控制模块。

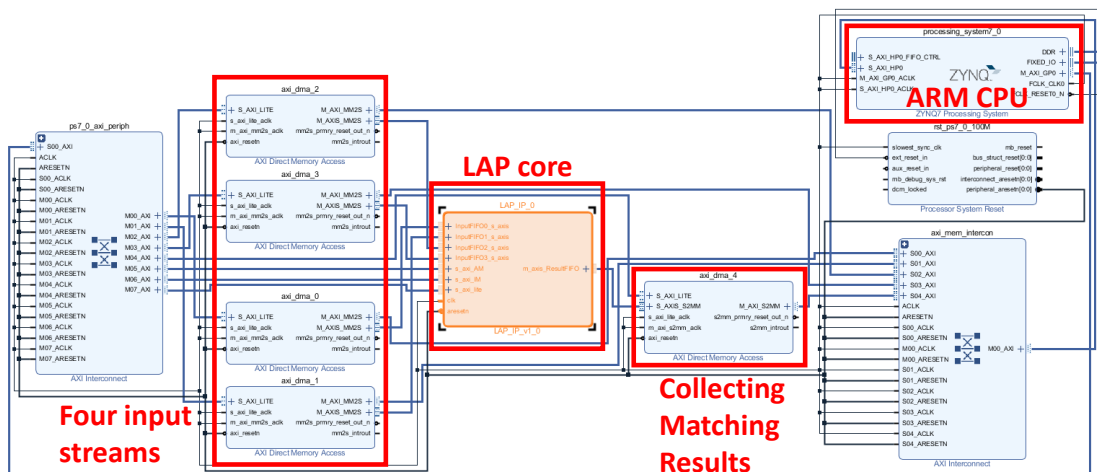


图 4.6 ZYNQ 系统的 Block Design

在完成 Block Design 后，我们还需要为系统中每个可以读写的模块配置一个统一的物理地址（ARM CPU 视角下的物理地址）。如图4.7所示，我们需要为 5 个 DMA 控制器的控制状态寄存器配置不同的物理地址，以便于 CPU 对他们进行控制。CPU 只需要简单的操作他们的控制状态寄存器，就可以在自己不参与的情况下进行大规模的流式数据传输工作。类似的，我们也为 LAP 的两个程序存储器和控制状态寄存器分配了少量物理地址。

```

    /processing_system7_0
    /processing_system7_0/Data (32 address bits : 0x40000000 [ 1G ])
    /axi_dma_0 S_AXI_LITE Reg 0x4040_0000 64K 0x4040_FFFF
    /axi_dma_1 S_AXI_LITE Reg 0x4041_0000 64K 0x4041_FFFF
    /axi_dma_2 S_AXI_LITE Reg 0x4042_0000 64K 0x4042_FFFF
    /axi_dma_3 S_AXI_LITE Reg 0x4043_0000 64K 0x4043_FFFF
    /axi_dma_4 S_AXI_LITE Reg 0x4044_0000 64K 0x4044_FFFF
    /LAP_IP_0 s_axi_AM reg0 0x43C0_0000 64K 0x43C0_FFFF
    /LAP_IP_0 s_axi_IM reg0 0x43C1_0000 64K 0x43C1_FFFF
    /LAP_IP_0 s_axi_lite reg0 0x43C2_0000 64K 0x43C2_FFFF
    
```

图 4.7 ZYNQ 系统的地址分配

4.4 LAP_SoC 软件系统

与 LAP_SoC 配套的软件系统主要包括两个部分。首先，我们需要为 ARM CPU 系统安装和配置一个操作系统。其次，我们需要为 LAP 编写设备驱动程序。

4.4.1 操作系统配置与部署

为了使得我们的异构系统易于操作，我们为该片上系统安装了嵌入式 Linux 作为其操作系统。为了便捷，我们使用 Petalinux 2020.1^[50] 工具来快捷地配置我们的操作系统并生成对应的 SD 卡启动镜像。Zedboard 平台上总共有 512 MB 大小的 DDR 空间。DDR 的存储空间可以由 PS 端的 ARM CPU 读写，同时也可以由 PL 端的各种模块进行读写。如图4.8所示，为了实现 CPU 端字符流到 LAP AXI-Strem 接口的 DMA 传输，我们为待传输的数据准备了 32 MB 的 DDR 空间当做缓存。由于 LAP 支持 4 个输入流，因此我们总共划分了 128 MB 空间预留给 DMA 传输当做缓存。对于 LAP 的输出，我们也在 DDR 中为其准备了 32 MB 的 DDR 空间。通过这样的方式，CPU 和 LAP 可以共享这部分 DDR 的内容。除此以外，我们将剩余的 352 MB 存储器空间划分给 Linux 操作系统。为了指定 Linux 操作系统使用的 DDR 地址范围，我们修改了 Linux 操作系统的设备树文件。

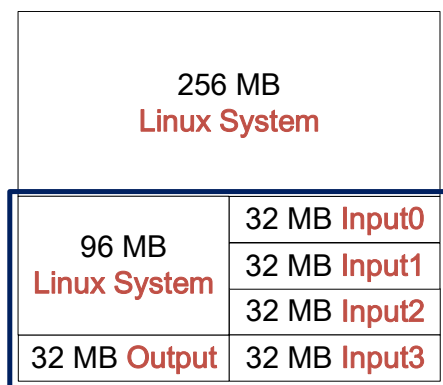


图 4.8 ZYNQ 系统 DDR 空间分配

4.4.2 驱动程序设计与实现

本文使用内存映射的方式读写外设，编写了运行在用户态的 LAP 的设备驱动 C 程序。linux 文件系统提供了一个特殊的接口用于 CPU 可见的访问物理地址。我们可以以如下方式打开一个特殊的文件并获得一个文件句柄：`open("/dev/mem", O_RDWR|O_SYNC)`。随后我们可以使用 `mmap()` 函数将这个文件的一部分映射到 C 程序可见的一段逻辑地址上。那么，我们就可以在 C 程序中读写这个文件。值得注意的是，这个特殊的文件对应着 CPU 的可见物理地址，对这个文件的读写就是对 CPU 可见的物理地址进行读写。通过这样的方式，我们就可以在 C 程序中读写 CPU 可见的物理地址。通过这样的方式，我们可以在 C 程序中读写 DMA 控制器的控制状态寄存器、LAP 的指令存储器和 LAP 的控制状态寄存器。

为了方便的对 LAP 的使用，我们实现了如表格4.1所示的 C 函数。ActivateLAP() 函数的输入为 LAP 的控制状态寄存器的物理地址和要激活的线程

号。该函数根据 LAP 的控制寄存器的地址修改该寄存器的值，从而启动想要启动的 LAP 线程。类似的，StopLAP() 函数则是可以关闭想要关闭的 LAP 线程。ReadLAPAcceptCount() 函数的输入也是 LAP 的控制状态寄存器地址。该函数访问 LAP 的状态寄存器并读取其值，从而得知 LAP 的输出 FIFO 中有多少有效的匹配，从而相应的读取 LAP 的匹配结果。相应的，InitLAPAcceptCount() 函数则是在 LAP 读取完输出 FIFO 的内容后，重置该寄存器的值（置为 0）。剩余的函数用于控制 LAP_SoC 的 DMA 控制器。值得注意的是，DMA 控制器分为两类。MM2S（Memory Map to Stream）类型的 DMA 控制器将来自 AXI-4 的数据搬运到 AXI-Stream 中。我们使用 MM2S 类型的 DMA 控制器将输入字符流从 DDR 中搬运到 LAP 的流式输入口。类似的，S2MM（Stream to Memory Map）类型的 DMA 控制器将来自 AXI-Stream 的数据搬运到 AXI-4 接口中。我们使用这一类 DMA 控制器将 LAP 的匹配结果从输出 FIFO 中拷贝到 DDR 中。对于每一类 DMA 控制器我们三个驱动函数。

- DMA_XXXX_Init() 函数负责初始化 DMA 控制器；
- DMA_XXXX_Is_Busy() 函数负责检测当前 DMA 控制器是否在忙碌，也就是检查当前 DMA 控制器被分配的数据拷贝任务是否已经完成；
- DMA_XXXX_Start() 则是启动一次 DMA 传输，该函数的输入包含数据传输的内存地址和传输的数据量。

表 4.1 驱动 LAP 的 C 函数

函数定义	函数功能
void ActivateLAP(int *Addr,int ThreadID)	启动 LAP 的某个线程
void StopLAP(int *Addr,int ThreadID)	停止 LAP 的某个线程
int ReadLAPAcceptCount(int *Addr)	读取 LAP 发现的匹配总数
void InitLAPAcceptCount(int *Addr)	初始化 LAP 发现的匹配总数
int DMA_MM2S_Init(int *Addr)	初始化 MM2S 类型的 DMA 控制器
int DMA_S2MM_Init(int *Addr)	初始化 S2MM 类型的 DMA 控制器
int DMA_MM2S_Is_Busy(int *Addr)	检测 MM2S 类型的 DMA 控制器是否在忙
int DMA_S2MM_Is_Busy(int *Addr)	检测 S2MM 类型的 DMA 控制器是否在忙
void DMA_MM2S_Start(int *addr, int src_addr, int size)	启动 MM2S 类型的 DMA 控制器
void DMA_S2MM_Start(int *addr, int dst_addr, int size)	启动 S2MM 类型的 DMA 控制器

4.5 LAP_SoC 部署与评估

为了评估基于LAP的片上系统的性能，我们实现了带有单核心LAP和ARM CPU的异构系统。我们将该系统实现，并部署到Zedboard FPGA开发板上实际运行，以评估多模式匹配的计算性能和功耗数据。如果装备更多的LAP核心，该系统的处理性能可以进一步提升。

4.5.1 实现平台与部署结果

图4.9展示的是用来部署LAP片上系统的Zedboard开发板的实物图。我们使用Verilog实现了LAP核心和接口电路，并导出为Xilinx IP核，然后使用Vivado的Block Design功能实现了完整的SoC结构并生成了相应的比特流文件。最终，我们将比特流文件烧写到该开发板上，实现了硬件系统的完整部署。

我们在Zedboard上部署了单个LAP核心+ARM CPU的异构系统，并整体运行在100 MHz的主频下。值得一提的是，LAP核心本身可以支持更高主频，但是其AXI接口电路和SoC的其他部分暂时无法运行在更高的主频下。如果再进一步进行多时钟域（LAP运行在更高主频下，而LAP的输入输出FIFO工作在低一些的主频下）的设计，LAP的主频理论上可以提升2.5倍（达到第3章提到的263 MHz）。即使基于这样一个Scale Down的一个原型系统，我们的LAP片上系统仍然可以展现出其优越性。

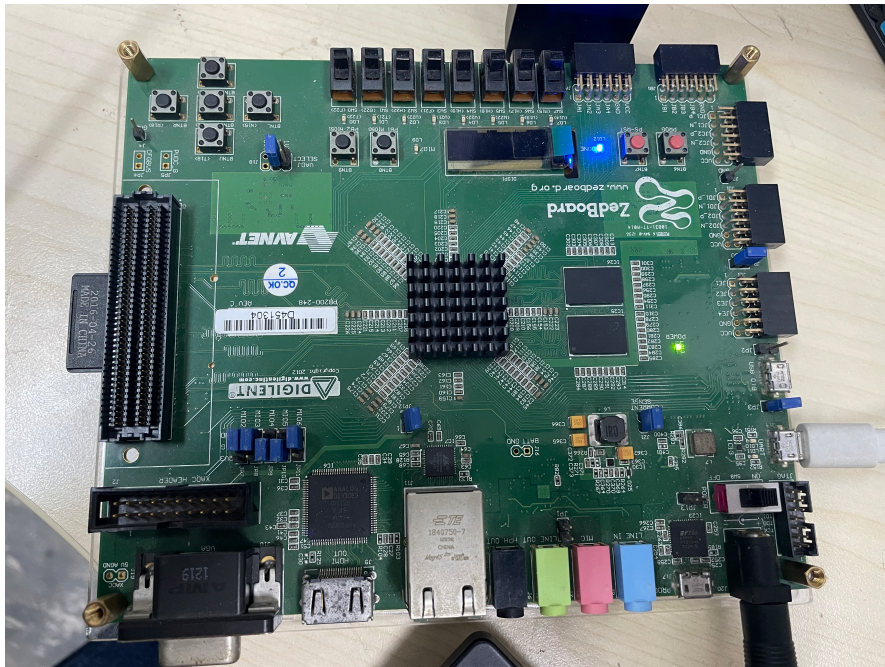


图4.9 LAP_SoC 的部署平台：Zedboard 开发板

表格4.2展示了该异构系统的整体的资源占用，可以看到LUT是整个系统的资源瓶颈。不过我们也可以看到，在该系统上部署更多的LAP核心是可行的。

表 4.2 LAP 片上系统总体资源占用

资源类型	占用数	总共可用数	占用比例
LUT	8655	53200	16.27%
LUTRAM	623	17400	3.58%
FF	12365	106400	11.62%
BRAM	12.5	140	8.93%

图4.10则进一步展示了 LAP 片上系统的层次结构。图中每个部件对应的框的大小反映了他们所占据的硬件资源。从图中可以发现，LAP_IP 其实只占据了比较少的硬件资源，而 PS 系统的 axi 外围总线占用了较多的资源。同时，5 个 DMA 控制器占据了较多的资源。事实上，如果引入更多的 LAP 核心，这些 DMA 控制器是可以复用的。因此我们可以得出结论，多添加一个 LAP 核心并不需要使用表格4.2中那么多资源，而是只需要增加一小部分资源。因此，该系统能容纳的 LAP 核的总数比我们根据表格4.2预估的数目要多一些。

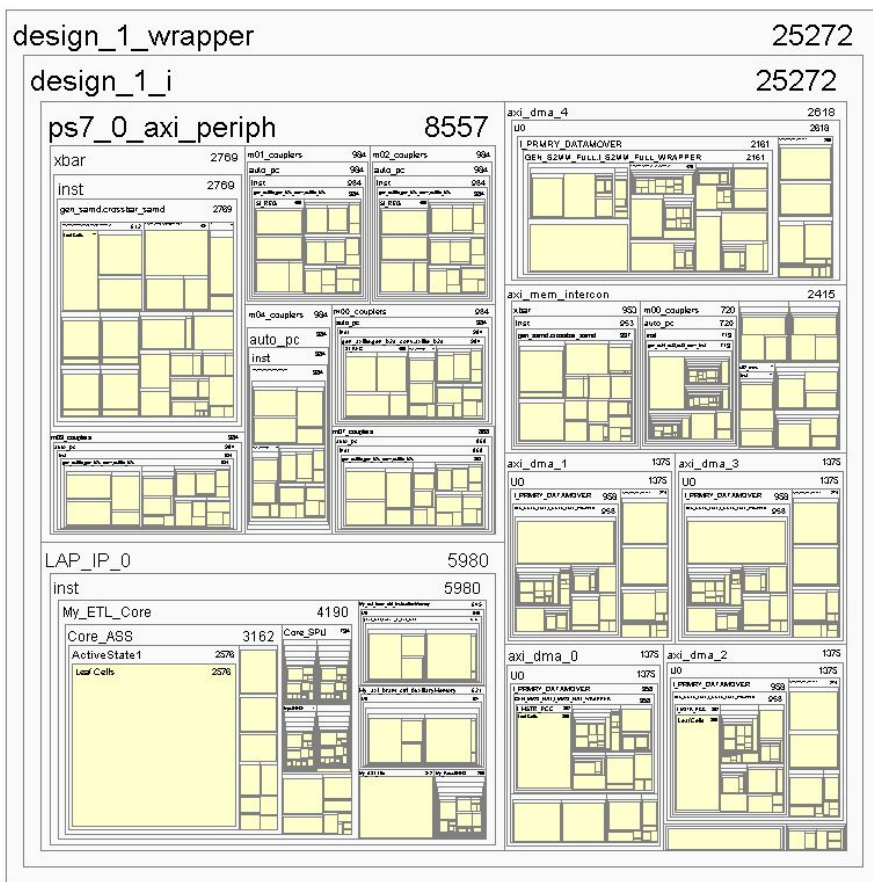


图 4.10 LAP 片上系统的层次结构

图4.11展示了该片上系统的功耗分布情况。可以看到 PS7（也就是 ARM CPU 系统）占据了 95% 的功耗，而我们的 LAP 核心加上 DMA 控制器总共就占据了 5% 的系统功耗。这意味着，我们每添加一个 LAP 核心，只会使得片上系统的总

功耗提升大约5%的功耗，而性能则是成倍数增长。由此可见，使用LAP核心进行多模式匹配计算可以极大地提升系统的整体能效。

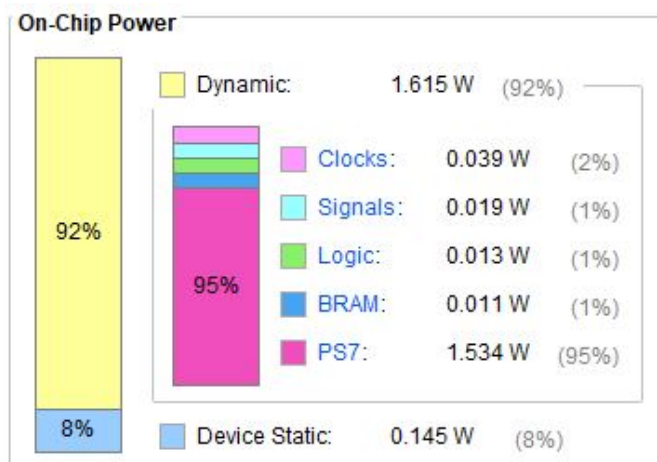


图 4.11 LAP 片上系统功耗分布

4.5.2 模式匹配性能的评估方法

为了评估LAP核心在实际的异构系统中的性能表现，我们将LAP的处理性能分别和嵌入式的ARM处理器以及面向桌面系统的Intel处理器对比。表格4.3详细介绍了评估性能所使用的三种硬件平台。其实，LAP使用本文介绍的专用的指令集和编译器将模式匹配任务编译成LAP支持的自动机模型，然后使用LAP的驱动函数驱动LAP进行高效的模式匹配计算。相应的，用于对照的两款CPU则是使用Linux Shell中的grep命令进行多模式匹配。Grep是linux系统中一个非常常用的用于模式匹配的标准工具，在该类型的计算方面有着强悍的性能。因此，本文选择该软件工具作为CPU平台上的测试程序。

表 4.3 评估性能所使用的三种硬件平台

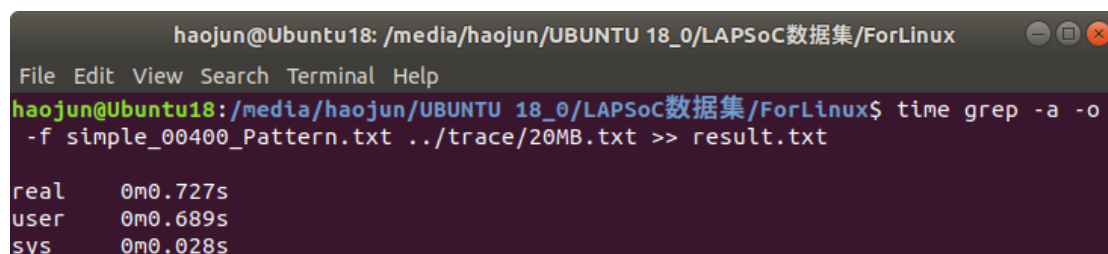
处理器架构	最大主频	指令吞吐	物理核心数
Cortex-A9 ARMv7	1.2 GHz	667 BogoMIPS ¹	2
Intel Core i5-6500	3.2 GHz	6400 BogoMIPS ¹	4
LAP (Our Work)	100 MHz	/	1

¹BogoMIPS: Linux 操作系统中衡量计算机处理器运行速度的一种尺度

对于多模式匹配计算，该任务的输入包括一个待匹配的正则表达式集合和一个输入字符流。为了进行严谨的对比，我们实验并记录了这三种平台在不同输入规模下的各自的处理时长。总的来说，我们的输入数据在两个维度上变化。第一个维度是同时匹配的正则表达式的数目，而第二个维度则是要处理的字符流的长度。我们所使用的正则表达式集合（simple100, simple200 和 simple400）取自

IBM PowerEN 处理器使用过的 benchmark^[14]。我们使用的字符流取自 simple400 对应的字符流。在此基础上我们对该字符流进行截取和复制，从而生成了不同大小的字符流文件。

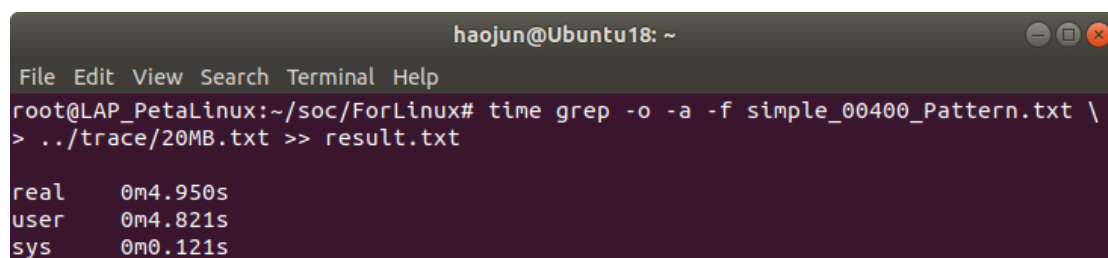
图4.12展示了在运行桌面 Ubuntu 的系统上进行多模式匹配的 linux 命令。我们使用 time 工具为模式匹配任务进行计时。同时，我们使用标准的工具 grep 进行多模式匹配。我们使用参数 -f，从 simple_400_Patterns.txt 中读取要匹配的正则表达式集合。此处表示，我们要同时匹配 400 个简单的正则表达式。随后，第二个重要参数是 20MB.txt，该文件存储了要处理的字符流。命令的最后，我们将匹配结果重定向到 result.txt 文件中。在 grep 执行结束后，终端打印出了总的执行时间。我们将该时间作为桌面系统中使用的 intel 处理器的单核模式匹配计算耗时。



```
haojun@Ubuntu18: /media/haojun/UBUNTU 18_0/LAPSoC数据集/ForLinux
File Edit View Search Terminal Help
haojun@Ubuntu18:/media/haojun/UBUNTU 18_0/LAPSoC数据集/ForLinux$ time grep -a -o
-f simple_00400_Pattern.txt ../trace/20MB.txt >> result.txt
real    0m0.727s
user    0m0.689s
sys     0m0.028s
```

图 4.12 桌面 Intel CPU 基于 grep 命令进行多模式匹配

图4.13展示了在 Cortex-A9 处理器上进行多模式匹配的 linux 命令。我们使用的正是 Zedboard 上的处理器系统进行评估。图中所示命令行和图4.12中的命令行是一致的。值得一提的是，图4.13中所示的终端也是显示在 Linux 台式机上的。Zedboard 本身并没有装备显示器，而是通过 UART 串口和 linux 台式机连接，并在 linux 台式机上通过 Minicom 充当 Zedboard 的终端窗口。

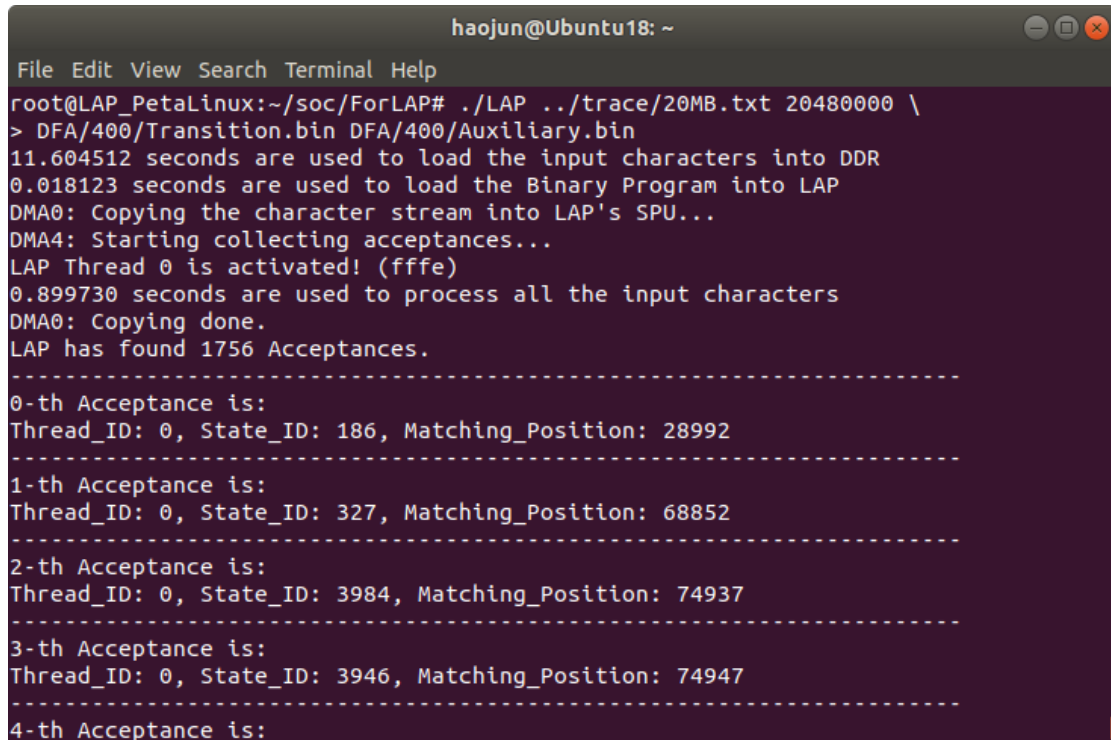


```
haojun@Ubuntu18: ~
File Edit View Search Terminal Help
root@LAP_PetaLinux:~/soc/ForLinux# time grep -o -a -f simple_00400_Pattern.txt \
> ../trace/20MB.txt >> result.txt
real    0m4.950s
user    0m4.821s
sys     0m0.121s
```

图 4.13 嵌入式 ARM CPU 基于 grep 命令进行多模式匹配

图4.14显示了使用 LAP 进行模式匹配计算时使用的命令行参数。我们通过 Minicom 连接到运行在 Zedboard ARM CPU 上的 linux 操作系统，并使用表格4.1中的驱动函数驱动 LAP 进行模式匹配计算。我们使用 C 语言编写了 ./LAP 程序。该程序读取 4 个输入参数：要处理的输入字符流文件、字符流文件大小和根据正则表达式集合编译得到的 LAP 二进制文件。./LAP 程序将 LAP 二进制文件写入

LAP 处理器核心，然后将字符流文件的内容通过 DMA 传输到 LAP 的流式输入端口。如图所示，我们还在该 C 程序中进行了各部分函数的执行时间的统计。最后，./LAP 将 LAP 的输出结果打印出来。



```

haojun@Ubuntu18: ~
File Edit View Search Terminal Help
root@LAP_PetaLinux:~/soc/ForLAP# ./LAP ../trace/20MB.txt 20480000 \
> DFA/400/Transition.bin DFA/400/Auxiliary.bin
11.604512 seconds are used to load the input characters into DDR
0.018123 seconds are used to load the Binary Program into LAP
DMA0: Copying the character stream into LAP's SPU...
DMA4: Starting collecting acceptances...
LAP Thread 0 is activated! (ffffe)
0.899730 seconds are used to process all the input characters
DMA0: Copying done.
LAP has found 1756 Acceptances.
-----
0-th Acceptance is:
Thread_ID: 0, State_ID: 186, Matching_Position: 28992
-----
1-th Acceptance is:
Thread_ID: 0, State_ID: 327, Matching_Position: 68852
-----
2-th Acceptance is:
Thread_ID: 0, State_ID: 3984, Matching_Position: 74937
-----
3-th Acceptance is:
Thread_ID: 0, State_ID: 3946, Matching_Position: 74947
-----
4-th Acceptance is:

```

图 4.14 使用 LAP 进行多模式匹配

4.5.3 实验结果与性能分析

基于上述实验方法，在三种硬件平台上我们实际的运行了不同规模下的模式匹配任务并得到如下的实验结果。值得一提的是，本部分进行比较时，只比较 intel/arm/LAP 处理器的单核心性能。

图4.15展示了运行 Ubuntu18.04 系统的 Intel 处理器和我们的 LAP_SoC 在不同的计算规模下所需要的执行时间。横轴表示的是要处理的字符流的总大小，而纵轴则反映了具体的执行时间。图中总共有 6 条线。其中，图上方的三条线代表着 Intel 处理器在同时匹配 100/200/400 个模式时需要的计算时间，而图下方的三条线则对应着 LAP 的执行时间。从图中，我们可以得到三个结论。第一，随着要处理的字符流的增大，这两种硬件平台所需要的计算时间也会线性增加。第二，随着同时匹配的模式串数目的增加，计算时间的变化不大。这一点可以从图中直观的看出：上面的三条线(或者说下面的三条线)非常接近彼此。第三，LAP 的处理时间比 Intel 处理器所需要的时间要少一些。图4.15的原始数据分别展示在表格4.4和表格4.5

表格4.6展示了 Cortex-A9 处理器在运行嵌入式 linux 系统时，grep 命令的执

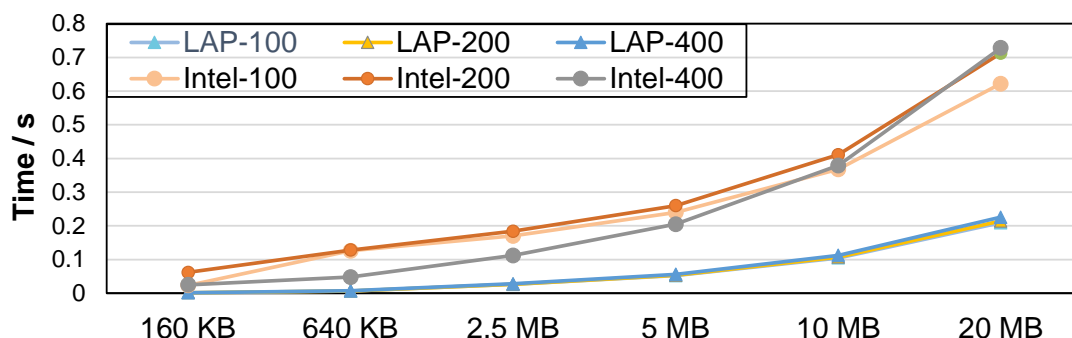


图 4.15 LAP 与 Intel 处理器进行模式匹配计算（不同输入规模下）的时间统计

表 4.4 LAP 在不同规模下运行 grep 命令耗时统计

字符流文件大小	160 KB	640 KB	2.5 MB	5 MB	10 MB	20 MB
匹配 100 模式耗时/秒	0.0016	0.0065	0.026	0.523	0.105	0.209
匹配 200 模式耗时/秒	0.0017	0.0068	0.027	0.054	0.107	0.214
匹配 400 模式耗时/秒	0.0016	0.007	0.028	0.056	0.113	0.226

表 4.5 Intel 处理器在不同规模下运行 grep 命令耗时统计

字符流文件大小	160 KB	640 KB	2.5 MB	5 MB	10 MB	20 MB
匹配 100 模式耗时/秒	0.023	0.126	0.17	0.24	0.368	0.622
匹配 200 模式耗时/秒	0.062	0.128	0.184	0.26	0.411	0.712
匹配 400 模式耗时/秒	0.025	0.048	0.112	0.205	0.379	0.728

行时间。之所以没和图4.15放在一起，是因为该处理器的处理速度较慢，与前两个硬件平台的处理时间相差较大。从表格4.6中可以发现，嵌入式 Linux 系统中的 grep 命令与桌面 Linux 系统中的 grep 命令的底层算法有一些差异。这可能是由于 linux 版本的问题。我们先观察表格的前两行数据。随着字符流文件的增大，处理时长在线性增加。这一点符合我们从图4.15中得到的第一个结论。不同的是，表格中同时匹配 200 个模式串的耗时是同时匹配 100 个模式串耗时的两倍。这一点，与我们从图4.15中得到的第二个结论不吻合。可以看到，由于采用这种低效的处理方式，Cortex-A9 处理器平台下的处理效率非常低下。有意思的是，当同时匹配 400 个模式时，Cortex-A9 的处理时间骤减。（相关数据已经被加粗。）结合上述观察，我们得到如下分析。第一，在该版本的 linux 下，grep 命令在处理较小规模数据时采用串行匹配的方法（每次匹配一个模式，依次匹配所有的模式）。第二，在处理较大规模输入时，grep 命令会采取更为高效的算法。在使用该算法时，处理时长将不再随着模式数目的增加而线性增加。

为了进行公平的比较，我们将只比较三种硬件平台在同时匹配 400 个模式时的处理速度。此时，Cortex-A9 处理器使用的是更为高性能的算法。图4.16展示了

表 4.6 Cortex-A9 处理器在不同规模下运行 grep 命令耗时统计

字符流文件大小	160 KB	640 KB	2.5 MB	5 MB	10 MB
匹配 100 模式耗时/秒	3.75	14.91	59.49	119.0	238.6
匹配 200 模式耗时/秒	7.46	29.76	119.14	238.2	476.5
匹配 400 模式耗时/秒	0.122	0.462	1.22	2.436	4.863

三种硬件平台上进行多模式匹配计算的加速比。图4.16的原始数据如表格4.7所示。其中，我们以 Cortex-A9 处理器作为基准，他的加速比恒为 1。观察图4.16可以发现，随着要处理的字符流的增大，LAP 的加速比先是显著降低。这是因为，当计算负载很小时，Cortex-A9 和 Intel 平台的操作系统开销就变得比较显著。因此，计算规模很小时，这两个平台的计算耗时的测量值其实偏大。随着计算规模的增大，操作系统带来的额外开销变得微乎其微。因此，在计算负载大于 2.5 MB 的大小后，LAP 的加速比维持恒定。更重要的是，尽管单核心的 LAP 的主频远远低于 Cortex-A9 和 Intel 处理器，但是 LAP 的处理速度却是 Cortex-A9 处理器的 40+ 倍，也是 Intel 处理器处理速度的 3 倍以上。

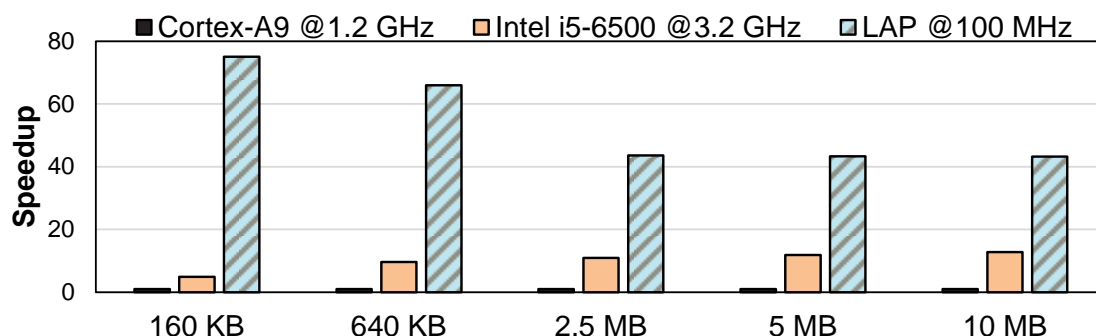


图 4.16 三种硬件平台下进行模式匹配计算（不同输入规模下）的加速比

表 4.7 三种硬件平台下进行模式匹配计算的加速比原始数据

字符流文件大小	160 KB	640 KB	2.5 MB	5 MB	10 MB
Cortex-A9	1	1	1	1	1
Intel i5-6500	4.88	9.625	10.893	11.88	12.83
LAP	75.08	66	43.57	43.3	43.23

这是因为，LAP 的一条指令就可以完成一次自动机模型的状态跳转，而通用 CPU 则需要将近 24 条指令^[51]。此外，通用 CPU 还会有 Cache Miss 和分支预测失败的开销，因此进行模式匹配计算时性能不高。虽然当代通用处理器都有着强大 SIMD 的计算能力，然而对于不便于并行化处理的多模式匹配计算却并没有很大增益。综上所述，我们充分证明了采用专用指令集的 LAP 在进行多模式匹配计算时的优越性。

第 5 章 总结与展望

5.1 本文总结

随着信息时代的到来，越来越多的信息被记录在计算机系统中。这其中包含了大量系统日志、网络流量、文字记录和生物学序列等非结构化的文本数据。在进行文本类数据的处理时，模式匹配仍然是一个常用手段。本文面向模式匹配计算任务，采用软硬件协同的设计方法，设计实现了一款自动机专用处理器。同时，我们提供了一个轻量级的解决方案，使得现有的通用处理器系统可以被方便的拓展为异构计算片上系统，从而实现高效率、高吞吐的多模式匹配计算。

在第 3 章，我们设计实现了一款专用的自动机处理器——LAP。该处理器非常的轻量，只需要非常小的存储器资源和其他硬件资源。但是，LAP 可以进行非常高效和高性能的模式匹配计算。我们首先介绍了 LAP 支持的 7 条硬件指令，并优化了其中用于支持 ADFA 模型的两条指令。基于优化后的 LAP 指令集，我们设计了四阶段的 LAP 流水线。通过使用细粒度多线程技术，我们消除了流水线内部的冲突。最终，我们实现了一个高效的面向多模式匹配计算的无停顿硬件流水线。不仅如此，我们为该处理器实现了配套的编译软件。该编译器将一组正则表达式编译为一个 ADFA 模型，并进一步生成 LAP 可以运行的二进制文件。我们只需要将该二进制文件加载进 LAP 的程序存储器，并将字符流文件加载进 LAP 的输入缓存中，便可以启动 LAP 核心，让 LAP 独立的运行 ADFA 模型。LAP 核心将顺序地读取和处理输入字符流，并在这些字符流匹配某个正则表达式时向外界汇报该匹配结果。最后，我们对 LAP 核心进行了资源、功耗以及性能上的评估。我们得出结论，在低成本的 Artix-7 FPGA 上，我们可以部署 5 个 LAP 核心；增加 0.6W 的动态功耗实现 9.5 Gbps 的额外处理吞吐。通过使用 ADFA 模型和高效的 Split EffCLiP 打包算法，我们可以生成紧致高效的 LAP 二进制程序。相比于业界两款知名的模式匹配处理器 RegX 和 AP，LAP 只需要不到 1/8 存储器空间用于存储其二进制程序。不仅如此，LAP 优化了支持 ADFA 模型的两条指令，同时在微体系结构方面做了调整。通过这样的方式，LAP 相比于其他基于 ADFA 模型的自动机处理器实现了 32% 到 91% 的处理速度的提升。

在第 4 章，我们在 Zedboard 上基于 LAP 核心和 CPU 系统构建了完整的片上异构系统。我们将 LAP 的对外接口封装为标准的 AXI4 协议，分别用于传输二进制程序、输入字符流、LAP 控制和匹配结果获取。随后，我们将 LAP 核心导出为 Xilinx 的 IP 核，并通过 Block Design 将 LAP 与 ARM Cortex-A9 处理器集成到一起，构成了完整的异构 SoC。同时，我们在该 SoC 的 CPU 系统上部署了 Linux 操作系统。基于 Linux 操作系统，我们开发了 C 语言程序，直接操作该 SoC 的物

理内存。该 C 语言程序包含多个驱动 LAP 核心的函数，可用于进行 LAP 和 CPU 间的数据交互，以及进行 CPU 对 LAP 的监控和控制。最终，我们对 LAP_SoC 进行了全面的评估，包括其资源占用、功耗和多模式匹配的性能。实验表明，在 Zedboard 的双核 Cortex-A9 处理器系统上添加一个 LAP 核心只会额外增加 5% 的功耗，同时将实现 40+ 倍的性能提升（面向多模式匹配计算）。同时，单个 LAP 核心的面向多模式匹配任务的处理速度也是桌面系统上的 Intel core i5-6500 处理器的 3 倍。这充分展现了 LAP_SoC 异构计算系统的有效性。

5.2 未来展望

本文实现了基于 ADFA 模型的自动机处理器 LAP，同时基于该处理器核心实现了异构计算系统 LAP_SoC。相比于已有的解决方案，它们展现出了一些优势。但是，它们也仍然需要进一步改进和扩充。

解决状态爆炸的问题：在同时匹配大量带有“.”（dot star）结构的正则表达式时，生成的 DFA 模型的状态数将会发生指数爆炸。为了解决这样的问题，LAP 仍然兼容了 NFA 模型。在第 2.1.6 小节我们简单介绍过几类新型的自动机模型，这些自动机模型可以很好地处理“状态爆炸”的问题。XFA^[34] 模型和 RegX^[14] 就很好地解决了状态爆炸的问题。这些新型的自动机模型，除了可以进行普通的状态切换外，还可以对其私有的寄存器组的各个比特进行读、写和检查。如果进一步拓展 LAP 的指令集，使其可以支持这样的寄存器操作，将会使得 LAP 可以在不使用 NFA 模型的情况下更好的处理更复杂的正则表达式集合。同时，这将进一步简化 LAP 的 ASS 的硬件结构，使得 LAP 的功耗和资源占用更低。

多 LAP 核心系统的构建：本文为了演示如何基于 LAP 构建完整的异构 SoC，在第 4 章构建了单 LAP 核心 + CPU 系统的片上系统。该 SoC 仍然有许多可以改进的地方。第一，该 SoC 的多个 IP 核（包括 BRAM Controller 和各类 AXI 总线）主频被物理原因所限制，无法进一步提升。因此，该 SoC 的全局时钟信号只能达到 100 MHz。这导致了 LAP 在 LAP_SoC 中主频只能部署为 100 MHz。事实上，LAP 可以采用多时钟域设计。LAP 核心的主频可以运行在 263 MHz，而将其 AXI 接口电路运行在 100 MHz。两者可以通过异步 FIFO 实现数据的一致传输。这将进一步提升 LAP_SoC 的计算吞吐。同时，在多 LAP 核心的系统中，这些 LAP 核心如何复用 DMA 控制器，也是一个值得考虑的问题。

支持更广泛的应用领域：基于 LAP 编译软件，我们可以实现对正则表达式的匹配。基于该核心功能，我们可以进一步开发编译器，使得 LAP 可以处理更多实际的任务：例如 CSV/Json/XML 文件解析，网络流量监控和基因串匹配等。

参考文献

- [1] GANDOMI A, HAIDER M. Beyond the hype: Big data concepts, methods, and analytics [J/OL]. *Int. J. Inf. Manag.*, 2015, 35(2): 137-144. <https://doi.org/10.1016/j.ijinfomgt.2014.10.007>.
- [2] LANEY D. 3-d data management: Controlling data volume, velocity, and variety[Z]. 2001.
- [3] KENNETH C. The economist, data, data everywhere: A special report on managing information[EB/OL]. 2010. <http://www.economist.com/node/15557443>.
- [4] PAN Y, ZHANG Y, CHIU K. Simultaneous transducers for data-parallel xml parsing[C]//2008 IEEE International Symposium on Parallel and Distributed Processing. 2008.
- [5] Bo C, Wang K, Fox J J, et al. Entity resolution acceleration using the automata processor [C/OL]//2016 IEEE International Conference on Big Data (Big Data). 2016: 311-318. DOI: 10.1109/BigData.2016.7840617.
- [6] Wang K, Qi Y, Fox J J, et al. Association rule mining with the micron automata processor [C/OL]//2015 IEEE International Parallel and Distributed Processing Symposium. 2015: 689-699. DOI: 10.1109/IPDPS.2015.101.
- [7] YU F, CHEN Z, DIAO Y, et al. Fast and memory-efficient regular expression matching for deep packet inspection[C/OL]//ANCS '06: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems. New York, NY, USA: Association for Computing Machinery, 2006: 93-102. <https://doi.org/10.1145/1185347.1185360>.
- [8] Roy I, Aluru S. Finding motifs in biological sequences using the micron automata processor [C/OL]//2014 IEEE 28th International Parallel and Distributed Processing Symposium. 2014: 415-424. DOI: 10.1109/IPDPS.2014.51.
- [9] WANG M H, CANCELO G, GREEN C, et al. Using the automata processor for fast pattern recognition in high energy physics experiments—a proof of concept[J/OL]. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 2016, 832: 219-230. <https://www.sciencedirect.com/science/article/pii/S0168900216306891>. DOI: <https://doi.org/10.1016/j.nima.2016.06.119>.
- [10] KNUTH D E, JR. J H M, PRATT V R. Fast pattern matching in strings[J/OL]. *SIAM J. Comput.*, 1977, 6(2): 323-350. <https://doi.org/10.1137/0206024>.
- [11] BOYER R S, MOORE J S. A fast string searching algorithm[J/OL]. *Commun. ACM*, 1977, 20(10): 762-772. <https://doi.org/10.1145/359842.359859>.
- [12] AHO A V, CORASICK M J. Efficient string matching: An aid to bibliographic search[J/OL]. *Commun. ACM*, 1975, 18(6): 333-340. <https://doi.org/10.1145/360825.360855>.

- [13] HOPCROFT J E, ULLMAN J D. Addison-wesley series in computer science and information processing: Formal languages and their relation to automata[M/OL]. Addison-Wesley, 1969. <https://www.worldcat.org/oclc/00005012>.
- [14] VAN LUNTEREN J, HAGLEITNER C, HEIL T, et al. Designing a programmable wire-speed regular-expression matching accelerator[C]//45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012. 2012: 461-472.
- [15] DLUGOSCH P, BROWN D, GLENDENNING P, et al. An efficient and scalable semiconductor architecture for parallel automata processing[J/OL]. IEEE Trans. Parallel Distrib. Syst., 2014, 25(12): 3088-3098. <https://doi.org/10.1109/TPDS.2014.8>.
- [16] WADDEN J, ANGSTADT K, SKADRON K. Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures[C/OL]//IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018. IEEE Computer Society, 2018: 749-761. <https://doi.org/10.1109/HPCA.2018.00069>.
- [17] SUBRAMANIYAN A, WANG J, BALASUBRAMANIAN E R M, et al. Cache automaton [C]//Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18. 2017: 259-272.
- [18] SADREDINI E, RAHIMI R, VERMA V, et al. eap: A scalable and efficient in-memory accelerator for automata processing[C]//Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019. 2019: 87-99.
- [19] LIU H, IBRAHIM M A, KAYIRAN O, et al. Architectural support for efficient large-scale automata processing[C/OL]//51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018. IEEE Computer Society, 2018: 908-920. <https://doi.org/10.1109/MICRO.2018.00078>.
- [20] MYTKOWICZ T, MUSUVATHI M, SCHULTE W. Data-parallel finite-state machines [C/OL]//BALASUBRAMONIAN R, DAVIS A, ADVE S V. Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014. ACM, 2014: 529-542. <https://doi.org/10.1145/2541940.2541988>.
- [21] SUBRAMANIYAN A, DAS R. Parallel automata processor[C/OL]//Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017. ACM, 2017: 600-612. <https://dl.acm.org/citation.cfm?id=3080207>.
- [22] ZHUO Y, CHENG J, LUO Q, et al. CSE: parallel finite state machines with convergence set

- enumeration[C/OL]//51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018. IEEE Computer Society, 2018: 29-41. <https://doi.org/10.1109/MICRO.2018.00012>.
- [23] ANGSTADT K, SUBRAMANIYAN A, SADREDINI E, et al. ASPEN: A scalable in-sram architecture for pushdown automata[C/OL]//51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018. IEEE Computer Society, 2018: 921-932. <https://doi.org/10.1109/MICRO.2018.00079>.
- [24] BO C, DANG V, SADREDINI E, et al. Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms[C/OL]//IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018. IEEE Computer Society, 2018: 737-748. <https://doi.org/10.1109/HPCA.2018.00068>.
- [25] GOGTE V, KOLLI A, CAFARELLA M J, et al. HARE: hardware accelerator for regular expressions[C/OL]//49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016. IEEE Computer Society, 2016: 44:1-44:12. <https://doi.org/10.1109/MICRO.2016.7783747>.
- [26] TANDON P, SLEIMAN F M, CAFARELLA M J, et al. HAWK: hardware support for unstructured log processing[C/OL]//32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016. IEEE Computer Society, 2016: 469-480. <https://doi.org/10.1109/ICDE.2016.7498263>.
- [27] FANG Y, HOANG T T, BECCHI M, et al. Fast support for unstructured data processing: the unified automata processor[C]//Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9. 2015: 533-545.
- [28] FANG Y, ZOU C, ELMORE A J, et al. UDP: a programmable accelerator for extract-transform-load workloads and more[C]//Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017. 2017: 55-68.
- [29] AHO A V, LAM M S, SETHI R, et al. Compilers: Principles, techniques, and tools (2nd edition)[M]. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [30] KUMAR S, DHARMAPURIKAR S, YU F, et al. Algorithms to accelerate multiple regular expressions matching for deep packet inspection[C]//Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Pisa, Italy, September 11-15, 2006. 2006: 339-350.
- [31] KRUSKAL J B. On the shortest spanning subtree of a graph and the traveling salesman problem[J]. Proceedings of the American Mathematical society, 1956, 7(1): 48-50.

- [32] BECCHI M, CROWLEY P. A-DFA: A time- and space-efficient DFA compression algorithm for fast regular expression evaluation[J/OL]. TACO, 2013, 10(1): 4:1-4:26. <https://doi.org/10.1145/2445572.2445576>.
- [33] FICARA D, PIETRO A D, GIORDANO S, et al. Differential encoding of dfas for fast regular expression matching[J/OL]. IEEE/ACM Trans. Netw., 2011, 19(3): 683-694. <https://doi.org/10.1109/TNET.2010.2089639>.
- [34] SMITH R, ESTAN C, JHA S. XFA: faster signature matching with extended automata[C/OL]// 2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA. IEEE Computer Society, 2008: 187-201. <https://doi.org/10.1109/SP.2008.14>.
- [35] SMITH R, ESTAN C, JHA S, et al. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata[C/OL]//BAHL V, WETHERALL D, SAVAGE S, et al. Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008. ACM, 2008: 207-218. <https://doi.org/10.1145/1402958.1402983>.
- [36] KUMAR S, CHANDRASEKARAN B, TURNER J S, et al. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia[C/OL]//YAVATKAR R, GRUNWALD D, RAMAKRISHNAN K K. Proceedings of the 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2007, Orlando, Florida, USA, December 3-4, 2007. ACM, 2007: 155-164. <https://doi.org/10.1145/1323548.1323574>.
- [37] BECCHI M, CROWLEY P. A hybrid finite automaton for practical deep packet inspection[C/OL]//KUROSE J, SCHULZRINNE H. Proceedings of the 2007 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2007, New York, NY, USA, December 10-13, 2007. ACM, 2007: 1. <https://doi.org/10.1145/1364654.1364656>.
- [38] BRODIE B C, TAYLOR D E, CYTRON R K. A scalable architecture for high-throughput regular-expression pattern matching[C/OL]//33rd International Symposium on Computer Architecture (ISCA 2006), June 17-21, 2006, Boston, MA, USA. IEEE Computer Society, 2006: 191-202. <https://doi.org/10.1109/ISCA.2006.7>.
- [39] VAN LUNTEREN J. Scalable DFA compilation for high-performance regular-expression matching[C/OL]//STUIJK S. Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPEs 2016, Sankt Goar, Germany, May 23-25, 2016. ACM, 2016: 10-19. <https://doi.org/10.1145/2906363.2907053>.
- [40] Zynq-7000 soc data sheet: Overview[EB/OL]. 2014. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.
- [41] Amba axi and ace protocol specification[EB/OL]. 2021. <https://developer.arm.com/documentation/ih0022/hc>.

- [42] Vivado design suite: Vivado axi referenc[EB/OL]. 2017. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.
- [43] Axi dma v7.1:logicore ip product guide[EB/OL]. 2019. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.
- [44] FANG Y, CHIEN A A. Udp system interface and lane isa definition[R]. <https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2017-05>, 2017.
- [45] LAUDON J, GUPTA A, HOROWITZ M. Interleaving: A multithreading technique targeting multiprocessors and workstations[C]//ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994. 1994: 308-318.
- [46] BECCHI M, FRANKLIN M A, CROWLEY P. A workload for evaluating deep packet inspection architectures[C]//4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008. 2008: 79-89.
- [47] FANG Y, LEHANE A, CHIEN A A. Effclip: Efficient coupled-linear packing for finite automata[R]. University of Chicago Technical Report, TR-2015-05, 2015.
- [48] Vivado design suite user guide: Using the vivado ide[EB/OL]. 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug893-vivado-ide.pdf.
- [49] Axi block ram (bram) controller v4.1[EB/OL]. 2019. https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_1/pg078-axi-bram-ctrl.pdf.
- [50] Petalinux tools documentation reference guide[EB/OL]. 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1144-petalinux-tools-reference-guide.pdf.
- [51] BECCHI M, WISEMAN C, CROWLEY P. Evaluating regular expression matching engines on network and general purpose processors[C/OL]//ONUFYK P Z, RAMAKRISHNAN K K, CROWLEY P, et al. Proceedings of the 2009 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2009, Princeton, New Jersey, USA, October 19-20, 2009. ACM, 2009: 30-39. <https://doi.org/10.1145/1882486.1882495>.

致 谢

入学时的情景还历历在目，三年研究生生活却即将落幕。在此，我想由衷的感谢我的母校中国科大，因为它改变了我的一生。在科大的七年，我努力地在成为一个数理基础扎实的人。更重要的是，这几年我结识了一群非常有趣的朋友，也完成了一些非常有意义的事。我也由衷的感谢这三年每一个关怀、帮助过我的人。你们编织了我在科大无数难忘的回忆，也成为了我记忆里不可或缺的元素。

我想首先感谢我的研究生导师，周学海教授。周老师治学严谨、为人宽厚，是我终身学习的目标。在他的影响下，我对于每件事都是全力以赴，不敢懈怠。在论文选题上，周老师给予了我非常大的帮助和支持。正是在他的引导下，我才找到了自己的研究方向，并最终完成了本文。在我的课题进展初期，周老师耐心听取我的每一次报告，并提出了非常多建设性的意见。不仅如此，周老师还不辞辛劳的为我仔细修改本论文，提出了几十项修改意见。再次感谢周老师这些年的照顾与指导！我还想感谢我的第二导师，宫磊老师。宫磊博士是一位非常有趣、也精于学术的人。在我完成小论文初稿后，宫磊老师认真仔细的帮我调整文章结构和梳理文章脉络。他一次次的帮我批阅论文稿件，并提出相应的修改建议。通过和他的反复讨论，我的小论文得到了充分地改进。他是我入门学术写作的引路人。同时，我也要感谢同实验室的王超副教授和陈香兰老师。在王老师的指导下，我学习了如何去组织学术论文，以及如何更好的呈现实验结果。陈老师不辞辛劳的管理着实验室的一些日常工作，帮我们解决了很多问题。

然后，我想感谢实验室的同学们。感谢章博师兄、赵彩旭师姐、王腾师兄、程玉明师兄和余辉煌师兄。在他们的帮助下，我很快适应了苏州的生活。我还要感谢我研二研三时期在苏州的三位室友，凌康志、席星宇和朱骁睿。在苏州的两年，感谢你们的陪伴与理解。在你们的帮助和支持下，我们的寝室才会如此整洁和温馨。除此以外，感谢和我同级的硕士和博士同学们，王轩、娄文启、吴昊、苑福利、王延龄、齐豪和易琦。有了他们的陪伴，我在苏州的生活才会如此丰富多彩。在王轩同学的帮助下，我高效快速的解决了许多代码编写相关的问题。还有认识一年的李玉婷师妹、姚袭欣师妹、汪超师弟、李松松师弟和刘翀师弟，我也要感谢你们。

我还要感谢我的老朋友们。韩浩宇和王浩辰是我大一到研一期间，连续五年的室友。这些年，我们一直在互相帮助，共同成长。我也要感谢我本科时期的另一位室友——郑俊。感谢他本科时期给我们寝室带来的快乐，也要感谢他在苏州给予我的两次重大帮助。感谢郑子涵同学，这些年在学习上给了我非常大的帮助。感谢王珺同学，他是我非常信赖的朋友。感谢刘梦境和于晓静，他们给我带

来了非常多欢乐的时光。同时，我也要感谢陈旻宇、辛媛、吴善驰、吴方唯。他们帮我解决了许多问题。

最后，感谢我的家人和我的女朋友权纯丽。我的爸爸和妈妈给了我无微不至的关怀与爱，让我在回家后能享受到舒适的生活。他们无条件的支持我的所有选择，给了我非常多的力量。他们总是督促我要注意身体，这让我内心一直非常温暖。权纯丽毫无缺席的陪我度过了研究生三年。她总是耐心的倾听着我的心声，陪我度过了每一个难过的时刻，也让我在开心的时候有人可以分享。再次感谢他们！他们是我奋力前进的不竭动力！

值此分离之际，再次感谢上述所有人，以及所有本文没有提及但是也帮助过我的人。正是因为你们，我的求学之路才会如此坦荡。谢谢你们，祝大家好运！

在读期间发表的学术论文与取得的研究成果

已发表论文

1. **XIA H**, GONG L, WANG C, et al. Lap: A lightweight automata processor for pattern matching tasks[C]//Proceedings of the 2021 Design, Automation & Test in Europe (DATE). 2021 {CCF B 类国际会议}



USTC

中国科学技术大学硕士学位论文
