一种无匹配时间损耗的DFA压缩算法的研究与实现

孙明乾,乔庐峰,陈庆华
(陆军工程大学通信工程学院,江苏南京 210000)

摘 要: 高性能深度包检测系统使用确定型有穷自动机DFA(Deterministic Finite Automata)来执行数据包的检测过程.然而,DFA所带来的存储消耗问题使其难以适用于片内资源稀缺的FPGA.目前已存在多种算法着眼于解决DFA的空间爆炸问题,但是其在带来较好压缩率的同时,也在一定程度上影响到了系统的检测速度.本文提出了一种无匹配时间损耗的DFA压缩算法,并在此基础上,基于FPGA硬件平台,设计实现了单个DFA匹配引擎.实验测试结果表明,本文所设计的算法,在未影响整个系统匹配性能的前提下,可以实现10% ~30%左右的压缩率.

关键词: 深度包检测; DFA; 存储压缩; FPGA
中图分类号: TP393.0 **文献标识码:** A **文章编号:** 0372-2112 (2020)06-1132-08
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2020.06.013

Research and Implementation of a DFA Compression Algorithm with No Matching Time Loss

SUN Ming-qian, QIAO Lu-feng, CHEN Qing-hua
(*Institute of Communication Engineering, Army Engineering University of PLA, Nanjing, Jiangsu* 210000, *China*)

Abstract: Start-of-the-art deep packet inspection system uses deterministic finite automata (DFA) algorithms to perform regular expression matching. Nevertheless, the storage consumption problem caused by DFA make it difficult to apply to FPGA with scarce on-chip resources. At present, there are many algorithms aiming at solving the space explosion problem of DFA, but it affects the detection speed of the system to some extent while bringing better compression ratio. In this paper, a DFA compression algorithm without matching time loss is proposed. Based on the hardware platform of FPGA, a single DFA matching engine is designed and implemented. Experimental results show that the algorithm can achieve a compression rate of 10% to 30% without affecting the matching performance of the whole system.

Key words: deep packet inspection; DFA; storage compression; FPGA

1 引言

深度包检测(Deep Packet Inspection, DPI)是网络流量监控与管理领域的一项关键技术,借助于以正则表达式为核心的模式匹配算法[1],DPI可以有效地识别出数据包中所包含的威胁信息及行为.通常,正则表达式需要被转换为NFA或DFA来进行数据包的检测操作.

在当前网络带宽日益增长的背景下,DFA因其$O(1)$的处理速度优势(将m条规则编译成一个FA,DFA匹配速度的时间复杂度为$O(1)$,而NFA为$O(n^2m)$[2])使其成为规则匹配的首选方案,但是其过高的空间复杂度$O(2^{nm})$,是不容忽视的.特别是在大规模规则匹配的应用中,这种现象尤为严重.当前已有许多工作着重于研究DFA的空间压缩,比较经典的算法有D^{2}FA[3]、δFA[4]、mDFA[5]等.

D^{2}FA针对不同状态之间对于相同的输入字符具有同样的跳转目标这一特点,通过在这些特定状态之间引入特定路径,进而可以消除状态间的此类冗余项.但是,压缩后的状态机在单个字符的匹配过程中可能会产生数次跳转,而非一次.对于δFA,在其整个DFA表项中,每个状态仅仅存储与其相邻状态不同的表项,而其余表项直接从其父节点继承而来,由此便可实现相

收稿日期:2019-07-29;修回日期:2019-11-17;责任编辑:梅志强

邻状态间的冗余消除. δFA 在匹配过程中必须引入一个临时转换表(local set),并且需要增加额外的操作来更新全局表项. 如果一个状态存储的有效状态比较多,则更新操作将会降低匹配的效率.

本文基于 FPGA 平台并结合其并行性优势,同时考虑到 DFA 的内在特性,提出了一种新颖的无匹配时间损耗的 DFA 压缩算法,称为 BmDFA(Bitmap DFA). 与传统的 DFA 压缩算法类似[3,4],BmDFA 的关注点在于 DFA 的内在冗余项,特别是单个状态内部的冗余转移边. BmDFA 结合了 Bitmap 算法及流水线技术,在未影响匹配速度的前提下,有效地实现了对原始 DFA 的存储压缩.

2　相关工作

2.1　DFA 的相关概念

正则表达式需要被转换为有限自动机(FA,简称为自动机)才能进行匹配操作. 在模式匹配领域,一个自动机是一个有限的状态集合 Q,包括:1 个初始状态 I(状态 $I \in Q$)和若干个终止状态 F(集合 $F \in Q$). 状态之间的转移用 $\Sigma \cup \{\varepsilon\}$ 中的元素做标记,其中,Σ 表示有限的输入字符集合,即字母表,表示自动机接收的输入字符范围(对于网络流量监测,一般为 ASCII 中的 256 个字符),ε 表示空字符串. 通常用状态转移函数 S 来形式地定义状态转移,对于 $\Sigma \cup \{\varepsilon\}$ 中的每一个字符 α,S 将状态 $q \in Q$ 关联到一个状态的集合 $\{q_1, q_2, \cdots, q_k\}$. 这样,一个自动机完全由 $A = (Q, \Sigma, I, F, S)$ 所定义.

对于 DFA,其状态转移函数 S 可用一个部分函数 $\delta: Q \times \Sigma \to Q$ 来表示,如果 $S(q,\alpha) = \{q'\}$,那么 $\delta(q,\alpha) = q'$. 由此可见,DFA 的状态转移函数 δ 在任意时刻对于输入字符仅返回一个确定的状态,即 DFA 中状态的跳转是唯一确定的,这也是其匹配速度优于 NFA 的原因. 下面以一个简单的正则表达式 ab. c 为例,简要说明上述概念. 整篇论文都使用了相同的例子.

正则表达式 *ab. c* 的 DFA 状态转移图和状态转移表分别如图 1 和表 1 所示.

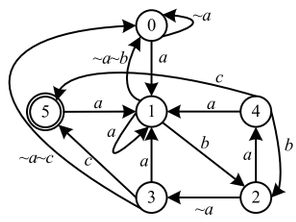

图1　正则表达式*ab.c*的DFA状态转移图

图 1 所示的 DFA 由 6 个状态组成,其内部的跳转关系都已标注在图上. 对于状态 5,使用双圈对其进行标记,这表明状态 5 是一个终止状态. 在整个逐字节读取字符进行状态跳转的匹配过程中,当 DFA 跳转至状态 5 时,表示匹配成功. 图中的符号“ ~ ”表示除其后紧跟的字符之外的所有字母表内的字符. 在这种情况下,图中的一条转移边实际上由多条转移边构成,这是因为在对原始 DFA 进行存储时,在每一状态中,字母表中的每一个字符需要分别与一条转移边对应. 实际上,一个具有 n 个状态 DFA 的主要存储消耗为一个 $n \times \Sigma$ 的二维表,与表 1 类似. 在表 1 中,假定字母表 Σ 为 $\{a, b, c, d, e\}$,存储一个整型变量需要 32bit,则其总共所需的存储资源为 $6 \times 5 \times 32\text{bits} = 960\text{bits}$. 目前,针对 DFA 的存储压缩主要在于减少其所对应的状态转移表的行数与列数.

表 1　正则表达式 *ab. c* 的 DFA 状态转移表

DFA 状态	输入字符				
	a	*b*	*c*	*d*	*e*
0	1	0	0	0	0
1	1	2	0	0	0
2	4	3	3	3	3
3	1	0	5	0	0
4	1	2	5	0	0
5	1	0	0	0	0

2.2　基于 FPGA 实现的 DFA 匹配引擎

传统的基于软件实现的 DFA 匹配引擎已无法适应当今深度包检测的线速要求[6],而 FPGA 作为一个良好的可编程硬件平台,其所具有的并发性及高速性优势,使其成为实现高性能 DFA 匹配引擎的首选. 当前已有许多研究在 FPGA 平台上设计新的算法,从而在一定程度上提升了匹配速度[6,7]或降低了资源消耗[8,9].

在 FPGA 平台上实现的 DFA 匹配引擎可获得较高的吞吐量,但这是以消耗大量的 FPGA 片内资源(查找表(Look Up Table,LUT)和触发器(Flip-flop,FF))为代价的. 文献[8]提出了两级存储的概念,通过筛选出 DFA 中的高频访问状态,存储至片内 RAM,而完整表项存储至片外,从而降低了 FPGA 的片内资源消耗. 但是这种方式的匹配速度会严重的受限于片内和片外存储器的带宽,访问片外存储器将会带来严重的读取延迟. 文献[9]通过引入三态内容寻址存储器(Ternary Content Addressable Memory,TACM),在实现了较好压缩率的同时,也在一定程度上保证了匹配速度. 但是 TCAM 的三个明显的缺点:成本高、功耗大、表项更新复杂,限制了其在 DFA 存储压缩中的进一步使用.

3　压缩算法

BmDFA 是一种适合硬件实现的 DFA 压缩算法,借助于“FPGA + 内存”架构,使用默认转移对 DFA 单个状态内部的冗余转移边进行替换,降低了 DFA 表项在 FP-

GA 片内的存储消耗. 结合 Bitmap 算法对各个状态内部的冗余转移进行记录,提供了一种便捷的 DFA 表项寻址方式,同时采用流水线技术优化字符匹配过程,从系统的角度实现了无匹配时间损耗.

3.1 冗余转移

本文提出的压缩算法 BmDFA,引入了冗余转移(redundant transition)这一概念. 特别地,在本文中,冗余转移可概念性地描述如下:

对于某一特定 DFA 中的任一状态 q,其所包含的所有状态转移函数 $\delta(q,\alpha \mid \alpha \in \Sigma)$ 分别与一特定状态转移边对应, 存在集合 $A_i = \{\alpha_1,\alpha_2,\cdots,\alpha_k\} \subseteq \Sigma$,使得:

$$\delta(q,\alpha_1) = \delta(q,\alpha_2) = \cdots = \delta(q,\alpha_k)$$

对于所有满足上述等式的集合 $A_1,A_2,\cdots,A_n$,存在某个集合 A_i,满足 $\mathrm{card}(A_i) = \max\{\mathrm{card}(A_1),\mathrm{card}(A_2),\cdots,\mathrm{card}(A_n)\}$,其中,$\mathrm{card}(A_i)$ 表示集合 A_i 中元素的个数,max 表示 $\mathrm{card}(A_1),\mathrm{card}(A_2),\cdots,\mathrm{card}(A_n)$ 中的最大值. 由此,所有状态转移函数 $\delta'(q,\alpha \mid \alpha \in A_i)$ 对应的状态转移边,便构成了针对特定状态 q 的冗余转移.

例如,对于表 1 所示的状态转移表来说,在状态 2 中,字符 b、c、d、e 所对应的状态转移边构成了状态 2 的冗余转移,指向状态 3. 字符 b、d、e 所对应的状态转移边构成了状态 3 的冗余转移,指向状态 0. 从上述冗余转移的定义可以看出,冗余转移由多条转移边组成,是多条转移边所构成的一个集合. 在传统的 DFA 存储中,针对此部分的存储消耗为:冗余转移所包含的转移边数量 × 存储单个整型变量的空间消耗. 同样基于先前的假设,存储一个整型变量需要 32bit. 在表 1 中,对于状态 2,其冗余转移的存储消耗为 4 × 32bits = 128bits,对于状态 3,其冗余转移的存储消耗为 3 × 32bits = 96bits.

实际上,对于任意状态的冗余转移来说,其内部所有的转移边构成了前文所提到的单个状态内部的冗余. 在此处,冗余的含义是指:冗余转移内部包含的所有转移边的目的状态相同. 因此,针对每一状态,引入“默认转移边”这一概念,来替代其内部的此类冗余. 在这种情况下,可以使用单条默认转移边来代替冗余转移内部所有的转移边. 对于由 ASCII 字符构成的字母表来说,针对特定规则集,这种方法可以取得较好的压缩效果. BmDFA 算法便是基于上述压缩思路进行设计与实现的.

使用单条默认转移边代替冗余转移,需要针对每一状态过滤出其所对应的冗余转移内的所有转移边,这一过程可通过算法 1 来实现:

算法 1 过滤单个状态内部冗余转移的方法

输入:原始 DFA 表项
输出:每一状态中的冗余转移

```
0   Function Filter_ReduTrans(DFA dfa)
1     For q ∈ Q Do
2       For i ∈ Σ Do
3         NextState[i] ← δ(q,i)
4       End of For
5       将数组 NextStateCount 初始化为 0.
6       将变量 StateNum 初始化为 DFA 中状态总数量.
7       For i ∈ StateNum Do
8         NextStateCount[NextState[i]] + +
9       End of For
10      ReduTransNum[q] ← 0
11      ReduTransNextState[q] ← 0
12      For i ∈ StateNum Do
13        IF NextStateCount[i] > ReduTransNum THEN
14          ReduTransNum[q] ← NextStateCount[i]
15          ReduTransNextState[q] ← i
16        End of IF
17      End of For
18      For i ∈ Σ Do
19        IF δ(q,i) = ReduTransNextState THEN
20          记录当前 i 值.
21        End of IF
22      End of For
23    End of For
24  End of Function
```

3.2 DFA 表项压缩

基于 FPGA + 内存架构实现的 DFA 匹配引擎,其内存所对应的存储器需要存储 DFA 表项信息,以供匹配引擎内部的控制器进行查询操作. 表 1 所示的 DFA 转移表无法直接用于硬件的匹配查找,需要对其进行二次转换,生成一种适合硬件查找的数据结构,具体如图 2 所示.

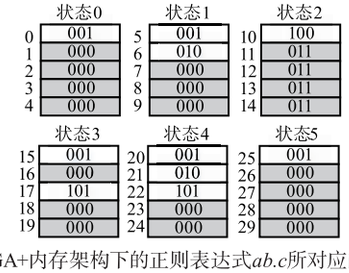

图2 FPGA+内存架构下的正则表达式*ab.c*所对应的DFA表项在存储器中的数据结构

在图 2 中,每一状态由 5 条表项组成,分别与字符 a、b、c、d、e 对应. 表项中记录的是下一跳状态标号. 例如,对于状态 1 中的第二条表项来说,其存储的内容为 010,表示当系统处于状态 1 时,读入字符 b,其下一跳状态是状态 2. 每一表项的左侧分别对应一个十进制数字,表示其在内存中的存储地址. 从图中可以看出,状态 0 至状态 5 中的表项是按照地址顺序依次进行存储的,这样做的目的在于为匹配引擎中的控制器提供一种便捷的寻址方式:内存地址 = 当前状态首表项地址 + 当

前字符偏移量. 在本例中,当前状态首表项地址 = 状态标号 × 每一状态所具有的表项数量,字符 a、b、c、d、e 的当前字符偏移量分别为 0、1、2、3、4. 假设当前状态为 3,当前输入字符为 c,则控制器需要对内存地址为 $3 \times 5 + 2 = 17$ 的表项进行读取,即可得到查询结果:101.

图 2 中使用了“阴影”效果对每一状态中的冗余转移进行标记,针对每一状态,使用单条默认转移边来代替冗余转移内部所有的转移边,可进一步得到如图 3 所示的数据结构. 相对于图 2,图 3 中所示的数据结构将原始的 DFA 表项由 30 条缩减到了 16 条,这是消除了单个状态内部冗余的结果. 同样采用与图 2 中相同的方式,对图 3 中的表项按照地址顺序依次进行存储,将其所对应的内存地址标注在左侧. 使用图 3 中压缩后的 DFA 表项进行数据包检测,需要解决的核心问题是内存寻址.

利用前文提到的公式,进行内存寻址需要获取两个信息:当前状态首表项地址、当前字符偏移量.

对于当前状态首表项地址,需要引入额外的存储空间分别记录每一状态的首表项在压缩后的 DFA 表项中的存储地址,具体如图 4 所示.

图 4 左侧的十进制数字,一方面表示每一表项在内存中的存储地址,另一方面表示相应的状态标号. 在当前状态已知的情况下,只需使用与状态标号相同的地址访问图 4 所对应的存储器,即可得到该状态首表项在图 3 中的内存地址. 对于图 4 中的每一表项,其最高位(虚线方框内所对应的比特)用于指示当前状态是否为终止状态,状态 5 为终止状态,于是使用 1 对其所对应的表项的最高位进行标记.

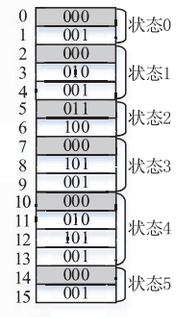

图3　存储压缩后的DFA表项

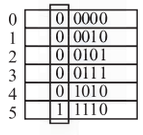

图4　状态首表项地址所对应的存储空间

对于当前字符偏移量,需要使用 Bitmap 技术,对每一状态中的所有表项进行标记,以确定某一字符所对应的表项是否属于冗余转移,以及该表项相对于首表项的位置,即偏移量.

Bitmap 技术是一种典型的数据结构压缩技术,广泛用于海量数据数据的查询和去重,同时在一些高性能网络算法中,也有其存在的身影.

针对图 2 中的 DFA 表项,对于每一状态,进行统计分析之后,可分别生成各自的 Bitmap 表项,具体可通过算法 2 实现.

算法 2　Bitmap 表项的生成

输入:原始 DFA 表项、每一状态中冗余转移所对应的下一跳状态
输出:Bitmap 表项

```
0   Function Bitmap_Table_Gen
    (DFAdfa,ReduTransNextState)
1     将变量 CharNum 初始化为 Σ 所包含字符个数.
2     For q ∈ Q Do
3       For i ∈ Σ Do
4         NextState[i] ← δ(q,i)
5         IF ReduTransNextState[q]
            = NextState[i] THEN
6           Bitmap_Table[i] ←1
7           ELSE
8           Bitmap_Table[i] ←0
9       END of IF
10        IF i = CharNum THEN
11            记录下当前 bitmap 表项的数值.
12          END of IF
13      End of For
14    End of For
15  End of Function
```

进一步地,可得到如图 5 所示的 Bitmap 表项:

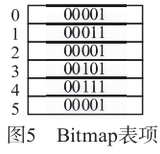

图5　Bitmap表项

图 5 左侧的十进制数字,一方面表示每一表项在内存中的存储地址,另一方面表示相应的状态标号. 在当前状态已知的情况下,只需使用与状态标号相同的地址访问图 5 所对应的存储器,即可得到该状态所对应的 Bitmap 表项. 例如,当前状态为 3,当前输入字符为 c,则通过查询内存地址 3,可得到状态 3 所对应的 Bitmap 表项为 10100,每一位分别与字符 e、d、c、b、a 对应. 在当前输入字符为 c 的情况下,令 bm = 10100,则 bm[0] + bm[1] + bm[2] = 0 + 0 + 1 = 1,即为当前字符偏移量. 结合图 4,对于状态 3,可查询得当前状态首表项地址为 0111,即 7. 利用公式:内存地址 = 当前状态首表项地址 + 当前字符偏移量,可计算得出,在当前状态为 3,当前输入字符为 c,控制器需要访问的内存地址是 7 + 1 = 8.

特别地,对于某一状态中冗余转移所对应的输入字符,通过查找图 5 所示的 Bitmap 表项,可查得相应的比特位为 0,这意味着当前字符偏移量为 0,也就是说,控制器需要访问的内存地址就是当前状态首表项地址. 例如,假设当前状态为 3,当前输入字符为 *d*. 在状态 3 下,当前状态首表项地址为 7,而 bm = 10100,bm[3] =0,所以控制器需要访问的内存地址是 7.

使用图 3 中的数据结构进行深度包检测,每处理单个字符,需要有两次独立的内存访问操作. 第一次内存访问,通过使用当前状态标号作为地址,对图 4 和图 5 中的表项进行查询,进而获得当前状态首表项地址和当前字符偏移量. 第二次内存访问,通过使用第一次内存访问的结果,查询图 3 中的 DFA 表项,进而获得下一跳状态标号. 与直接使用图 2 中的数据结构相比,这种操作方式多了一次内存访问操作,且第二次内存访问需要使用第一次内存访问的结果,这样的话就无法并行执行这两次的内存访问操作. 倘若不采取某种优化方式的话,系统的速度性能便会降低为原始系统的1/2,这也是大多数 DFA 压缩算法的一个共性:压缩效果的获取是以牺牲部分速度性能为代价的.

3.3 流水线优化

本设计使用流水线技术对内存访问过程进行优化,从 DFA 匹配引擎的角度来看,单个时钟周期便可以完成一个字符的处理.

使用片内高速 RAM 分别对对压缩后的 DFA 表项、状态首表项地址及 Bitmap 表项进行存储. 片内 RAM 具有一个时钟周期的读延迟,即发出读命令后,需要等待一个时钟周期,才能从 RAM 的输出数据端读取到所需数据. 由此,针对本设计,设计一种 4 级流水线结构,4 级流水线共用一套存储资源,具体如图 6 所示.

图 6 中使用 *A*1、*W*1、*A*2、*W*2 分别表示首次访存操作、首次访存后的等待、二次访存操作、二次访存后的等待. 实际上,当希望获得最高性能或吞吐率时,应尽量保证所有的电路都持续工作,而不是有的电路处于工作状态,而有的电路处于空闲或等待状态. 在未使用流水线技术时,完成单个字符的处理需要花费 4 个时钟周期. 原因在于当系统进行 *A*1、*W*1、*A*2、*W*2 中的某一项工作时,该部分的电路处于工作状态,而其余三项工作所对应的电路处于空闲状态,这在一定程度上限制了其速度性能的提升.

引入流水线技术后,单个时钟周期内便可以完成一个字符的处理,与传统的基于 FPGA 硬件逻辑实现的 DFA 匹配引擎的处理速度一致,这是各部分电路同时进行工作的结果. 在每一时钟周期内,*W*2 所对应的电路单元都会完成一个字符的处理,从而使得整个 DFA 匹配引擎的处理速度为 $O(1)$.

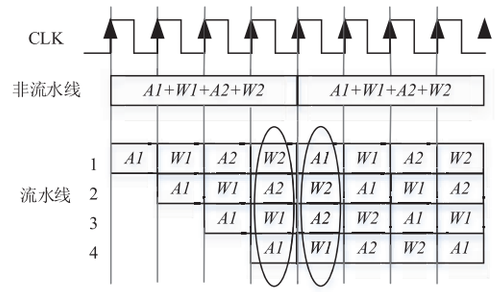

图6 本设计的流水线概念图

4 系统架构

本设计基于 NETFPGA-SUME[10] 平台实现,其上搭载了 Xilinx 公司的 Virtex-7 系列型号为 XC7V690T 的 FPGA 芯片,系统时钟频率为 200Mhz,是一个理想的高性能网络设计的实现平台. 在上述平台,实现了融合有 BmDFA 算法的 DFA 匹配引擎,其系统架构如图 7 所示.

整个系统架构分为两个层级:软件层与硬件层. 软件层主要负责正则表达式的编译,在将其转换为 DFA 之后,使用 BmDFA 算法对其进行压缩,并生成图 3、图 4 和图 5 中的数据结构,通过 CPU 配置,分别存储至图中的 RAM1、RAM2 和 RAM3 中.

在硬件层,使用 Verilog 硬件描述语言对该系统进行设计与实现. 为使本文所设计的匹配引擎具备一定的通用性,在其前级和后级接口,分别设置有针对待匹配数据包和已匹配完成数据包的数据通路(datapath),以及相应的指示信号. 特别地,对于前级接口,提供标准的数据输入接口和引擎状态指示信号,引擎状态指示信号用于指示当前匹配引擎是否可以接收一个新的待匹配数据包,供前级模块读取. 对于后级接口,提供标准的数据输出接口和数据包状态指示信号,数据包状态指示信号用于指示当前输出的数据包是否含有威胁信息及威胁信息在当前数据包中的位置.

对于待匹配数据包,首先对其进行预处理,依据该数据包的长度以及每一级流水线的剩余数据量,将其分配至某一特定流水线 *n*. 然后,按照固定位宽对数据包进行分割,分割后的数据依次写入流水线 *n* 所对应的数据 FIFO 中,同时生成该数据包所对应的指针,指针指示了当前数据包的长度信息.

图 7 所示的四级流水线管理器维护着各级流水线的数据分发操作,其前级面向流水线 1、流水线 2、流水线 3 和流水线 4 所对应的各个数据 FIFO 与指针 FIFO,后级面向状态转移控制单元. 特别地,在整个 DFA 匹配引擎的运转过程中,四级流水线管理器可以准确地控制某一级流水线上数据包检测过程的开始和结束. 状态转移控制单元维护着各级流水线上数据的检测操

作,可同时进行四个数据包的检测过程,同一时钟周期内,同时对 RAM1、RAM2 和 RAM3 进行读取,分别完成针对某一数据包的子操作. DFA 匹配引擎针对单个字符的检测,包含有 4 个子过程,每个子过程内部都未包含复杂的组合逻辑,因而各个子过程的处理延迟(包括门延迟、寄存器延迟、线延迟等)可被限制在单个时钟周期以内,这一点可由相对应的综合工具保证实现.

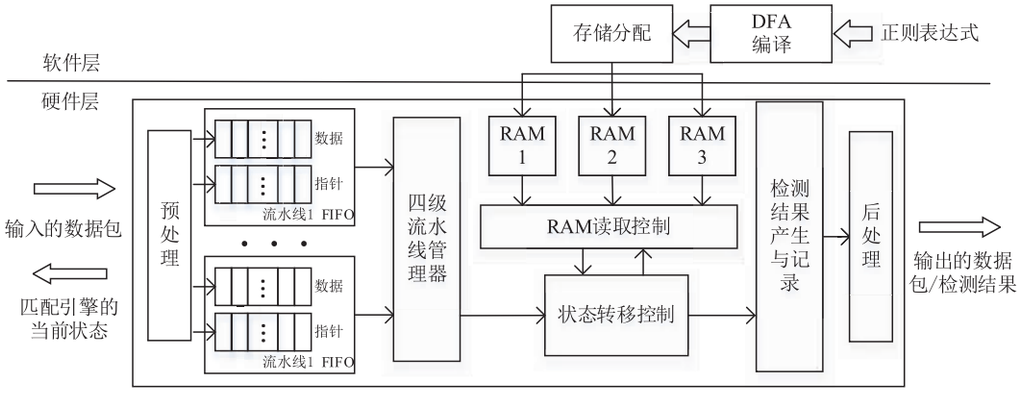

图7　DFA匹配引擎的系统架构

流水线管理器电路的另一个主要功能是实现多级流水线之间的指令调度. 为了保证片内 RAM 在特定的时钟周期内能正确接收到特定流水线读操作所对应的地址,各级流水线读操作所对应的地址需要按照特定顺序送入片内 RAM 的读数据通路,以保证后续过程中各级流水线能够从 RAM 中读取出其所需要的表项内容. 流水线管理器电路内部设置有指令调度器以实现这种功能.

5　性能评估

5.1　压缩率与匹配速度

使用开源入侵检测和防御系统 Snort[11] 中的规则集 snort24、snort31 和 sonrt34,以及网络安全检测软件 Bro[12] 中的规则集 bro217 对本文所设计的压缩算法的压缩率进行分析,具体如表 2 所示.

例化片内 RAM 主要消耗的是 FPGA 内部的 Block-RAM 资源,不同规则集经过软件编译后得到的 DFA,由于其状态数量以及每一状态的冗余转移内部所包含的转移边数量都不相同,所以消耗的 BlockRAM 数也不同,进而导致不同的压缩率. 表 2 中四个规则集的压缩率在 10% ~ 30%左右不等,具体原因将在 5.3 节进行定量分析.

表 2　各规则集的特征及压缩率

规则集	规则数量	DFA 状态数	存储消耗/BRAM 数		压缩率
			原始 DFA	BmDFA	
Snort24	24	8335	811	113	13.9%
Snort31	31	4864	440	55	12.5%
Snort34	34	9754	948	129	13.6%
Bro217	217	6533	590	178	30.2%

从压缩率和匹配速度两个角度,对本文所设计算法 BmDFA 与文献[3,4,13,14]中的算法进行比较,如表 3 所示.

表 3　各算法之间压缩率与匹配速度的比较

	压缩率	匹配速度
初始 DFA	100%	单个时间单位可以处理一个字符
文献[3](D²FA)	5%	至少两个时间单位处理一个字符
文献[13](A-DFA)	<10%	一个时间单位或两个时间单位处理一个字符
文献[4](δFA)	<10%	需引入额外的时间单位进行全局表项的更新操作
文献[14]	10%左右	两到三个时间单位处理一个字符
BmDFA	10% ~30%左右	单个时间单位可以处理一个字符

由于上述算法的实现平台不同,在对其进行匹配速度比较时,需统一评判标准. 在这里,使用处理单个字符需要消耗的时间单位对上述算法进行比较. 由表 3 可以看出,大部分压缩算法都能实现 10%左右的压缩效果,但都在一定程度上影响到了速度性能. BmDFA 牺牲了部分压缩效果(20%左右),由此实现了与原始 DFA 相同的匹配速度.

5.2　预处理时间

预处理时间指的是由原始 DFA 构造压缩后 DFA 的时间复杂度,即压缩算法复杂度. 在很多情况下,要

求系统能够尽快生成压缩后的 DFA,也就是说压缩算法的复杂度应当不宜过高.

表 4 中给出了不同算法的时间复杂度. 对于本文所设计算法,由于只关注单个状态内部的冗余,因此其时间复杂度相对较低.

表 4 各算法之间时间复杂度的比较

	时间复杂度
初始 DFA	$O(0)$
文献[3](D^{2}FA)	$O(n^2\log n)$
文献[12](A-DFA)	$O(n^2)$
文献[4](δFA)	$O(n\times\|\Sigma\|^2)$
文献[13]	$O(n^3\log n)$
BmDFA	$O(n)$

5.3 定量分析

对本文所设计算法的压缩率进行定量分析,有如下等式:

$$r=\frac{256N\rho\lceil\log_2 N\rceil+N(\lceil\log_2 256N\rho\rceil+1)+256N}{256N\lceil\log_2 N\rceil}$$

上式中,r 表示压缩率,N 指 DFA 中的状态总数,ρ 表示整个 DFA 表项中所有非冗余表项占总表项的比值. 对于规则集 snort24、snort31 和 snort34 来说,这个比值在 5% 左右,而对于规则集 bro217,这个比值在 20% 左右.

对于真实的规则集,ρ 的取值一般在 0.05 ~ 0.2 左右,而编译后的 DFA 状态数量也在千单位数量级以上,DFA 表项的字母表 Σ 为 ASCII 字符集,在这种前提下,本算法可以实现 10% ~ 30% 左右压缩率.

6 结语

本文通过对 DFA 表项中单个状态内部的冗余项进行分析,提出了一种适合硬件实现的无匹配时间损耗的 DFA 压缩算法 BmDFA,并通过一个简单的示例详细地讲解了该算法的思想,在此基础上,使用 FPGA 平台设计实现了融合有上述算法的单个 DFA 匹配引擎. 最后,对该算法的存储消耗进行了定量分析,结果表明,本文所设计的算法,在未影响匹配速度的前提下,实现了 10% ~ 30% 左右压缩率.

由此,进一步地,本文所设计的融合有 BmDFA 算法思想的 DFA 匹配引擎可以作为一种轻量级的模块内嵌于高性能路由器或其他网络设备中,使得这些设备能够具备基本的基于应用层过滤的安全防护功能,亦或是将多个匹配引擎集群,从而形成更加强悍的深度包检测系统.

参考文献

[1] 付哲,李军. 高性能正则表达式匹配算法综述[J]. 计算机工程与应用,2018,54(20):1 - 13.
Fu Zhe, Li Jun. Survey on high performance regular expression matching algorithms[J]. Computer Engineering and Applications, 2018, 54(20): 1 - 13. (in Chinese)

[2] Xu C, Chen S, Su J, et al. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms[J]. IEEE Communications Surveys & Tutorials, 2016, 18(4): 2991 - 3029.

[3] Kumar S, Dharmapurikar S, Yu F, et al. Algorithms to accelerate multiple regular expressions matching for deep packet inspection[A]. ACM SIGCOMM 2006 [C]. Pisa, Italy: ACM, 2006. 339 - 350.

[4] Ficara D, Giordano S, Procissi G, et al. An improved DFA for fast regular expression matching[J]. ACM SIGCOMM Computer Communication Review, 2008, 38(5): 29 - 40.

[5] 邵翔宇,刘勤让,谭力波. 基于规则模板的正则表达式分组算法[J]. 电子学报,2016,44(01):236 - 240.
Shao Xiangyu, Liu Qinrang, Tan Libo. A regular expression grouping algorithm based on signature templates [J]. Acta electronica Sinica, 2016, 44(01): 236 - 240. (in Chinese)

[6] Orosz P, Tóthfalusi T, Varga P. FPGA-assisted DPI systems: 100 Gbit/s and beyond[J]. IEEE Communications Surveys & Tutorials, 2018, 21(2): 2015 - 2040.

[7] Matousek D, Matousek J, Korenek J. High-speed regular expression matching with pipelined memory-based automata[A]. FCCM 2018[C]. Boulder, CO, USA: IEEE, 2018. 214 - 214.

[8] Su J, Chen S, Han B, et al. A 60GBps DPI prototype based on memory-centric FPGA [A]. ACM SIGCOMM 2016 [C]. New York, NY, USA: ACM, 2016. 627 - 628.

[9] Liu A X, Torng E, Liu A X, et al. Overlay automata and algorithms for fast and scalable regular expression matching [J]. IEEE/ACM Transactions on Networking (TON), 2016, 24(4): 2400 - 2415.

[10] Zilberman N, Audzevich Y, Covington G A, et al. NetFPGA SUME: Toward 100 Gbps as research commodity [J]. IEEE micro, 2014, 34(5): 32 - 41.

[11] Marty Roesch, Joel Esler, Christine Council, et al. Snort [EB/OL]. https://www.snort.org, 2019.

[12] Vern Paxson. The bro network security monitor [EB/OL]. https://www.bro.org, 2019.

[13] Becchi M, Crowley P. A-dfa: A time-and space-efficient dfa compression algorithm for fast regular expression evaluation [J]. ACM Transactions on Architecture and Code Optimization (TACO), 2013, 10(1): 4.

[14] Becchi M, Cadambi S. Memory-efficient regular expression search using state merging [A]. INFOCOM 2007 [C]. Barcelona, Spain: IEEE, 2007. 1064 - 1072.

作者简介

孙明乾　男,1995 年出生于山东鄄城,现为陆军工程大学通信工程学院硕士研究生. 主要研究方向为网络安全、高性能交换机和路由器相关技术.
E-mail:sunmq1995@163.com

乔庐峰(通讯作者)　男,1971 年出生于河南南乐,硕士生导师,现为陆军工程大学通信工程学院教授. 主要研究方向为通信和计算机网络中关键芯片和电路技术.
E-mail:13357837783@189.cn

陈庆华　男,1976 年出生于湖北红安,现为陆军工程大学通信工程学院副教授. 主要研究方向为交换技术和通信网络.
E-mail:whitenny@126.com