多域网包分类算法研究

（申请清华大学工学博士学位论文）

培 养 单 位： 自动化系

学　　科： 控制科学与工程

研 究 生： 亓 亚 烜

指 导 教 师： 李　　军　研究员

二〇一一年四月

多域网包分类算法研究

亓 亚 烜

Multi-dimensional Packet Classification Algorithms

Dissertation Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Doctor of Engineering

by

QI Yaxuan

(Control Science and Engineering)

Dissertation Supervisor : Professor LI Jun

April, 2011

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后遵守此规定）

作者签名：_____ 导师签名：_____

日　　期：_____ 日　　期：_____

摘　要

随着互联网架构的不断演进以及互联网新应用的不断涌现，基于单一 IP 地址域的传统路由技术已经不能满足日益增长的网络业务和网络安全需求。例如，多媒体业务需要的服务质量保证、企业网络需要的访问控制等均难以通过传统路由转发技术来实现。由于多域网包分类能够依据网包信息对网络流量进行细粒度的划分，该技术已在新一代路由器、安全网关及流量控制系统中得到了广泛应用。与此同时，随着云计算、物联网、移动互联网等前沿技术的发展，高性能网包分类已成为下一代互联网发展和演进中的研究热点。为了满足不断增长的网络带宽和日益丰富的网络业务需求，本文从理论分析、算法设计和系统实现三个层面对高性能多域网包分类算法进行了深入的研究。

在理论分析层面，本文将网包分类问题的数学解法归纳为规则投影区间查找算法和网包搜索空间分解算法两类，根据查找算法中的维度、回溯和复制等约束条件对不同数学解法的复杂度进行了进一步的归纳，从而提出了网包分类算法设计的理论依据及评价准则。在算法设计层面，本文分别设计了基于投影区间查找和搜索空间分解的 HyperSplit 和 AggreCuts 算法。HyperSplit 算法利用启发式方法构建多域区间树，解决了现有区间查找算法内存使用过高的问题。AggreCuts 则利用深度为常数的决策树结构，结合压缩和编码技术解决了现有空间分解算法时间性能不可控的问题。作为多域网包分类算法研究的拓展，本文还提出了基于网包载荷分类的 FEACAN 算法，通过状态聚类和二维压缩技术提高了现有深度检测算法的空间性能。在系统实现层面，本文所提出的算法分别实现于多核网络处理器平台以及 FPGA 硬件平台上。其中，AggreCuts 算法和 HyperSplit 算法分别在多核网络处理器平台上实现了 10Gbps 和 8Gbps 的吞吐率，而 HyperSplit 算法和 FEACAN 算法分别在 FPGA 硬件平台上实现了超过 100Gbps 和 40Gbps 的吞吐率。

综上所述，本文的主要贡献是从理论、算法、系统三个层面对高性能网包分类算法进行分析、归纳和创新。理论依据是算法的研究的基础；新算法可以满足高性能网包分类应用的需求；基于多核网络处理器和 FPGA 的算法实现则验证了理论和算法的正确性，并为高性能网包分类系统和芯片的设计提供了参考依据。

关键词：网包分类；模式匹配；网络处理器；FPGA

Abstract

With the evolution of Internet architecture and the flourishing of Internet applications, traditional IP forwarding is not capable of meeting the increasing demands for network services and security. For example, multimedia applications require good quality of services, while enterprise network needs secure access control. Because multi-dimensional packet classification can distinguish network traffic in fine grain, it has been widely used in modern routers, firewalls and traffic management systems. In this dissertation, multi-dimensional packet classification algorithms are studied mainly in three aspects: theoretical analysis, algorithm design and system implementation.

In theory, the packet classification problem can be seen as a point location problem in computational geometry. This dissertation first categorizes theoretical solutions into segment-search algorithms and space-decomposition algorithms. Different algorithms are then analyzed according to their classification strategies. In contrast to theoretical analysis, practical performance metrics are also summarized for an objective algorithm evaluation.

Three novel packet classification algorithms, HyperSplit, AggreCuts and FEACAN are presented in this dissertation. HyperSplit is based on segment search and uses multiple heuristics to reduce memory usage. AggreCuts, a space decomposition algorithm, achieves deterministic worst-case search time with fixed decision tree stride. As an extension to multi-dimensional packet classification, the FEACAN algorithm supports regular expression matching based packet classification. It uses a 2-D compression technique to reduce size of DFA transition table without significant loss of classification speed.

For performance evaluation, all algorithms presented in this dissertation are implemented on both multi-core network processors and FPGA hardware platform. The maximum throughput achieved with Intel IXP2850 and Cavium OCTEON3860 multi-core network processors are 10Gbps and 8Gbps respectively. In comparison, over 100Gbps throughput is achieved with the FPGA implementation.

In conclusion, this dissertation studies the multi-dimensional packet classification problem by theory analysis, algorithm design and system implementation. The computational geometry based theory analysis is the basis of this research. The

proposed algorithms achieve better performance than existing work. The system implementation on multi-core and FPGA platforms provide reference design for high-performance network processing chips and systems.

Key words: packet classification; pattern matching; multi-core; FPGA

目 录

主要符号对照表

p	网包（Packet）
r	规则（Rule）
R	规则集合（Rule set）
d	维度（Dimensionality）
W	位宽（Bit Width）
IP	Internet 协议（Internet Protocol）
TCP	传输控制协议（Transmission Control Protocol）
UDP	用户数据包协议（User Datagram Protocol）
HiCuts	层级智能切分算法 （Hierarchical Intelligent Cuttings）
HSM	层级空间映射算法（Hierarchical Space Mapping）
RFC	递归网流分类算法 （Recursive Flow Classification）
DFA	确定性有限状态自动机 （Deterministic Finite Automata）
NFA	非确定性有限状态自动机 （Non-deterministic Finite Automata）
FW	防火墙（Firewall）
ACL	访问控制列表（Access Control List）
QoS	服务质量（Quality of Service）
IDS	入侵检测系统（Intrusion Detection System）
IPS	入侵防御系统（Intrusion Prevention System）
UTM	统一威胁管理（Unified Threat Management）
NP	网络处理器（Network Processor）
FPGA	现场可编程门阵列（Field Programmable Gate Array）
ASIC	专用集成电路（Application Specific Integrated Circuit）
RAM	随机存储器（Random Access Memory）
SRAM	静态随机存储器（Static RAM）
DRAM	动态随机存储器（Dynamic RAM）
BRAM	块随机存储器（Block RAM）
CAM	内容可寻址存储器（Content Addressable Memory）
TCAM	三态内容可寻址存储器（Ternary CAM）
Mbps	兆比特每秒 （Megabit per second）

Gbps	千兆比特每秒 （Gigabit per second）
MB	兆字节（Megabyte）
GB	千兆字节（Gigabit）
MHz	兆赫兹（Megahertz）

第 1 章　引言

1.1　研究意义

随着互联网架构的不断演进以及互联网新应用的不断涌现，基于单一 IP 地址域的传统路由技术已经不能满足日益增长的网络业务和网络安全需求。例如，多媒体业务需要的实时性保证、企业网络需要的安全防护等均难以通过传统路由转发技术来实现。由于多域网包分类能够依据网包信息对网络流量进行细粒度的划分，该技术已在路由器、安全网关及流量控制系统等各类网络设备中得到了广泛应用[1-4]。与此同时，随着业务感知网络、数据中心网络以及软件定义网络等前沿网络技术的发展，高性能网包分类技术已成为下一代互联网发展和演进中的研究热点[5-8]。

随着云计算、移动互联网、宽带多媒体等互联网业务的发展与普及，下一代互联网需要满足融合性、安全性、可控性等多方面需求。网包分类作为支撑业务融合、网络安全、流量监控等网络功能的核心技术，具有多方面的重要研究意义（参考图 1.1）。

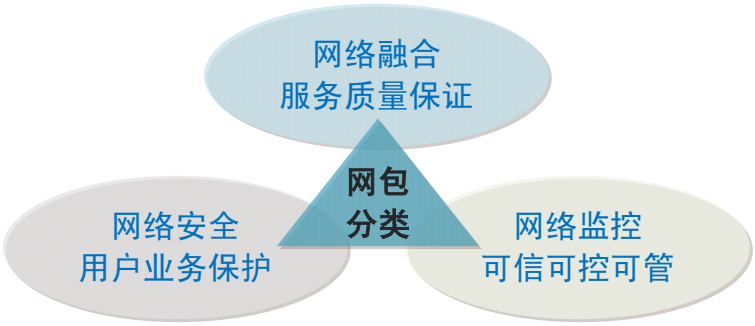


图 1.1　网包分类的研究意义

- **网包分类对网络融合具有重要意义**

在网络融合的大趋势下，如何提高互联网基础设施的多业务承载力直接影响着我国下一代互联网的发展方向。多业务融合网络的一个基本需求就是网络设备能够对流量依据业务进行分类，使得不同业务的流量能够在融合网络中满足各自的认证计费、带宽需求、服务质量保证。因此，高性能网包分类技术的研究将有力推动网络融合的业务发展及网络基础设施建设。

- **网包分类在网络安全中有广泛应用**

网络安全是保障互联网及其业务健康发展的基础。无论在企业网络、移动网络还是数据中心，网络安全设备均已得到广泛的部署。主要的网络安全设备，例如防火墙、入侵防御系统等，均使用安全策略对网络用户进行保护，而安全策略的匹配过程实际就是基于规则查找的网包分类过程。因此，网包分类在网络安全设备中有广泛应用，研究网包分类理论和算法对网络安全设备性能的提升具有重要意义。

- **网包分类在网络监控中有重要价值**

网络监控技术的发展对构建可信可控可管的下一代互联网起着重要作用。随着网路带宽和业务的不断增加，网络监控设备必须具备对海量网络数据进行细粒度分析和处理的能力。由于网包分类技术可以提供基于流量、内容、和业务的全息统计信息，使得网络监控设备可以据此实现网络测量、业务管理、舆情分析等大规模网络监控应用。因此，网包分类技术的相关研究，尤其是高性能细粒度分类算法的研究，在网络监控中具有重要价值。

综上所述，网包分类技术在业务融合、网络安全、及网络监控等诸多领域都有广泛的应用，是关系到下一代互联网发展的核心技术。因此，研究和分析网包分类问题的理论基础，设计并实现高性能实用网包分类算法，具有极为重要的科学价值和应用场景。

1.2　研究内容

多域网包分类是指依据网包中的多域信息，按照给定规则集合对网包进行分类的过程。图 1.2 描述了一个典型的基于 IPv4 五元组（5-tuple）的网包分类系统。该五元组包括网包包头中的源/目的 IP 地址域（各 32 比特），源/目的传输层端口域（各 16 比特），以及传输层协议域（8 比特）。网包分类规则（如表 1.1 例）存储于网包分类系统中。网包分类引擎根据输入网包的五元组信息与分类规则进行匹配。网包分类系统将依据匹配规则的决策（action）对输入网包进行相应的处理，例如接受转发（ACCEPT）、拒绝转发（DENY）、重置连接（RESET）或丢弃网包（DROP）。

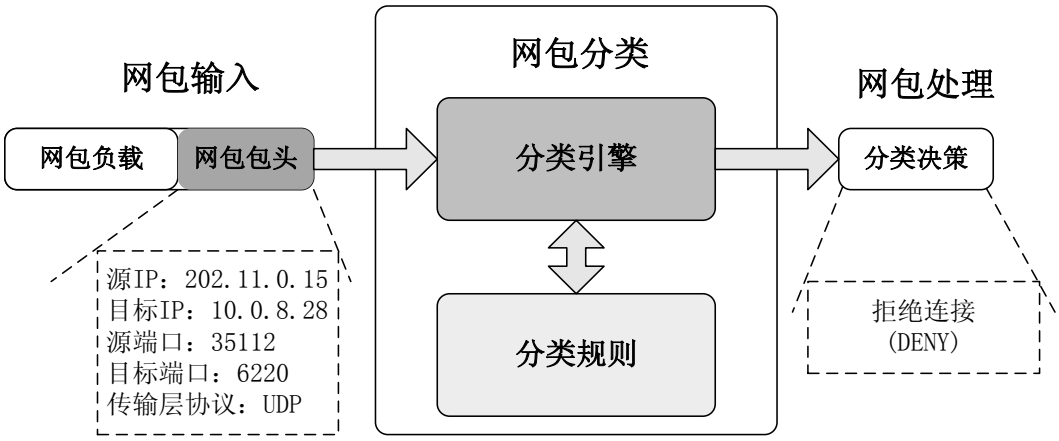


图 1.2　网包分类系统

表 1.1　网包分类规则示例

规则	目标 IP	源 IP	目标端口	源端口	L4 协议	优先级	决策
r_1	10.0.8.28/32	64.10.8.20/32	80	0~65535	TCP	1	RESET
r_2	10.0.8.28/32	0.0.0.0/0	53	0~65535	TCP	2	ACCEPT
r_3	10.0.0.0/16	202.11.0.15/32	0~65535	0~65535	UDP	3	DENY
r_4	10.0.0.0/16	0.0.0.0/0	0~65535	0~65535	TCP	4	DENY
r_5	10.1.3.20/32	0.0.0.0/0	80	6110~6112	UDP	11	ACCEPT
r_6	10.1.3.0/24	0.0.0.0/0	0~65535	1024~65535	ANY	12	ACCEPT
r_7	0.0.0.0/0	0.0.0.0/0	0~65535	0~65535	ANY	999	DROP

多域网包分类算法通过对给定规则集合进行预处理，生成高效的查找数据结构，能够有效提高网包分类系统的性能。多域网包分类算法的研究内容主要包括：

- **算法理论**：理论依据对于高性能多域网包分类算法的设计具有指导性作用。由于网包分类问题是计算几何中的多域空间点定位问题，算法的理论依据研究包括点定位问题的数学解法及其理论复杂度分析。其中，数学解法是所有网包分类算法设计中需要遵循的基本方法，而理论复杂度分析则是网包分类算法性能的重要依据。
- **算法设计**：建立在点定位数学解法之上，算法设计利用启发式方法（heuristics）进一步挖掘规则集合的特性来优化网包分类的数据结构。算法设计的优化目标包括提高分类速率、降低存储空间、支持快速更新等。对不同类型的规则集合的自适应性也是算法设计中的重要研究内容。
- **算法实现**：为了满足高性能网包分类系统对分类速率的要求，网包分类算法需要基于硬件的实现。由于硬件平台对计算单元、存储单元都有严格的约束，网包分类算法需要针对不同的平台进行改进和优化。这方面的研究包括算法并行化、内存管理、以及规则更新等。

1.3　国内外研究现状

网包分类算法在国内外学术界及工业界都有广泛和深入的研究。无论在核心网络中还是在边界网关上，业务感知路由器、应用层防火墙以及入侵检测系统等网络设备都普遍使用网包分类来满足业务处理的需求。下面依据不同的研究内容，从算法理论、算法设计和算法实现三个方面介绍国内外的研究现状及发展动态。

从算法理论层面来看，近年来网包分类问题的理论方法研究未取得重大进展。以加州大学圣地亚哥分校、斯坦福大学、加州大学圣地亚哥分校、清华大学以及微软研究院为主导的网包分类算法研究[9-19, 31, 33-35]，均以 Overmars 等提出的使用回溯查找的多域区间树算法作为网包分类问题的理论最优解[20]。由于该算法的时间复杂度不能满足大多数网包分类系统的性能要求，工业界流行的网包分类技术并未普遍应用这些算法，而大多采用了不使用回溯查找的方法以提高处理效率。因此，目前的网包分类算法依然缺乏坚实的数学理论依据。

从算法设计层面来看，现有网包分类算法虽然能够充分发掘实际网包分类规则的特性，利用启发式方法构建优化的查找数据结构，却往往偏重于单一性能指标的提高。例如 RFC 算法具有极高的分类速率，却需要大量的存储空间[12]。与之相反，EffiCuts 算法虽然存储空间得到了极大优化，但存储访问次数却增加了数倍之多[21]。因此，已有算法缺乏对时间和空间性能的全局优化。

从算法实现层面来看，多数的研究工作依赖于专用处理芯片（例如 ASIC）或专用存储芯片（TCAM）[22-29]。基于 ASIC 的网包分类系统实现成本高、设计周期长且难以复用或更新。而基于 TCAM 的网包分类算法则由于前缀到范围转换以及自身功耗等问题，难以处理复杂的规则集合并广泛用于各类网络处理设备上[15]。随着芯片技术的发展，基于多核网络处理器（multi-core network processor）和 FPGA（Field Programmable Gate Array，现场可编程门阵列）的网包分类算法实现成为了当前的研究热点[30-43, 96-100]。

综上所述，当前的网包分类算法的研究缺乏可靠的理论依据和全局的优化设计。随着网络带宽的逐步提高以及网络应用的层出不穷，通过改进现有网包分类算法并堆叠功能单一的专用芯片将无法满足下一代网络处理设备高性能、多功能和低功耗的需求。要从根本上解决网包分类问题必须建立坚实的理论依据并依此设计全局（功能、性能、功耗）优化的分类算法。网包分类算法的实现则需要尽可能提高系统计算和存储资源的利用率，最终设计和实现一体化高性能的网包分类算法。

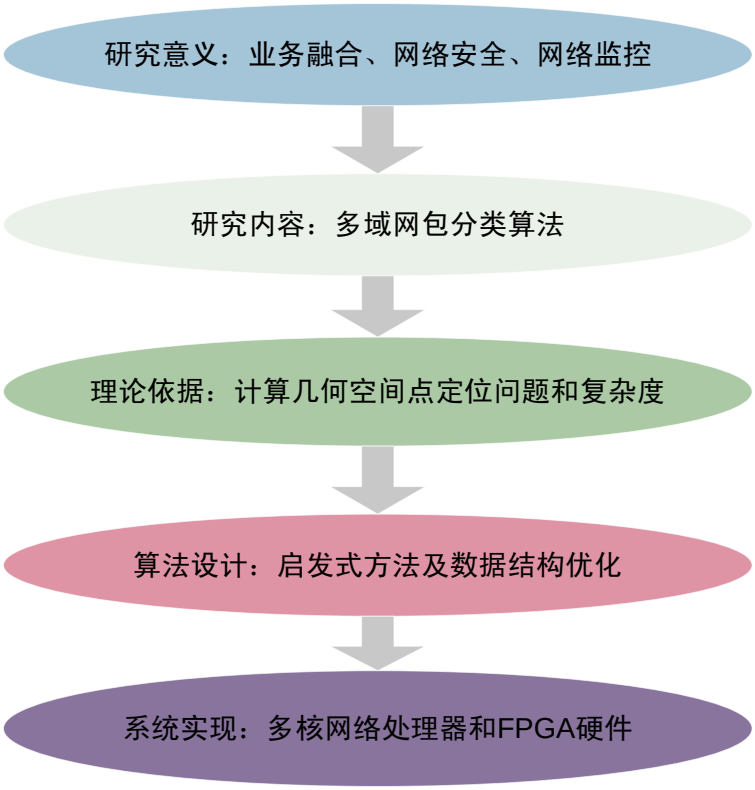


图 1.3　研究路线

1.4　研究方法

多域网包分类问题在计算几何学中已被证明具有极高的时间和空间复杂度，即在最坏情况（worst case）下无法同时达到高速率的查找和低消耗的内存[12, 44]。然而，在实际的网络应用中，一个现实的网包分类问题往往不会达到数学上的最坏情况。斯坦福大学的 Gupta 在其博士论文中指出[45]：

“理论上的复杂度告诉我们，在最坏情况下无法得到多域网包分类问题的实用的求解。所幸，我们并不需要那样去做。没有单独的任何一种算法能够适用于任何情况。因此，采用混合的求解策略或许能够结合不同方法各自的优势，从而得到更好的分类算法。”

由此可见，多域网包分类算法的研究并不是寻求单一的、适用于所有网包分类问题的最优解，而是针对不同的网包分类问题，采用不同的方法和策略来寻找对于特定问题在特定要求下的可行的解。基于前人的成果，本文采用了图 1.3 中的研究路线：首先为网包分类问题建立基于计算几何方法的理论依据，然后结合规则特性分析和启发式方法进行算法的优化设计，最后在可编程的硬件平台上实现高性能网包分类系统。

- **算法理论的研究方法**

本文的理论研究基于计算几何方法。首先将多域网包分类问题转化为计算几何中的多域空间点定位问题。然后从点定位问题的经典数学解法入手，并结合不同的约束条件，总结出不同维度下、具有不同复杂度的多域空间点定位问题的数学解法。最后，结合实际的规则集合特性，说明网包分类算法的理论依据，并提出评价网包分类算法实际性能的方法。

- **算法设计的研究方法**

本文的算法设计首先依据计算几何理论，分别从规则投影区间查找和搜索空间分解两个思路对已有算法进行分类总结。然后通过启发式方法的创新和数据结构的优化设计全新的网包分类算法。最后，将新算法从多域空间拓展到无限域空间，使其具备基于网包包头和网包载荷的全网包分类能力。

- **算法实现的研究方法**

新的网包分类算法的实际性能最终需要通过系统实现，并跟已有算法进行客观公正的比对实验来评价。本文的算法实现将基于最新的多核网络处理平台以及 FPGA 硬件平台，并使用实际网络流量和分类规则评价算法的实际性能。其中，多核网络处理器主要用于新算法和经典算法的比对实验，而 FPGA 硬件系统主要用于高性能网包分类系统的实现。

1.5　主要贡献

本文的主要贡献包括凝练网包分类算法的理论依据，设计全新的网包分类算法，以及实现高性能网包分类系统。

- **凝练网包分类算法的理论依据**

理论依据对于高性能网包分类算法的设计具有重要指导性作用。现有的网包分类算法研究工作大多依据算法数据结构进行归类、分析和比较。这种方法掩盖了不同算法之间共通的理论依据，进而导致算法在设计和优化过程中缺乏基本的数学分析手段。本文依据计算几何理论对网包分类问题的数学解法及复杂度进行了归纳，总结出广泛适用于各类网包分类算法的理论依据及性能评价方法。本文首先将网包分类问题的数学解法归纳为规则投影区间查找算法和网包搜索空间分解算法两类。然后根据查找算法中的维度、回溯和复制等约束条件对不同数学解法的复杂度进行了进一步的归纳。最后提出了网包分类算法设计的理论依据及评价准则。

- **网包分类算法设计与实现**

以网包分类的理论为依据，本文分别基于规则投影区间查找和网包搜索空间分解设计了 HyperSplit[33]和 AggreCuts[34]两个网包分类算法。HyperSplit 算法利用启发式方法构建多域区间树，解决了现有区间查找算法的内存使用过高的问题。AggreCuts 则利用深度可控的多比特 trie 结构，并结合压缩和编码技术解决了现有空间分解算法时间性能不可控的问题。此外，作为传统的基于网包包头分类的网包分类算法的拓展，本文提出了基于网包载荷分类的 FEACAN 算法[46]，通过引入中继节和纵向压缩技术实现了基于网包载荷的正则表达式匹配。实验结果表明，HyperSplit 算法比典型的区间查找算法 HSM[30, 31]提高了 20%的分类速度，同时减少了 1～2 个数量级的内存使用。AggreCuts 算法比典型的空间分解算法 HiCuts[14]提高了 2～10 倍的查找速度，同时减少了 1～2 个数量级的内存使用。FEACAN 算法在内存使用与当前最流行的正则表达式算法 D^2FA[47]相同的情况下，减少了 10 倍以上的内存访问带宽。

- **实现了基于多核网络处理器和 FPGA 的网包分类系统**

本文在多核网络处理器平台以及 FPGA 硬件平台上实现了 HSM、HyperSplit、HiCuts、AggreCuts、D^2FA、FEACAN 等多种算法。基于 Cavium OCTEON3860 [48]和 Intel IXP2850[49]多核网络处理器，本文通过实际的网络流量测试比较了不同类型算法在不同系统平台上的实际性能，并验证了新算法在分类速率，内存使用以及预处理时间上的优势。在 FPGA 硬件平台[50]上的算法实现达到了 100Gbps 的网包分类性能。实验中所用的测试方法、数据以及自行开发的算法源代码将对外公开，以推进高性能网包分类算法及其相关研究的进一步发展。

综上所述，本文的主要贡献是从理论、算法、系统三个层面对高性能网包分类技术进行总结和创新。理论依据是网包分类算法的进一步研究的基础；新算法的提出可以满足高性能网包分类应用的需求；基于多核网络处理器和 FPGA 的算法实现则为高性能网包分类系统和芯片的设计提供了参考依据。

1.6　论文章节安排

本文的后续章节安排如下：

第 2 章归纳和总结网包分类算法的理论依据。第 3 章介绍和分析区间查找算法，并提出 HyperSplit 算法。第 4 章介绍和分析空间搜索算法，并提出 AggreCuts

算法。第 5 章介绍网包分类算法的硬件实现，并提出了基于 FPGA 的网包分类算法。第 6 章介绍了基于网包载荷的网包分类算法，并提出了 FEACAN 算法。第 7 章为全文总结及未来研究的展望。

第 2 章　算法理论依据

本章首先通过数学定义，将网包分类问题归结为计算几何领域中的多域空间点定位问题；接下来介绍和分析点定位问题的多类数学解法及其理论复杂度分析；最后结合实际的网包分类规则的统计特性总结网包分类算法性能的评估标准。

2.1　网包分类问题的数学描述

网包分类问题本质上是多域空间中的点定位问题（point location problem）[20, 52]。为了便于复杂度分析，首先介绍网包分类问题中三个基本概念：网包，搜索空间及分类规则：

- 网包（packet）

网包 p 包含 d 个域的网包包头。网包包头的各个域分别表示为 $p[1], p[2], \ldots, p[d]$, 其中每个域的取值都是特定长度的比特串。例如 32 比特的 IPv4 网络层 IP 地址，16 比特的传输层端口号等。

- 搜索空间（search space）

网包 p 在 d 维空间所有可能的取值构成搜索空间 S。S 的各个维度值域不同，对于 IPv4 五元组网包分类问题，$S = [0, 2^{32} - 1] \times [0, 2^{32} - 1] \times [0, 2^{16} - 1] \times [0, 2^{16} - 1] \times [0, 2^8 - 1]$。

- 分类规则（rule）

每个分类规则包含三个部分。各域范围 $r[1], r[2], \ldots, r[d]$，规则优先级（priority）*r.pri*，和规则决策（action）*r.act*。若网包 p 与规则 r 匹配（match），则 $\forall 1 \le i \le d, p[i] \in r[i]$。对于包含 n 个规则的规则集合 $R=\{r_1, r_2, \ldots, r_n\}$，$p$ 可能与其中多个规则匹配①。

以上是网包分类问题的三个最重要的概念，下面进一步定义网包分类问题中的三种匹配方式：

① 对于防火墙等应用，通常执行匹配规则中优先级最高的那条规则所包含的规则决策。

- 严格匹配（exact match）

若规则 r 在所有域上的范围 $r[1]$, $r[2]$, …, $r[d]$均为一个数值，且 p 满足 $\forall 1 \le i \le d,\ p[i] = r[i]$，那么就称 p 与 r 严格匹配。在实际应用中，严格匹配通常用于 TCP/UDP 会话表查找（session table lookup）。

- 前缀匹配（prefix match）

若规则 r 在所有域上的范围 $r[1]$, $r[2]$, …, $r[d]$均由前缀表达，如 IP 地址 10.0.0.0/8，且 p 满足$\forall 1 \le i \le d,\ p[i] \in r[i]$，那么 p 与 r 满足前缀匹配。在实际应用中，前缀匹配广泛用于基于 IP 地址的路由查找（routing lookup）。

- 范围匹配（range match）

若规则 r 在所有域上的范围 $r[1]$, $r[2]$, …, $r[d]$均由范围表达，如端口号[1, 1024]，且 p 满足$\forall 1 \le i \le d,\ p[i] \in r[i]$，那么 p 与 r 满足范围匹配。范围匹配涵盖了前缀匹配和严格匹配，适用于本文研究的网包分类问题。

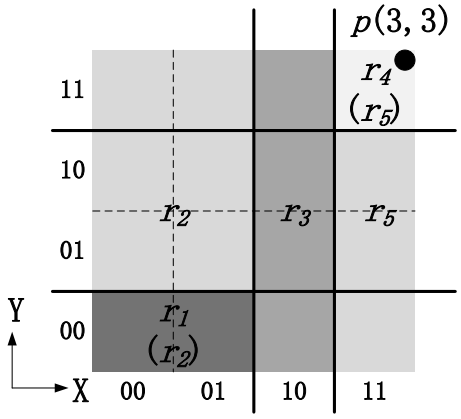


图 2.1　二维网包分类问题示例

表 2.1　二维网包分类规则示例

规则	X 域	Y 域	优先级	决策
r_1	[00,01]	[00,00]	1	DENY
r_2	[00,01]	[00,11]	2	ACCEPT
r_3	[10,10]	[00,11]	3	ACCEPT
r_4	[11,11]	[11,11]	4	DENY
r_5	[11,11]	[00,11]	5	ACCEPT

若从几何角度来理解多域范围匹配，分类规则 r 对应于搜索空间 S 中的一个超长方体（hyper-rectangle），而网包 p 则对应于 S 中的一个点。当 p 落入 r 所表示的

超长方体中时，p 即与 r 匹配。为了便于理解，图 2.1 给出了一个二维（d=2）网包分类问题的示例。其中搜索空间$S = [0,3]_X \times [0,3]_Y$，网包 p 的点坐标为$(p[x] = 3，\ p[y] = 3)$，规则集合 R={r_1, r_2,…, r_5}如表 2.1 所示。由于 p 落入 r_4 和 r_5 对应的矩形（二维超长方体）中，p 与 r_4 和 r_5 匹配。若考虑匹配优先级，则由于 r_4 的优先级高于 r_5，网包分类系统执行 $r_4.act$ 决策（DENY）。

2.2　网包分类问题的数学解法

对于空间点定位问题，线性查找算法（linear search）是最简单的解法。通过将输入网包 p 与规则集合 R 中的所有规则逐一匹配，即可得到匹配结果。对于 d 维空间的 n 个规则，线性查找的时间复杂度为$\Theta(d * n)$，空间复杂度为$\Theta(n)$。由于查找时间随规则增加而呈线性增长，线性查找仅适用于小规模的规则集合。

在计算几何领域，多域空间点定位问题存在多种数学解法。每种解法具有不同的时间和空间复杂度，下面将分别进行介绍。

2.2.1　规则投影区间查找算法

规则投影区间查找算法（下文简称**区间查找算法**）源自 Overmars 等提出的高维区域树（hierarchical segment tree, 本文简称为 H-Tree）算法[20]。

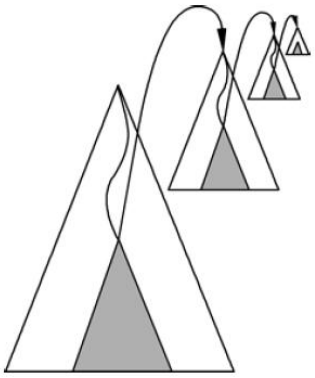


图 2.2　高维区域树

算法 2.1　H-Tree 算法

当 d=1 时，由于 n 个规则的端点（end point）在一维空间中最多构成 2n+1 个连续区间（segment），因此可以构建一个空间复杂度为$\Theta(n)$的平衡二分查找树（balanced binary search tree）来进行查找，查找时间为$\Theta(\log n)$。

当 d>1 时，首先依据规则集合在第 d 维的投影区间构造平衡二分查找树 T_d。树的每一个节点 v 对应一个区间 I_v，其中 I_v 表示所有 v 的所有子节点构成的连续区

间。对于满足：

1）在 d 维上的投影区间完全包含 I_v；

2）在 d 维上的投影区间不完全包含 v 的父节点 v' 对应的区间 $I_{v'}$；

的规则子集 R_v，用同样的方式在另外 d-1 个维度上依次构造平衡二分查找树 $T_{v,d\text{-}1}$，直到最后一个维度即可。※

图 2.2 位 H-Tree 算法的示意图。H-Tree 算法结合 Chazelle 等提出的分散叠层（fractional cascading）技术[53]，能够以$\Theta(\log^{d-1}n)$的时间复杂度和$\Theta(n*\log^{d-1}n)$的空间复杂度解决点定位问题[20]。

虽然 H-Tree 具有良好的空间复杂度，但时间复杂度依然过高（优于线性查找算法）。下面介绍降低时间复杂度的 Set-pruning Segment Tree（本文简称为 S-Tree）算法[33, 54]。

算法 2.2　S-Tree 算法

当 d=1 时，S-Tree 和 H-Tree 算法相同。由于 n 个规则的端点（end point）在一维空间中最多构成 2n+1 个连续区间，因此可以构建一个空间复杂度为$\Theta(n)$的平衡二分查找树来进行查找，查找时间为$\Theta(\log n)$。

当 d>1 时，S-Tree 采用规则复制的方法避免回溯查找。首先依据规则集合在第 d 维的投影区间构造平衡二分查找树 T_d。T_d 的每一个叶节点 v 对应一个区间 I_v，其中 I_v 不包含任何子区间。对于规则子集$R_v=\{r_i|r_i\cap I_v\neq\emptyset, 1\le i\le n\}$，依次为其构造子树 $T_{v,d\text{-}1}$ 即可。※

与 H-Tree 相比，S-Tree 的时间复杂度降低到了$\Theta(d*\log n)$，但依据定理 2.1，S-Tree 的空间复杂度将增至$\Theta(n^d)$。

定理 2.1　d 域空间中的 n 个规则至多可构成$(2n+1)^d$ 个相互不重叠的超长方体。

证明：

首先证明 d=1 时命题成立。一维空间中的规则退化为线段，即数轴上的一个区间。

当 n=1 时，一个规则构成一个区间，命题成立。

假设$k(k\ge 1)$个规则可构成 2k+1 个相互不重叠的区间。那么当 n=k+1 时，第 k+1 个规则的两个端点最多落入 2k+1 个不重叠的区间中的两个不同的区间中，并将这两个区间划分为四个区间，因此最多增加两个不重叠区间。又由

$2k+1+2=2(k+1)+1$，所以命题对于 $n=k+1$ 亦成立，即一维情况下命题成立。

多域情况下，即当 $d>1$ 时，由于每个域上最多有 $2n+1$ 个不重叠区间，经过交叉相乘，d 个域上最多出现 $(2n+1)^d$ 个不重叠的超立方体，由此命题得证。※

2.2.2　网包搜索空间分解算法

与区间查找算法不同，网包搜索空间分解算法（下文简称**空间分解算法**）通过对空间的均匀切分构建网包分类的数据结构。空间分解算法源于路由查找算法，通常使用 trie 结构对空间进行逐级等分。这里首先讨论规则均为最长前缀匹配的情况，后面会讨论一般情况。下面首先介绍基于 Hierarchical Trie（本文简称 H-Trie）的空间分解算法[20]。

算法 2.3　H-Trie 算法

当 $d=1$ 时，构建一个 W 深度的 trie 结构，每一个规则均存储于 tire 的一个节点上，此 trie 结构空间复杂度为 $\Theta(n*W)$，时间复杂度为 $\Theta(W)$。

当 $d>1$ 时，首先用 $\{r_1[d], r_2[d], \ldots, r_n[d]\}$ 构造 trie 结构 T_d。然后为每一个不同的前缀 P_i（T_d 中不同的节点）所对应的规则子集 R_i 构建 d-1 维度上 trie 结构 $T_{i,\,d\text{-}1}$。依次类推，直到第一个维度即可。※

H-Trie 在搜索过程中需要回溯查找（back tracking search），因此时间复杂度为 $\Theta(W^d)$。由于 H-Trie 中规则集合只存储一次，因此算法的空间复杂度为 $\Theta(n*W)$。注意，此时的空间复杂度只对应前缀匹配的规则，而本文讨论的网包分类问题是基于范围匹配的，因此这里讨论范围到前缀转换问题（range-to-prefix）。首先给出定理 2.2。

定理 2.2　一个 $[0, 2^W]$ 区间中的范围可以用至多 $2(W\text{-}1)$ 个前缀表示[9]。

证明：给出存在性证明。

首先考虑当范围为 $[0, b]$ 时至多需要多少个前缀表示。

当范围为 $[0, b]$，其中 $2^{W-1} < b \le 2^{W-1}$ 时，首先记录表示区间 $[0, 2^{W-1})$ 的前缀 P_0，然后将区间 $[2^{W-1}, 2^W]$ 等分，生成两个新区间 $[2^{W-1}, 2^{W-1}+2^{W-2})$ 和 $[2^{W-1}+2^{W-2}, 2^W]$，则 b 必然落入两个区间中的一个，如果 $2^{W-1}+2^{W-2} < b \le 2^W$，则记录表示区间 $[2^{W-1}, 2^{W-1}+2^{W-2})$ 的前缀 P_1；依次不断等分 b 所在子区间，并当 b 落入右半区间时记录左半区间的前缀 P_i；经过至多 W-1 次等分，区间 $[2^{W-1}, 2^W]$

将不可再分，由于整个过程中最多记录下 W 个前缀（W-1 次等分得到的前缀和 P_0），而这些前缀恰好可以表达范围[0, b]。因此，范围[0, b]需要至多 W 个前缀表示。

再考虑[0, 2^W]空间中的范围[a, b]，通过将[0, 2^W]二等分，范围[a, b]最多被分为两个范围，并且这样的两个范围通过平移和镜像，都可以等价的转化为[0, 2^{W-1}]空间中以 0 为起点的范围。根据上面的证明，这两个范围都可以用至多 W-1 个前缀表示，因此范围[a, b]可用至多 2 (W-1)个范围表示。※

根据定理 2.2 可知，一个范围匹配的规则在 d 个域上最多可以转换为$(2(W-1))^d$个前缀匹配的规则。因此，对于范围匹配，H-Trie 的空间复杂度为$\Theta(n * W^d)$。当$d > 1$时，最后两个维度的查找可以使用 Srinivasan 等提出的 Grid-of-Trie 结构进一步减少查找时间[9]。此时 H-Trie 的时间和空间复杂度可以改进为$\Theta(W^{d-1})$及$\Theta(n * W^{d-1})$。由于 H-Trie 的时间复杂度过高，Tsuchiya 等提出了 Set-pruning Trie（本文简称 S-Trie）算法，通过规则复制来降低时间复杂度[55]。

算法 2.4　S-Trie 算法

当 d=1 时，S-Trie 算法与 H-Trie 相同。构建一个 W 深度的 trie 结构，每一个规则均存储于 tire 的一个节点上，此 trie 结构空间复杂度为$\Theta(n * W)$，时间复杂度为$\Theta(W)$。

当 d>1 时，首先对{$r_1[d]$, $r_2[d]$, …, $r_n[d]$}构造 trie 结构 T_d。然后对于每一个叶节点 v，将包含 v 代表的前缀 P_v 的所有规则子集 R_v 构建 d-1 维度上 trie 结构 $T_{v,\,d\text{-}1}$。依次类推，直到第一个维度即可。※

由于 S-Trie 的查找过程不需要回溯，因此时间复杂度为$\Theta(d * W)$，空间复杂度为$\Theta(n^d * dW)$。由此可见，H-Trie 和 S-Trie 分别是对空间和时间性能的折中，但两者的时间性能在 n 较大时均优于线性查找算法。

2.2.3　数学解法总结

多域点定位问题的各类数学解法总结于表 2.2。这些数学解法为网包分类算法设计提供了理论依据。从该表可知，允许回溯查找的算法具有较好的空间特性，而允许规则复制的算法则具有较快的查找速率。由于查找速率决定网包分类系统的吞吐率，因此典型的网包分类算法大多采用了允许规则复制的方法。关于查找的策略，区间查找算法和空间分解算法均有各自的优势，并广泛用于不同的网包

分类算法。空间分解算法由于不需要存储规则投影点，且可以利用多比特 trie 进一步加快查找速率，但空间分解法需要考虑范围到前缀匹配，因此最坏情况下存储空间为投影区间二分法的$\Theta(W^d)$倍（参见定理 2.2）。关于网包分类算法时间和空间性能折中的进一步分析可以参阅文献 [44]。

表 2.2　多域正交区域点定位问题的算法比较

维度	复杂度	回溯查找		规则复制	
		空间分解	区间查找	空间分解	区间查找
		H-Trie	H-Tree	S-Trie	S-Tree
d=1	时间 空间	$\Theta(W)$ $\Theta(n*W)$	$\Theta(\log n)$ $\Theta(n*\log n)$	$\Theta(W)$ $\Theta(n*W)$	$\Theta(\log n)$ $\Theta(n*\log n)$
d>1	时间 空间	$\Theta(W^{d-1})$ $\Theta(n*W^{d-1})$	$\Theta(\log^{d-1} n)$ $\Theta(n*\log^{d-1} n)$	$\Theta(d*W)$ $\Theta(n^d*d\mathrm{W})$	$\Theta(d*\log n)$ $\Theta(n^d*d\log n)$

2.3　网包分类算法的评价方法

由上述算法分析可知，网包分类问题的各类数学解法均具有较高的时间或空间复杂度，即在最坏情况下无法同时满足查找速率和存储空间的两方面要求。幸运的是，在实际的网络应用中网包分类问题往往不会达到理论上的最坏情况[13, 33]。当前的网包分类算法设计大多通过引入规则集合的特征来提高分类速率、降低内存使用。

Gupta 等通过对大量实际的（real-life）规则集合的研究，总结并归纳出一系列规则集合特征[13]：

- 实际规则集合中规则数目不会太多，一般从几十条规则到数千条规则。规则数目不多可能是由于网络应用本身规模的限制，也可能是基于当前路由器处理能力的考虑。
- 规则在协议域通常只有很少的几个取值。绝大多数规则集合中只出现 TCP 和 UDP 两种传输层协议。个别规则集合中可能涉及 ICMP、IGMP 和 GRE 等协议。
- 传输层端口域取值范围很广，采用范围前缀转换很可能是非常低效的。
- 与同一个网包匹配的规则通常少于 5 个，最多出现过 10 个。①
- 同一规则集合中的多个规则往往在某些域上具有相同的设置。
- 规则集合中所有规则在单一域的不同取值的个数通常远小于规则个数。
- 规则集合出现的重叠个数远远小于理论上界。

① 此处结论结合本文实验进行了部分修正。Gupta 在文献[13]的原文中为 7 个。

表 2.3　网包分类规则复杂性比较[33]

规则集	规则数	各域非重叠区间理论最大值	目标 IP 地址域非重叠区间实际值	源 IP 地址域非重叠区间实际值	目标端口域非重叠区间实际值	源端口域非重叠区间实际值	非重叠超长方体理论最大值	非重叠超长方体理论实际值
FW1-100	92	185	45	19	48	20	1.17×10^{9}	8.21×10^{5}
FW1-1K	791	1583	314	221	75	23	6.28×10^{12}	1.20×10^{8}
FW1-10K	9311	18623	13901	7270	77	22	1.20×10^{17}	1.71×10^{11}
ACL1-100	98	197	120	48	70	1	1.51×10^{9}	4.03×10^{5}
ACL1-1K	916	1833	559	143	165	1	1.13×10^{13}	1.32×10^{7}
ACL1-10K	9603	19207	1325	7921	181	1	1.36×10^{17}	1.90×10^{9}
IPC1-100	99	199	119	118	26	26	1.57×10^{9}	9.49×10^{6}
IPC1-1K	938	1877	796	559	78	49	1.24×10^{13}	1.70×10^{9}
IPC1-10K	9037	18075	4604	2377	94	59	1.07×10^{17}	6.07×10^{10}

另外，对于不同应用下的规则集合，也会出现各自不同的统计特性。例如在 WUSTL（华盛顿大学圣路易斯分校）公开数据集中，核心路由器上的访问控制规则（ACL），防火墙的安全策略（FW），以及 Linux 网关的 iptables 规则（IPC）就有相当大的差异[18, 33]。Qi 等通过统计这些规则集合中各个维度上的投影区间个数说明了网包分类问题的实际复杂性。

从表 2.3 中的统计结果可以看出[33]：

- 同一类型规则集合在不同维度上的统计特性不同
- 不同种类的规则集合的统计特性不同
- 所有规则集合的实际复杂度均远小于理论复杂度

解决多域网包分类问题需要设计高效的分类算法。一般来说，评价一个算法的好坏，需要从三个方面进行综合考虑：

- **分类速率：**网包分类系统通常需要满足线速（wire-speed）处理速度。对于实际的网包分类系统，通常使用系统吞吐率（throughput）来评价分类速率。例如要满足 100Gbps 的网络带宽，网包分类系统需要每秒钟处理

150M 个 64 字节的网包。在算法分类速率的评价中，考虑到处理单元（CPU）的计算速度比外围存储设备（DRAM）的访问速度快得多，因此当计算量大小在可接受的范围内时，通常使用内存访问次数（memory access times）来评价一个算法的分类速率。

- **内存使用：**网包分类算法的内存使用不仅仅指存放规则集合本身所占用的存储空间，还包括算法建立的用于查找的数据结构存储空间。考虑到网包分类系统的内存空间有限，网包分类算法应尽可能压缩其数据结构以支持更多的分类规则。另外，有时候还必须考虑算法预处理过程中的内存使用。例如在递归生成多级 trie 的数据结构过程中，有时需要大型的堆栈支持。如果系统无法满足此内存需求，那么即使最终数据结构较小也无法实现。

- **预处理时间：**由于网包分类规则并非固定不变，网包分类算法需要依据规则更新生成新的数据结构。本文讨论的算法的预处理时间指依据新的规则集合重新生成网包分类数据结构的全部时间。

从算法设计的角度来看，目前的大多数算法都比较重视分类速率和内存使用两方面的性能，与之对应的是算法的时间性能和空间性能。两个因素在算法设计中往往互相制约，通常在满足某一方面性能的情况下，尽可能优化另一方面的性能。从算法实现的角度来看，网包分类算法必须要兼顾多种系统平台的硬件约束，能够自适应部署于各类平台并最大程度利用系统资源以满足网包分类设备的性能需求。因此，对于一个出色的网包分类算法其性能并不仅仅体现在理论分析中，还必须考虑到算法实现、运行环境、软硬件平台和特殊需求等多种问题。只有这样的算法才真正具有研究意义和现实的价值。

2.4　本章小结

本章总结了网包分类算法的理论依据，并结合实际规则分析给出了算法评价的常用方法。从计算几何中的复杂度分析表明，不存在某种通用的算法适用于所有的多域网包分类问题。H-Trie 和 H-Tree 算法具有较好的空间特性，但需要耗时的回溯查找；S-Trie 和 S-Tree 算法通过规则复制大幅降低了搜索时间，但代价是指数级的内存增长。

幸运的在实际应用中极少会遇到理论中的最坏的情况。实际应用中的网包分类问题往往具有各类结构和统计特性。将这些特性应用到网包分类算法的设计中

去，通常可以得到“足够快”的分类速率，同时满足“足够少”的内存使用。“足够快”和“足够少”在这里表明了一种权衡（tradeoff）的思想，是算法时间性能和空间性能的折中。算法设计的最终目标就是在理论依据之上，充分挖掘规则特性并考虑系统约束，寻求最优的权衡。

第 3 章　区间查找算法

本章研究网包分类算法设计中的区间查找算法。首先依据第 2 章中的计算几何理论对区间查找算法的设计思想进行归纳；接下来介绍和分析几类典型的区间查找算法；然后提出一种创新的区间查找算法 HyperSplit[33]；最后利用实际数据集合和多核网络处理器对区间查找算法的性能进行测试和比较。

3.1　理论依据

依据第 2 章的理论分析，区间查找算法采用了一种分而治之的设计思想：将网包分类规则投影在各个域上，每个域上相邻的两个投影点构成一个投影区间。分类过程中首先确定网包在各个域上属于哪个投影区间（子空间），然后再通过子空间求交的方法完成最终匹配。

对于第i $(1 \le i \le d)$个域来说，投影区间将该域划分为 m_i 个子空间：

$$S = S_1^i \cup S_2^i \cup \ldots \cup S_{m_i}^i$$

其中$\forall j,k$ $(1 \le j < k \le m_i)$有：

$$S_j^i \cap S_k^i = \emptyset$$

且通常有：

$$|S_j^i| \ne |S_k^i|$$

与之对应的规则集合被分解为 m_i 个规则子集：

$$R = R_1^i \cup R_2^i \cup \ldots \cup R_{m_i}^i$$

若已知网包 p 落入子空间：

$$S_p = S_p^1 \cap S_p^2 \cap \ldots \cap S_p^d$$

则与之匹配的规则子集为：

$$R_p = R_p^1 \cap R_p^2 \cap \ldots \cap R_p^d$$

由此可见，绝大多数区间查找算法都采用了规则复制的策略避免回溯查找，因此区间查找算法对应于空间点定位问题中的 S-Tree 算法（算法 2.1），并具有相同理论复杂度。在算法设计中，各类区间查找算法对 S-Tree 算法都进行优化和拓展。一方面利用启发式方法改进区间查找的数据结构提高分类速率，另一方面利用规则冗余特性通过迭代求交降低内存使用。下面几个小节将分别介绍已有区间投影算法以及创新的 HyperSplit 算法。

3.2　已有算法分析

最早的基于规则投影区间查找的网包分类算法是由 Srinivasan 等提出的 Cross-producting 算法[9]。基于 Cross-producting 算法，学术界提出了一系列的网包分类算法[11, 13, 30-33, 54,]。其中最典型的是 Gupta 等提出的 RFC（Recursive Flow Classification）算法[13]和 Xu 等提出的 HSM（Hierarchical Space Mapping）算法[30, 31]。作为算法创新的依据，这里首先对几类典型区间查找算法进行介绍和分析。

3.2.1　Cross-producting算法

Cross-producting 算法[9]是最早出现的多域网包分类算法之一，其主要思想是采用最长前缀匹配方法进行各域上的规则投影区间查找，并使用求交表（cross-producting table）完成空间求交。

从数据结构角度分析，Cross-producting 首先使用 trie 结构进行最长前缀匹配，并存储每个域的不同前缀及其对应的前缀编号；然后使用一个 d 维的求交表进行空间求交。其中，d 维求交表各个维度的取值大小由该维度上不同前缀的个数决定，而每个表项存储内容为优先级最高的规则序号。

从查找过程分析，Cross-producting 算法的查找过程分为两步：首先在第一级 d 个 trie 结构上进行一维的最长前缀匹配，确定网包包头在各个域上的前缀索引；然后通过查询 d 维求交表来确定分类结果。

Cross-producting 算法巧妙地将规则投影区间查找和最长前缀查找相结合，成为最早流行的网包分类算法之一。Cross-producting 算法为其后续算法 RFC、HSM 等提供了设计思路。不过，Cross-producting 的缺点也十分明显。首先，Cross-producting 算法要求对规则集合进行范围前缀转化。按照第 2 章中的相关理论和实际数据分析，这种转化会大幅降低算法的空间性能。其次，由于 Cross-producting 算法的时间复杂度为$\Theta(d*W)$，对于典型的五元组分类需要上百次的内存访问（典型的五元组网包包头为 104 比特）。所以网包分类速率不能得到有效保障。此外，单一的 d 维求交表无法利用启发式方法消除分类规则的结构冗余。

3.2.2　RFC算法

为解决 Cross-producting 算法中分类速率低，内存使用过高的问题，Gupta 等人提出了 RFC 算法[13]。RFC 算法的基本思想是：利用数组结构存储各域的投影区间，将每个维度上$\Theta(W)$的最长前缀查找时间提高为$\Theta(1)$，同时采用多级求交表进行迭代求交，逐级消除空间冗余。

在 RFC 算法中，如果将网包分类问题看作 W 比特的网包包头到 U 比特的规则序号（U=log$n$）的映射过程，那么可以通过逐级映射的方式来实现从 W 比特网包包头到 U 比特规则的匹配。由于规则在空间的重叠数目远小于理论上的可能值（参见第 2 章中的表 2.3），RFC 算法能够消除部分空间冗余。图 3.1 是 RFC 算法的基本思想，通过三级空间映射，完成了 104 比特网包包头到 12 比特（约 4K 规则）规则序号的映射。

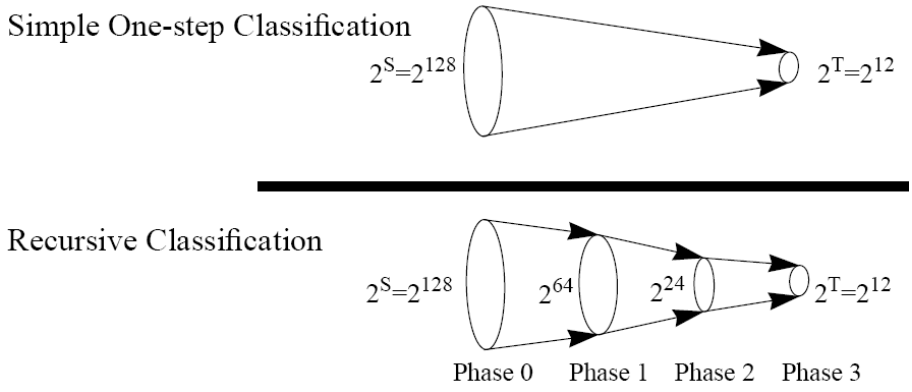


图 3.1　RFC 算法基本思想[13]

RFC 算法的数据结构为多级表结构。这些表被称为 chunk，表结构的级数（phase）以及每一级中 chunk 的个数都是预先设定的，图 3.2 给出了一种可能的结构。这些表的建立分多步进行：在第一级（Phase 0），对网包包头的各个域进行规则投影区间划分，以地址或端口为索引为各个维度建立对应的表。每个表项存放对应的子空间序号；在下一级中，每一个表都由前一个 Phase 的若干个表求交生成，索引由各个域的子空间序号计算得出，每个表项存储当前搜索空间划分方法的子空间序号；最后一级求交表中，每个表项存储优先级最高的匹配规则序号。

RFC 算法的优点是一方面利用数组查找避免了每个域上的最长前缀的查找，将分类速率提高到了$\Theta(d)$，另一方面通过多级映射消除各级表项中的空间冗余。RFC 算法的不足之处包括：首先，表结构级数和每级 chunk 个数都是由提前设定的。因此当规则集合变化时，这种设定可能不是最优。其次，利用数组进行区间查找，无法实现对 32 位 IPv4 地址的索引。因此，RFC 将 32 位的 IP 地址域拆分为两个 16 位的维度建立两张 chunk 表，这实际上增加了下一级 chunk 的表项数。尤其对于 IPv6 中的 128 位的 IP 地址，这种拆分方式更不适用。

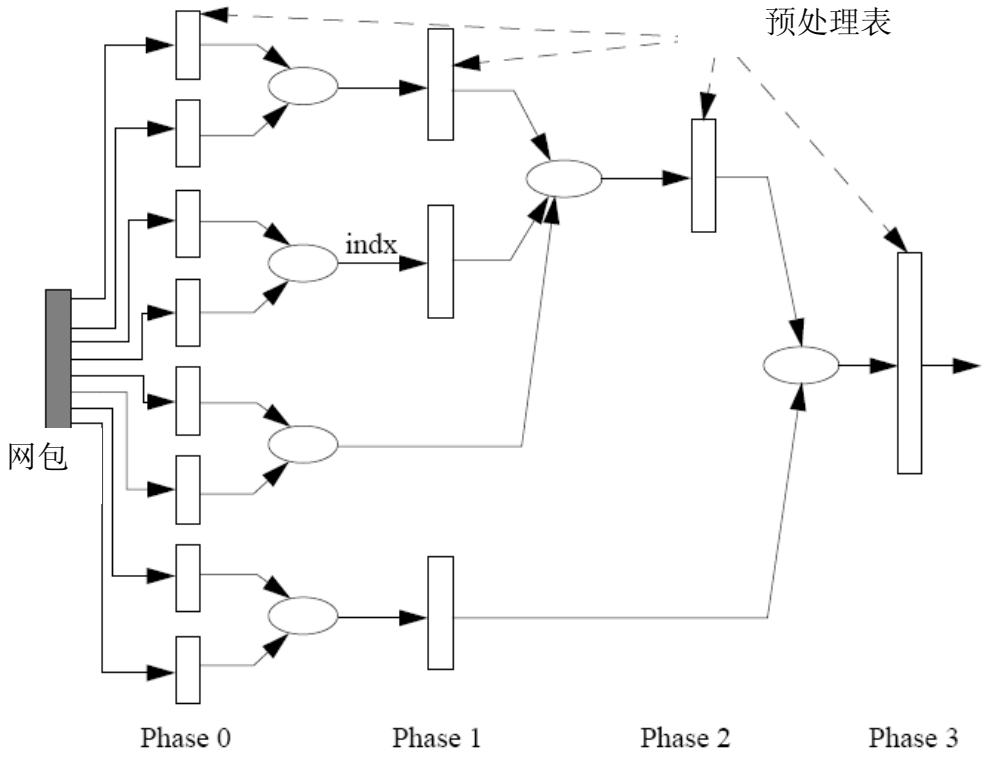


图 3.2　RFC 算法结构示例[13]

3.2.3　HSM算法

Xu 等提出的 HSM 算法改善了 RFC 算法中存在的一些问题。HSM 算法的基本思想包括：首先，利用二分查找法进行各域的区间查找；其次使用两级映射表进行空间求交。HSM 能够进一步降低 RFC 算法的内存空间，同时能够支持任意位数的多域网包分类。图 3.3 为一个四域的 HSM 算法示例。依据该图，HSM 算法的查找过程分为三个阶段：

- 第一阶段：根据规则集合在源 IP 地址（SA）、目标 IP 地址（DA）、源端口（SP）和目标端口（DP）四个域上进行投影区间查找。为了加快查找过程，HSM 对于 32 位的 IP 地址使用平衡二分查找树查找，而对于 16 位的端口号使用 RFC 中的数组索引查找。
- 第二阶段：利用 SA 和 DA 的区间编号为索引，对地址映射表（Address Mapping Table, AMT)进行查找。AMT 为 SA 和 DA 两个域的空间求交表，其中的每个表项为指定地址组编号（Address Group Number, AGN）。第二阶段同时利用 SP 和 DP 的区间编号为索引，对端口映射表（Port Mapping Table, PMT）进行查找。PMT 为 SP 和 DP 两个域的空间求交表，其中的每个表项为指定端口组编号 （Port Group Number, PGN）。
- 第三阶段：利用 AGN 和 PGN 为索引，对策略查找表（Policy Lookup Table,

PLT）进行查找。PLT 为 AMT 和 PMT 两个表中独立元素个数的求交表，其中每个表项存储最后的匹配规则编号。

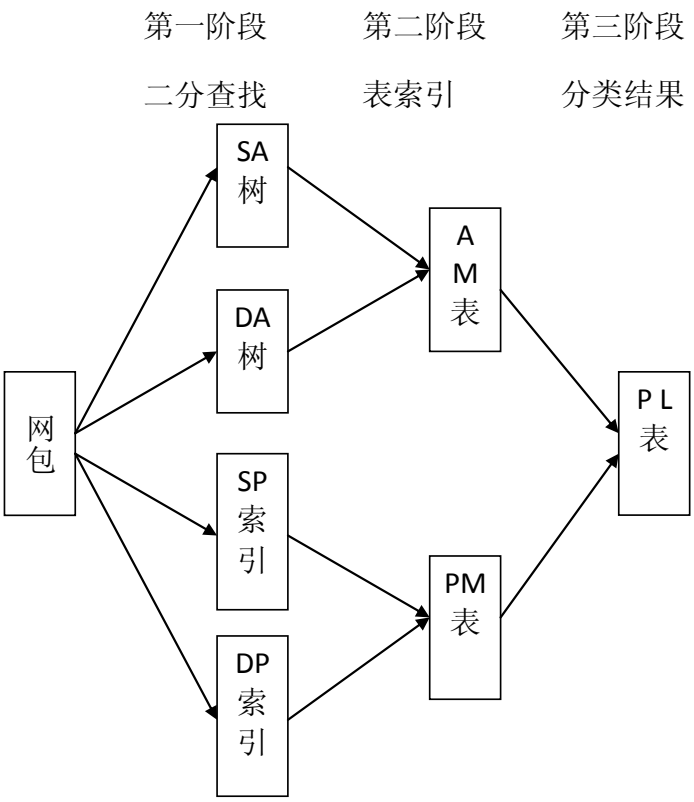


图 3.3　HSM 算法原理图[30]

由于使用了二分查找代替数组索引，HSM 算法的时间复杂度为$\Theta(d*\log n)$。虽然时间复杂度较 RFC 算法有所提高，但由于实际应用中源 IP 域的区间数极小，同时目标端口和源端口域又采用了数组索引，因此 HSM 的分类速率实际仅由目标地址域决定。对于 n 为 1000 时，HSM 算法通常能够在 10～15 次内存访问中完成查找。因此，HSM 和 RFC 一样，被认为是分类速率最快的一类算法。

由于 HSM 算法在空间求交过程中依然采用了确定性的求交方法，因此并未从根本上解决 RFC 算法的存储空间过大的问题。此外，无论 RFC 还是 HSM 算法在处理大规模规则集合时，预处理时间都会很长。为了解决现有区间查找算法存在的种种问题，本文提出了 HyperSplit 算法，将在下一节中详细介绍。

3.3　HyperSplit算法

HyperSplit[33]是本文提出的一种基于规则投影区间查找的网包分类算法。HyperSplit 的设计目的是在保持现有区间查找算法的高效分类能力的前提下，大幅降低已有算法的内存使用。下面分别介绍 HyperSplit 的设计思想、理论依据和算法设计。

3.3.1　设计思想

HyperSplit 算法设计的基本目标是：

- **高分类速率**：由于分类速率通常是网包分类系统最关键的性能指标，因此 HyperSplit 算法必须具有稳定高效的分类速率。
- **低内存使用**：对于内存使用，一个最基本的要求是算法的内存使用不能超过系统支持的最大内存；此外，HyperSplit 算法应当尽量减少内存使用，使得算法数据结构能被有效缓存，以降低每次访存的延迟。

第 2 章的理论分析证明，高分类速率和低内存使用在网包分类算法中难以同时满足。因此，HyperSplit 算法采用了优先保证分类速率，尽量优化内存使用的策略。具体来讲 HyperSplit 算法的主要设计思路是：

第一步，采用没有回溯的二分查找

为满足这个设计要求，HyperSplit 采用了没有回溯的二分查找。因此 HyperSplit 的时间复杂度为$\Theta(d*\log n)$，与 HSM 算法一致。

第二步，采用启发式方法构建二分查找树

不同于 HSM 算法在每个域上进行独立的、全规则集投影区间的二分查找，HyperSplit 采用了启发式的区间查找策略。HyperSplit 每次选取最优的二分点，以过该点的垂直于坐标轴的平面将搜索空间二分。由于每次划分后规则集合也分为两个子集，那么下一次区间二分仅对两个规则子集进行分别处理即可，因此实际区间数逐级降低。此外，由于使用二分查找，HyperSplit 算法的每个内部节点仅需要保存一个二分点，因此节点大小得到了有效控制，数据结构相对简单。

为了进一步说明 HyperSplit 算法的核心设计思想，先举一个实际的例子。图 3.4 和图 3.5 比较了 HSM 和 HyperSplit 两种算法的不同分解策略。可以看出，HSM 算法中 r_2，r_3，r_5 均出现了复制，而 HyperSplit 并未复制规则，因此节省了存储空间。从搜索过程来看，HSM 算法分别查找两个树结构需要最多 4 次二分查找，而 HyperSplit 算法只需要最多 3 次查找。

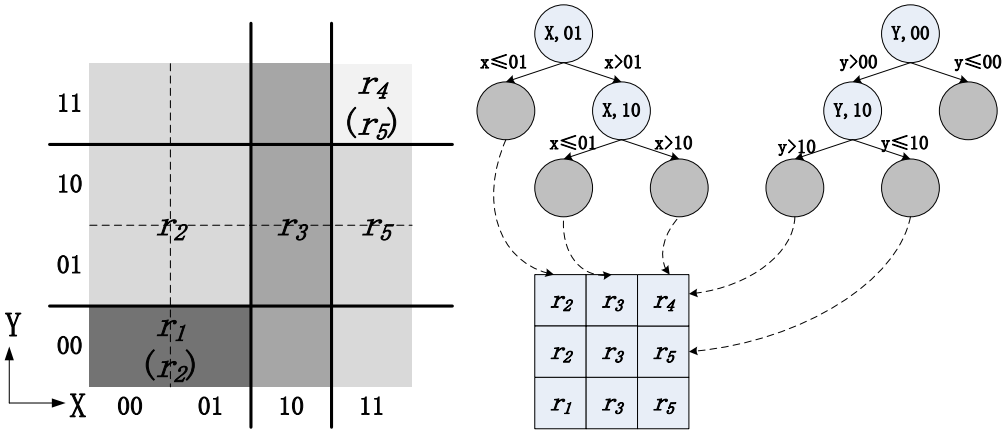


图 3.4　HSM 算法示例[33]

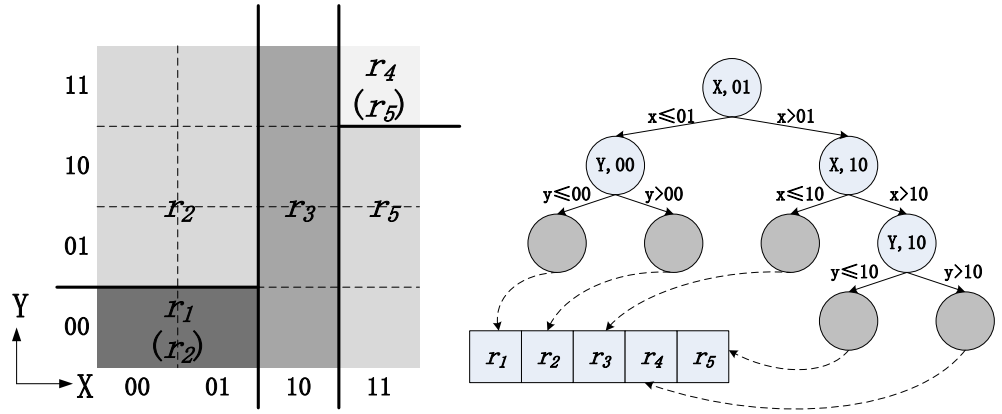


图 3.5　HyperSplit 算法示例[33]

3.3.2　二分点选择

HyperSplit 算法的核心设计思想是：

尽可能利用当前查找结果简化下一级查找，尽量保证每一次的二分查找都是当前最优。

对于基于二分查找的数据结构来说，二分点的选择最能体现上述思想。因此，这里首先介绍 HyperSplit 算法的二分点选择方法。

首先给出如下定义：

- **空间二分**：空间二分是指在特定一个维度上对当前规则搜索空间的一个划分。例如对于 IPv4 的五元组分类的初始搜索空间，S={[0, 2^{32}-1], [0, 2^{32}-1], [0, 2^{16}-1], [0, 2^{16}-1], [0,2^{8}-1]}，若在第 1 个域上以二分点 P 进行二分，则得

到两个搜索子空间 S_1={[0, P-1], [0, 2^{32}-1], [0, 2^{16}-1], [0, 2^{16}-1], [0, 2^8-1]}，S_2={[P, 2^{32}-1], [0, 2^{32}-1], [0, 2^{16}-1], [0, 2^{16}-1], [0, 2^8-1]}。

- **规则投影点**：规则集合 R 中所有的规则在第 f ($1 \leq f \leq d$)个域上的投影点（每一个范围的起始点和终节点）构成的投影点集合，设为 m 个，$1 \leq m \leq 2n$。
- **投影点数组**：将 m 个投影点按照从小到大的顺序排列并存储在数组 $Pt[i]$ 中，其中 $0<i<m+1$。
- **投影区间数组**：两个相邻的端点构成一个区间，m 个点构成 m-1 个区间，用数组 $Sg[j]$标识，其中 $0<j<m$。每个区间中落入的规则数记为 $Sr[k]$，其中 $0<k<m$。
- **区间包含**：两个区间[a_1, a_2], [b_1, b_2]，若$a_1 \leq b_1$且$a_2 \geq b_2$，则[a_1, a_2]包含[b_1, b_2]。
- **区间相交**：两个区间[a_1, a_2], [b_1, b_2]，若$a_1 \leq b_2$且$a_2 \geq b_1$，则[a_1, a_2]与[b_1, b_2]相交。

基于上述定义，HyperSplit 算法二分点 P 依据下面三类方法

- **二分点 *P* 选择方法 1**

$$P=Pt[m/2]$$

此启发式方法使得区间二分后两个子树内的区间数相等。

- **二分点 *P* 选择方法 2**

$$P=Pt[i],\ \text{s. t.}$$

$$|\{r_j | r_j \text{ 与 } [Pt[1], Pt[i]] \text{相交}\}| \cong |\{r_k | r_k \text{ 与 } [Pt[i+1], Pt[m]] \text{相交}\}|$$

此启发式方法使得区间二分后两个子树内的规则数目近似相等。

- **二分点 *P* 选择方法 3**

$$P=Pt[i],\ \text{s. t.}$$

$$\sum_{j=1}^{i} Sr[j] \cong \sum_{j=i+1}^{m-1} Sr[j]$$

此启发式方法是将与每个区间相交的规则个数作为区间权重，然后使得区间二分后两个子树所包含区间的总权重相等。

与二分点选择对应的二分维度 F 选择方法为：

- **二分点维度 F 选择方法 1**

对于二分点选择方法 1 和二分点选择方法 2，选择

$$F = \max_{1 \le F \le d} m$$

此启发式方法选择不同投影点最多的维度进行区间二分。

- **二分点维度 F 选择方法 2**

对于二分点选择方法 3，选择

$$F = \min_{1 \le F \le d} \frac{1}{m} \sum_{j=1}^{m} Sr[j]$$

此启发式方法是将与每个区间相交的规则个数作为区间权重，然后选择平均权重最小的维度进行区间二分。

3.3.3　预处理

HyperSplit 的数据结构为多域二分查找树。每个内部节点选择二分点对当前规则集合的投影区间进行二分。每个叶节点上存储最终匹配的规则子集。HyperSplit 算法数据结构生成过程见图 3.6，主要包括下面的步骤：

1) 生成规则全集 R0、搜索空间 S0 对应的根节点 V0，并将其存入 R’、S’、V’。
2) 将 R’、S’和 V’送入队列 Q，队列 Q 为先进先出队列。
3) 从队列 Q 中取 R’、S’、V’，分别记为 R、S、V。
4) 判断 R 中所有规则是否都包含 S，若是，执行步骤 12，若否，执行步骤 5。
5) s107，按照 3.3.2 节内启发式算法，选择二分点 P（维度为 F）。
6) 将 S 在划分维度 F 上以划分点 P 进行空间划分，得到子空间 S1 和子空间 S2。
7) 将 R 中所有与子空间 S1 有相交的规则记为 R1，将 R 中所有与子空间 S2 相交的规则记为 R2。
8) 生成与 R1、S1 对应的子节点 V1，及与 R2、S2 对应的子节点 V2，并使得 V1 和 V2 的存储地址空间连续。

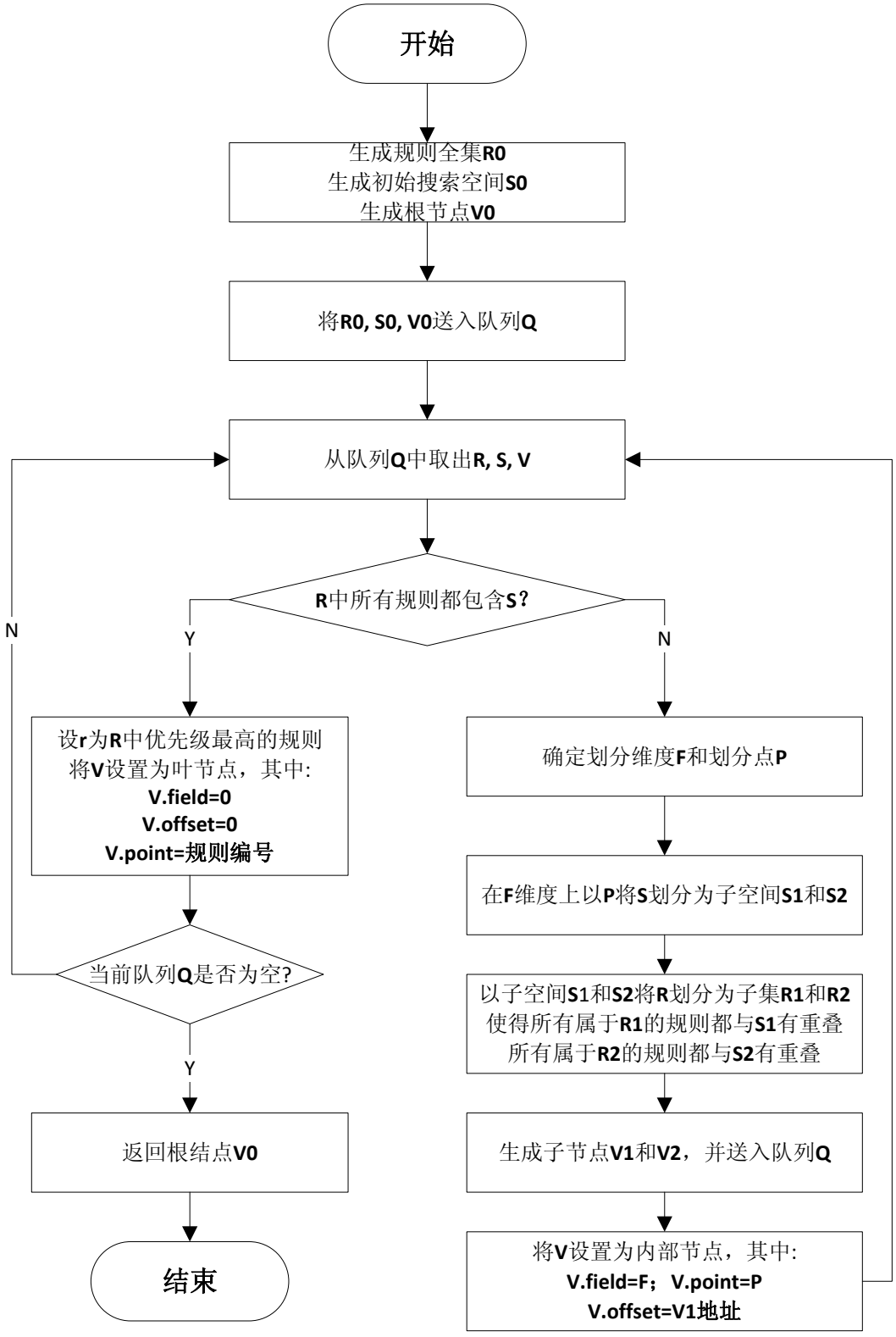


图 3.6　HyperSplit 算法预处理流程图

9) 将 V 设置为内部节点，并将 V 赋值为：V.field=F，V.point=P，V.offset=V1 的起始地址。
10) 生成新的一组 R’、S’和 V’，赋值为 R1、S1 和 V1 之后送入队列 Q。
11) 生成新的一组 R’、S’和 V’，赋值为 R2、S2 和 V2 之后送入队列 Q，返回执行步骤 3。
12) 获取 R 中优先级最高的规则 r，将 V 设置为叶节点，并将 V 赋值为如下数据结构：V.field=r.action，V.point=r.pri，V.offset=0；其中，r.action 为规则 r 中的分类结果，r.pri 为规则 r 中的优先级。
13) 判断当前队列 Q 是否为空，若是，执行步骤 14，若否，返回执行步骤 3。
14) 返回根节点 V0，将其作为数据结构索引的起点。

为了提高数据节点的访问效率，HyperSplit 算法采用了图 3.7 所示的数据结构。内部节点（白色）和叶节点（灰色）均被设计为 64 比特大小以提高访存效率。

对于每个内部节点，前 32 比特包含 28 位子节点地址信息（V.offset）和 4 位的二分域信息（V.field）。由于每个节点大小为 8 字节，则 28 位地址信息可以表示 $2^{28}\times 8$=2GB 地址空间。4 位的二分域信息可以支持多达 16 个域的网包分类查找。内部节点的后 32 位信息存放二分点的数值（V.point）。对于每个叶节点， 前 32 比特为 0，后 32 比特为匹配的决策。

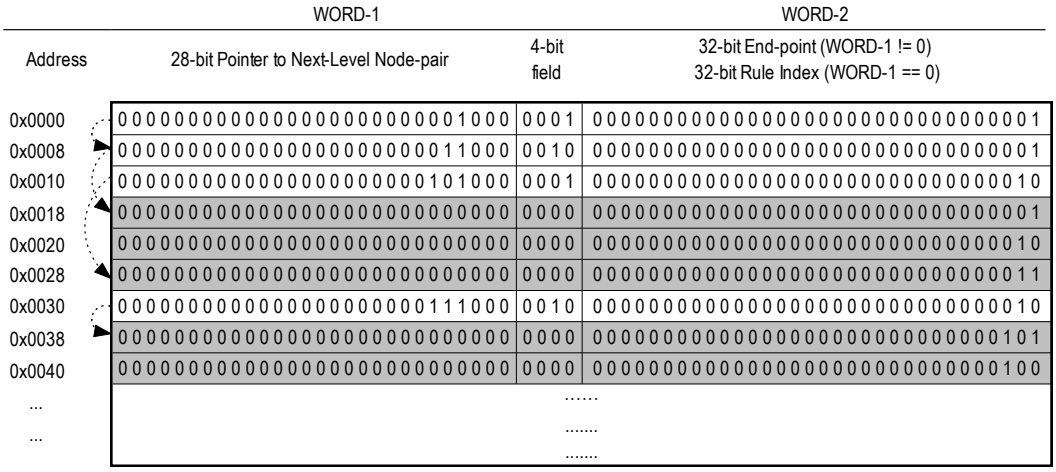


图 3.7　HyperSplit 数据结构[33]

3.3.4　查找算法

基于预处理的数据结构，HyperSplit 使用图 3.8 所示的步骤进行查找：

1) 设当前节点 V=V0。
2) 若 V.offset 为 0，则返回 V.point，执行步骤 6；否则执行步骤 3。
3) 若 V.field 小于等于 V.point，则执行 4，否则执行 5。
4) 令 V 为子节点 V1，其中 V1 的存储地址为：&(V0)+V.offset。执行步骤 2。
5) 令 V 为子节点 V2，其中 V2 的存储地址为：&(V0)+V.offset+sizeof(V)。执行步骤 2。
6) 根据 V.point 获取匹配规则，并根据规则决策执行转发、丢弃决策。

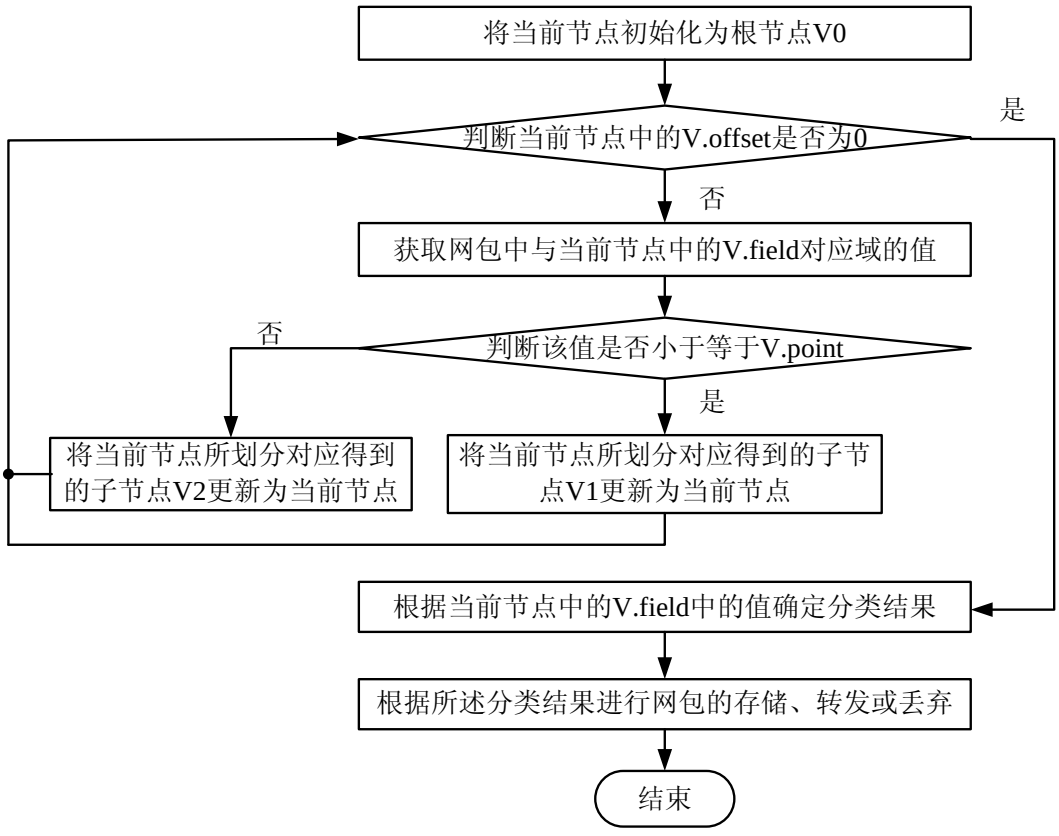


图 3.8 查找过程

3.4　性能评价

为了客观评价网包分类算法，本节比较了 HyperSplit 算法、HSM 和 HiCuts 算法的内存访问次数、内存使用、预处理时间和基于多核平台的吞吐率。由于 HyperCuts 和 HiCuts 都可以在叶节点使用线性查找算法，测试中使用 HiCuts-1 表示叶节点中只有唯一规则匹配的 HiCuts 算法；HiCuts-8 表示叶节点中存放 8 条规则，并需要进行线性查找的 HiCuts 算法。与之类似，HyperSplit-1 表示叶节点中只有唯一规则匹配的 HyperSplit 算法；HyperSplit -8 表示叶节点中存放 8 条规则，并需要进行线性查找的 HyperSplit 算法。

3.4.1　测试数据及平台

本文的算法比较采用公开的网包分类规则集合[18]。该数据集合为网包分类研究广泛使用，主要包含 ACL，FW 和 IPC 三类规则。其中，ACL 规则为路由器访问控制列表，FW 规则为防火墙策略，IPC 规则为 Linux IP Chains 规则。测试中每个规则集合的命名为“规则类型-规则数量”。例如，FW1-10K 为 10,000 条防火墙安全策略规则。所有规则都由 IPv4 五元组构成：包括 32 比特源 IP 地址、32 比特目标 IP 地址、16 比特源端口号、16 比特目标端口号、8 比特传输层协议。

算法预处理在一台 Intel x86 架构的服务器上进行。该机器拥有一颗 2.0GHz 的 Intel core2 处理器，以及 4GB DDR2 内存。算法预处理中使用的代码均使用 C 语言开发，并使用 gcc –O2 指令编译。操作系统为 Ubuntu 8.04 LTS。

网包分类系统的吞吐率测试采用测试采 Cavium OCTEON3860 多核网络处理器平台实现[48]。该处理器包含 16 个运行在 500 MHz 的 MIPS 核心，8 个千兆 RGMII 的网络接口。存储体系包括 32K 字节的每核独享 L1 缓存，1M 字节的共享 L2 缓存以及 2G 字节的 DDR2 主存。在开发编程工作中，使用了基于 Cavium SDK version 1.5 的 Simple Executive 模式以保证最大程度的发挥处理器性能。图 3.9 为 Cavium OCTEON3860 芯片的架构。

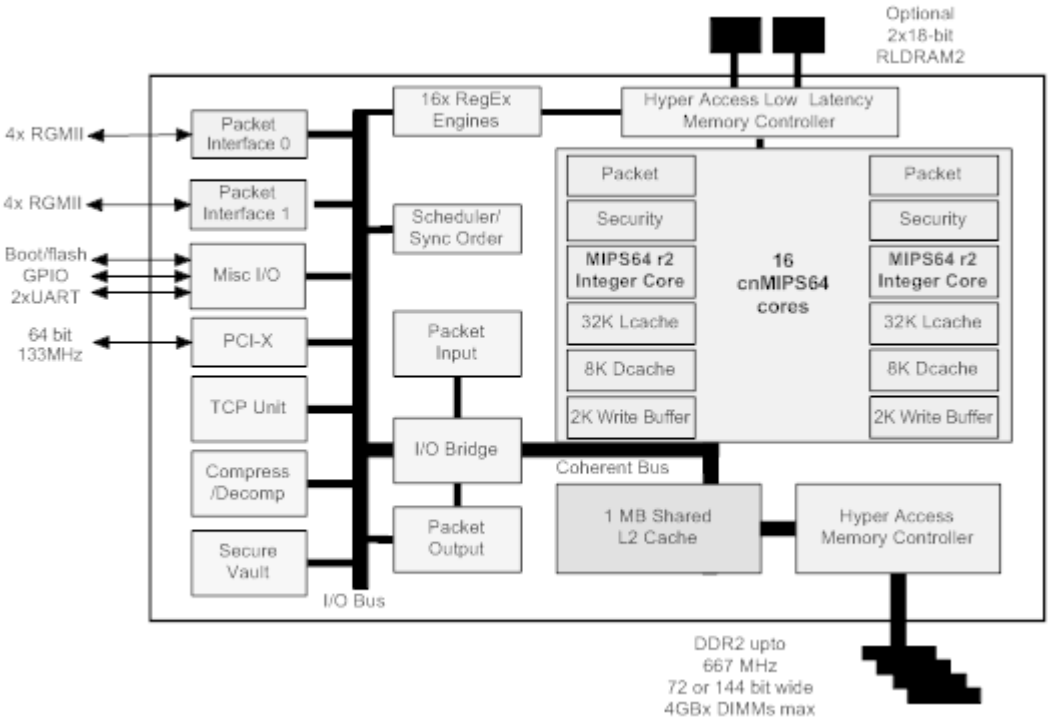


图 3.9　Cavium OCTEON3860 多核网络处理器[48]

3.4.2　分类速率

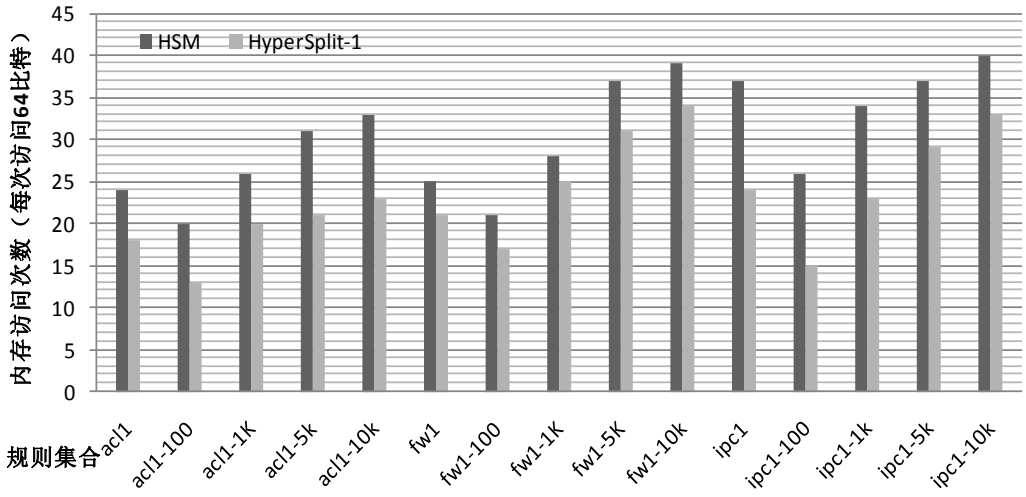


图 3.10　HyperSplit-1 和 HSM 内存访问次数[33]

图 3.10 比较了 HyperSplit-1 和 HSM 算法的分类速率。评价指标为 64 比特长字的内存访问次数。从图中可以看出，HyperSplit-1 算法的平均内存访问次数为 HSM 算法的 70%。

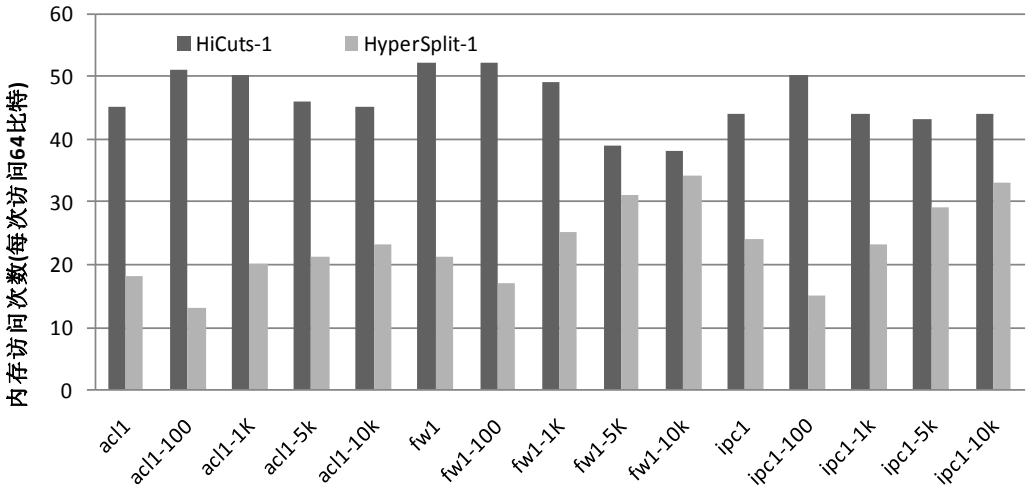


图 3.11　HyperSplit-1 和 HiCuts-1 内存访问次数[33]

图 3.11 比较了 HyperSplit-1 和 HiCuts-1 算法的分类速率。评价指标为 64 比特长字的内存访问次数。从图中可以看出，HyperSplit-1 算法的平均内存访问次数为 HiCuts-1 算法的 50%。

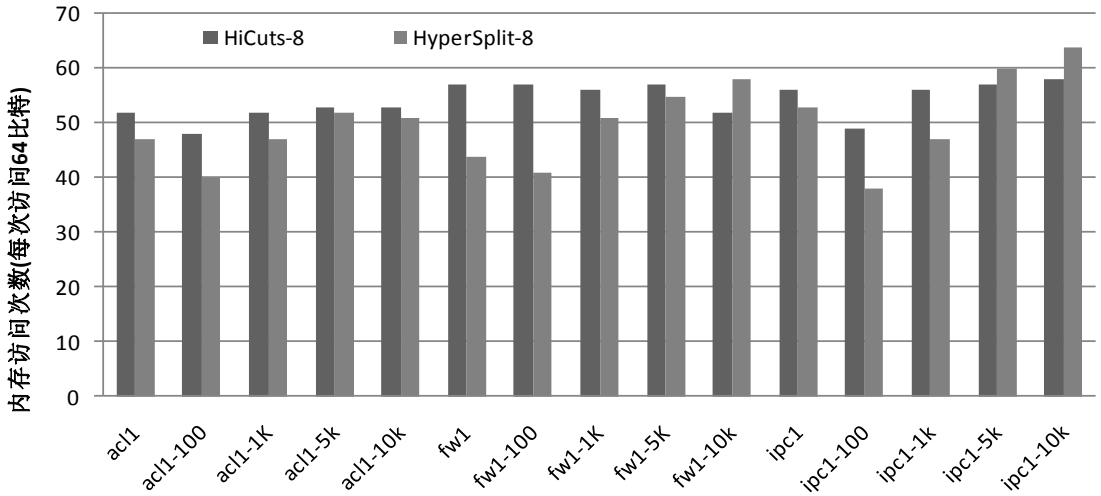


图 3.12　HyperSplit-8 和 HiCuts-8 内存访问次数[33]

图 3.12 比较了 HyperSplit-8 和 HiCuts-8 算法的分类速率。评价指标为 64 比特长字的内存访问次数。从图中可以看出，HyperSplit-8 算法的平均内存访问次数和 HiCuts-8 算法的相差不到 10%。这是因为 HyperSplit-8 和 HiCuts-8 算法中 70%以上的内存访问都来自叶节点的线性查找。

由上述实验可知，HyperSplit 算法的分类速率要优于现有的 HSM 算法以及 HiCuts 算法。后面会通过算法实现说明，HyperSplit 在多核网络处理器平台上具有良好的分类性能。

3.4.3　内存使用

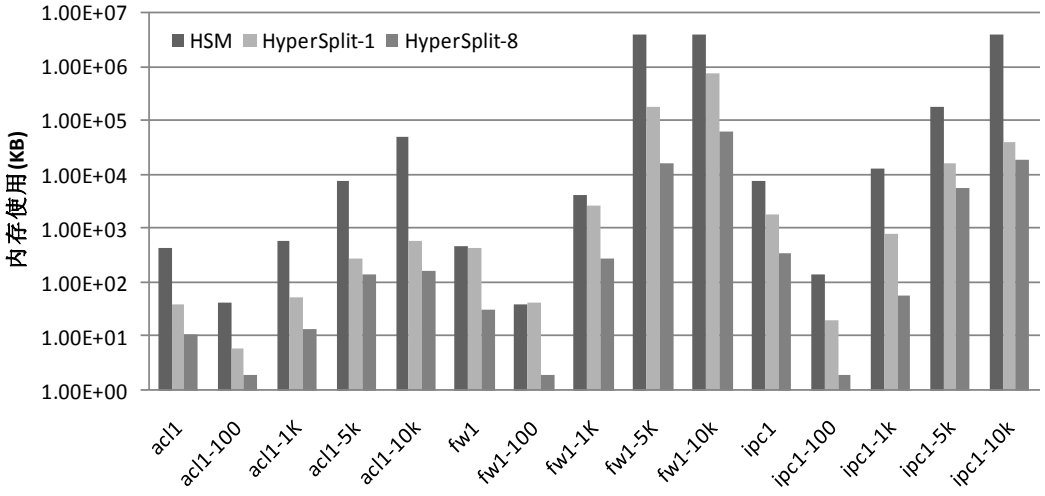


图 3.13　HyperSplit 和 HSM 的内存使用[33]

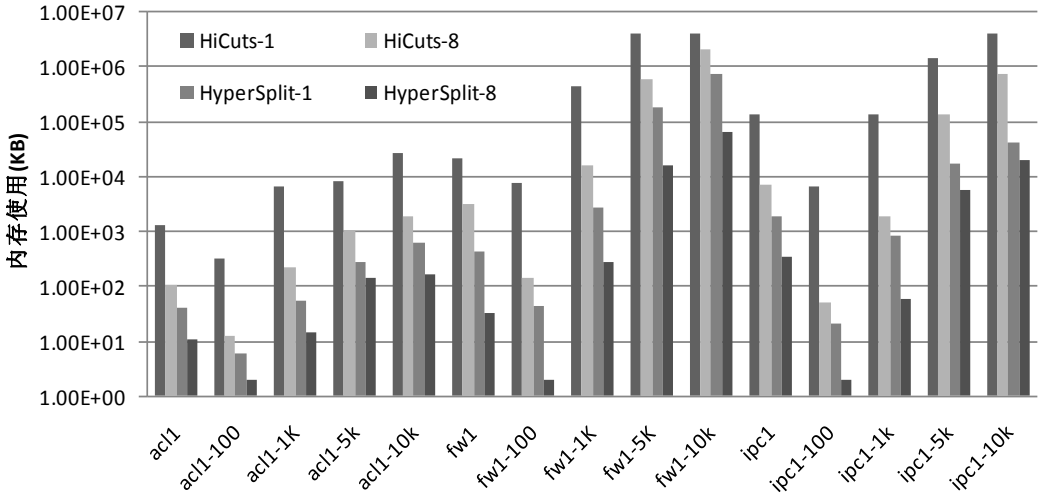


图 3.14　HyperSplit 和 HiCuts 的内存使用[33]

从图 3.13 和图 3.14 可以看出，对于大多数规则集合，HyperSplit-1 和 HyperSplit-8 算法的内存使用比 HSM 和 HiCuts 算法分别少 1～2 个数量级。对于特别复杂的规则集合，例如 FW1-10K，HSM 算法和 HiCuts-1 算法的内存使用均超

过了 4GB 的系统内存。与之相比，HyperSplit-1 和 HyperSplit-8 的内存使用分别为 753MB 和 66MB。

3.4.4　预处理时间

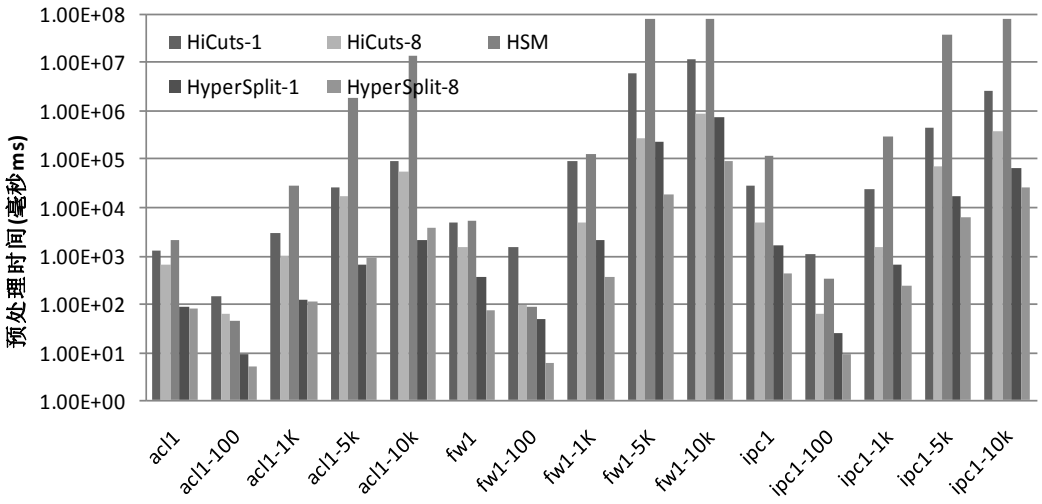


图 3.15　HyperSplit、HSM 和 HiCuts 算法的预处理时间[33]

从图 3.15 可知，HyperSplit 算法的预处理时间仅为 HSM 算法和 HiCuts 算法的 1/10～1/100。对于小规模的规则集合，例如 ACL1-100，HyperSplit 算法仅适用不到 10 毫秒的预处理时间。对于大规模的复杂规则集合，例如 FW1-10K，HyperSplit-1 和 HyperSplit-8 分别需要大约 10 分钟和 1 分钟进行预处理。与之相比，HSM 和 HiCuts-1 算法对 FW1-10K 的规则集合的预处理时间均超过 24 小时，且最终由于系统内存耗尽而无法完成预处理。

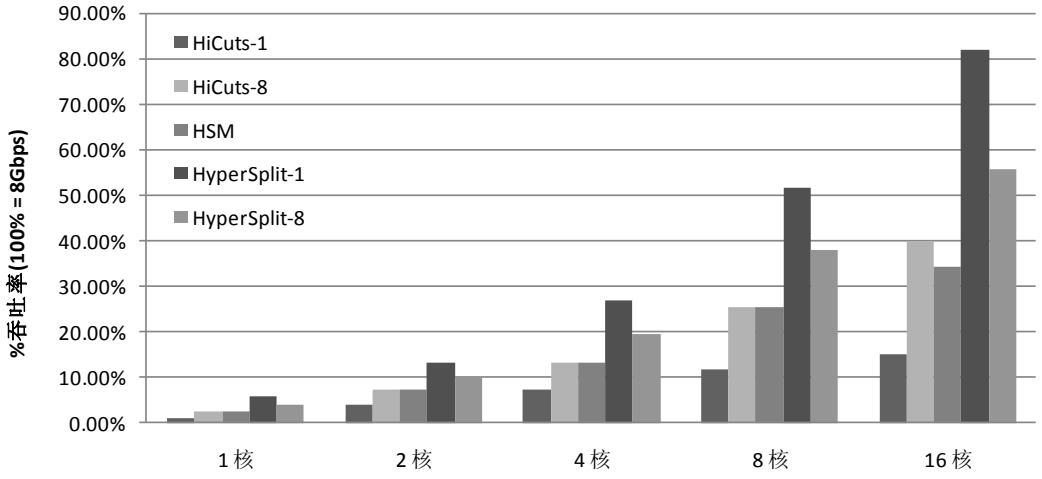


图 3.16　OCTEON3860 不同核数的网包分类吞吐率[33]

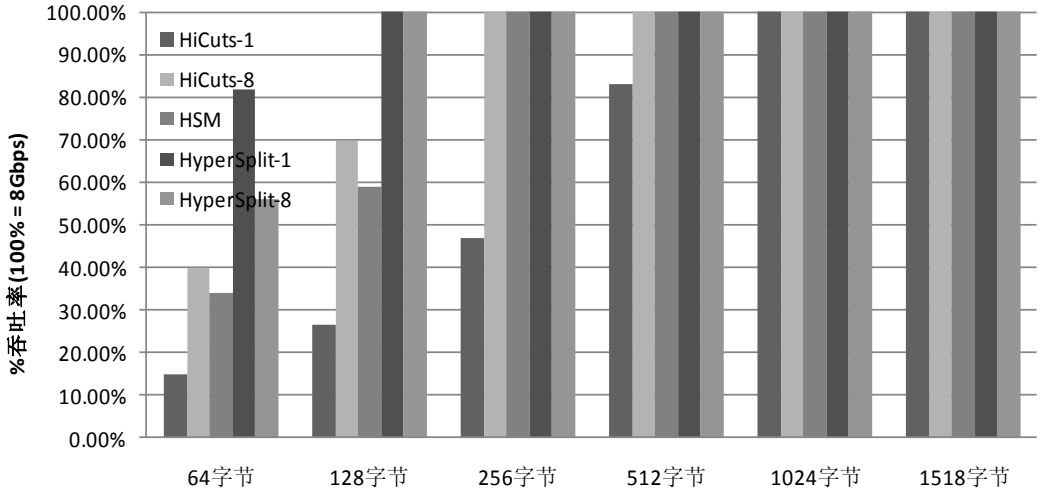


图 3.17　OCTEON3860 不同包长的网包分类吞吐率[33]

3.4.5　系统吞吐率

图 3.16 为 HyperSplit、HSM、HiCuts 算法实现于 OCTEON3860 多核网络处理器平台上的网包分类吞吐率。测试中使用了 ACL1-10K 的规则集合。测试网包为 64 字节以太网网包（由 SmartBit 测试仪[100]产生）。从图 3.16 可以看出，HyperSplit-1 算法在 16 核同时处理时有 6.4Gbps 的最大吞吐率。与之相比，HSM 算法和 HiCuts 算法仅能达到 3Gbps 左右的最大吞吐率。

图 3.17 为 HyperSplit、HSM、HiCuts 算法实现于 OCTEON3860 多核网络处理器平台上的对于不同包长的网包分类吞吐率。测试中使用了 ACL1-10K 的规则集合，并使用全部 16 个核进行测试。从图 3.17 可以看出，HyperSplit 算法在网包包长大于等于 128 字节时即能达到 OCTEON3860 的 100%吞吐率（8Gbps）。与之相比，HSM 算法和 HiCuts-8 算法只有当网包包长大于等于 256 字节时才能达到 100%吞吐率。HiCuts-1 算法的实际性能最低，仅当网包包长大于等于 1024 字节时才能达到 100%吞吐率。

3.5　本章小结

本章主要研究了基于规则投影区间查找的网包分类算法，并基于已有算法的分析和研究，提出了一种新算法 HyperSplit。研究中发现，算法的理论复杂度和实际性能并非完全一致。

首先，虽然理论上 HyperSplit 和 HSM 具有相同的时间复杂度，但 HyperSplit 的实际内存访问次数却明显少于 HSM。这是由于 HyperSplit 算法每次二分后仅对

规则子集而非规则全集的投影区间继续二分。

其次，虽然 HyperSplit 算法每次只将搜索空间二分，而 HiCuts 算法每次将空间多分，但 HyperSplit 算法的二分树深度却远小于 HiCuts 算法的多分树深度（HiCuts 算法将于第 4 章详细介绍）。其原因在于，HyperSplit 算法每次二分均选择的是最优的二分点，而 HiCuts 算法则仅仅对空间进行均匀切分。

另外，在大多数网包分类算法的研究中，通常认为网包分类速率跟内存访问次数成正比关系，即内存访问次数越少网包分类速率越高。然而，在基于多核网络处理器的网包分类系统实现中这一假设与网包分类的实际吞吐率并不相符。例如 HyperSplit-8 的吞吐率比 HSM 的更高，尽管 HyperSplit - 8 需要更多的内存访问次数。与之类似，虽然 HiCuts-8 需要更多的内存访问，但其实际性能却超过了 HiCuts-1。

这些实验结果揭示了理论分析的局限性。在实践中，实际网包分类系统处理一个网包的时间往往与内存访问次数并不成正比。原因在于内存访问可以在不同类型的内存中发生，而不同类型内存具有不同的延迟。根据 OCTEON3860 的处理器架构可知，由于 HyperSplit 内存使用小于 1MB，其内存访问将大多集中于低延迟的 L2 缓存。相比之下，虽然 HSM 的内存访问次数小于 HyperSplit-8，但由于 HSM 的多级求交表（ACL1-10K 的求交表大小为 49MB）无法存放于 L2 缓存，需要访问高延迟的 DRAM，因此 HSM 的吞吐率低于 HyperSplit-8。

综上所述，本章研究基于网包分类算法的理论分析，但同时兼顾了算法设计和实现。研究中提出的 HyperSplit 算法，无论分类速率、内存使用还是预处理时间均优于 HSM 和 HiCuts 等经典的网包分类算法。基于 HyperSplit 算法的网包分类系已经部署于高性能网关设备中，并达到了 20Gbps 的线速[56]。

第 4 章　空间分解算法

本章研究网包分类算法设计中的空间分解算法。首先依据第 2 章中的计算几何理论对空间分解算法的设计思想进行归纳；接下来介绍和分析几类典型的空间分解算法；然后提出一种创新的空间分解算法 AggreCuts[34, 57]；最后利用实际数据和多核网络处理器对空间分解算法的性能进行测试和比较。

4.1　理论依据

空间分解算法采用的基本策略也是分而治之，但空间分解算法不同于区间查找算法的根本之处在于采用对网包搜索空间而非规则投影区间的划分。依据第 2 章理论，使用空间分解的网包分类算法将原始搜索空间 S 分解为 m 个子空间，且满足如下约束：

$$S = S_1 \cup S_2 \cup \ldots \cup S_m$$

且 $\forall 1 \le i,j \le m$，有 $S_i \cap S_j = \emptyset$

与区间查找算法不同，绝大多数空间分解算法都对搜索空间进行了均匀分解，因此对于每一次空间分解，通常满足：

$$|S_i| = |S_j|$$

与每个子空间对应的规则子集合满足：

$$R = R_1 \cup R_2 \cup \ldots \cup R_m$$

且 $\forall r \in R_k, 1 \le k \le m$，有

$$S_k \cap r \neq \emptyset$$

由此可见，经过空间分解后，落入每个子空间的网包具有唯一的规则子集与之匹配。因此只要确定子空间归属即可完成网包分类。由第 2 节理论分析可知，空间分解算法的理论依据是 S-Trie 算法。该算法源自路由查找算法，使用多级 tire 结构并结合规则复制实现网包分类。该算法的时间复杂度为$\Theta(d * W)$空间复杂度为$\Theta((nW)^d)$。在算法设计中，各类空间分解算法对 S-Trie 算法进行了优化和拓展。一方面利用启发式方法改进 S-Trie 算法的空间分解效率，另一方面利用规则集合的冗余通过压缩数据结构。下面几个小节将分别介绍经典的空间分解算法以及创新的 AggreCuts 算法。

4.2　已有算法分析

最早的基于空间分解的网包分类算法是由 Tsuchiya 等提出的 Set-pruning Trie 算法[55]。基于 S-Trie 算法，学术界提出了一系列的网包分类算法[14-17, 21, 34, 58-64]。其中最典型的是 Gupta 等提出的 HiCuts（Hierarchical Intelligent Cuttings）算法[14]和 Singh 等提出的 HyperCuts 算法[17]。作为算法设计创新的依据，这里首先对这些典型空间分解算法进行介绍和分析。

4.2.1　S-Trie算法回顾

与第 2 章的理论算法分析不同，这里介绍 S-Trie 算法在网包分类中的应用。S-Trie 算法的基本思想是用空间换时间，通过规则复制避免 H-Trie 算法中的回溯查找。S-Trie 算法虽然简单，但却和当前性能最优秀的空间分解算法包含相同的设计思想，因此这里首先对 S-Tire 算法做一介绍。

S-Trie 对搜索空间的每一个维度依次进行逐级分解。每一次对搜索空间的分解均为单一维度上的二等分，即将当前搜索空间均匀划分为两个相同大小的子空间。每个叶节点存放包含当前搜索子空间的所有规则。

S-Trie 采用 *d* 级二叉树结构进行网包分类。每个内部节点包含两个指针，分别指向两个子节点；对于前 *d*-1 级 trie 结构，每个叶节点则包含一个 trie 指针指向下一级 trie 结构的根节点，而最后一级 trie 结构的叶节点则包含与当前子空间匹配的规则子集。图 4.1 为 S-Trie 算法进行二维网包分类的数据结构示例。

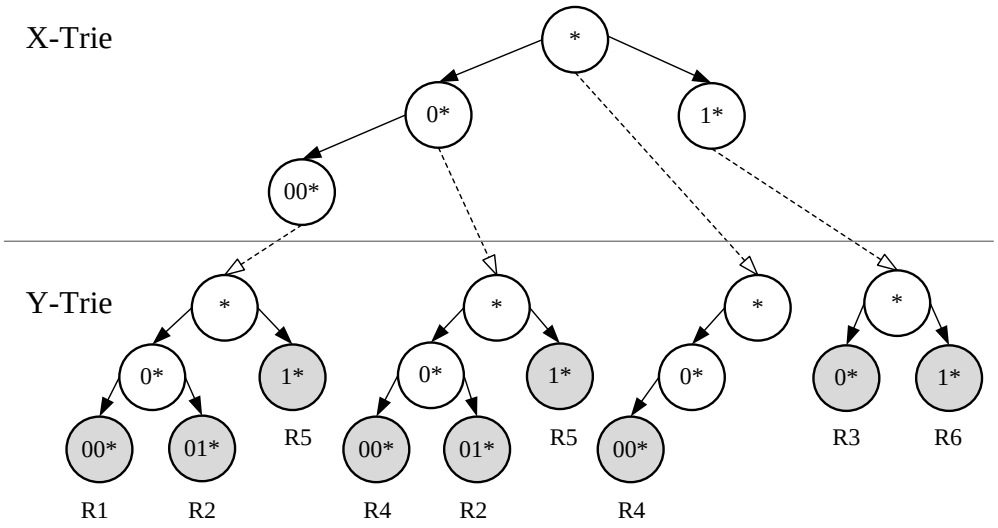


图 4.1　Set-pruning Trie 算法示例

S-Trie 的空间复杂度很高，而且算法需要范围－前缀转换，因此不适用于大规模的规则集合。另外，由于每次仅对搜索空间进行二等分，因此从空间分解效率来看，S-Trie 算法的效率并不高。

针对这一点，现有的空间分解算法采用各种启发式方法提高空间分解效率。一种可能的改进是使用多比特 trie 结构，即每次将空间分解为 2^w 等分，其中 $1 \le w \le W$，称为 trie 的跨度（stride）。但多比特 trie 会带来生成的树结构扇出（fan-out）过大，导致存储冗余。因此，典型的空间分解算法在设计中会控制 w 大小，并采用各种数据结构压缩方法来减少冗余的空间。下面介绍 HiCuts 和 HyperCuts 算法均基于上述设计思想进行算法设计。

4.2.2　HiCuts算法

Gupta 提出的 HiCuts 被公认为当前最优秀的网包分类算法之一。HiCuts 利用可变 w 的多比特 trie 结构进行空间分解，并结合叶节点上的线性查找进一步降低内存使用。HiCuts 使用多种启发式方法挖掘规则集合的特性，对网包搜索空间进行了合理有效的分解，达到了较好的时域和空域性能。下面从空间分解、数据结构两个方面来具体分析 HiCuts 算法。

- HiCuts 算法的空间的分解

HiCuts 采用多级空间分解，每一级的空间分解在一个特定维度上进行。在每一级空间分解中，HiCuts 首先通过可分离性判别函数（discrimination function）选取最具有可分性的维度，然后在该维度上对当前搜索空间进行均匀切分（cut）。

HiCuts 每一级的空间分解都由切分维度和切分次数来共同决定。HiCuts 总是选择最具有可分性的维度进行切分，因此与当前搜索空间相交的规则子集将在尽可能少的步骤内被划分成若干个规模较小的规则集合，从而提高查找效率。

HiCuts 对切分次数有明确的限制，即切分后每个子空间不能包含过多的规则。以此来保证内存的有效利用。由于采用了均匀切分，且切分次数为 2^w，HiCuts 在每一个节点上的查找效率为$\Theta(1)$。但这样的空间分解同时也使得同一条规则可能与大量子空间相交，从而产生大规模的规则复制。HiCuts 算法使用图 4.2 所示的指针数组来解决规则复制的问题。被相同规则子集所包含的子空间将进行合并，从而减少规则复制。

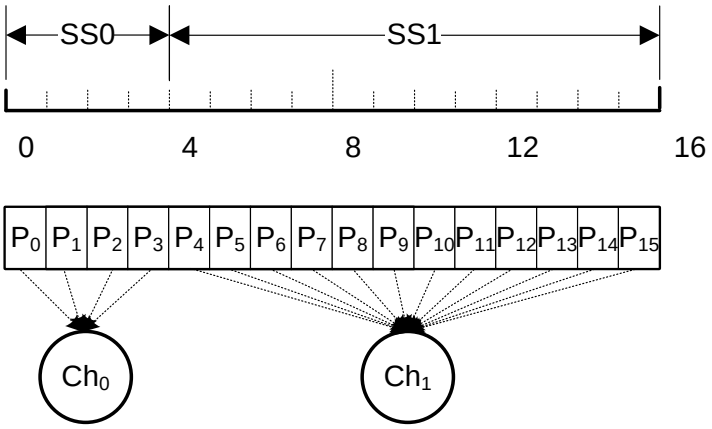


图 4.2　HiCuts 的子空间合并[64]

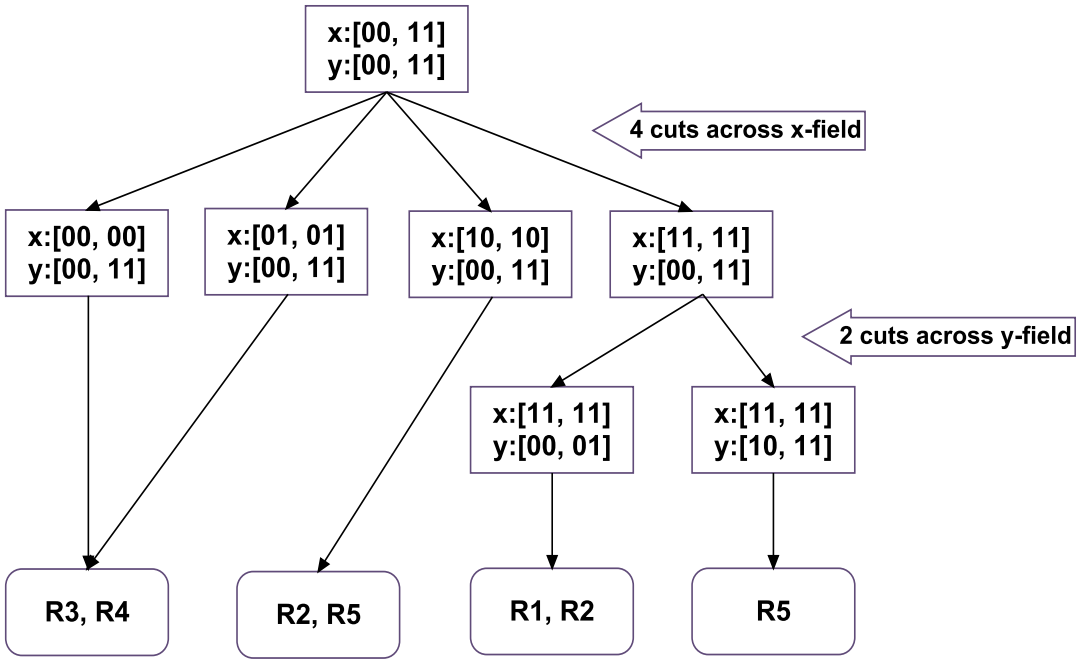


图 4.3　HiCuts 的数据结构示例

- HiCuts 算法的数据结构

HiCuts 采用基于多比特 trie 结构的决策树（decision tree）数据结构（如图 4.3 例）。每一个内部节点对应一个搜索子空间和与该子空间相交的规则子集，并存放切分维度，切分次数，指针数组。每一个叶节点对应一个不需进一步分解的搜索子空间，并存放与该子空间相交的规则子集。

HiCuts 的搜索过程首先做决策树查找，在决策树的每一个内部节点，由切分维度和切分次数等信息，求得网包归属的子空间的序号，再由此序号作为索引从

指针数组中读取下一级节点存储位置；当搜索过程到达叶节点后，HiCuts 对叶节点存放的规则子集进行先行查找，确定最终的匹配规则。

HiCuts 具有良好的可拓展性。具体表现在以下几个方面：

- 层次化空间分解

HiCuts 每一次对空间的分解为多等分而非 S-Trie 算法的二等分，因此空间分解效率得到提高。与此同时，HiCuts 并不是对空间进行一次性的完全分解，在其算法实现上表现为每一次空间分解都只涉及当前搜索空间的最高位的 w 个比特，即不同的节点可以采用不同的尺度进行空间分解，这样进一步提高了算法的适用性，有效的控制了内存使用。

- 启发式方法应用

HiCuts 在维度选择，切分数量选择上都使用了启发式方法。多种启发式方法的使得 HiCuts 算法能够有效消除数据结构中的冗余，使得每一次的空间分解都在特定目标函数下达到最优。

- 支持流水线搜索

HiCuts 的查找过程是顺序的，且与之前节点无关。因此在 FPGA 等硬件实现中，HiCuts 算法可以通过流水线映射实现并行处理。

HiCuts 算法的不足之处主要表现在以下几个方面：

- 分类速率不确定

虽然 HiCuts 算法比 S-Trie 算法对空间的分解效率有了提高，但由于其空间分解方式依赖于规则集合特性，因此对于不同特性的规则集合，HiCuts 的分类性能可能出现较大起伏。关键一点是，HiCuts 算法无法保证最坏情况下的分类速率，而这往往是网包分类问题中最重要的性能指标。

- 冗余的指针数组

HiCuts 在子空间去冗余过程中采用指针数组进行空间压缩。虽然子节点个数大大降低，但指针数组本身也需要大量的存储空间。实验发现，在 HiCuts 算法中一个节点通常只有不到 10 个子节点，但往往需要存储 256 个指针进行空间压缩。从整体内存使用分布来看，指针数组需要占用 HiCuts 数据结构的 90%以上存储空间。

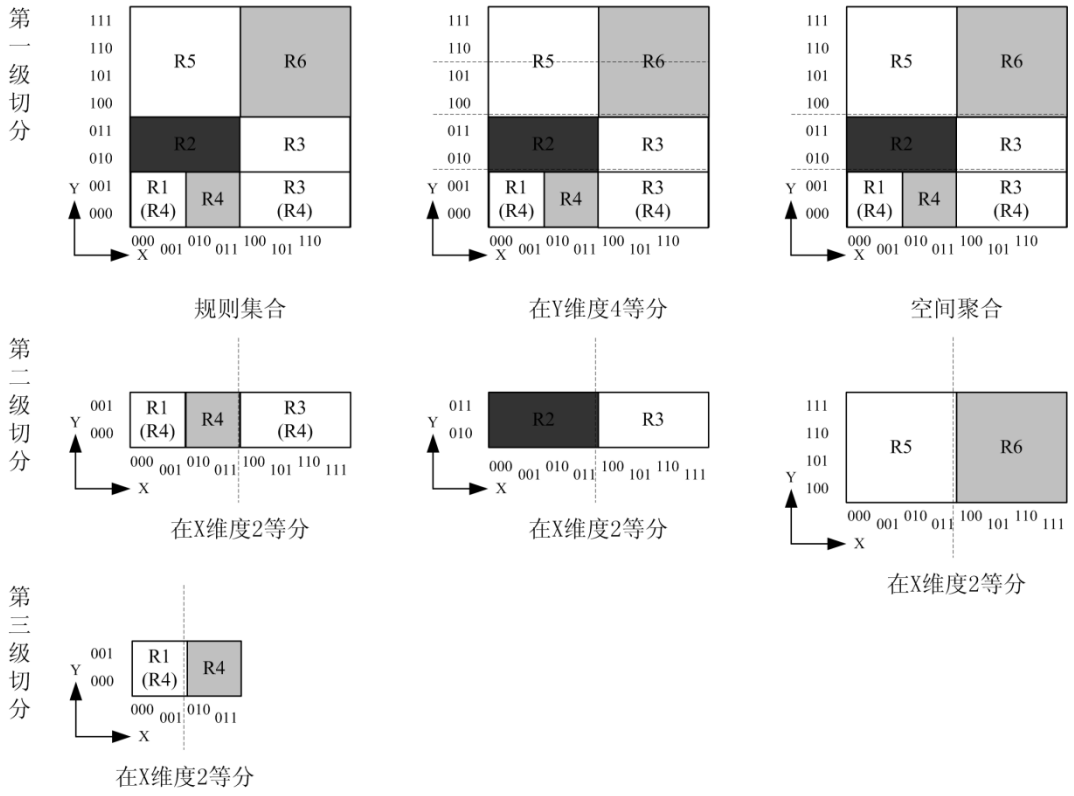


图 4.4 HiCuts 算法的空间分解

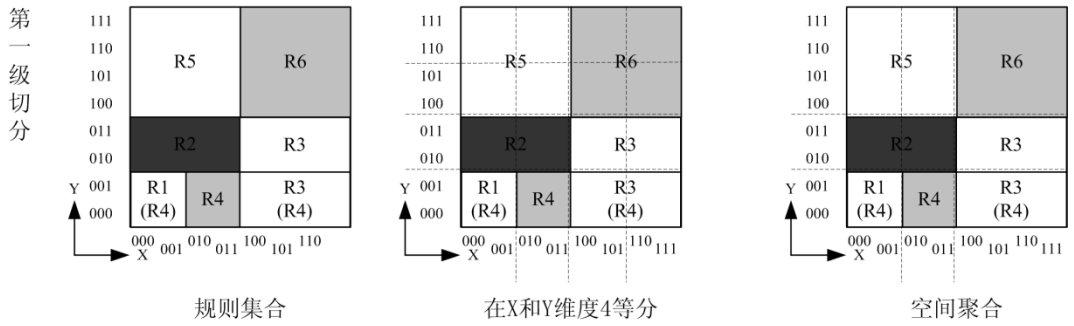


图 4.5　HyperCuts 算法的空间分解

- 预处理时间过长

HiCuts 在决策树建立过程中使用了多种启发式算法，这些算法大量使用枚举的方式进行局部寻优。随着规则数量的增加， HiCuts 算法的预处理时间越来越长。实验表明，对于 10000 条防火墙规则，HiCuts 算法需超过 1 小时的预处理时间。[33]

4.2.3　HyperCuts算法

HyperCuts 通过一个节点上同时对多个维度进行空间分解来提高 HiCuts 的空间分解效率。在查找效率上，HyperCuts 算法中的一步的查找相当于多步的 HiCuts 查找，因此查找效率得到提高。在内存利用率上，HyperCuts 实际上省去了 HiCuts

中可能出现的部分中间节点，从这个角度上来讲，HyperCuts 也使内存利用率得到了提高。

图 4.4 和图 4.5 分别给出了 HiCuts 和 HyperCuts 的空间分解示意图。图 4.4 中的 HiCuts 算法通过三级切分完成搜索空间分解而图 4.5 中的 HyperCuts 算法仅通过一级切分即完成搜索空间的分解。

与 HiCuts 算法不同，HyperCuts 算法通过指针矩阵而非指针数组进行子空间压缩。虽然指针矩阵能够大幅降低子节点的个数，但大量的指针矩阵本身会占用相当多的内存空间。例如，一次空间分解同时在两个维度上进行，每个维度切分次数为 2^8，那么该节点将需要包含 $2^8\times2^8=2^{16}$ 个指针的指针矩阵。对于 64 位（8 字节）的指针地址，这个指针矩阵将占用 $2^{16}\times8=512$KB 的内存空间。对于一棵有数万节点的 HyperCuts 决策树，这样的内存使用是无法接受的。

大量的指针矩阵的出现引发了一系列算法设计中的问题：

首先，HyperCuts 算法每一个维度上的切分次数不能过大，因为必须对切分次数进行有效的限制才能避免指针矩阵过大的问题。而较小的切分次意味着空间分解效率的降低，将导致决策树深度增加。

其次，如何限制切分次数，尤其是在多个维度上对切分次数进行优化，本身是一个很难解决的问题。虽然 HiCuts 和 HyperCuts 算法都给出了各种优化方法，包括各种形式的维度选择函数和内存约束函数等，但实验表明没有一种优化方法在不同的规则集合中都能表现出很好的性能。

另外，即使找到了某种可行的优化方法，那么它可能是非常复杂的，会带来很多计算上的负担。而过于复杂的计算势必影响到预处理时间。实验表明，目前用于 HiCuts 和 HyperCuts 等算法中的相对简单的启发式方法已经需要超过一小时的时间处理较为复杂的防火墙规则集合（如 FW1-10K）。因此更复杂的优化方法可能会带来更加耗时的预处理过程。

4.3　AggreCuts算法

AggreCuts 是本文提出的一种搜索空间分解的网包分类算法。AggreCuts 算法基于 HiCuts 算法，但解决了 HiCuts 算法中指针数组的压缩问题。同时，AggreCuts 算法在分类速率和预处理性能上也比 HiCuts 算法有了显著提高。下面分别介绍 AggreCuts 的设计思想、理论依据和算法设计。

4.3.1　设计思想

近年来的研究指出，将 HiCuts 和 HyperCuts 直接用于实际网包分类系统中存

在诸多缺陷[21]。首先，由于启发式算法每级切分次数不同，导致决策树深度不确定，进而使得系统吞吐率无法保障。其次，由于每个内部节点都使用指针数组或指针矩阵连接子节点，在切分次数增长的情况下，指针结构的存储空间随之增长。在实际系统中，过高的内存需求将导致网包分类算法难以利用有限的高速内存资源实现快速的网包分类。此外，由于 HiCuts 和 HyperCuts 每个节点的大小不一致，使得查找过程中每次访存（memory access）所读取的字节数以及相应的节点处理过程不一致。这种不一致使内存分配单元变得复杂，并降低硬件处理单元的效率。为解决这些问题，本文提出了 AggreCuts 算法。

AggreCuts 算法的基本设计思想包括：

- 确定性切分次数

AggreCuts 算法中采用了常数的 w，每次空间分解均使用 2^w 的切分次数。假设网包包头长度为 W 位，则 AggreCuts 可以保证通过 W/w 次空间分解完成网包分类。由此可见，确定性的切分次数选择保证了 AggreCuts 具有明确的时间复杂度。若取 w=8，则对于典型的五元组分类 AggreCuts 可以通过(32+32+16+16+8)/8=13 次查找完成。

- 指针数组压缩

AggreCuts 算法使用了 BITMAP 压缩技术对指针数组进行压缩。对于每个内部节点的 256 个指针， AggreCuts 使用 256 位的 BITMAP 进行压缩，大大减少了内存使用。在查找过程中的 BITMAP 计算则利用多核网络处理器中的硬件指令进行高速处理。

- 节点数据结构优化

由于指针数组中的指针个数经过比特串压缩后大大减少，而且节点信息与指针大小相比近似，因此 AggreCuts 将节点信息和指针数组融为一体，使每个节点的访问减少了一次内存读取。

4.3.2　算法设计

由于每级节点使用确定性的空间分解次数，原始 HiCuts 算法的指针数组会大幅增长。因此，主要的优化任务变成了如何有效地减少内存使用。为了有效地减少这些指针数组的内存使用情况，AggreCuts 采用了比特串压缩技术

。

AggreCuts 算法使用聚合比特串 ABS（aggregation bit string）记录每个指针数组中不同的指针排列，然后消除冗余的指针，得到压缩后的指针数组 CPA（compressed pointer array）。具体来讲，ABS 中的每一个比特代表原始指针数组中的一个指针，而 CPA 中存放的是原指针数组中所有与前一个指针不同的指针。

ABS 和 CPA 的设置方法如下：

- ABS 的第 1 个比特设为 1，CPA 的第 1 个指针设为原始指针数组中的第 1 个指针，令 j=1。
- 若原始指针数组的第 i（i>1）个指针与第 i-1 个指针相同，则 ABS 的第 i 个比特为 0。
- 若原始指针数组的第 i（i>1）个指针与第 i-1 个不同，则 ABS 的第 i 个比特为 1，并令 j=j+1，设 CPA 中的第 j 个指针为原始指针数组中的第 i 个指针。

基于上述设置，原指针数组的第 i 个指针可以通过下列方法获取：统计 ABS 中前 i 个比特中 1 的个数（比特求和），设为 j；则该指针为 CPA 中的第 j 个指针。图 4.6 为比特串压缩示例。

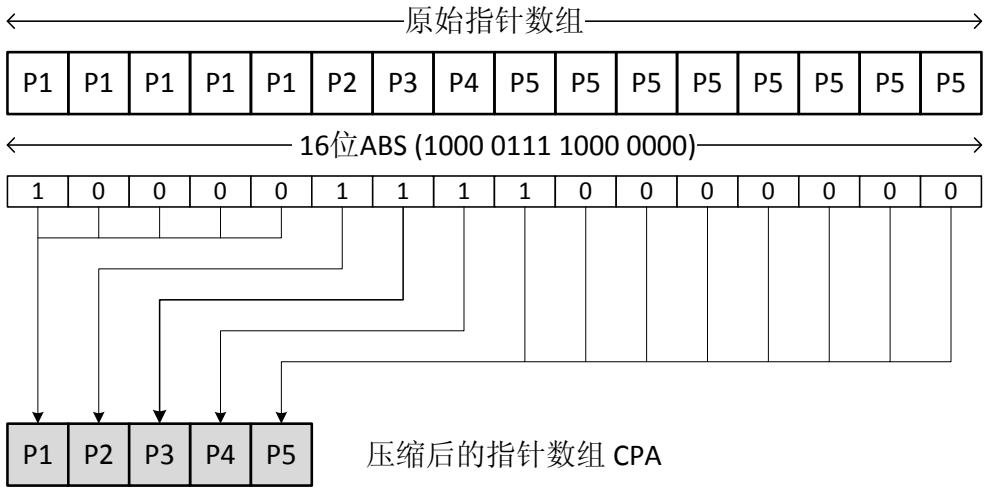


图 4.6　AggreCuts 比特串压缩

理想的情况下，所有指针数组（包括 HyperCuts 中的指针矩阵）都可以使用 ABS 和 CPA 压缩。然而，由于 AggreCuts 算法选择 w=8，即 2^8=256 作为切分次数，每个 ABS 的长度将为 256。虽然在高端网络处理器（如 Intel IXP2850）和 FPGA 上进行 256 长度的比特串求和仅需要几个时钟周期，但为在更多的系统平台上高效运行，AggreCuts 采用了层级化的比特串 HABS（hierarchical aggregation bit string）压缩技术。可以使用不同长度的比特串对原始指针数组进行压缩。

HABS 的设计来源于大量实验数据的观察。实验中发现，AggreCuts 算法生成的决策树中，每一个内部节点仅产生为数不多的子节点。对于 256 切分，通常只会生成不到 10 个节点。上述实验结果与文献 [3] [9] [10]中的实验结果基本吻合。由此可见，256 长度的原始指针数组是非常稀疏的，与之对应的 ABS 绝大多数位为 0。基于这个观察，HABS 使用了 1 个比特来代表 ABS 中的多个比特，代价是少量复制 CPA 中的指针。

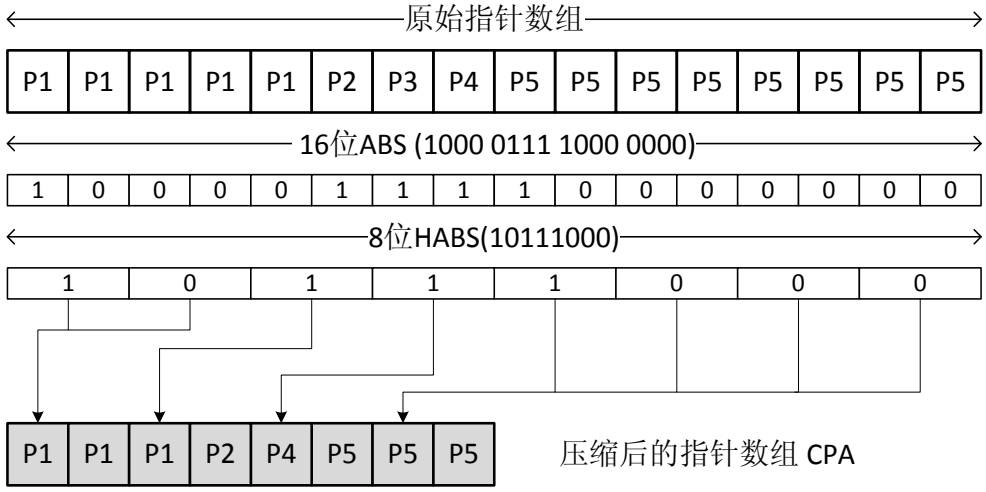


图 4.7　层次化比特串压缩

HABS 的压缩方法见图 4.7 示例。设 HABS 的长度为 2^v， 原始指针数组的大小为 2^w，且 $u=w-v$，则压缩算法为：

- 将 2^w 长度的指针数组分为 2^v 长度的子指针数组。
- 将 HABS 的第 1 位设为 1，并在 CPA 中存储第 1 个子指针数组。
- 若第 i（i>1）个子指针数组与第 i-1 个子指针数组内容一致，则设 HABS 的第 i 位为 0。
- 若第 i（i>1）个子指针数组与第 i-1 个子指针数组内不一致，则设 HABS 的第 i 位为 1，并在 CPA 中存储第 i 个子指针数组。

经过 HABS 压缩后，可以通过下列步骤得到第 $k(1 \le k \le 2^w)$个子空间对应的子节点地址：

- 取 k 的最高 v 比特，作为值存于 m，即 $m=k>>u$
- 取 k 的最低 u 比特，作为值存于 j，即 $j=k-(m<<u)$
- 统计 HABS 中前 m 个比特中 1 的个数（比特求和），得到 i
- 以$(i<<u)+j$ 为索引读取 CPA 中的指针，该指针等于原始指针数组中的第 k 个指针。

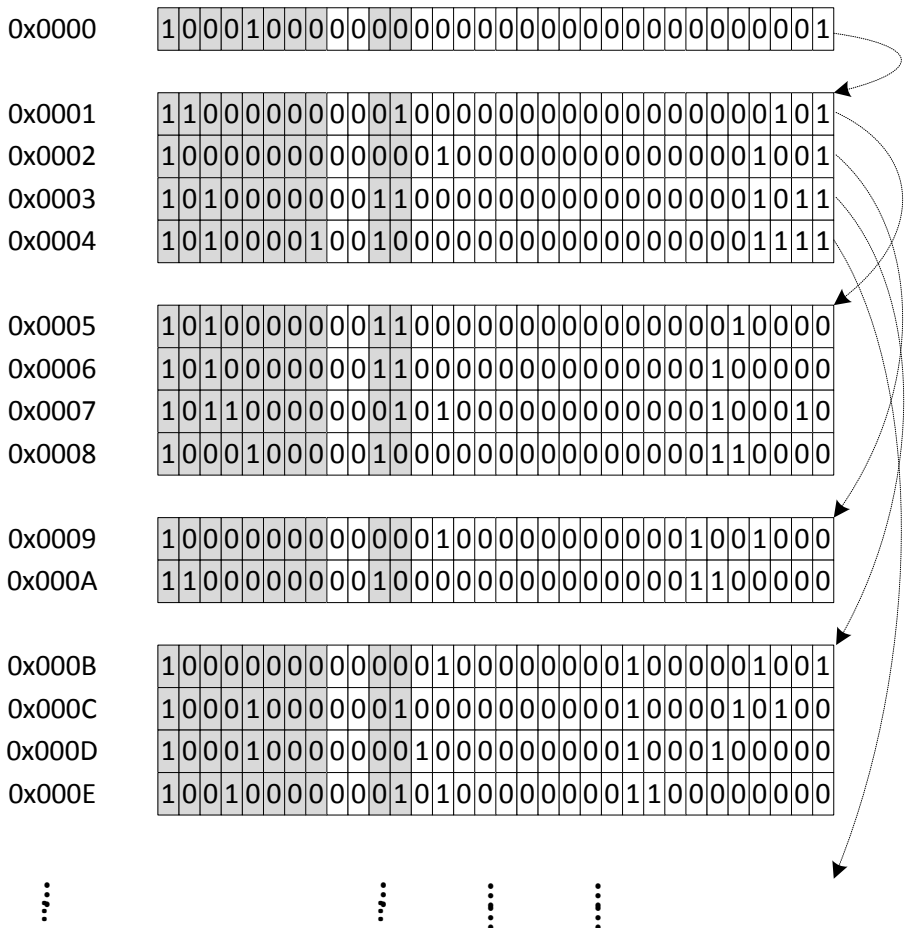


图 4.8　AggreCuts 的数据结构示例[34]

表 4.1　AggreCuts 节点数据结构

位	存储信息	取值含义
31:30	切分维度	d2c=00: src IP; d2c=01: dst IP; d2c=10: src port; d2c=11: dst port.
29:28	切分位数	b2c=00: 31~24; b2c=01: 23~16 b2c=10: 15~8; b2c=11: 7~0
27:20	8 比特 HABS	若 w=8，每比特表示 32 个子空间 若 w=4，每比特表示 2 个子空间
19:0	20 比特子节点起始地址	因为最小的内存块为 $2^w/8\times4$ 字节，所以 若 w=8, 20 比特可表示 128MB 内存空间 若 w=4，20 比特可以表示 8MB 内存空间

实验表明，即使使用 8 比特的 HABS，依然能得到良好的压缩效果。因此，在 AggreCuts 算法中，采用 32 比特的节点数据结构。表 4.1 总结了在 32 位处理器上实现 AggreCuts 算法的节点数据结构。图 4.8 则为 AggreCuts 算法在实际应用中的存储示例。

4.4　性能评价

本节比较了 AggreCuts 算法和 HiCuts 算法的内存访问次数、内存使用和基于 Cavium OCTEON3860 和 Intel IXP2850 多核网络处理器平台的吞吐率。

4.4.1　测试数据及平台

表 4.2　测试用规则集合[34]

规则集合名称	规则个数	规则特点
SET1	68	目标 IP 的掩码多为 30~32 位 源 IP 的掩码多为 0 或 16 位
SET2	136	
SET3	340	
SET4	500	
SET5	1,000	目标 IP 的掩码多为 32 位 源 IP 的掩码在 0～32 位间变化
SET6	1,530	
SET7	1,945	

本章算法测试使用的规则集合见表 4.2。这些规则集合来自网络核心路由器和大型校园网。规则命名从 SET1～SET7，不同规则集合中包含的规则数目从 68 到 1945 不等。所有规则都由 IPv4 五元组构成：包括 32 比特源 IP 地址、32 比特目标 IP 地址、16 比特源端口号、16 比特目标端口号、8 比特传输层协议。

算法预处理在一台 Intel x86 架构的服务器上进行。该机器拥有一颗 2.0GHz 的 Intel core2 处理器，以及 4GB DDR2 内存。算法预处理中使用的代码均使用 C 语言开发，并使用 gcc –O2 指令编译。操作系统为 Ubuntu 8.04 LTS。

算法的吞吐率实验基于 Intel IXP2850 多核网络处理器[49]。IXP2850 是 Intel IXA（Internet exchange architecture）架构芯片的高端产品，是网络处理器设计中的典型代表。其体系结构图如图 4.9 所示。IXP2850 是典型的多核架构，其内部封装有 16 个可编程微引擎（micro engine，ME）和一个工作在 600MHz 的 Xscale 处理器。64 位 IX Bus 将微引擎、Xscale、存储器以及 MAC 等片外设备连接到一起。PCI 总线则用来和外部设备通信完成四层以上的更高层网络处理。IXP2850 的主要特点如下：

- 并行处理：16 个微引擎（ME）和 1 个 Xscale 处理器构成。这些处理器可以并行工作，ME 负责数据平面（data-plane）的高速网包处理，通过对 16 个 ME 及其 128 个硬件线程分配不同功能的微码程序，可以实现网络负载的静态或动态的均衡。
- 分布式数据存储结构：每个 ME 独立使用 256 个 32 位寄存器。其中 128 个寄存器是传送寄存器集。每个微引擎将数据载入传送寄存器集，对传送

寄存器集进行操作，然后通过传送寄存器集写到数据目的地。数据载入传送寄存器集后，微引擎可在单指令周期完成访问。

- 硬件多线程：每个微引擎有 8 个编程计算器，支持 8 个硬件线程。每个线程可以执行相同或不同的微码程序，采用内部线程通信机制实现线程同步，提高系统效率。微码指令采用 5 级流水线机制，执行周期为 1 个时钟周期。
- 内存管理：IXP 系统的 SDRAM 和 SRAM 支持多个读写队列进行优先级排队以优化带宽。允许多个线程同时提交对内存单元的读写请求，内存单元根据特定优化指令对读写请求硬件优先级排队。

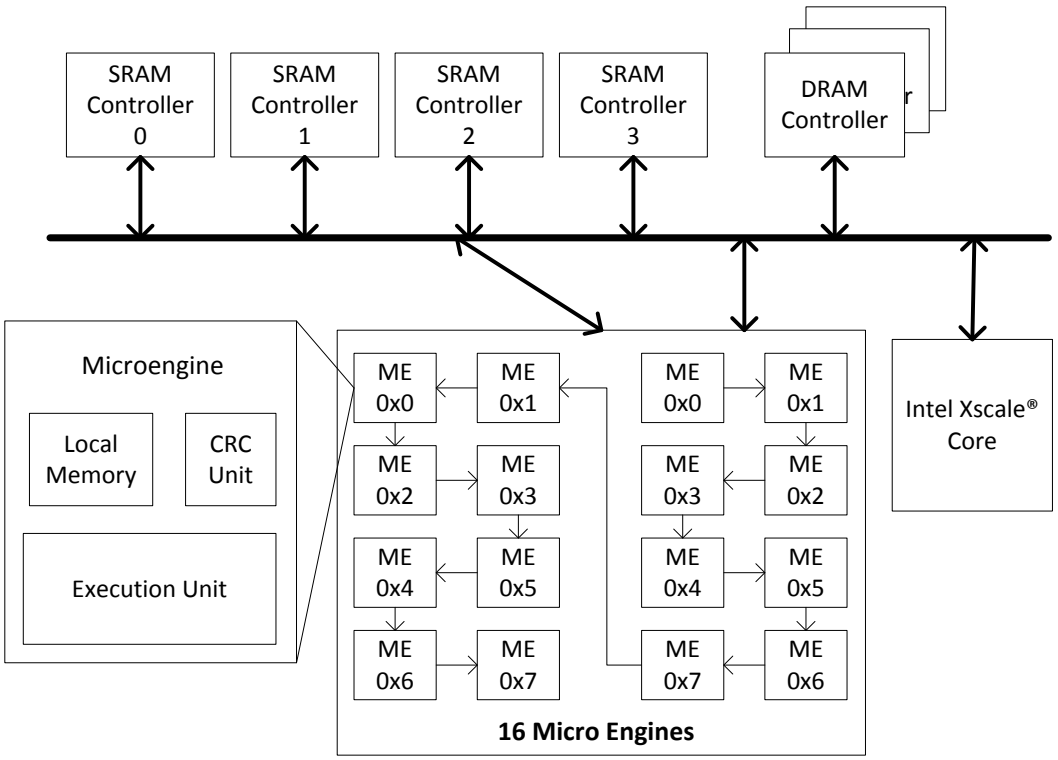


图 4.9　Intel IXP2850 多核网络处理器[49]

4.4.2　分类速率

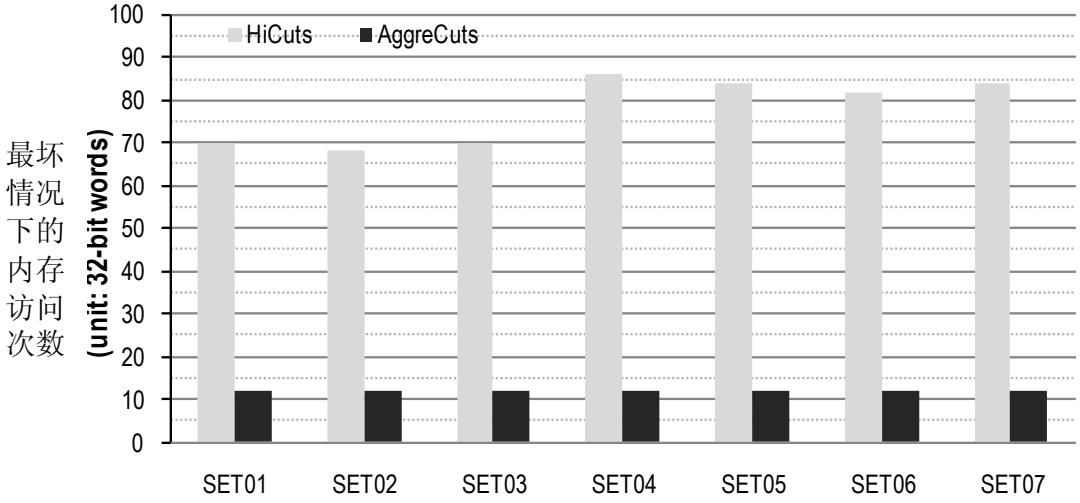


图 4.10　AggreCuts 与 HiCuts 的内存访问次数[57]

图 4.10 比较了 AggreCuts 算法和 HiCuts 算法的分类速率。评价指标为 32 比特长字（考虑到 Intel IXP2850 处理器的 32 比特寄存器）的内存访问次数。从图中可以看出，AggreCuts 算法的内存访问次数为固定值，且仅为 HiCuts 算法内存访问次数的不到 20%。

4.4.3　内存使用

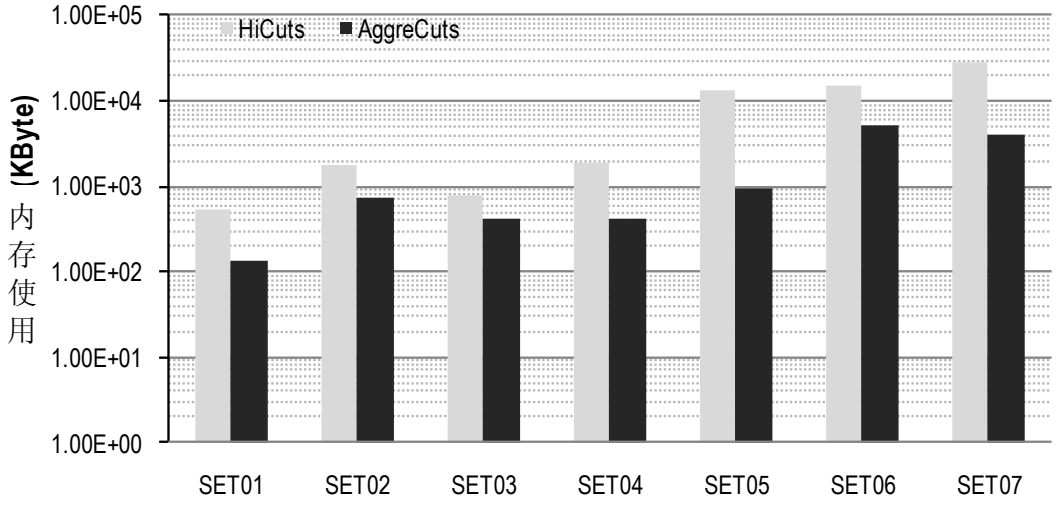


图 4.11　AggreCuts 和 HiCuts 的内存使用[57]

从图 4.11 可以看出，对于所有规则集合，AggreCuts 算法均比 HiCuts 算法的内存使用少 1～2 个数量级。

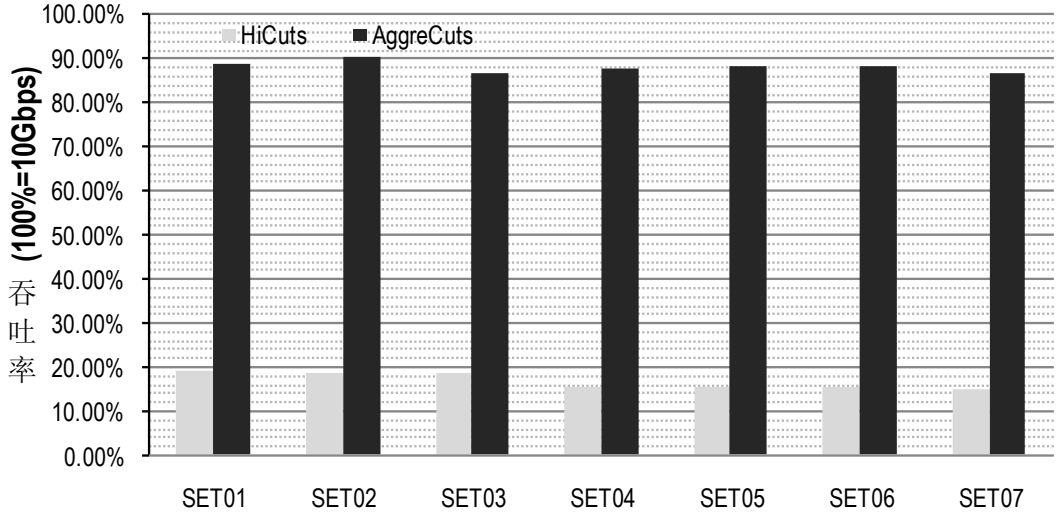


图 4.12　AggreCuts 和 HiCuts 在 Intel IXP2850 上的吞吐率[57]

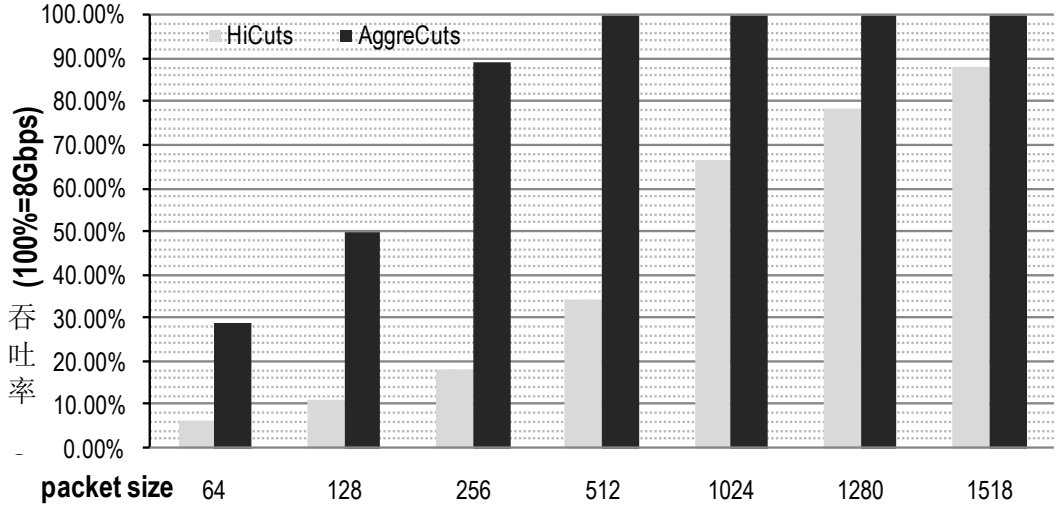


图 4.13　AggreCuts 和 HiCuts 在 Cavium OCTEON3860 上的吞吐率[57]

4.4.4　系统吞吐率

AggreCuts 与 HiCuts 实现于 Intel IXP2850 及 Cavium OCTEON3860 多核网络处理器上的吞吐率分别由图 4.12 和图 4.13 给出。由图 4.12 可见，测试网包大小为 64 字节时，AggreCuts 对于所有的测试规则集合，均有近 9Gbps 的吞吐率。与之相比，HiCuts 算法的吞吐率不到 2Gbps。对于大小超过 64 字节的网包，AggreCuts 算法均能在 IXP2850 网络处理器上达到 10Gbps 的线速。由图 4.13 可见，当网包包长大于等于 512 字节时，Aggrecuts 算法在 OCTEON3860 多核网络处理器上能够达到 8Gbps 的线速。

4.5　本章小结

本章主要研究了基于网包搜索空间分解的网包分类算法，并提出了一种新算法 AggreCuts。AggreCuts 通过固定的切分次数，保证了算法分类速率。同时利用比特串压缩方法，大幅降低了指针数组的内存需求，获取了较好的空间性能。

从数据结构设计来看，AggreCuts 的数据结构在全局上是使用类似 HiCuts 的决策树，但通过层次化比特串等结构完全消除了 HiCuts 算法中的指针数组。AggreCuts 算法在设计中充分考虑了在新一代网络处理器平台上实现的可能性。AggreCuts 算法在查询过程中每一次内存访问都限制在 32 比特内，并且只需要移位和加减运算。AggreCuts 算法也考虑到了相关硬件指令（如比特串求和指令的使用），这进一步为其在网络处理器平台上的顺利实施提供了保证。

综上所述，本章研究基于网包分类算法的理论分析，但同时兼顾了算法设计和实现。研究中提出的 AggreCuts 算法，无论分类速率、内存使用均优于典型的空间分解算法。AggreCuts 算法用于 Intel IXP2850 和 Cavium OCTEON3860 多核网络处理器，可分别获取 10Gbps 和 8Gbps 的网包分类吞吐率.

第 5 章　算法硬件实现

虽然在过去的十多年中，出现了许多多域网包分类算法，但仍无法满足不断增长的性能需求。一方面，基于多核网络处理器的高性能包分类系统虽具有较好的灵活性和可编程特性，但其固有缺点是缺少高可并行性和大量的片上存储。因此，即使是运行于多核网络处理器的最好的软件网包分类算法，也只能达到 10Gbps。这远远低于大的 ISP 骨干网和数据中心网络所要求的 100Gbps 处理能力。另一方面，基于 TCAM 的解决方案可以达到线速性能。但 TCAM 因其功耗问题、范围前缀转换问题始终难以支持大且复杂的规则集合。

因此，可扩展的高性能网包分类系统一直是研究热点。本章介绍了基于 FPGA 硬件平台的高性能网包分类算法设计及实现，主要内容包括硬件算法综述，以及基于 HyperSplit 算法的网包分类引擎设计实现。

5.1　基于FPGA的已有算法分析

与软件算法设计不同，基于硬件的网包分类算法具有更严格的约束条件。以 FPGA 平台为例，进行硬件算法设计需要综合考虑如下因素：

- 计算单元：由于 FPGA 中可编程的逻辑单元资源有限，同时复杂的逻辑指令会导致最高可用时钟频率下降，因此算法在查找过程中的计算单元必须精简高效。
- 存储单元：FPGA 中的片上存储资源总量非常有限。因此硬件算法需要在严格控制内存使用，尽量避免对外部存储的依赖。
- 算法并行性：虽然 FPGA 的计算和存储单元资源有限，但它们均可并行工作。要得到高性能的硬件算法，必须充分利用 FPGA 平台的平行特性。

基于 FPGA 平台的网包分类算法包括[35-43, 67-69]。大部分现有的基于 FPGA 的包分类引擎都是基于并行或流水线搜索算法。Jedhe 等人在 Xilinx Virtex-2 Pro FPGA 上实现的 DCFL 架构达到了 128 条规则 16Gbps 的吞吐率（测试 128 条规则，包长 40Bytes），并声称如果采用 Virtex-5 FPGAs，吞吐率可以高达 24Gbps[67]。

Luo 等人提出的被称为 Explicit Range Search 的方法基于 HyperCuts 算法[68]，通过在每个节点上进行更多次的空间分解，决策树的深度随着存储的增长而递减。由于该算法每个内部节点内存访问次数不确定，因此难于实现高效的流水线映射。

从系统功耗考虑，Kennedy 等人在 Altera Cyclone-3 FPGA 上实现了一个简化的 HyperCuts 算法[69]。由于高达一百条规则被存储于每个叶节点进行并行化查找，该算法只能够运行于低的时钟周期。更近一步的，由于决策树搜索并没有进行流水线映射，使得该算法在处理大规则集合时只能达到 0.47Gbps 的吞吐率。该算法在查找 20K 防火墙规则时，需要花费 23 个时钟周期才能够完成一次网包分类。

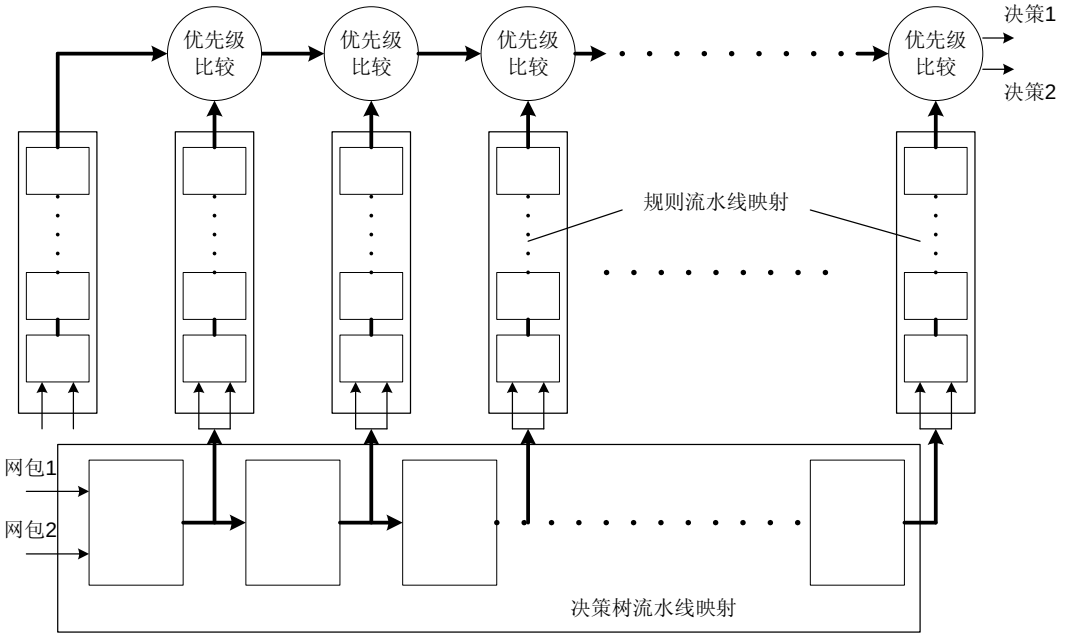


图 5.1　HyperCuts 的二级流水线映射[37]

Jiang 等人提出了 HyperCuts 算法的两个优化方法以降低内存消耗[37]。如图 5.1 所示，该算法将 HyperCuts 的各级节点映射到横向的决策树查找流水线上，而将各级节点对应的内部规则列表（internal rule list）映射到纵向的规则查找流水线上。算法的查找从根节点开始，每经过一级决策树节点，都会分两路进行下一步查找。一路转入下一级决策树节点继续查找，另一路则转入当前决策树节点对应的规则查找流水线进行一一匹配。该算法通过引入二维流水线，巧妙解决了 HyperCuts 算法中的内部规则查找问题。一方面使每个时钟周期（cycle）都能处理一个网包，另一方面减少了节点计算单元的复杂性，从而获取了超过 100MHz 的时钟频率（对应 80Gbps 的 64 字节以太网网包吞吐率）。通过深度流水线作业，FPGA 实现可以达到 80Gbps 的吞吐率。然而，由于流水线深度由决策树深度来决定，它们的架构对深度可变的决策树而言并不适用。

5.2　基于FPGA的HyperSplit算法

为了实现高性能的多域网包分类系统，基于 FPGA 的解决方案需要满足如下设计要求[35]：

- 算法并行性：为了达到线速网包分类，算法实现需要能够在 FPGA 设备上进行有效地并行计算。
- 逻辑复杂性：为了达到高的时钟频率，每个流水线阶段的组合逻辑需要简单有效。
- 内存利用率：为了支持大的规则集，内存的利用应当小到与片上 BRAM 容量相匹配。

本章选用区间查找算法 HyperSplit 进行 FPGA 的硬件实现，理由如下：

- 首先，二分查找树的树形结构可以流水线实现，使得每个时钟周期都能达到高吞吐率。
- 其次，每个树节点的操作比较简单。二分点的值比较和取地址操作都可以通过简单的逻辑单元实现。
- 此外，HyperSplit 的内存使用少。实验结果显示，50,000 条规则也只需要不高于 6MB 的内存空间[33]。

这些特性都表明，HyperSplit 的所有的数据结构都能够很好的适用于现代的 FPGA 芯片（如 Xilinx Virtex-6）。下面从算法映射、内存管理及并行处理三个层面介绍基于 FPGA 硬件平台的 HyperSplit 算法实现。

5.2.1　算法映射

在 HyperSplit 算法的硬件实现中，首先将 HyperSplit 的二分查找树映射到 FPGA 上的流水线作业。如图 5.2 所示，二分查找树上具有相同深度的节点将被映射到流水线的一个阶段。由于流水作业在每个阶段并行处理多个网包，且每个阶段仅需要一个时钟周期，因此网包分类速率等于 FPGA 芯片的时钟频率。

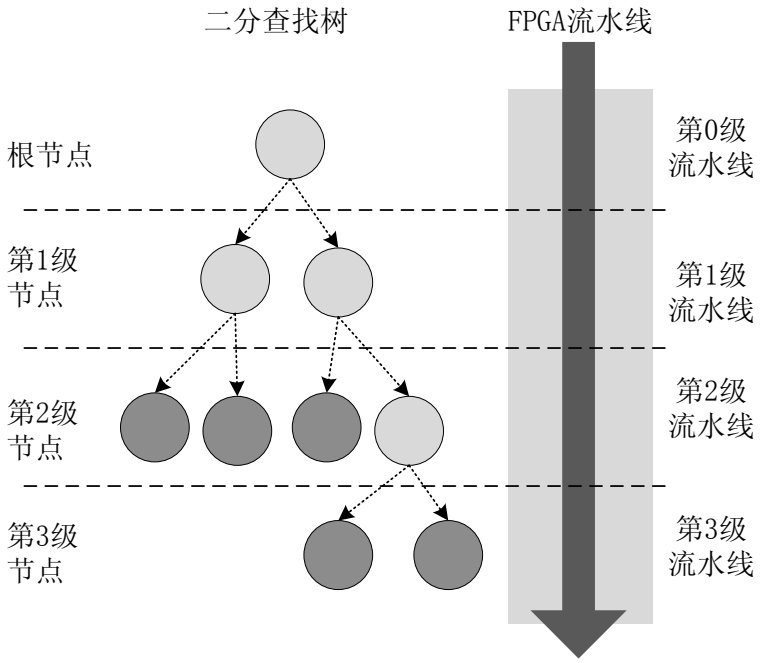


图 5.2　HyperSplit 算法映射[35]

在图 5.2 的算法映射中，由于执行相同的二分查找，每个阶段的处理逻辑是相同的。如图 5.3 所示，每个节点的处理逻辑有三个输入，分别为 packet_in, node_in 和 found_in。其中:

- packet_in：104 比特包头，由 32 比特目的 IP 地址，32 比特源 IP 地址，16 比特目的端口，16 比特源端口和 8 比特的传输层协议号组成。
- node_in：64 比特的二分查找树的节点数据，包括 32 比特的域值，4 比特的域指示和 28 比特的下一节点地址。
- found_in：1 比特的信号，表明在前一阶段，是否最佳匹配规则已找到。

每个节点的输出包括 104 比特的 packet_out，64 比特的 node_out 和 1 比特的 found_out。

在每一阶段的处理过程中，输入网包首先读入寄存器中。寄存器的输出是相关的二分点取值。多路器由 node_in 中 4 比特的域指示控制，用来选择相应的域去比较。类似的，32 比特二分点值和多路器的输出都作为比较分析器的输入。比较结果决定多路器的输出是否不大于二分点值。该值将驱动另一个多路器来选择 next_node_address 指示的左子节点或者 next_node_address+1 指示的右子节点（在 HyperSplit 算法映射中，两个孩子节点顺序存储于 BRAM 中的连续地址空间）。这个地址将作为输入，在下一个时钟周期送入下一级流水线阶段。

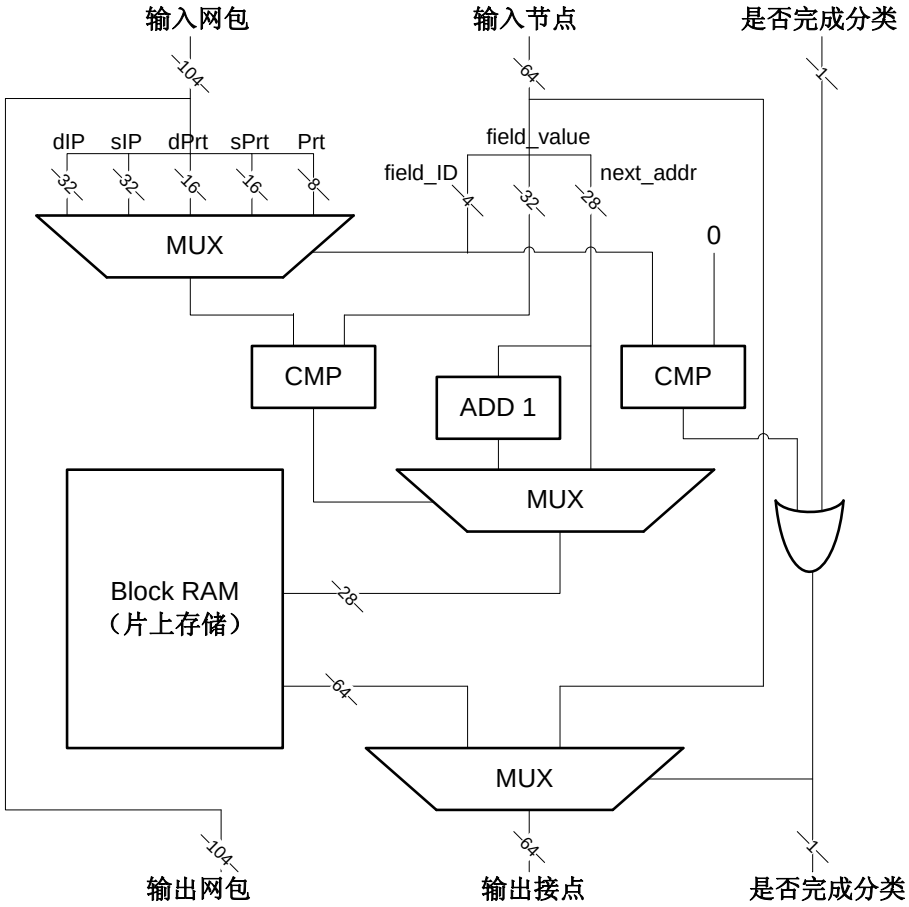


图 5.3　FPGA-Split 算法硬件逻辑单元[35]

为了确定当前查找过程是否已达到叶节点，也就是说最佳匹配规则是否找到，算法映射中使用了如下设计：首先，硬件实现中的叶节点可以根据域指示来指定。当指示值为 0 时，表明到达叶节点，found_out 作为信号量置 1，流水线的下一阶段将根据该值找到最佳匹配规则。

流水线作业中的内存通过插入 Write Bubble 来更新[70-71]。每一流水线阶段中新的树节点更新内容首先进行离线计算。然口启动更新过程，将 write bubble 插入到流水线中。每个 write bubble 分配了一个阶段指示标识 write bubble ID。如果可写位被置位，write bubble 将用新的内容去更新 write bubble ID 级的 BRAM。这一更新机制支持实时内存更新。之前的包处理是基于原来的树，write bubble 之后的包处理则是查找新树的。图 5.4 给出了 write bubble 的设计思想[71]。

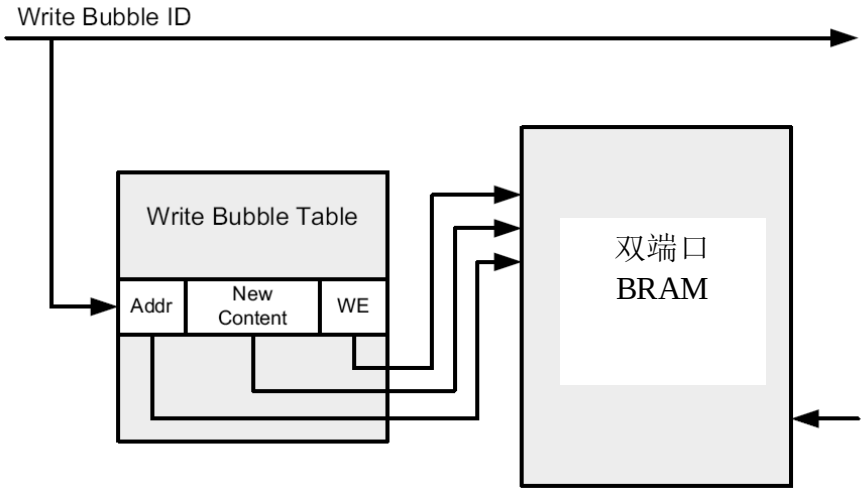


图 5.4　Write Bubble 内存更新方法[71]

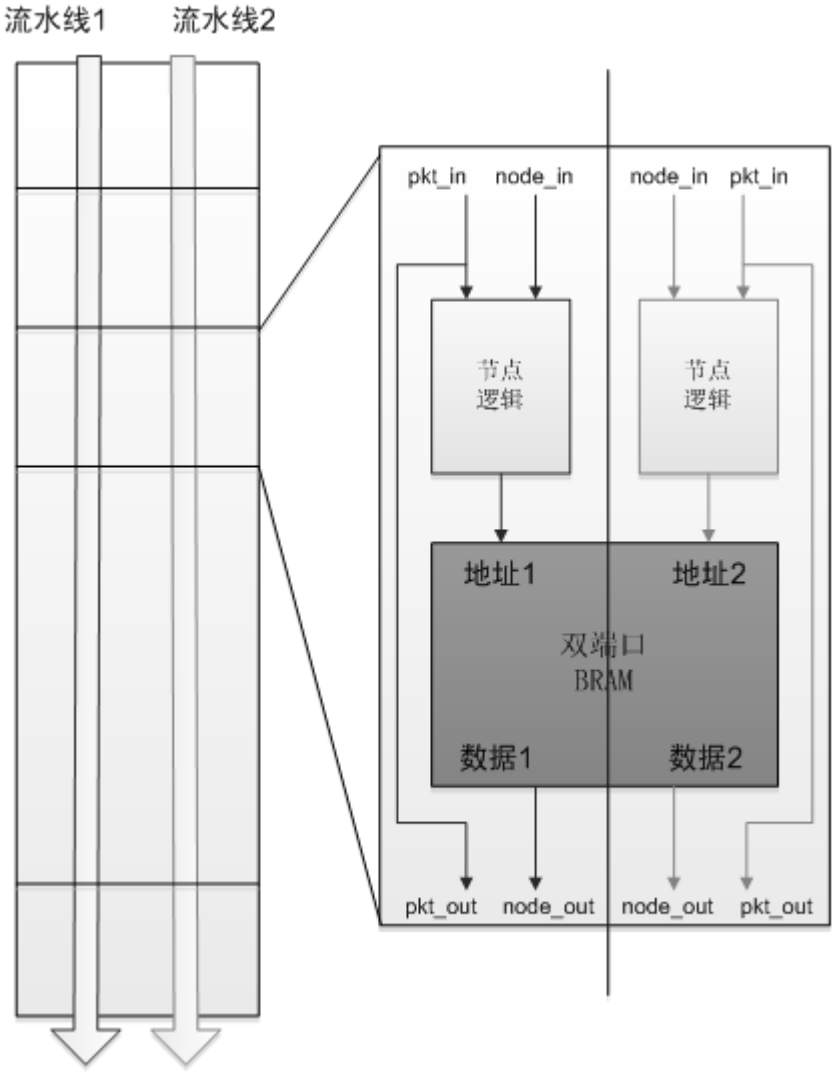


图 5.5　双通道并行查找[35]

5.2.2　并行处理

FPGA 平台为算法提供了良好的并行支持，因此一个优秀的硬件算法应当充分利用 FPGA 的并行性。一种简单有效的并行处理方法是利用 FPGA 平台上的双通道（dual-port）BRAM。具备双通道特性的 BRAM 允许在一个时钟周期内并发完成两个地址的访问，因此属于同一级的两条流水线可以利用不同通道访问同一块 BRAM 进行节点查找（如图 5.5）。双通道特性的利用，使得基于 FPGA 的 HyperSplit 算法在不增加 BRAM 使用的前提下，提高了近一倍的吞吐率。

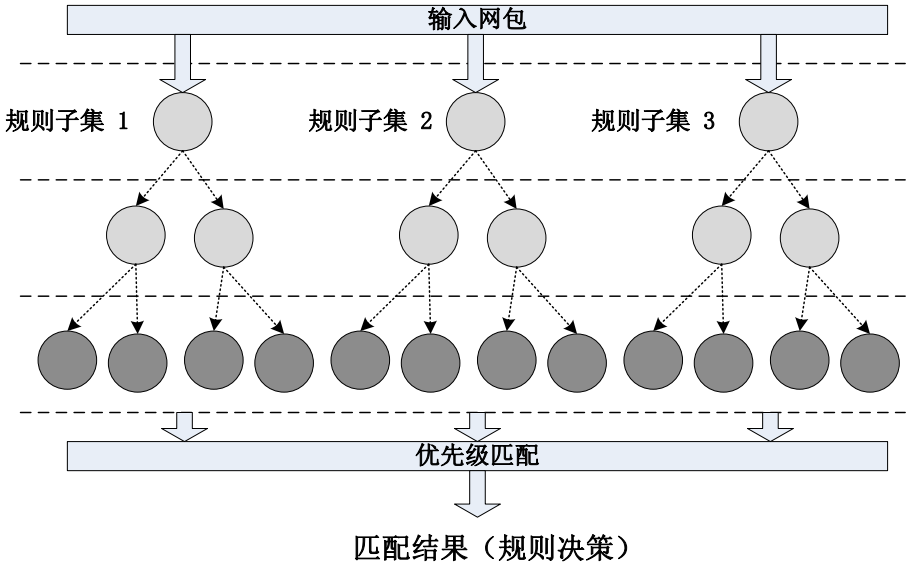


图 5.6　多引擎并行查找[36]

硬件并行处理的另一种思路是将原始的单一数据结构拆分为多个数据结构进行并行查找。如图 5.6 所示，利用 Jiang 等和 Fong 等提出的规则分组算法[36, 39]，可以将 HyperSplit 的单一二分查找树拆分为多个二分查找树并进行并行查找。由规则分组的实验结果可知，数据结构拆分后的总内存使用将大大减少，因此在存储资源有限的 FPGA 平台上可以支持更大规模的规则集合。与此同时，由于每棵子树对应一个规则子集，其查找方法可以根据子集特性不同而进行调优，从而可以使用异质（heterogeneous）的数据结构更加优化地控制各子树的深度及 BRAM 存储。这方面的研究对未来的网络处理芯片设计具有重要的参考价值。

5.2.3　内存管理

由于二分查找树的深度是$\Theta(\log_2 n)$的，基于大规模规则集合的流水线可能会很长。比如，10K 大小的访问控制列表规则，其流水线可达 28 级。深度流水线将极大地增加包分类结果的延时，导致包缓存和系统同步方面的很多问题。

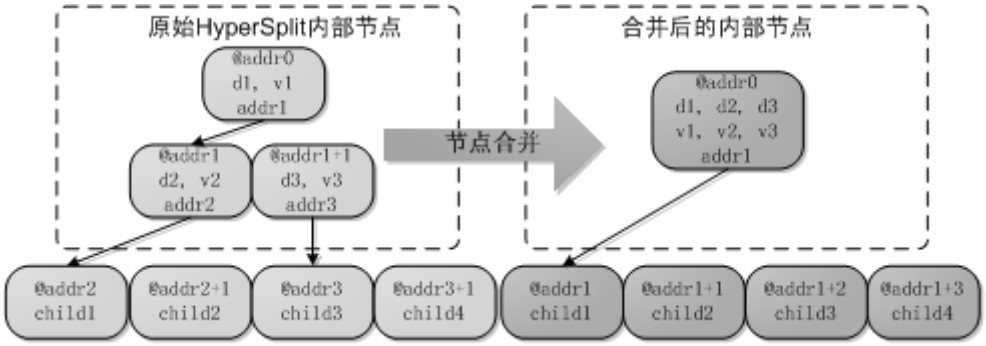


图 5.7　节点合并算法原理[35]

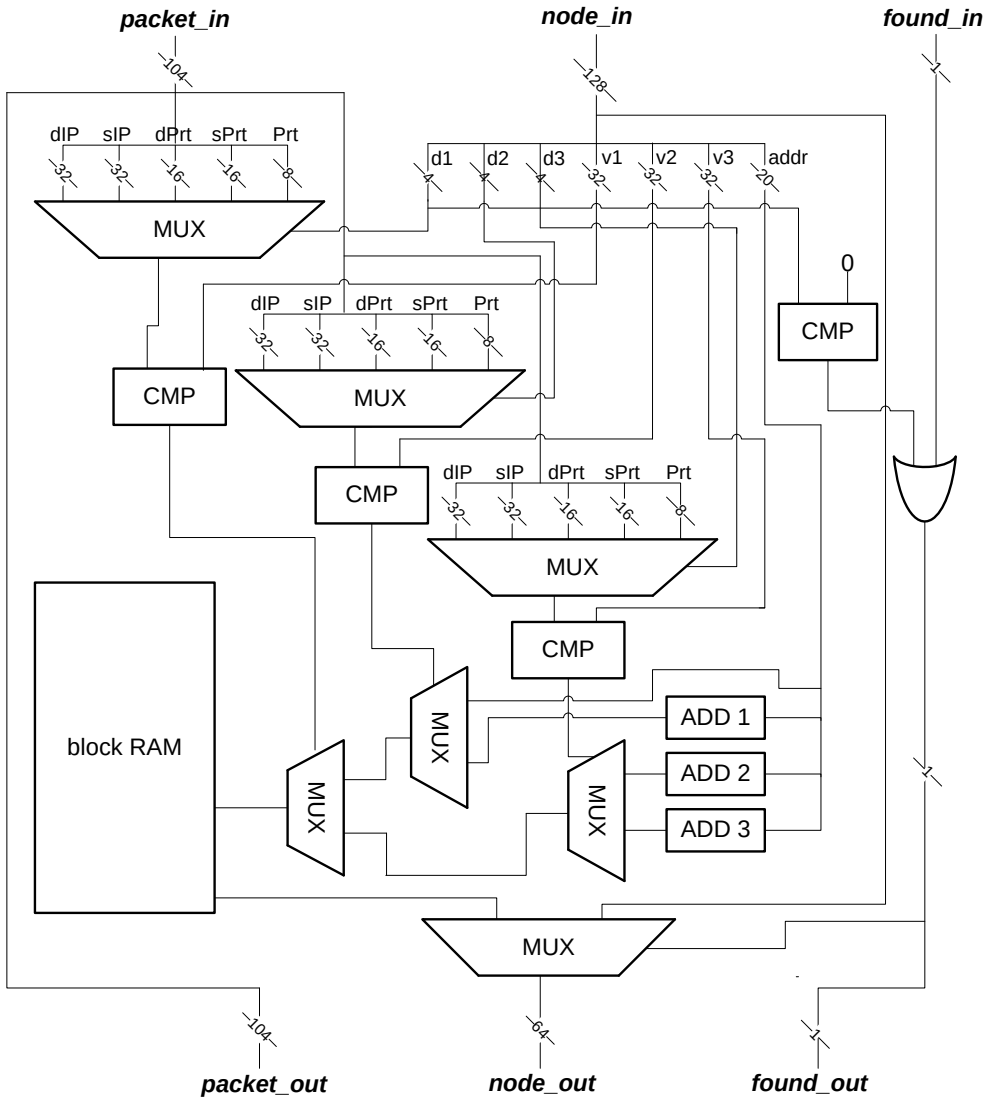


图 5.8　节点合并后的硬件逻辑[35]

为了解决这个问题，HyperSplit 在实现中使用了节点合并算法，以降低树的深度。节点合并机制将非叶节点和它的两个子节点合并为一个节点。从而一个两阶

段搜索将在一个阶段内完成。图 5.7 显示了该算法的基本原理。从该图中可以看出，原始的 HyperSplit 树中的三个内部节点被合并为一个，合并后的节点存储了所有的域指示和二分值。合并后的节点处理逻辑见图 5.8。

```
// Algorithm 1: Node merging
// before merging
if  pkt[d1] <= v1  then
  node_next = read_ram(addr1+0)
  if pkt[d2] <= v2 then
    node_next = read_ram(addr2+0)
  else
    node_next = read_ram(addr2+1)
  end if
else
  node_next = read_ram(addr1+1)
  if pkt[d3] <= v3 then
    node_next = read_ram(addr3+0)
  else
    node_next = read_ram(addr3+1)
  end if
end if

// After merging
if pkt[d1] <= v1 then
  node_next = (pkt[d2] <= v2) ? read_ram(addr1+0) :
  read_ram(addr1+1)
else
  node_next = (pkt[d3] <= v3) ? read_ram(addr1+2) :
  read_ram(addr1+3)
end if
```

图 5.9　HyperSplit 节点合并算法

节点合并之后，树的深度从$\log_2 n$降到$\log_4 n$，因而流水线阶段的数目将是原设计方案的一半。此外，由于在合并的节点中并未存储额外的信息，合并后的节点并不大于原来的三个节点的总和。因而，节点合并机制并不增加全局的额外内存存储。节点合并后的查找算法见图 5.9。

内存管理中的另一个问题在于各级节点分布不均匀。由于节点数在树的每一层都不同，每一阶段的内存使用情况也都不相等。为了支持实时规则更新，每一流水线阶段中，BRAM 的大小都在设计实现时确定。

由于每一流水线阶段都有它自己独有的 BRAM，如果节点数目改变，则需要为每一阶段重新分配 BRAM。然而，BRAM 的重分配将导致 FPGA 设备的重置（reconfiguration），所以无法实现实时的规则更新。

为解决上述问题，HyperSplit 实现了叶节点下移算法。首先，由于树的 0～l-1 层的节点数目较少（小于 4^l），在 BRAM 中为这 l 层建立 1024 甚至更多的表项是不必要的，因此使用分布式 RAM 存储 0～l-1 级节点。对于 l 级之后的节点，由于叶节点的推迟查找不改变原始树的搜索语义，而且叶节点的数目和内部节点的数目在同一数量级，因此算法将超出本级 BRAM 空间大小的叶节点移到下一级进行查找。

```
// Algorithm 2: Leaf pushing
Initialize bucket[i], i=0, …, tree_height-1
node = root
depth = 0
Reshape (node, depth) begin
if  node is leaf then
  while bucket[depth]==0 do depth ++
  end while
  bucket[depth]--
  node->depth = depth
else
  Reshape (node->child[0], depth+1)
  Reshape (node->child[1], depth+1)
  Reshape (node->child[2], depth+1)
  Reshape (node->child[3], depth+1)
end if
end
```

图 5.10　叶节点下移算法

图5.10是节点下移算法的伪代码。第l阶段的内存使用情况可以限制在$bucket[l]$内。$bucket[l]$的最大值由 FPGA 芯片内可用 BRAM 总量决定。一旦所有的 $bucket[l]$ 设置了相同的值，就可以达到一个平衡的内存分配，也就是预分配给所有流水线第 l 阶段之后的 BRAM 的数目具有相同的值。

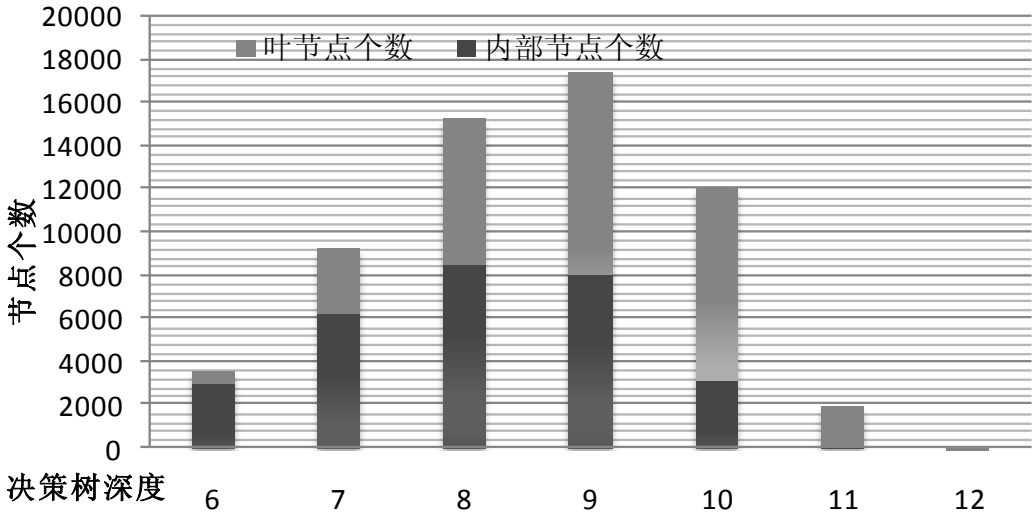


图 5.11　节点下移前的内存分布[35]

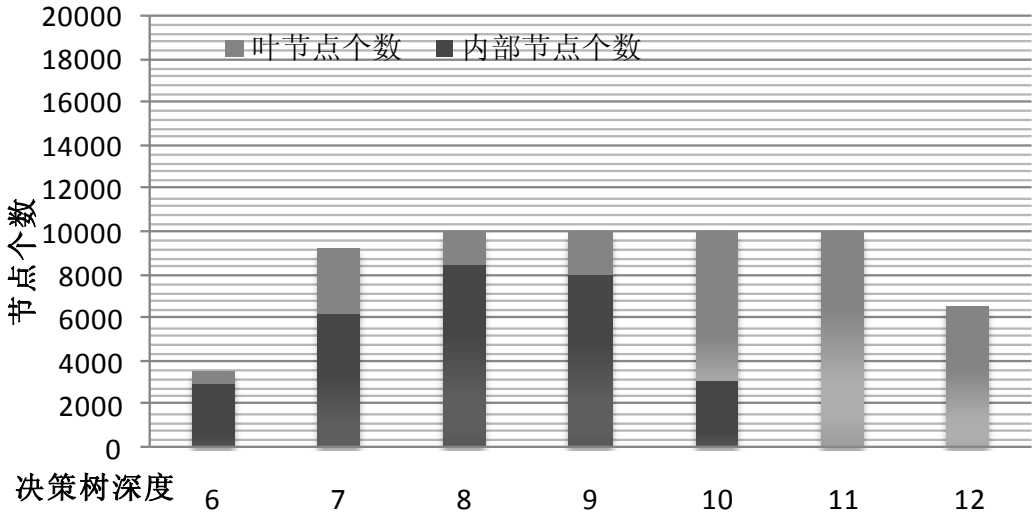


图 5.12　节点下移后的内存分布[35]

图 5.11 和图 5.12 分别显示了节点下移前和节点下移后各级节点数量的分布情况。从这两幅图可以看出，经过节点下移，每级流水线上的节点个数有了严格控制。因此，采用节点下移算法可以有效解决 FPGA 内存分布不均的问题。

5.3　硬件仿真实验

5.3.1　测试数据与平台

本章的硬件测试使用公开的网包分类规则集合[18]。该数据集合为网包分类研究广泛使用，主要包含 ACL，FW 和 IPC 三类规则。其中，ACL 规则为路由器访

问控制列表，FW 规则为防火墙策略，IPC 规则为 Linux IP Chains 规则。测试中每个规则集合的命名为“规则类型-规则数量”。例如，FW1-10K 为 10,000 条防火墙安全策略规则。所有规则都由 IPv4 五元组构成：包括 32 比特源 IP 地址、32 比特目标 IP 地址、16 比特源端口号、16 比特目标端口号、8 比特传输层协议。

FPGA 算法测试中使用的是 Xilinx Virtex-6（model: XC6VSX475T) 芯片[50]。该芯片的计算资源为 37,440 可编程逻辑单元（configurable logic blocks）。包含 7,640K 比特分布式存储单元（Distributed RAM）以及 38,304K 比特块存储单元（Block RAMs）。所有实验结果均使用 Xillinx 的 ISE 仿真平台获取[51]。

5.3.2　实验结果

表 5.1 和表 5.2 给出了基于 FPGA 的 HyperSplit 算法性能。最大时钟速率从 *post place and route report* 的报告中获取。随着规则的变大，最大时钟频率会随着 BRAM 的使用增多而减少。但即使是 10K 规则，HyperSplit 的时钟频率依然可以达到 115.4MHz，这相当于对最小以太网包（64 字节）可达 118Gbps 的吞吐率。

表 5.1　最大时钟频率和吞吐率[35]

规则	最大时钟频率 (MHz)	最高吞吐率 (Gbps)	流水线深度
acl1_100	139.1	142	7
acl1_1K	134.0	137	11
acl1_10K	115.4	118	12

表 5.2　不同规则集合[35]

规则集合	计算资源使用率（已用/ 总共）	存储资源使用率（已用/ 总共）
acl1_100	444/37440	10/516
acl1_1K	602/37440	18/516
acl1_10K	747/37440	103/516

表 5.3　不同 FPGA 算法的性能比较[35]

算法	最多支持规则	最高吞吐率 (Gbps)	使用 acl1_10K 测试		
			流水线深度	计算资源 #Slice	存储资源 #BRAM
HyperSplit on FPGA[35]	50K	142	12	747	103
HyperCuts on FPGA[37]	10K	128	20	10307	407
HyperCuts Simplified[69]	10K	7.22	--	--	--

表 5.4　HyperSplit 算法实现于不同平台的性能[35]

网包分类系统平台（HyperSplit 算法）	最高吞吐率 (Gbps)
Xillinx Virtex-6[35]	142
Cavium OCTEON3860 [33]	6.4
Intel IXP2850 [34]	10

表 5.3 比较了基于 FPGA 的 HyperSplit 和其他基于 FPGA 的网包分类算法的性能。从表中可以看出，首先，基于 FPGA 的 HyperSplit 算法使用了远少于其他算法的 BRAM。例如，对 acl1_10K 规则集合，基于 FPGA 的 HyperSplit 算法使用 103 个 BRAM 资源，而基于 FPGA 的 HyperCuts 算法需要 407 个 BRAM。其次，基于 FPGA 的 HyperSplit 算法在一个 Virtex-6 芯片上支持多于 50,000 条规则。其计算资源使用量（slice 数目）也远小于已有算法。例如对 acl1_10K 规则，基于 FPGA 的 HyperSplit 算法仅适用 747 个 slice，而基于 FPGA 的 HyperCuts 算法需要多大 10307 个 slice。

计算资源和存储资源能够节省的主要原因是，基于 FPGA 的 HyperSplit 算法使用单一的流水线，且在每一流水线阶段不存储原始规则；而基于 FPGA 的 HyperCuts 算法则使用二维流水线，且每级均需要线性查找。

最后，表 5.4 比较了基于 FPGA 硬件平台和基于多核网络处理器平台的 HyperSplit 算法性能。如表所示，基于 FPGA 的硬件网包分类系统比基于多核网络处理器的系统吞吐率高 10 倍以上。

5.4　本章小结

随着网络带宽的高速增长，多域网包分类算法面临着 100Gbps 的性能挑战。本章将 HyperSplit 算法实现于 FPGA 硬件平台，得到了 100Gbps 的网包分类引擎。研究中提出了两个优化算法。其中，内部节点合并算法用来降低流水线阶段的数目，叶节点下移算法则用来控制每一阶段的内存使用上限。

实现结果显示，基于 FPGA 硬件平台的 HyperSpit 算法可以获得超过 100Gbps 的吞吐率（基于 64 字节以太网包）。与现有基于 FPGA 的网包分类算法相比，HyperSplit 算法使用资源更少，并且支持规则更多。与基于多核网络处理器的网包分类系统相比，基于 FPGA 硬件的 HyperSplit 算法具有 10 倍以上的吞吐率。

第 6 章　基于载荷的网包分类算法

传统的网包分类系统依据多域网包包头进行分类，而现代网络设备越来越注重基于网包载荷（packet payload）的网包分类，以改善数据传输并提高网络安全。基于载荷的网包分类是实现深度检测（deep inspection）的核心技术，广泛应用于入侵防御、协议识别、内容过滤等网络设备中。

图 6.1 是一个典型企业网络的网关架构[1]。网络流量来自于连接 Internet 路由器的前端外部网络接口。安全网关中的多核网络处理器主要负责进行 L2~L4 层处理，如路由查找、五元组网包分类、传输层状态检测等。而基于网包载荷的分类通常由硬件协处理器完成，如 LSI Tarari T2000/2500[72]、NetL7 NLS2008[73]和 Cavium DFA 引擎[74]等。这些深度检测协处理器具有高带宽 I/O（如 PCI-Express，XAUI）以保证主处理器和专有内存（如片上 SRAM 或者低延迟的 RLDRAM）的数据交换，进而完成快速并行处理。前端处理设备将根据网包包头分类和网包载荷分类的结果最终决定将网包的后端处理。

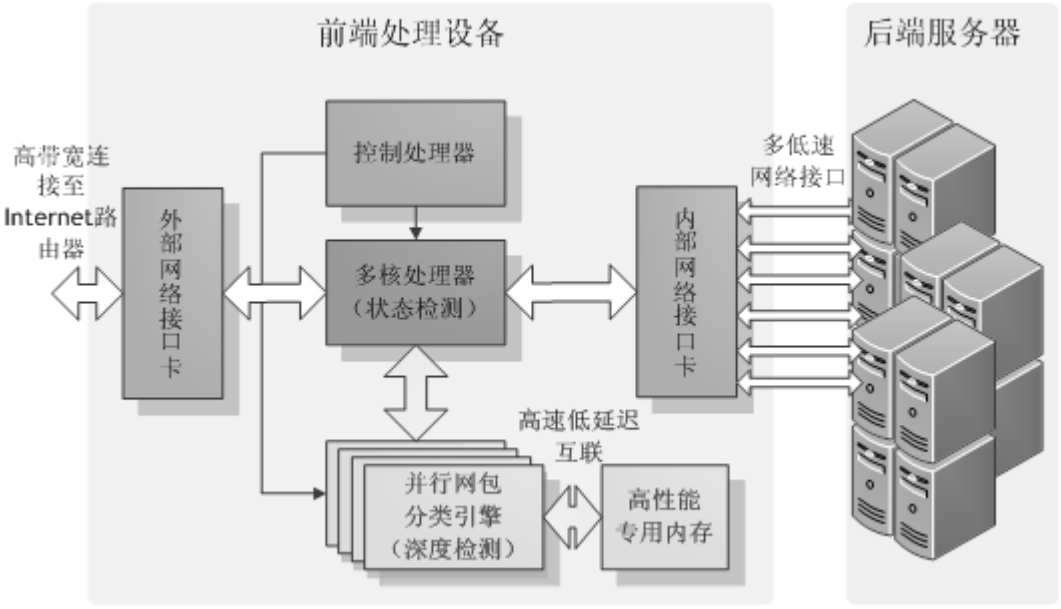


图 6.1　前端深度检测系统架构

本章将首先介绍基于网包载荷的网包分类问题，然后介绍相关算法，并提出用于前端设备进行网包载荷分类的 FEACAN 算法[46]。最后结合 FPGA 硬件平台设计出基于 FEACAN 算法的网包分类引擎。

6.1　基于网包载荷的网包分类问题

基于网包载荷的网包分类是将网包载荷中的字节流与分类规则一一匹配，并依据匹配结果对网包进行分类。由于网包载荷的长度不定，因此依据网包载荷的网包分类是多域网包分类的拓展，即从多域向无穷域的拓展。基于载荷的分类规则通常使用正则表达式（regular expression）。下面具体介绍基于正则表达式匹配的网包分类问题。

6.1.1　正则表达式匹配

正则表达式匹配已被证明在网络内容识别处理中具有高效性和灵活性[75]。理论上可以证明[76-78]，对于正则表达式匹配问题，利用不确定和确定的有限自动机都可以解决（NFAs 和 DFAs）。

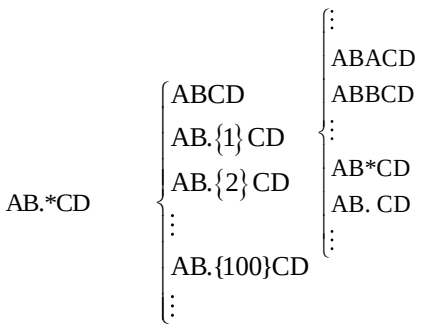


图 6.2　正则表达式与字符串的关系

上图就是一个简单的正则表达式的例子，最左边和中间的都是指正则表达式，最右边的都是子字符串（图 6.2 只是将部分正则表达式展开成了字符串）。从中可以发现，正则表达式具有简单、灵活、能够表达无穷域的特性。

在网络应用中，通常使用基于 PCRE（Perl-Compatible Regular Expressions）的正则表达式库[79]。本文对正则表达式的描述也基于 PCRE。表 6.1 给出了 PCRE 各版本中相兼容的、最常见的一些正则表达式符号。

表 6.1　正则表达式中常见的符号

分类	字符	示例	描述
限定符		$r_1 r_2$	匹配相连的字符串 r_1 和 r_2 （连接）
	|	$r_1 \| r_2$	匹配字符串 r_1 或 r_2 （并）
	*	$r*$	匹配出现零次或更多次的字符串 r （闭包）
	+	$r+$	匹配出现一次或更多次的字符串 r
	?	$r?$	匹配出现零次或一次的字符串 r
	{}	r{m,n}	匹配出现 m 到 n 次的字符串 r
	[]	[abc] [a-c]	匹配 a、b、c 中的任意字符
	[^]	[^abc] [^a-c]	匹配除 a、b、c 之外的任意字符
转义符	\	*	匹配字符 *
元字符	.	.	匹配除换行符以外的任意字符
	\w	\w	匹配字母、数字或下划线
	\s	\s	匹配任意空白符
	\d	\d	匹配数字
	\b	\b	匹配单词的开始或结束
	^	^r	匹配字符串 r 的开始
	$	r$	匹配字符串 r 的结束

其实从本质上来说，上表中的连接、并、闭包运算是传统正则表达式的三个基本运算，其他的符号运算都是在此基础上扩展而来的。比如：

$$x+ == xx*$$

$$x\ \{1,3\} == x \mid xx \mid xxx$$

$$[x\text{-}z] == x \mid y \mid z$$

下面以判断具有 HTTP 协议的正则表达式为例说明特殊字符的使用[93]：

http/(0\.9|1\.0|1\.1)[1-5][0-9][0-9][\x09-\x0d-~]*(connection:|content-type:|content-length:|date:)|post[\x09-\x0d -~]*http/[01]\.[019]

其中：

- http/：匹配 http/
- (0\.9|1\.0|1\.1)：匹配 0.9 或 1.0 或 1.1
- [1-5][0-9][0-9]：匹配一个三位数，范围从 100 到 599
- [\x09-\x0d -~]*：匹配任意个从\x09 到\x0d 以及空格到’~’之间的所有字符，即任意个可打印字符

- (connection:|content-type:|content-length:|date:)：匹配 connection: 或 content-type:或 content-length:或 date:
- http/[01]\.[019]：匹配 http/后跟版本号，版本号范围为 0.0，0.1，0.9，1.0，1.1，1.9

6.1.2　基于DFA的匹配算法

Hopcroft 等[76]证明了正则表达式所表示的正则语言与有限自动机所识别的语言是完全等价的，只是表示形式不同而已。同一个语言，既可以用有穷自动机描述，也可以用正则表达式描述。确定性有穷自动机、带 ε 跳转或者不带 ε 跳转的非确定性有穷自动机、以及正则表达式之间的等价转换关系如图 6.3 所示：

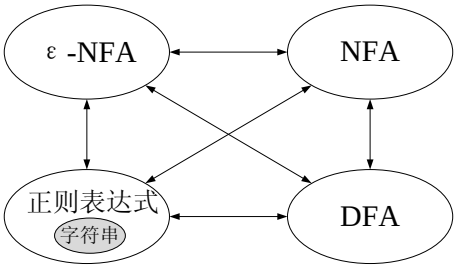


图 6.3　正则表达式与有穷自动机的语言集合等价

图 6.3 中的非确定性有穷自动机 NFA 的数学模型形式化定义为

$$A=（Q，\Sigma，\delta，q_0，F），$$

其中：

- Q 是一个有穷的状态集合；
- Σ 是一个有穷的输入符号集合（字符表）；
- δ 是一个跳转函数，以 Q 中的一个状态和 Σ 中的一个输入符号作为变量，返回 Q 的一个状态子集；
- q_0 是一个初始状态，是 Q 中的一个状态；
- F 是一个接受状态（终结状态）的集合，是 Q 的一个状态子集。

图 6.3 中的确定性有穷自动机 DFA 的数学模型形式化定义为

$$A=（Q，\Sigma，\delta，q_0，F），$$

其中：

- Q 是一个有穷的状态集合；

- Σ 是一个有穷的输入符号集合（字符表）；
- δ 是一个跳转函数，以 Q 中的一个状态和 Σ 中的一个输入符号作为变量，返回 Q 中的一个状态；
- q_0 是一个初始状态，是 Q 中的一个状态；
- F 是一个接受状态（终结状态）的集合，是 Q 的一个状态子集。

NFA 与 DFA 的根本区别是它们的跳转函数不同。DFA 对每一个可能的输入只有一个状态的跳转。NFA 则对每一个可能的输入可以有多个状态跳转，接受到输入时从这多个状态跳转中非确定地选择一个。前者意味着在任何时候自动机不能处在一种以上状态中，后者意味着自动机能同时处在几种状态中。

根据前面所述的一个正则表达式可以转换成与之等价的有限自动机的结论，因此可以通过构造成有穷自动机来解决正则表达式的匹配问题将正则表达式构造成等价的 NFA 的算法当中，最经典的是由 Thompson 于 1968 年首先提出的 Thompson 构造法[78]，其基本思想是利用 ε 转换将正则表达式的机器片段（对应于子表达式）组合在一起，最终形成与整个表达式相对应的机器，因此该结构是归纳的。

将正则表达式转换成 DFA 的基本方法是先把正则表达式转换成 NFA，然后使用子集构造法（subset construction）将 NFA 转换成 DFA，最后再将 DFA 最小化[77]。子集构造法本质上来说类似于模拟 NFA 的匹配过程，其核心是要为 DFA 构造一个状态跳转表。DFA 的每个状态是一个 NFA 状态集合，使得 DFA“并行地”模拟 NFA 在遇到一个给定输入字符串时可能执行的所有动作。

对于任意的正则表达式，都可以找到一个与之等价的具有最少状态数的 DFA。换句话说，在状态数无冗余的情况下，用于存储一个 DFA 所需的内存大小决定于状态内部的跳转个数，即 DFA 状态跳转表中带标记的边的数目。

6.1.3　改进的DFA算法

由于网络应用中的正则表达式匹配一般都使用扩展的 ASCII 字符集作为字母表 Σ，这样 DFA 的状态跳转表的每个状态都会有 256 条出边。由如此庞大规模的边中存在大量的重复，比如很多跳回到起始状态的边等。去除这些冗余边，是减少 DFA 本身所占空间的主要手段，而如何有效地去除，则是 DFA 优化算法研究的主要任务。

在此研究领域，Tuck 等人使用比特串（BITMAP）技术压缩具有相同下级状态的跳转[80]。因为他们的工作中只考虑了一个状态中的冗余，当特殊的转变增加

的时候，系统的表现将会不稳定。根据实验结果，Tuck 等人的算法用于少量正则表达式时有 90%的压缩率，而在大量的规则上只有少于 30%的压缩率。另一个问题是比特串的存储是和相应的状态一致的，如：*N* 状态 DFA 将会有 *N* 个比特串。因此大数量的比特串存储和计算成为高效硬件执行的又一个问题。

Kumar 等注意到 DFA 中的许多状态有着相似的相邻状态跳转，于是根据模式匹配中 AC（Aho-Corasick）算法[81]，提出了默认跳转（default transition）的概念，并基于此提出了一种优化的 DFA 结构——D^2FA（Delayed Input DFA）[47, 82]。D^2FA 通过默认路径和默认跳转实现 DFA 表的压缩。每一条默认路径上的状态仅存储不同的跳转（unique transitions），而默认跳转可以由其同路径上的父状态获得。在不控制默认路径深度的前提下，D^2FA 算法对由常见正则表达式规则集合所生成的 DFA 有 95%的压缩率。

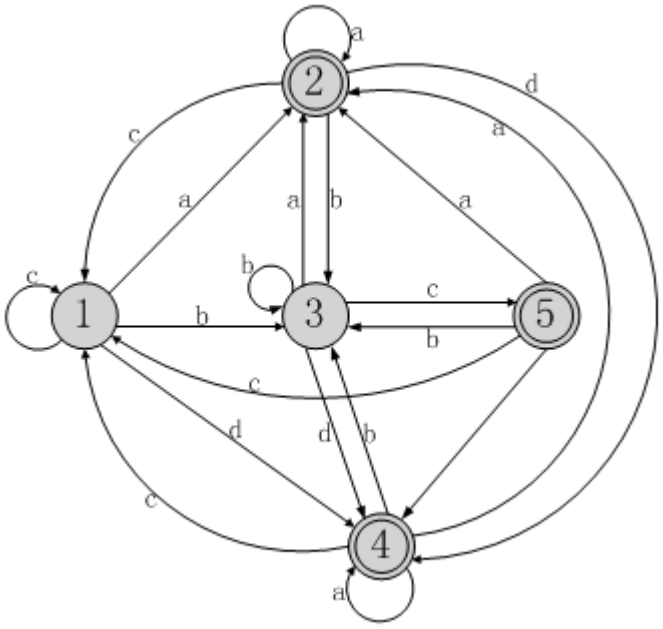


图 6.4　对应正则表达式 a+，b+c，c*d+的 DFA[47]

图 6.4[47]是一个字母表 Σ 为{*a*，*b*，*c*，*d*}的 DFA，它被用来识别 3 个正则表达式：*a*+，*b*+*c*，*c***d*+，状态 2、状态 5、状态 4 分别对应 3 个正则表达式匹配成功的状态。图 6.5[47]是与图 6.4 中的 DFA 等价的 D^2FA。当输入字符串为 *aabdbc* 时，DFA 的状态访问序列为 1223435，D^2FA 的访问序列为 1212314135。匹配状态（序列中下划线标识的部分）与 DFA 完全等价。图 6.4 中 DFA 有 20 条边，而 D^2FA 只有 9 条边，可见 D^2FA 对 DFA 进行了有效的压缩。但上面的访问序列也表明处理同样的字符串 D^2FA 比 DFA 要访问更多的状态，即 D^2FA 以时间性能为代价提高了空间性能。

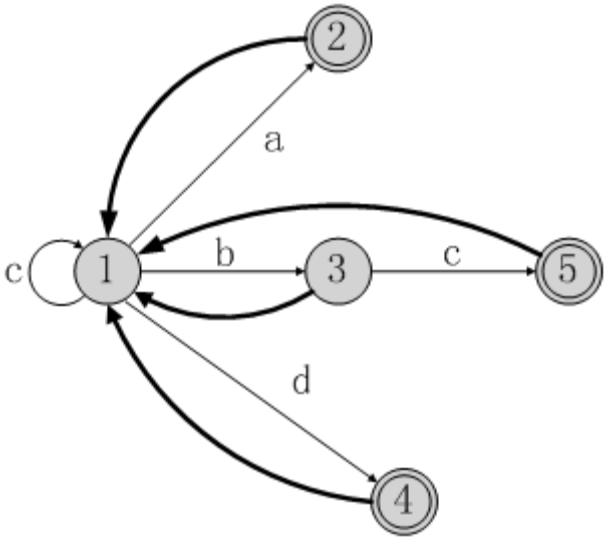


图 6.5　对应正则表达式 a+，b+c，c*d+的 D^{2}FA[47]

D^{2}FA 以默认跳转取代了大量重复的标号跳转，在不考虑默认跳转路径深度的情况下，取得了非常高的空间压缩比。然而，D^{2}FA 实际上是以增加处理每个字符时经过的状态数或跳转数为代价的，这在一定程度上降低了 D^{2}FA 的时间性能。Kumar 等提出了基于哈希访问的对策限制 D^{2}FA 的最大默认路径长度，但该方法对于包含标号跳转较多的状态并不适用。Becchi 等指出了 D^{2}FA 存在的三个问题[83]：

- D^{2}FA 性能与最长默认路径直接相关，对其长度限制的参数是需要经过大量实验之后才能确定的，因为对于不同的规则集，这个参数最优的数值往往不同；
- D^{2}FA 的构造算法构造出的数据结构，最坏情况下最长默认路径每一次都会被遍历；
- D^{2}FA 的构造算法需要多次扫描全局状态，其构造时间复杂度为$\Theta(N^2 \log N)$，因此实现效率低下。

针对上面三个问题，Becchi 提出的改进的 D^{2}FA 具有如下特点：

- 对于长度为 N 的输入字符串最多只需要遍历 $2N$ 个状态，与默认路径的最大长度无关；
- 它的空间压缩比与 D^{2}FA 非常接近，但是最坏情况下所需内存带宽要小的多。

Becchi 的这种方法基于对大量 DFA 的实际观察，即大量标号跳转的目标状态是起始状态或者是起始状态的临近子状态。通过定义状态深度（state depth），即从

起始状态到该状态所需遍历的最小状态数，可以用比 D^{2}FA 构造算法更加简洁的方法来选择更好的默认跳转。

Becchi 指出，若定义起始状态 S0 的深度为 0，则 S0 的所有下一级子状态的深度为 1。这些深度为 1 的状态的下一级子状态深度为 2。以此类推，可以计算出 DFA 中所有状态的状态深度。Becchi 利用上述定义证明[83]：如果在 D^{2}FA 中所有默认路径只指向状态深度小的状态，则处理任何长度为 N 的输入字符串最多只需要遍历 $2N$ 个状态。因此，Becchi 提出了一种改进的 D^{2}FA 算法，在保证压缩率的同时获取每字符平均访问不超过 2 个状态的时间性能。

6.2　FEACAN算法设计

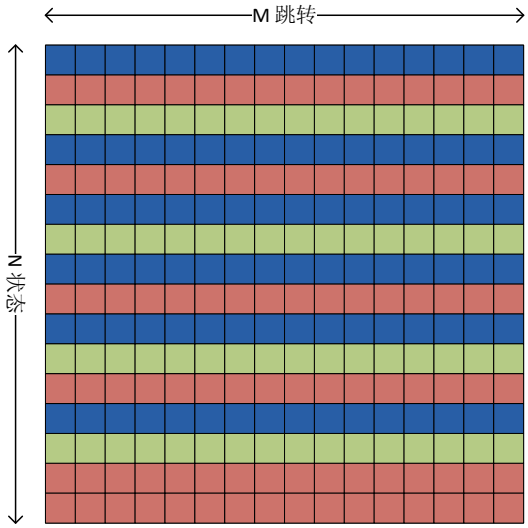
虽然 D^{2}FA 算法实现了很好的 DFA 压缩率，但是 D^{2}FA 难以用于高速硬件实现。这是因为：

- 每一个状态跳转在执行默认转换之前都要比较当前状态中的所有非默认跳转；
- 在一个非默认跳转被发现之前这样的线比较需要递归所有默认路径中的状态。

D^{2}FA 实验发现，每输入字符所遍历的跳转数量是状态数量的 10～100 倍。这意味着为达到同样分类速率，D^{2}FA 算法需要的内存带宽是原始 DFA 算法的 10～100 倍。由此可见，当前已有的正则表达式匹配算法仍不能满足高吞吐、低延迟的处理需求。

6.2.1　设计思想

与 HyperSplit 算法和 AggreCuts 算法的设计思想一样，FEACAN 算法的核心思想是在保证 DFA 查找速率的前提下，尽可能压缩 DFA 跳转矩阵的大小。



$$\begin{bmatrix} t(1,1), t(1,2), & \cdots & t(1,M) \\ t(2,1), t(2,2), & \cdots & t(2,M) \\ \vdots & \vdots & \vdots \\ t(N,1), t(N,2) & \cdots & t(N,M) \end{bmatrix}$$

图 6.6　状态跳转矩阵[46]

$$\begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \iddots \\ \cdots & 1 & 1 & 1 & 1 & 1 & 9 & 35 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\ \cdots & 1 & 1 & 36 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\ \cdots & 1 & 1 & 1 & 1 & 1 & 9 & 121 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\ \cdots & 1 & 1 & 1 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\ \cdots & 1 & 1 & 123 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\ \cdots & 1 & 1 & 1 & 1 & 1 & 9 & 1 & 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots \\ \iddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

图 6.7　状态跳转矩阵示例[46]

如图 6.6 所示，对于一个定义在字符集 Σ（$M = |\Sigma|$为字符集大小）上有 N 个状态的 DFA，其跳转矩阵总共有 $N * M$ 个跳转，不妨使用一张二维状态跳转表表 T_{N*M} 来表示. T_{N*M}中的第 n 行：

$$S_n = \{t(n,1), t(n,2), \dots, t(n,M)\}$$

表示第 n 个状态，而其中的每一个元素$t(n,m)$表示下一跳状态。若给定输入字符$c \in \Sigma$ 及当前状态S_{n0}，则可以读取S_{n0}的第 c（假设 c 从 1 编号至 M）个跳转$t(n0,c)$来获取下一个状态S_{n1}，即:

$$S_{n1} = t(n0,c)$$

大量实验发现，状态跳转矩阵 T_{N*M} 是非常稀疏的。此结论跟已有文献[47, 82-84]

中的结论一致。图 6.7 显示了一个由实际正则表达式集合生成的 DFA 状态跳转矩阵片段，从中可以看出状态跳转矩阵在每个状态内部和状态两两之间均有冗余：

- 状态内冗余：在状态S_n内部，不同跳转（unique transition）的个数很小。通常，不同的$t(n,m)$的个数远小于 M.
- 状态间冗余：对于任一状态大多数状态S_{n1}，总能找到一组状态$S_{n1},S_{n2},\ldots,S_{nk}$，使得这些状态的跳转相似，即对于大多数的$1 \le m \le M$，满足：

$$t(n1,m) = \cdots = t(nk,m)$$

鉴于以上两个发现，FEACAN 算法使用状态内（inter-state）和状态间（intra-state）的二维压缩算法来压缩原始 DFA 的状态跳转矩阵。对于状态内压缩，FEACAN 算法使用类似 AggreCuts 算法的层级比特串方法，对于状态间压缩，FEACAN 使用 2 级状态聚类的方法。FEACAN 算法能够在减少内存空间的同时，保证查找速率。

6.2.2　数据结构

FEACAN 算法通过下述步骤来实现状态跳转矩阵的压缩：

首先定义第 n（$1 \le n \le N$）个状态的 UT （unique transition）为：

- 第一个跳转$t(n,1)$为 UT
- 若第 m 个跳转满足$t(n,m) \ne t(n,m-1)$则$t(n,m)$为 UT

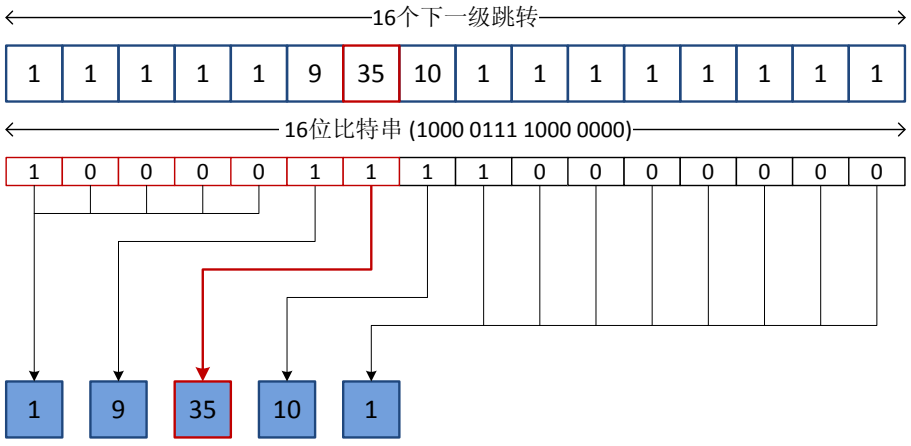


图 6.8　使用比特串压缩一个状态[46]

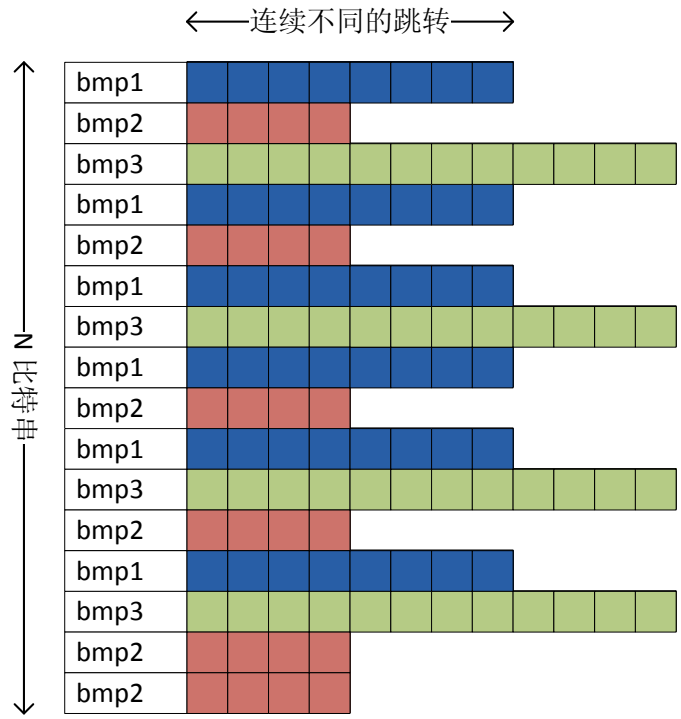


图 6.9　状态内压缩[46]

类似于 AggreCuts 算法，UT 可以使用图 6.8 所示的比特串方法进行有效的压缩。为了确定第 *m* 个跳转对应的 UT，可以通过计算比特串中的 1 的个数。经过比特串压缩，原始状态跳转矩阵可以表示为图 6.9 所示的结构。

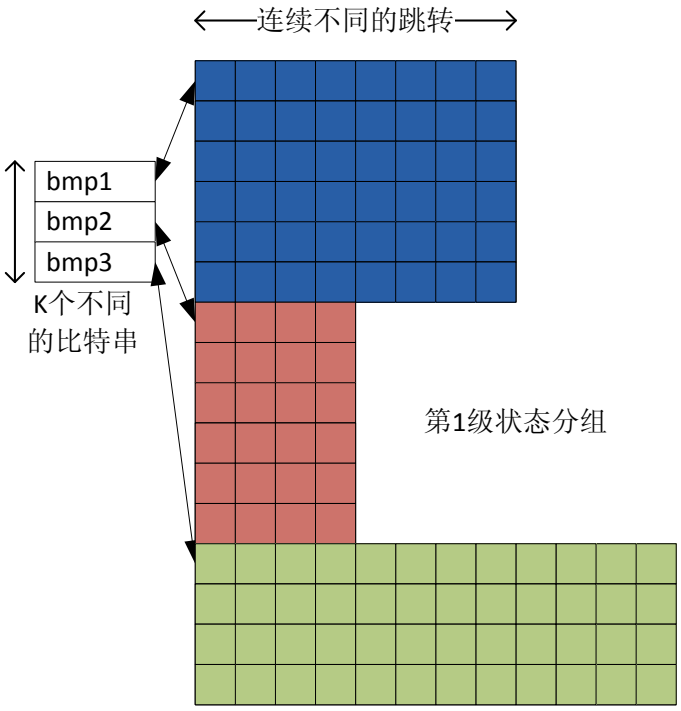


图 6.10　第一级状态聚类[46]

对于压缩后的状态，FEACAN 使用第一级聚类把相同的比特串对应的状态排列在一起，这样可以减少比特串的存储数量（即只需存储不同的比特串）。如图 6.10 所示，在进行了第一级聚类（总共 *K* 个组）之后，只留下了 *K* 个不同的比特串，且他们每一个都对应着一个组中所有的状态。第一级聚类不仅把比特串的数量从 *N* 个减少到 *K* 个，同时由于对应相同比特串的状态更可能有相同的跳转表（见第 6 部分），因此第一级聚类能有助于下面要进行的状态间压缩。

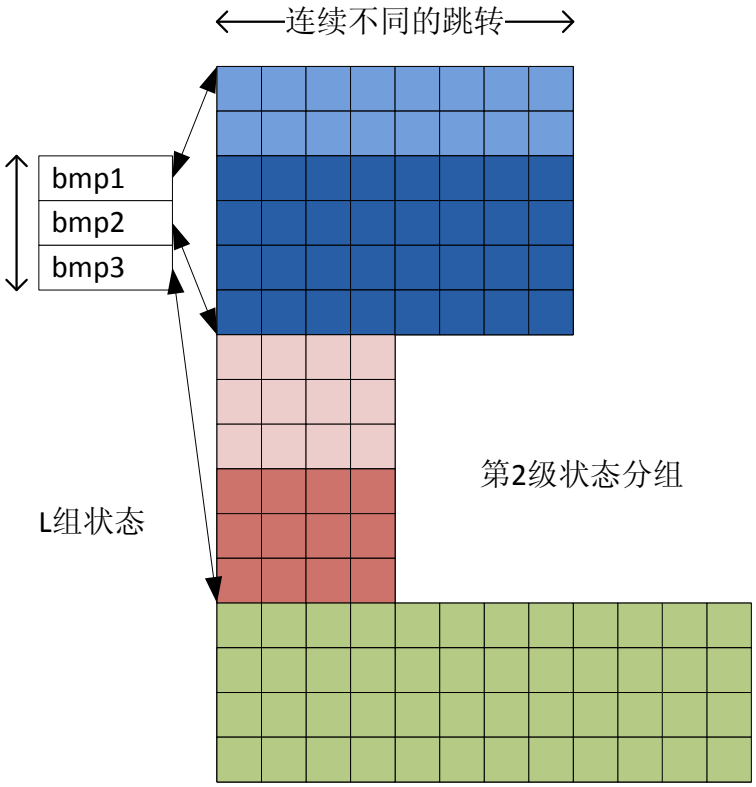


图 6.11　第二级状态聚类[46]

第二级聚类是把 *K* 个组中相同的组再细化为 *L* 个子组。判断两个状态是否相似需根据如下条件：

- 共享相同的比特串（这样他们才会有相同的跳转个数）；
- 多于 *thresh*%的跳转相互是有联系的。

根据实验结果，当 *thresh*%的值为 80%~95%时，均能取得良好且稳定的压缩率。第二级聚类的结果显示与图 6.11，其中 *L* 个组以不同的颜色表示。

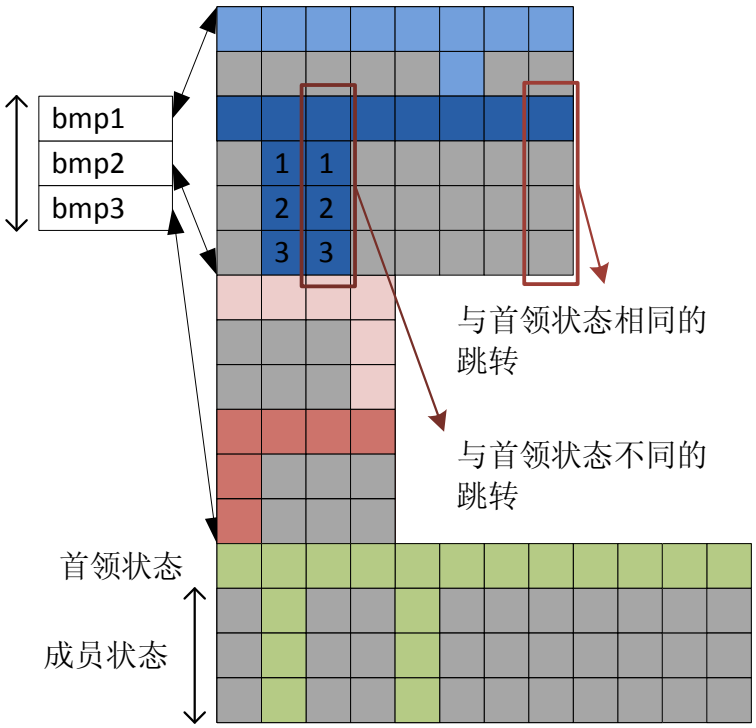


图 6.12　状态间压缩[46]

在两个层的聚集之后，FEACAN 算法选每组中的第一个状态作为首领状态（*leader state*），并把其他状态作为成员状态（*member states*）。算法通过消除成员状态中与首领状态的相同的跳转进行状态间压缩。如图 6.12 所示，每个灰色状态均为成员状态中与首领状态对应项相同的跳转。

在经过了状态内压缩和状态间压缩后，图 6.13 显示了最终的数据结构。压缩后的状态 ID 被相应的<groupID，memberID>对来代替。其中，groupID 是用来寻找下一状态组的索引，而 memberID 代表组内成员状态的索引。对于首领状态而言，所有的 memberID 值都为 0。首领状态中的每一个跳转都与一个 memberOffset 参数联系在一起，从而能够记录查找成员跳转的基地址。所以每个首领状态中的跳转将被一个三元组所替代，它们是<groupID，memberID，memberOffset>。除此之外，还需要维护一个 L 项的地址映射表，表中每一个条目记录如下信息：<bitmapID，leaderBase，memberBase>。其中，bitmapID 用来查找该组对应的比特串，leaderBase 记录成员状态的跳转基址，memberBase 记录首领状态的地址。

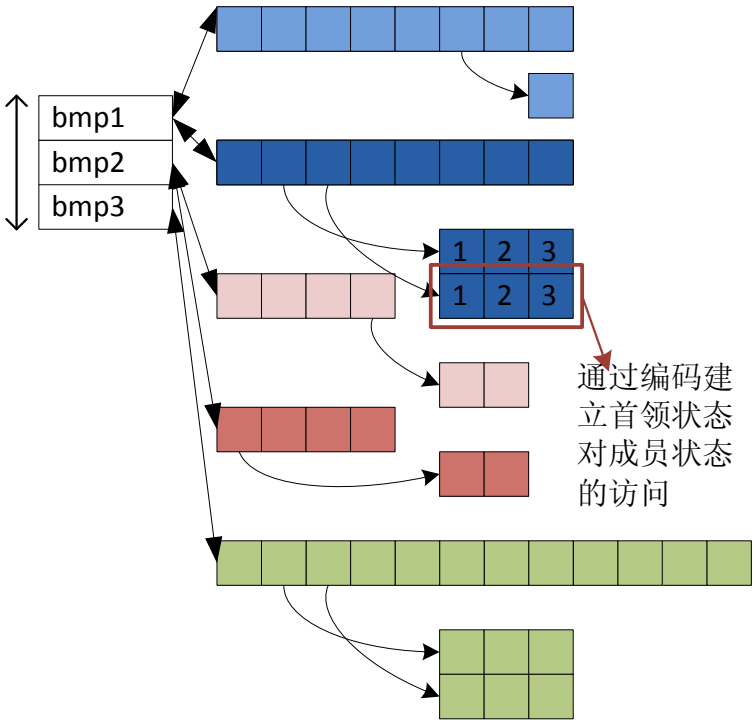


图 6.13　二级压缩后的数据结构[46]

查找图 6.13 中显示的数据结构需要进行如下步骤：

- 使用 groupID 参数来读取地址映射表中的<bitmapID，leaderBase，memberBase>参数。
- 使用 bitmapID 参数读取组中相关的比特串并计算比特串前 c（c 是输入字符的值）比特中 1 的个数 u。
- 使用（leaderBase+u）参数作为读取 leaderTransition 的索引。
- 如果 memberID 或者 memberOffset 值为 0，下一跳状态就是存储在跳转中的<groupID，memberID>。
- 如果 memberID 或者 memberOffset 值为均不为 0，则使用 memberBase+memberOffset +memberID 作为读取 memberTransition 的索引，并且下一跳状态是存储在跳转中的<groupID，memberID>。

根据上述设计，FEACAN 算法在系统部署上有如下优点：

- **高效内存压缩：**相比于 Tuck 等提出的原始比特串一维压缩算法，FEACAN 算法使用二维压缩技术消除 DFA 状态内和状态间。实验结果表明二维压缩技术相比于原始比特串压缩算法减少了最高 80%的内存使用。
- **确定的内存存取：**与 D^2FA 算法不同， FEACAN 算法的状态跳转不需要遍历当前状态内部的所有跳转。对于每一个输入字符的最坏的跳转次数是

2，其中一个属于首领状态，另一个属于成员状态。

- **快速预处理：**原始的 D^{2}FA 算法在处理时的时间复杂度为$O(N^2logN)$，而 FEACAN 算法处理的时间复杂度仅为$O(NlogN)$。由于 N 代表了状态的数量（通常很大），实际预处理速度比 D^{2}FA 有明显提高。

值得注意的是，对于一些复杂的正则表达式规则，原始 DFA 的状态数会呈爆炸式增长。这种情况下将无法直接使用 D^{2}FA 或 FEACAN 等算法对 DFA 进行压缩（DFA 跳转矩阵本身就会无法生成）。这种情况下，FEACAN 可以结合 H-cFA[85]、HybridFA[86, 87]和 XFA[88, 89]等算法解决 DFA 状态爆炸问题。

6.2.3　算法优化

要将 FEACAN 算法实现于硬件系统，需要进行算法优化。FEACAN 的算法优化包括两个方面：一方面是控制比特串数量以及每个比特串的大小，从而使得比特串的读取和计算能够快速进行；另一方面是通过对状态访问次数的统计来减少每一输入字符的内存访问次数，从而进一步降低访问延迟。

表 6.2　实际规则集合中的状态数和比特串数[46]

规则类型	DFA 状态数 (N)	比特串数(K)
正则表达式规则	6533	73
精确字符串规则	56280	112

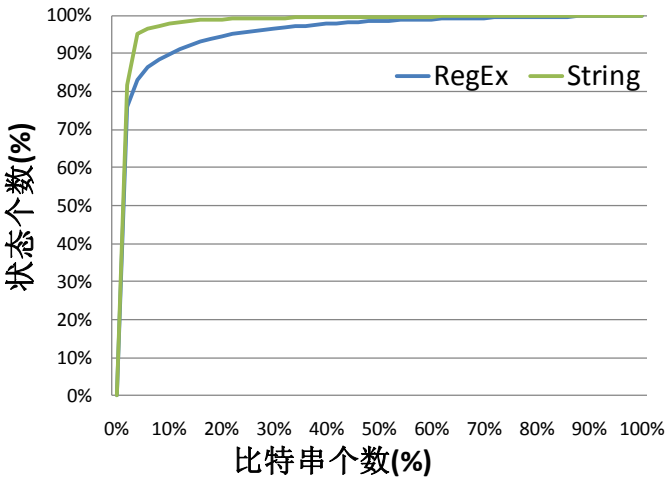


图 6.14　对应不同比特串的状态数分布[46]

在最坏的情况下，比特串的数量 K 等于状态的数量 N。但实际测试发现，绝大多数情况下 K 远小于 N。表 6.2 给出了基于两个典型的正则表达式规则集合的统计结果。从这个表可以看出，比特串的数量是非常小的。即使对于拥有 56280 个状态（由所有的 Snort[93]字符串生成）的大型 DFA，比特串的数量只有 112。图 6.14

表明，到超过 90%的状态只对应不到 10%的比特串。这意味着绝大多数的状态跳转只需要操作最常访问的 10-20 个比特串。

通过对两个比特串进行“OR”操作，可以合并结构相似的比特串。假设对于两个比特串 bmp_1 和 bmp_2 只有 m 位不同，则可使用 $bmp_0 = bmp_1$ OR bmp_2 来合并这两个比特串。虽然比特串结合增加 m 个跳转数量，但这一增长经过状态间压缩后基本得到了消除。表 6.3 给出的实验结果表明，比特串合并后并不会使跳转总数有显著增加。

表 6.3　比特串合并前后比较[46]

规则类型	合并前比特串数量	合并后比特串数量	跳转总数变化
正则表达式规则	73	22	6%
精确字符串规则	112	18	3%

另外，可以使用 AggreCuts 算法中的层级比特串压缩技术减少每个比特串的长度。如图 6.15 所示，原始的 16 位比特串被 8 位的层级比特串代替。实验结果表明使用 64 位比特串能够实现与 256 位比特串相近的压缩率。

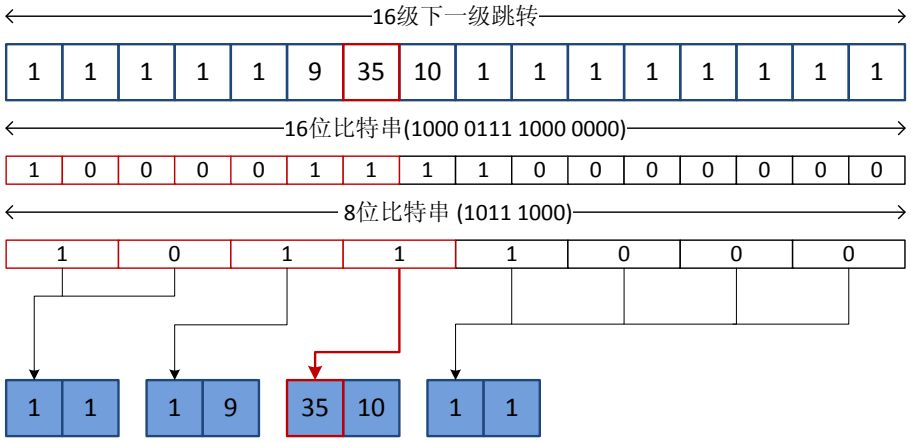


图 6.15　状态跳转表的层级比特串压缩[46]

根据 FEACAN 算法，首领状态可以是组中的任何一个状态。由于首领状态的访问只需要一次内存存取即可到达下一状态，优化首领状态的选择将会很大程度上提高分类速率。实验发现，超过 90%的输入字符仅访问不到 10%的状态（如图 6.16 所示）。因此，可以选择每组中访问最为频繁的状态作为首领状态。

优化后的状态聚类算法见图 6.17。

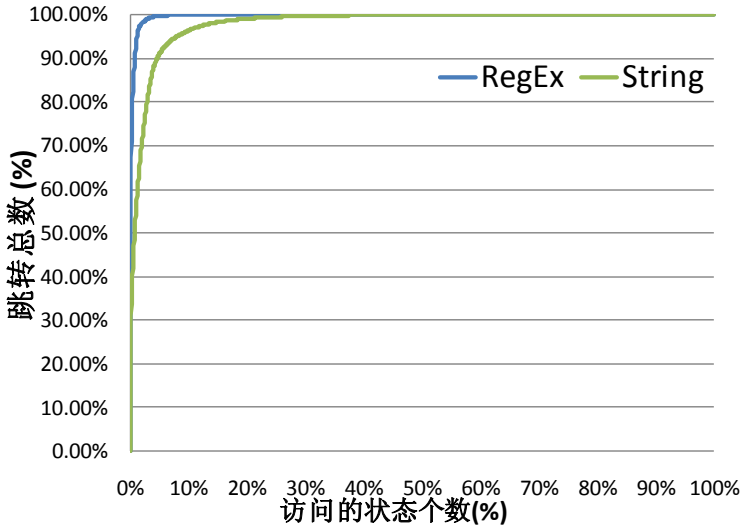


图 6.16　不同状态的访问次数[46]

```
Grouping(bitmaps, states, thresh);
while (TRUE)
   for (state s ∈ states && s.done == NO)
    allover = NO;
    if (s.hotval > thresh)
           s.done = YES; break;
        else if (hotover == YES)  s.done = YES; break;
        else allover = YES; continue;
        end if
   end for
   if (s ∉ states)
        if (allover == NO)
        hotover = YES; continue;
        else break;
        end if
   else
        add s in a new group G as the leader state;
   end if
   for (state t ∈ states && group.size < max_size)
        if (t.bitmapID == s.bitmapID && t.done == NO
          && t.hotval <= thresh)
           add state t in G as a member state;
           if (diversity(G) < max_div) t.done = YES;
           else delete t from G;
           end if
        end if
   end for
end while
```

图 6.17　优化的状态聚类算法[46]

6.3　硬件实现

Floyd 等人首先指出，基于正则表达式匹配的 NFA 能够在可编程逻辑阵列上有效实现[90]。Sindhu[91]和 Clark[92]等分别在 FPGA 设备上实现了基于 NFA 的正则表达式匹配系统，并且在分类速率和内存使用上都取得了良好的效果。然而，基于 NFA 的算法需要在硬件逻辑上对 NFA 自动机进行编码，在线升级将难以实现[46]。因此，面向存储而非面向逻辑的正则表达式匹配引擎通常具备更高的灵活性。

6.3.1　算法映射

FEACAN 算法在 FPGA 上的硬件实现使用图 6.18 的 4 级流水线映射。此系统的输入是当前状态 state_in 和输入字符 char_in，输出为下一个状态 state_next。

- 在第 1 级流水线，硬件系统通过 state_in 的 groupID 获取各类表项的地址信息，包括比特串索引（bitmapID）、首领状态基址（leaderBase）以及成员状态基址（memberBase）等
- 在第 2 级流水线，系统使用 bitmapID 读取比特串，之后利用位操作来计算首领状态索引（leaderID）。
- 在第 3 级流水线，系统通过（leaderBase+leaderID）来读取 leaderTransition，之后通过检查 memberID 和 memberOffset 的数值来决定是否找到下一状态 state_out。如果找到，将 state_out 设置为 leaderTransition 的<groupID,memberID>值；否则，设置 state_out 为 0。
- 在第 4 级流水线，如果 state_out 值为 0，系统读取（memberBase+ memberOffset + memberID）中的的 memberTransition 参数并将 state_out 设置为 memberTransition 中的<groupID,memberID>。

由于每一级流水线逻辑单元都在一个时钟周期内完成处理，因此每一个输入字符可以在 4 个时钟周期内查找。为了支持在线升级，查找引擎中的 SRAM 可以通过图 6.19 中的 Write Bubble 技术实现[71]。

6.3.2　硬件优化

FEACAN 算法首先使用 input-interleaving 技术来提高并行处理能力[23]。相比于从单一网包载荷中向引擎输入字符，FEACAN 使用来自四个不同网包的字节流依次作为输入。这种方法使得每级流水线单元在每个时钟周期都有全新的输入，因此将每个字节的平均处理时间从 4 个时钟周期降低到了 1 个。

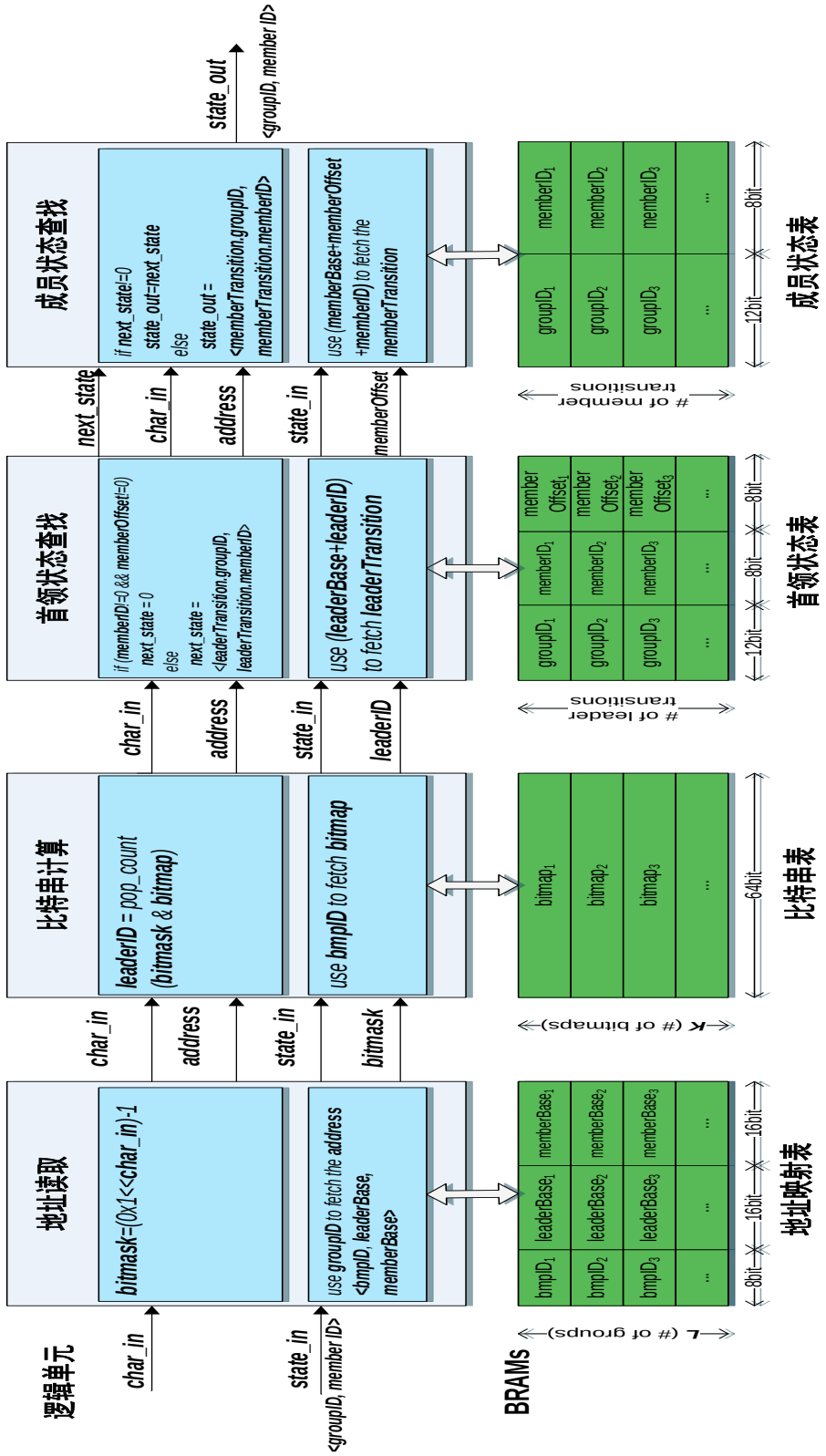


图 6.18　FEACAN 算法映射[46]

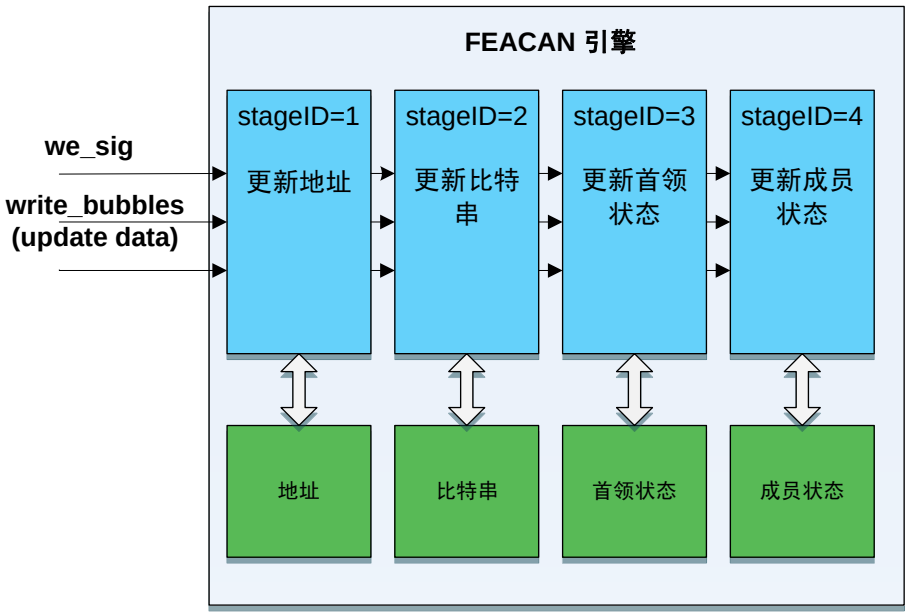


图 6.19　查找引擎的在线更新[46]

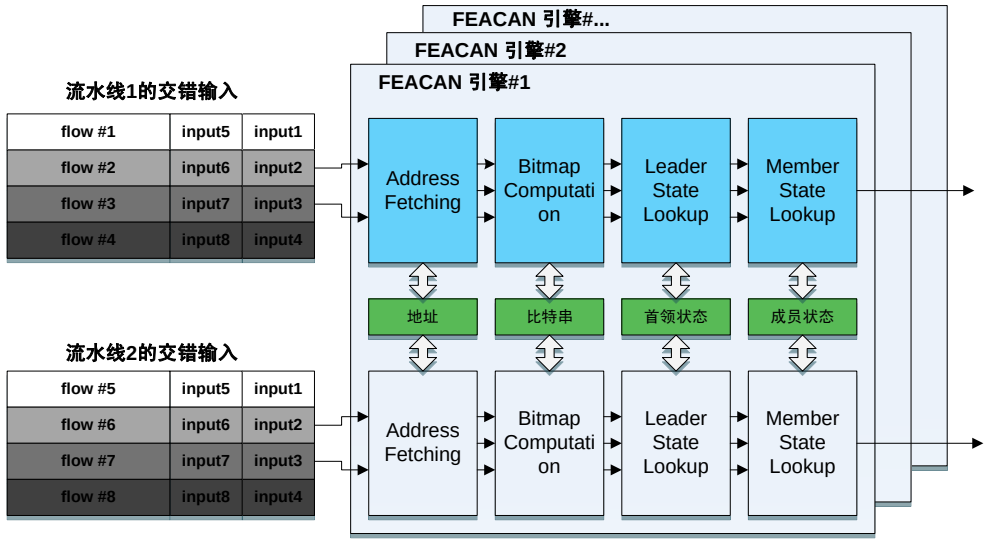


图 6.20　查找引擎的并行处理机制[46]

类似第 5 章中的双流水线方法，FEACAN 算法也可以利用双通道的 BRAM 实现双流水线引擎的并行查找，从而将分类速率提高一倍。若利用内存复制，还可实现多引擎并行查找进一步拓展 FEACAN 算法的性能。图 6.20 为多引擎并行查找的 FEACAN 算法实现。

6.4　性能评价

6.4.1　测试数据及平台

测试中所使用的正则表达式规则都是包括：snort24.re 正则表达式（Snort[93]中的 24 个规则，包含.*及大量字符区间等，生成）、bro217.re 正则表达式（Bro[94]中的 217 个规则，包含大量字符串及少量字符区间）、snortPart（Snort 中的 1982 个短字符串规则）和 snortAll.str（Snort 中所有 5766 个字符串）。测试中所使用的网络流量包括 ap020.tr 和 ap060.tr（踪来自于大学校园网络）、C20010.tr（来自一个较大规模的企业网络）、inside.tr 和 outside.tr（来自 MIT Lincoln Lab[95]）、trace0seg.tr 和 trace15seg.tr（来自于一个拥有 1000 多个用户的研究中心）。

FPGA 算法测试中使用的是 Xilinx Virtex-6（model: XC6VSX475T) 芯片[50]。该芯片的计算资源为 37,440 可编程逻辑单元（configurable logic blocks）。包含 7,640K 比特分布式存储单元（Distributed RAM）以及 38,304K 比特 BRAM。所有实验结果均使用 Xillinx 的 ISE 仿真平台获取[51]。

性能评价测试的算法包括原始的 DFA 算法，Kumar 等提出的 D^2FA 算法[47]，Becchi 等提出的改进 D^2FA 算法[83]，以及 FEACAN 算法。为了方便起见，测试中分别将这些算法分别记为 DFA、D^2FA、D^2FA improved 和 FEACAN。测试中使用的评价准则包括：内存访问次数，内存使用，以及预处理时间。

6.4.2　内存访问

由于基于 DFA 的算法均是面向存储的实现，因此内存访问的效率决定了查找速度。为了进行公平的性能比较，实验中 D^2FA 的默认路径深度设为 2，使其与 FEACAN 算法具有相同的最坏情况内存访问次数。图 6.21 和图 6.22 显示了各类算法在两个不同的正则表达式规则集合 *bro217.re* 和 *snortPart.str* 中每个字符的平均访问状态个数。

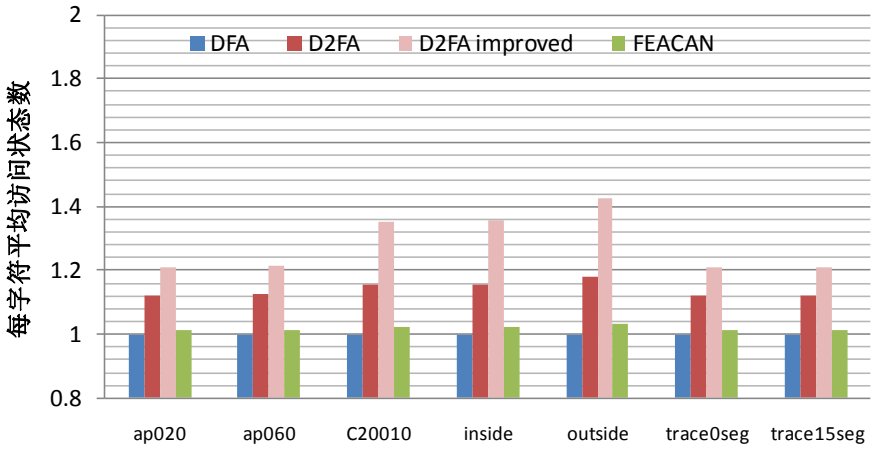


图 6.21　每字符平均访问状态数（*bro217.re* 规则集合）[46]

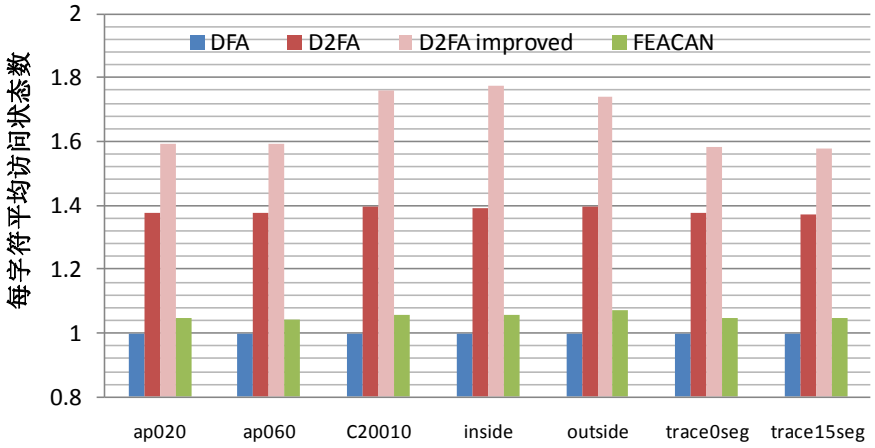


图 6.22　每字符平均访问状态数（*snortPart.str* 规则集合）[46]

从图 6.21 和图 6.22 可以看出，FEACAN 算法每个输入字符平均访问不到 1.05 个状态（DFA 算法每字符访问 1 个状态），而无论是 D^2FA 算法还是 D^2FA improved 算法都需要访问更多的状态（1.1～1.8 个）。

由于每个状态包含跳转的个数不同，因此仅仅依据状态的遍历数并不能反映实际的内存访问次数。表 6.4 中给出了每字节平均遍历的跳转个数。从该表可以看出，D^2FA 算法和 D^2FA improved 算法的每字节平均跳转次数位 80～120 次，而 FEACAN 算法只需要 1 次。如果一个跳转用 3 个字节存储，那么 D^2FA 算法需要 100*（3/1）*10Gbps = 3Tbps 的内存带宽来支持 10Gbps 线速度处理。与之相比，FEACAN 算法只需要约 1*（3/1）*10Gbps = 30Gbps 的内存带宽进行状态访存。

表 6.4 每字符平均访问跳转数 (*bro217*.re)[46]

流量	DFA	D^2FA	D^2FA imprvd	FEACAN
ap020	1	121.86	121.81	1.01
ap060	1	120.05	120.00	1.01
C20010	1	85.89	85.59	1.02
inside	1	82.97	82.70	1.02
outside	1	86.11	84.81	1.03
trace0seg	1	123.33	123.27	1.01
trace15seg	1	122.87	122.82	1.01

6.4.3　内存使用

这部分实验以每个算法的总跳转访问次数来进行内存使用的比较。由图 6.23 可知，FEACAN 算法相比于原始 DFA 算法的压缩率大于 90%。即使是用于最大的 snortAll.str 规则集合，FEACAN 算法的内存使用仍然少于 2MB。相比于 D^2FA 算法和 D^2FA improved 算法，FEACAN 也在大多数规则集下具有最大的压缩率。

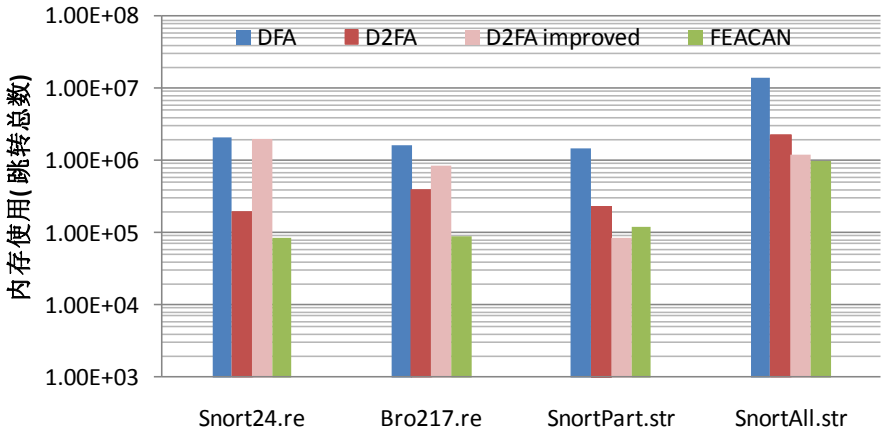


图 6.23　内存使用[46]

6.4.4　预处理时间

算法的预处理时间决定了一个网络系统对于新的规则的响应速度。图 6.24 的实验结果给出了 D2FA 算法、D2FA improved 算法和 FEACAN 算法对同一 DFA 进行压缩的预处理时间（不包括 DFA 状态转移矩阵的构造时间）。对于测试中所有的规则集合，FEACAN 算法的预处理时间平均为 D2FA 算法的 1/100 或 D2FA improved 算法的 1/10。对于最大的规则集合 snortAll.str，FEACAN 算法的预处理时间少于 10 秒。相比而言，D2FA improved 算法需要两分钟的时间，而 D2FA 算法需要多于 20 分钟的时间。

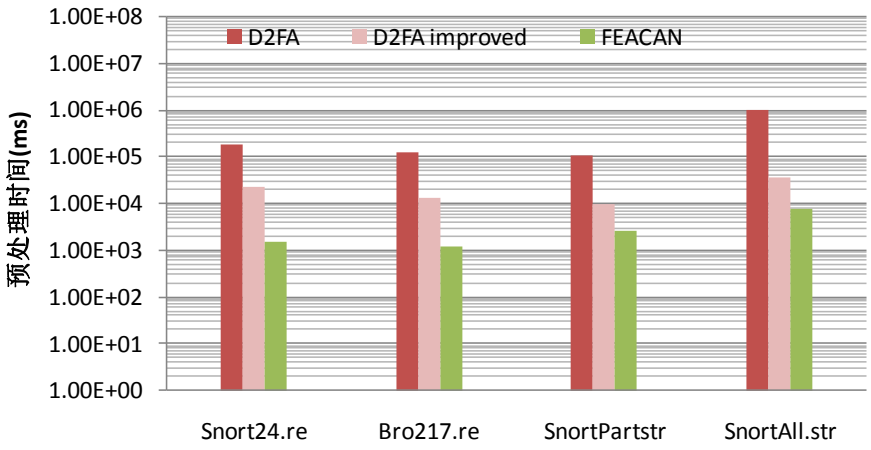


图 6.24　预处理时间[46]

6.4.3　系统吞吐率

由于 FEACAN 引擎每个时钟周期处理一个字节的流量，因此其吞吐率可以根据最大时钟频率获得。根据 ISE 仿真器上的实验，制约 FEACAN 引擎最大时钟频率的是比特串计算单元。使用不同长度的比特串，可以获取不同的最大时钟频率。例如，使用 256 位比特串所得的最大时钟频率为 156.006MHz，使用 32 位层级比特串则可获取 227.583MHz 的最大时钟频率。

若基于 150MHz 的时钟频率的 FPGA 实现，每个 FEACAN 引擎可以达到$150\text{MHz} * 8\text{bit} * 2 = 2.4\text{Gbps}$的吞吐率。若使用 ASIC（专用集成电路）技术实现，通常可以获得 500MHz 的时钟频率。因此，每个基于 ASIC 的 FEACAN 引擎吞吐率可以达到$500\text{MHz} * 8\text{bit} * 2 = 8\ \text{Gbps}$。

使用并行 FEACAN 查找引擎，可以实现分类速率的进一步拓展。由于 FEACAN 是面向存储器的设计，并行引擎的数量将受到 FPGA 平台中可用的 BRAM 总量的限制。表 6.5 给出了单一的 Xilinx Virtex-6 芯片能实现的最大分类引擎个数。从表 6.5 可以看出，FEACAN 能够在 FPGA 芯片上实现 10～40Gbps 吞吐率，并在 ASIC 芯片上实现超过 100Gbps 的吞吐率。

表 6.5　FEACAN 算法在硬件平台上的吞吐率[46]

规则集合	单片 FPGA 可部署最大引擎数	FPGA 吞吐率 (150MHz)	ASIC 吞吐率 (500MHz)
Snort24.re	15	36.0 Gbps	120 Gbps
Bro217.re	17	40.8 Gbps	136 Gbps
SnortAll.str	4	9.6 Gbps	32 Gbps

6.5　本章小结

随着网包分类技术的发展，依据多域网包包头信息的网包分类系统已经不能满足日益增长的网络业务需求。本章研究了广泛用于深度检测、业务识别等网络处理的基于网包载荷的网包分类算法。基于载荷的分类通常使用正则表达式（regular expression）规则与网包载荷中的字节流进行匹配，并依据匹配结果对网包进行分类。

由于网包载荷的长度不定，依据网包载荷的网包分类算法从多域向无穷域进行了拓展。本章基于现有的 DFA 算法，提出了一种新型的基于网包载荷的正则表达式匹配算法 FEACAN。FEACAN 使用状态内和状态间的二维压缩方法，在保证原始 DFA 算法时间性能的同时，降低 DFA 状态跳转矩阵的内存使用。基于 Snort 和 Bro 等实际规则集合的实验表明，FEACAN 算法可以取得平均 90%的 DFA 压缩率。硬件仿真表明，FEACAN 算法能够在 FPGA 芯片上实现 10～40Gbps 吞吐率，并在 ASIC 芯片上实现超过 100Gbps 的吞吐率。

第 7 章　结 论

随着云计算、物联网、移动互联网等前沿技术的发展，高性能网包分类已成为下一代互联网发展和演进中的研究热点。本文从理论分析、算法设计和系统实现三个层面对多域网包分类算法进行了深入的研究。

在网包分类的理论研究中，本文第 2 章依据计算几何中多域空间点定位的数学解法，归纳和总结了网包分类算法的理论依据。从分类规则的几何投影出发，一大类已有网包分类算法采用投影区间查找的方法解决网包分类问题；从子空间划分的角度来看，另一大类已有算法采用各种不同的空间分解方法实现网包分类。各类方法的理论复杂度分析表明，不存在一种同时具有良好时间和空间性能的完美算法。而现有网包分类算法均是利用各种启发式方法，通过挖掘规则集合特征来提高特定情况下的算法性能。因此，时间和空间性能的权衡，以及启发式方法的效率，决定网包分类算法的实际性能。本文在算法设计中均采用了优先保证分类速率，尽量减少内存使用的权衡策略。

本文第 3 章在总结区间查找算法的基础上，提出了 HyperSplit 算法，利用多类启发式方法构建二分查找的多域区间树，一方面保持了同类算法分类速率高的特性，另一方面通过避免不必要的二分查找，大幅降低了内存使用。与经典的同类算法 HSM 相比较，HyperSplit 在保证内存访问次数不增加的情况下，将内存使用降低了 1 至 2 个数量级，且在 Cavium Octeon3860 多核网络处理器上实现了 8Gbps 的吞吐率。

与区间查找算法想对应，本文在第 4 章总结了空间分解算法的设计思想和典型算法，并在此基础上提出了 AggreCuts 算法。AggreCuts 利用确定性的空间切分次数来限制最坏情况下的决策树深度，从而使时间性能得到保证。在决策树构建过程中，AggreCuts 采用层级比特串压缩和数据结构归并的方法降低内存使用。与典型的同类算法 HiCuts 相比，AggreCuts 的内存访问次数不到其 20%，而内存使用则降低了 1 个数量级，且在 Cavium Octeon3860 和 Intel IXP2850 多核网络处理器上分别实现了 8Gbps 和 10Gbps 的吞吐率。

当基于多核网络处理器的网包分类系统不能满足吞吐率需求时，基于 FPGA 可编程硬件平台的网包分类系统可以达到更高的性能。然而，基于硬件的网包分类算法设计必须考虑严格的计算和存储的资源限制。本文第 5 章总结了网包分类算法在 FPGA 硬件平台上实现和优化的思路和方法，并提出了基于 FPGA 平台的 HyperSplit 算法。该算法通过内部节点合并、叶节点下移、多子树并行等方法针对

FPGA 硬件进行了优化。硬件仿真实验表明，实现于 Xilinx Virtex-6 芯片上的 HyperSplit 算法可以在支持 5 万条分类规则的同时，获取 100Gbps 以上的吞吐率。

作为多域网包分类算法的拓展，本文第 6 章研究了基于正则表达式匹配的网包分类问题，在对现有正则表达式匹配算法分析的基础上，提出了基于 DFA 的 FEACAN 算法。与 HyperSplit 和 AggreCuts 算法的设计思想一样，在 FEACAN 的算法设计中依然采用了在优先保证时间性能的前提下尽可能提高空间性能的策略和二维的空间压缩算法，首先使用 AggreCuts 中的层级比特串压缩技术消除 DFA 状态内的冗余，然后使用聚类和编码技术进一步压缩 DFA 状态间的冗余。实验表明，FEACAN 算法能够在保持与原始 DFA 算法接近的分类速率的同时，获得与 D^2FA 算法相同的 DFA 压缩率，且在 FPGA 硬件平台上实现了 8～40Gbps 的吞吐率。

随着互联网架构的不断演进以及互联网业务的不断增长，多域网包分类问题的研究也会不断发展和延续。从近年来兴起的云计算数据中心网络、移动互联网、软件定义网络来看，现有网包分类算法面临着全新的挑战。首先，传统的五元组分类将被更高维度的分类问题取代。例如在基于 OpenFlow 的软件定义网络中，每一个 OpenFlow 交换机都需要进行 12 元组的网包分类。因此未来的网包分类算法需要支持任意维度的网包分类。其次，随着越来越多的网络业务基于应用层实现，传统的网包分类技术将无法满足依据业务对流量进行分类的需求。因此，基于应用层的网包分类技术将成为今后网包分类算法研究的另一个重要内容。最后，随着网络业务融合、数据中心建设，越来越多的网络设备将需要高效的网包分类功能。因此，具备网包分类功能的高性能低功耗的芯片技术，也具有重要的研究价值。

综上所述，严谨的数学方法和充分的复杂度分析构成算法研究的根基，巧妙的启发式方法和高效的数据结构设计则是算法实用性的保障。基于多核网络处理器的网包分类系统易于实现和部署，而基于 FPGA 的硬件分类引擎可以满足更高的处理需求。

参考文献

[1] Varghese G. Network algorithmics: an interdisciplinary approach to designing fast networked devices. Morgan Kaufmann Publishers, 2005.

[2] Chao J, Liu B. High performance switches and routers. Wiley, 2007.

[3] 林闯, 单志广, 任丰原. 计算机网络的服务质量(QoS). 清华大学出版社, 2004.

[4] 徐恪, 吴建平, 徐明伟. 高等计算机网络--体系结构、协议机制、算法设计与路由器技术(第 2 版), 机械工业出版社, 2009.

[5] Casado M, Freedman J, Pettit J, Luo J, McKeown N, Shenker S. Ethane: taking control of the enterprise. Proceedings of ACM SIGCOMM, 2007.

[6] Joseph D, Tavakoli A, Stoica I. A policy-aware switching layer for data centers. Proceedings of ACM SIGCOMM, 2008.

[7] Koponen T, Casado M, Gude N, Stribling J, Poutievski L, Zhu M, Ramanathan R, Iwata Y, Inoue H, Hama T, Shenker S. Onix: A distributed control platform for large-scale production networks. Proceedings of OSDI, 2010.

[8] Popa L, Egi N, Ratnasamy S, Stoica I. Building extensible networks with rule-based forwarding. Proceedings of OSDI, 2010.

[9] Srinivasan V, Varghese G, Suri S, et al. Fast and scalable layer four switching. Proceedings of ACM SIGCOMM, 1998, 191-202.

[10] Srinivasan V, Suri S, Varghese G. Packet classification using tuple space search. Proceedings of ACM SIGCOMM, 1999, 135-146.

[11] Lakshman T, Stiliadis D. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. Proceedings of ACM SIGCOMM, 1998.

[12] Gupta P, McKewon N. Algorithms for packet classification. IEEE Network, 2001, 15(2): 24-32.

[13] Gupta P, McKeown N. Packet classification on multiple fields. Proceedings of ACM SIGCOMM, 1999, 147-160.

[14] Gupta P, McKeown N. Packet classification using hierarchical intelligent cuttings. Proceedings Hot Interconnects VII, 1999.

[15] Baboescu F, Varghese G. Scalable packet classification. Proceedings of ACM SIGCOMM, 2001, 199-210.

[16] Baboescu F, Singh S, Varghese G. Packet classification for core routers: Is there an alternative to CAMs? Proceedings of IEEE INFOCOM, 2003, 53-63.

[17] Singh S, Baboescu F, Varghese G, et al. Packet classification using multidimensional cutting. Proceedings of ACM SIGCOMM, 2003, 213-224.

[18] Taylor D, Turner J. ClassBench: a packet classification benchmark. IEEE/ACM Trans. Netw.

(TON), 2007, 15(3):499-511.

[19] Taylor D. Survey and taxonomy of packet classification techniques. ACM Computer Survey, 2005, 37(3):238–275.

[20] Overmars M, Van der Stappen A. Range searching and point location among fat objects. Journal of Algorithms 1996, 21(3).

[21] Vamanan B, Voskuilen G, Vijaykumar T. EffiCuts: Optimizing packet classification for memory and throughput. Proceedings of ACM SIGCOMM, 2010.

[22] Spitznagel E, Taylor D, Turner J. Packet classification using extended TCAMs. Proceedings of ICNP, 2003.

[23] Lakshminarayanan K, Rangarajan A, Venkatachary S. Algorithms for advanced packet classification with ternary CAMs. Proceedings of ACM SIGCOMM 2005.

[24] Zheng K, Che H, Wang Z, Liu B. TCAM-based distributed parallel packet classification algorithm with range-matching solution. Proceedings of IEEE INFOCOM, 2005.

[25] Meiners C, Liu A, Torng E. TCAM Razor: a systematic approach towards minimizing packet classifiers in TCAMs. Proceedings of ICNP, 2007.

[26] Meiners C, Liu A, Torng E. Topological transformation approaches to optimizing TCAM-based packet classification systems. Proceedings of ACM SIGMETRICS, 2009.

[27] Meiners C, Liu A, Torng E. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. Proceedings of ICNP, 2009.

[28] Pao D, Li Y, Zhou P. Efficient packet classification using TCAMs. Computer Networks (CN), 2006, 50(18):3523-3535.

[29] Sun Y, Kim M. Bidirectional range extension for TCAM-based packet classification. Proceedings of Networking, 2010.

[30] Xu B, Jiang D, Li J. HSM: a fast packet classification algorithm. Proceedings of the International Conference on Advanced Information Networking and Applications (AINA), 2005.

[31] Xu B, Qi Y, He F, Zhou Z, Xue Y, Li J. Fast path session creation on network processors. Proceedings of ICDCS, 2008.

[32] Liu D, Hua B, Hu X, et al. High-performance packet classification algorithm for many-core and multithreaded network processor. International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2006.

[33] Qi Y, Xu L, Yang B, et al. Packet classification algorithms: from theory to practice. Proceedings of IEEE INFOCOM, 2009.

[34] Qi Y, Xu B, He F, Yang B, Yu J, Li J. Towards high-performance flow-level packet processing on multi-core network processors. Proceedings of ACM/IEEE ANCS, 2007.

[35] Qi Y, Fong J, Jiang W, Bo X, Li J, Prasanna V. Multi-dimensional packet classification on FPGA: 100Gbps and beyond. Proceedings of Internation Confererence on Field-Programmable Technology (FPT), 2010.

[36] Fong J, Qi Y, Li J. ParaSplit: an FPGA-optimized packet classification engine. Unpublished NSLab Technical Report, 2011.

[37] Jiang W, Prasanna V. Large-scale wire-speed packet classification on FPGAs. Proceedings of FPGA, 2009.

[38] Jiang W, Prasanna V. Field-split parallel architecture for high performance multi-match packet classification using FPGAs. Proceedings of SPAA, 2009.

[39] Jiang W, Prasanna V. Yamagaki N. Decision forest: a scalable architecture for flexible flow matching on FPGA. Proceedings of FPL 2010.

[40] Xu Y, Liu Z, Zhang Z, Chao J. An ultra high throughput and memory efficient pipeline architecture for multi-match packet classification without TCAMs. Proceedings of ACM/IEEE ANCS, 2009.

[41] Zhang T, Wang Y, Zhang L, Yang Y. High throughput architecture for packet classification using FPGA. Proceedings of ACM/IEEE ANCS, 2009.

[42] Yan T, Wang Y. Design of packet classification co-processor with FPGA. Proceedings of ESA, 2005.

[43] Song H, Lockwood J. Efficient packet classification for network intrusion detection using FPGA. Proceedings of FPGA, 2005.

[44] Feldmann A, Muthukrishnan S. Tradeoffs for packet classification. Proceedings of IEEE INFOCOM, 2000.

[45] Gupta P, McKewon N. Algorithms for routing lookups and packet classification. [Ph. D Thesis]. Computer Science Dept., Stanford University, 2000.

[46] Qi Y, Wang K, Fong J, Jiang W, Xue Y, Li J, Prasanna V. FEACAN: Front-end acceleration for content-aware network processing. Proceedings of IEEE INFOCOM, 2011.

[47] Kumar S, Dharmapurikar S, Yu F, Crowley P, Turner J. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. Proceedings of ACM SIGCOMM, 2006.

[48] Cavium OCTEON3860 [EB/OL]. [2011-04-01]. http://www.caviumnetworks.com/OCTEON-Plus_CN38XX_solutions.html.

[49] IXP2850 [EB/OL]. [2011-04-01]. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[50] Virtex-6 FPGA [EB/OL]. [2011-04-01]. http://www.xilinx.com/products/virtex6/.

[51] ISE Design Tool [EB/OL]. [2011-04-01]. http://www.xilinx.com/tools/designtools.htm.

[52] 邓俊辉（译著）. 计算几何-算法与应用（第二版）. 清华大学出版社, 2005.

[53] Chazelle B, Guibas L. Fractional cascading I: A data structuring technique. Algorithmica, 1986.

[54] Chang Y, Lin Y, Lin C. Grid of segment trees for packet classification. Proceedings of AINA, 2010.

[55] Tsuchiya P. A search algorithm for table entries with non-contiguous wildcarding.

Unpublished report, Bellcore.

[56] Qi Y, He F, Wang X, Chen X, Xue Y, Li J. OpenGate: Towards an open network services gateway. Computer Communications, 2011, 34(2): 200-208.

[57] Qi Y, Xue Y, Li J. High-performance packet classification on multi-Core network processing platforms. Journal of Tsinghua Science and Technology, 2011.

[58] Woo T. A modular approach to packet classification: algorithms and results. Proceedings of IEEE INFOCOM, 2000.

[59] Pao D, Liu C. Parallel tree search: An algorithmic approach for multi-field packet classification. Computer Communications (COMCOM), 2007, 30(2):302-314.

[60] Lu W, Sahni S. Packet classification using space-efficient pipelined multibit tries. IEEE Trans. Computers (TC), 2008, 57(5):591-605.

[61] Zheng B, Lin C, Peng X. AM-Trie: an OC-192 parallel multidimensional packet classification algorithm. Proceedings of IMSCCS, 2006.

[62] 陈友，张艳军，郭莉，程学旗. 基于决策树的递归包分类算法. 北京邮电大学学报, 2006.

[63] 张树壮，罗浩，方滨兴. 一种支持实时增量更新的并行包分类算法. 计算机研究与发展, 2010.

[64] Qi Y, Xu B, He F, et al. Towards optimized packet classification algorithms for multi-core network processors. Proceedings of International Conference on Parallel Processing (ICPP), 2007.

[65] Song H, Turner J, Dharmapurikar S. Packet classification using coarse-grained tuple spaces. Proceedings of ANCS, 2006.

[66] Dharmapurikar S, Song H, Turner J, Lockwood J. Fast packet classification using bloom filters. Proceedings of ANCS, 2006.

[67] Jedhe G, Ramamoorthy A, Varghese K. A scalable high throughput firewall in FPGA. Proceedings of FCCM, 2008.

[68] Luo Y, Xiang K, Li S. Acceleration of decision tree searching for IP traffic classification. Proceedings of ANCS, 2008.

[69] Kennedy A, Wang X, Liu J, Liu B. Low power architecture for high speed packet classification. Proceedings of ANCS, 2008.

[70] Basu A, Narlikar G. Fast incremental updates for pipelined forwarding engines. Proceedings of IEEE INFOCOM, 2003.

[71] Le H, Jiang W, Prasanna V. Scalable high-throughput SRAM-based architecture for IP lookup using FPGA. Proceedings of Intl. Conf. on Field Programmable Logic and Applications (FPL), 2008.

[72] LSI Tarari T2000/T2500 [EB/OL]. [2011-04-01]. http://www.lsi.com/networking_home/networking_products/tarari_content_processors/.

[73] NetLogic NETL7 [EB/OL]. [2011-04-01]. http://www.netlogicmicro.com/Products/Layer7/Layer7.htm.

[74] Cavium OCTEON5860 [EB/OL]. [2011-04-01]. http://www.cavium.com/OCTEON_MIPS64.html.

[75] Sommer R, Paxson V. Enhancing byte-level network intrusion detection signatures with context. Proceedings of ACM CCS, 2003.

[76] Hopcroft J, Motwani R, Ullman J. Introduction to automata theory, languages, and computation (third edition). Addison Wesley, 2008.

[77] Aho A, Lam M, Sethi R, Ullman J. Compilers: principles, techniques and tools, second edition. Addison Wesley, 2009. 2008, 57(5):591-605.

[78] Thompson K. Regular expression search algorithm. Communication of ACM, 1968, 11(6): 419-422.

[79] PCRE-Perl Compatible Regular Expressions [EB/OL]. [2011-04-01]. http://www.pcre.org.

[80] Tuck N, Sherwood T, Calder B, et al. Deterministic memory-efficient string matching algorithms for intrusion detection. Proceedings of IEEE INFOCOM, 2004.

[81] Aho A, Corasick M. Efficient string matching: An aid to bibliographic search. Communications of the ACM, 1975, 18(6):333–340.

[82] Kumar S, Turner J, Williams J. Advanced algorithms for fast and scalable deep packet inspection. Proceedings of ACM/IEEE ANCS, 2006.

[83] Becchi M, Crowley P. An improved algorithm to accelerate regular expression evaluation. Proceedings of ACM/IEEE ANCS, 2007.

[84] Brodie B, Cytron R, Taylor D. A scalable architecture for high-throughput regular-expression pattern matching. Proceedings of ISCA, 2006.

[85] Kumar S, Chandrasekaran B, Turner J, Varghese G. Curing regular expressions matching algorithms from insomnia, amnesia, and Acalculia. Proceedings of ACM/IEEE ANCS, 2007.

[86] Becchi M, Crowley P. A hybrid finite automaton for practical deep packet inspection. Proceedings of CoNEXT, 2007.

[87] Becchi M, Wiseman C, Crowley P. Evaluating regular expression matching engines on network and general purpose processors. Proceedings of ACM/IEEE ANCS, 2009.

[88] Smith R, Estan C, Jha S. XFA: Faster signature matching with extended automata. Proceedings of IEEE Symposium on Security and Privacy (S&P), 2008.

[89] Smith R, Estan C, Jha S, Kong S. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. Proceedings of ACM SIGCOMM, 2008.

[90] Floyd R, Ullman J. The compilation of regular expressions into integrated circuits. Journal of ACM, 1982, 29(3): 603-622.

[91] Sidhu R, Prasanna V. Fast regular expression matching using FPGAs. Proceedings of IEEE FCCM, 2001.

[92] Clark C, Schimmel D. Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns. Proceedings of FPL, 2003.

[93] Snort Open Source IDS/IPS [EB/OL]. [2011-04-01]. http://www.snort.org.

[94] Bro Intrusion detection system [EB/OL]. [2011-04-01]. http://www.bro-ids.org/.

[95] DARPA intrusion detection data sets [EB/OL]. [2011-04-01]. http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html

[96] Kounavis M, Kumar A, Vin H, Yavatkar R, Campbell A. Directions in packet classification for network processors. Proceedings of Network Processor Workshop, 2003.

[97] Cohen E, Lund C. Packet classification in large ISPs: design and evaluation of decision tree classifiers. Proceedings of ACM SIGMETRICS, 2005.

[98] Srinivasan D, Feng W. Performance analysis of multi-dimensional packet classification on programmable network processors. Proceedings of LCN, 2004.

[99] Pus V, Korenek J. Fast and scalable packet classification using perfect hash functions. Proceedings of FPGA, 2009.

[100] Spirent SmartBits [EB/OL]. [2011-04-01]. http://www.spirent.com/Solutions-Directory/Smartbits.aspx.

致 谢

衷心感谢导师李军研究员,他不仅在科研中给予启发的指导，在生活中也对我关怀备至。李老师开阔的国际视野、严谨的治学态度、渊博的专业知识和饱满的工作热情将使我受益终身。

感谢薛一波研究员、陈震老师、周晋老师以及清华大学网络安全实验室每一位同学在学习和生活中给我的帮助。特别感谢徐波、何飞、杨保华、王凯、Jeffrey Fong 在科研工作中给我的宝贵建议和无私协助。感谢 MSU 的 Alex Liu 教授、USC 的 Viktor Prasanna 教授、IBM 的谭伟博士、Juniper 的蒋蔚荣博士，感谢你们在我赴美留学期间给予的指导和帮助。

本课题承蒙国家 863 计划基金资助，特此表示感谢!

最后，衷心感谢我的父亲、母亲以及夫人李君，感谢你们对我无微不至的关怀和永远的鼓励与支持!

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1979 年 8 月 24 日出生于陕西省西安市。

1998 年 9 月考入清华大学自动化系自动化专业，2002 年 7 月本科毕业并获得工学学士学位。

2002 年 9 月考入清华大学自动化系控制科学与工程专业，2005 年 7 月研究生毕业并获得工学硕士学位。

2008 年 9 月考入清华大学自动化系攻读工学博士至今。

2009 年 11 月入选国家留学基金委公派留学博士联合培养项目，在美国密歇根州立大学（合作导师 Alex X. Liu）和南加州大学（合作导师 Viktor K. Prasanna）进行为期 12 个月的研究工作。

发表的学术论文

[1] Yaxuan Qi, Fei He, Xiang Wang, Xinming Chen, Yibo Xue, Jun Li. OpenGate: Towards an open network services gateway. **Computer Communications**, 2011, 34(2): 200-208. (SCI 索引, IF: 0.933).

[2] 亓亚烜，李军．高性能网包分类理论与算法研究综述．**计算机学报**, 2011. (EI 索引，已录用)

[3] Yaxuan Qi, Yibo Xue, Jun Li. High-performance packet classification on multi-Core network processing platforms. Journal of Tsinghua Science and Technology (**清华学报**), 2011. (EI 索引，已录用)

[4] Yaxuan Qi, Kai Wang, Jeffrey Fong, Weirong Jiang, Yibo Xue, Jun Li and Viktor Prasanna, FEACAN: Front-End Acceleration for Content-Aware Network Processing, Proc. of the 30th IEEE **INFOCOM**, 2011. (EI 索引)

[5] Yaxuan Qi, Fei He, Kai Wang, Xinming Chen, Jeffrey Fong, Feng Xie, Yiyang Shao, Yang Gao, Yibo Xue, Jun Li. LiveSec: OpenFlow-based Security management for production networks. Proc. of the 30th IEEE **INFOCOM**, 2011. (live demo)

[6] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, Jun Li. Packet classification algorithms: from theory to practice. Proc. of the 28th IEEE **INFOCOM**, 2009. (EI 索引)

[7] Yaxuan Qi, Jeffrey Fong, Weirong Jiang, Bo Xu, Jun Li, Viktor Prasanna. Multi-dimensional packet classification on FPGA: 100 Gbps and beyond. Proc. of the International Conference on Field-Programmable Technology (**FPT**), 2010. (EI 索引)

[8] Yaxuan Qi, Zongwei Zhou, Yiyao Wu, Yibo Xue, Jun Li. Towards high-performance pattern matching on multi-core network processing platforms. Proc. of the IEEE **GLOBECOM**, 2010. (EI 索引)

[9] Yaxuan Qi, Fei He, Xiang Wang, Xinming Chen, Yibo Xue, Jun Li. OASis: towards extensible open-architecture services platforms. Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (**ANCS**) 2009. (short paper) (EI 索引)

[10] Kai Wang, Yaxuan Qi, Yibo Xue, Jun Li. Reorganized and compact DFA for efficient regular expression matching. Proc. of the IEEE International Conference on Communications (**ICC**), 2011. (EI 索引)

[11] Weirong Jiang, Yaxuan Qi. Power-aware parallel forwarding: An optimization study. Proc. of the IEEE/ACM International Conference on Green Computing and Communications (**GREENCOM**), 2010. (EI 索引)

[12] Xiang Wang, Yaxuan Qi, Baohua Yang, Yibo Xue, Jun Li. Towards high-performance network instrusion prevention system on multi-core network service processor. Proc. of the IEEE International Conference on Parallel and Distributed System (**ICPADS**), 2009. (EI 索引)

[13] Baohua Yang, Guodong Li, Yaxuan Qi, Yibo Xue, Jun Li: DFC: Towards effective feedback flow management for datacenters. Proc. of **GCC** 2010. (EI 索引)

[14] Fei He, Yaxuan Qi, Yibo Xue, Jun Li. YACA: Yet another cluster-based architecture for network intrusion prevention. Proc. of the IEEE **GLOBECOM**, 2010. (EI 索引)

研究成果

[1] Yaxuan Qi, Jun Li. Method for multi-core processor based packet classification on multiple fields: 美国专利, 申请号: 20100192215, 公开日: 2010 年 7 月 29 日.

[2] 亓亚烜, 李军, 薛一波. 基于 ATCA 的多业务处理系统及方法, 中国专利, 公开号: CN101729413, 公开日: 2010 年 6 月 9 日.

[3] 亓亚烜, 李军. 基于多核处理器的多域网包分类方法: 中国专利, 公开号: CN101478551, 公开日: 2009 年 7 月 8 日.

参与的主要科研项目

[1] 2007~2009: 一体化网络数据深度安全检测与分析的技术与系统(国家 863 目标导向项目，No. 2007AA01Z468). 本人负责系统整体及核心算法设计.

[2] 2009~2010: A Distributed IA-based Intrusion Prevention System (Intel-清华联合研究项目). 本人为项目负责人.

[3] 2010~2011: 基于平滑分析理论的高性能网络深度检测算法研究 (清华信息科学与技术国家实验室学科交叉项目). 本人负责算法设计与实现.