计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220656
61(1): 66−85, 2024

面向申威众核处理器的规则处理优化技术

张振东[1,2] 王 彤[1,2] 刘 鹏[1,2,3]
[1](浙江大学信息与电子工程学院 杭州 310027)
[2](之江实验室智能超算研究中心 杭州 311100)
[3](数学工程与先进计算国家重点实验室 江苏无锡 214125)
(zhendong404@zju.edu.cn)

Rule Processing Optimization Technologies on the Sunway Many-Core Processor

Zhang Zhendong[1,2], Wang Tong[1,2], and Liu Peng[1,2,3]
[1](*College of Information Science and Electronic Engineering*, *Zhejiang University*, *Hangzhou* 310027)
[2](*Research Center for Intelligent Supercomputing*, *Zhejiang Lab*, *Hangzhou* 311100)
[3](*State Key Laboratory of Mathematical Engineering and Advanced Computing*, *Wuxi*, *Jiangsu* 214125)

Abstract High-performance password recovery system is one of the important application scenarios of the Sunway many-core processor. Many popular password recovery systems and tools adopt rule processing as a mainstream password generation method due to its relatively high hit rate compared with the dictionary/mask based password generation methods. However, current researches lack optimization for the rule processing algorithm on the Sunway processor, which makes the rule-based password generation speed become the bottleneck of the password recovery systems. By analyzing the parallelism of the rule processing algorithm at different levels, we propose several optimization techniques for the rule processing on the Sunway processor. For the thread-level optimizations, we explore the optimal scheme to parallelize the rule processing algorithm, which includes the optimal task mapping technique, the optimal local data memory allocation technique, the load balancing technique, and the variable-length rule storage technique. For the data-level optimizations, we analyze the computing patterns of the rule functions and leverage the Sunway SIMD instructions to vectorize the rule functions and reduce the execution time. The experimental results based on the SW26010 processor show that the proposed optimization techniques effectively eliminate the performance bottleneck of rule processing and the rule-based password recovery speed is increased by 30 to 101 times.

Key words Sunway many-core processor; password recovery; rule processing; heterogeneous computing; SIMD

摘 要 高性能口令恢复系统是申威众核处理器的重要应用场景之一,规则处理是主流口令恢复工具中被广泛应用的一种口令生成方式.现有相关研究工作缺少对规则处理算法的优化,导致申威处理器上基于规则的口令生成速度成为口令恢复系统的性能瓶颈.通过分析规则处理算法的多层次可并行性,提出了面向申威众核处理器的线程级、数据级优化方案.在线程级优化方案中,探索了规则处理算法的最优任务映射方式,设计了主从核任务分配机制、从核缓冲区配比优化机制、负载均衡机制、变长规则存储机制等技术以提高并行效率;在数据级优化方案中,分析了规则处理算法中规则函数的计算模式,并通过申威

收稿日期: 2022-07-24; **修回日期**: 2023-05-15
基金项目: 数学工程与先进计算国家重点实验室开放基金项目(2020A11, 2017A07); 之江实验室科研攻关项目(2021PB0AC02)
This work was supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (2020A11, 2017A07) and the Key Research Project of Zhejiang Lab (2021PB0AC02).
通信作者: 刘鹏(liupeng@zju.edu.cn)

SIMD 指令集对规则函数进行向量优化以提高执行效率. 在 SW26010 处理器上的实验结果表明，上述优化方案有效解除了规则处理的性能瓶颈，使规则模式下的口令恢复速度提升了 30~101 倍.

关键词 申威众核处理器；口令恢复；规则处理；异构计算；单指令多数据流

中图法分类号 TP391

随着以申威系列处理器[1]和“神威•太湖之光”[2]为代表的国产处理器和国产超算性能的突飞猛进，越来越多的应用[1, 3]开始利用申威处理器的主核+从核阵列的架构优势实现性能提升. 口令恢复系统因其对信息安全、国防军工的重要意义也成为了申威处理器平台上的热门研究问题.

一般口令恢复系统可以分为口令生成和口令验证 2 个部分，其中口令生成负责产生可能包含正确口令的口令搜索空间，口令验证则负责验证口令搜索空间中的口令的正确性. 当前大多数基于申威处理器的口令恢复系统研究主要关注对口令验证部分的并行优化. 常见的方案是利用主核产生口令，然后将口令验证部分划分并分配给申威处理器的从核阵列，利用多个从核并行执行实现加速. 对于口令生成部分，这些研究均只采用了掩码方式生成口令. 由于掩码方式的生成速度较快，这种方案不会产生口令生成性能瓶颈，但掩码方式的猜测命中率较低，因此不能覆盖实际应用场景.

基于规则的口令生成是另一种主流的口令生成方式，相关研究[4]表明使用规则可以获得比字典和掩码方式更高的猜测命中率，因此规则模式也成为了 Hashcat[5]和 John the Ripper[6]等主流口令恢复工具中流行的口令生成方式. 然而由于规则处理的过程比掩码更加复杂，规则模式下的口令生成速度成为性能瓶颈. 以神威•太湖之光超算使用的 SW26010 处理器为例，其单个主核的规则处理速度约为 3 MPPS（million passwords per second），而 SW26010 处理器单个从核阵列执行 NTLM, MD5 等加密算法时速度可达到 100 MPPS 以上，现有方案下申威处理器的规则模式性能大约只有其理论峰值性能的 1/30.申威处理器平台上规则处理实现方案造成性能瓶颈的原因有 2 个方面：一是现有方案下规则处理算法由主核完成，一个主核要负责生成 64 个从核需要的口令，对于包括 NTLM, MD5, SHA1 在内的众多加密算法，64 个从核每秒消耗的口令数量远大于一个主核每秒生成的口令数量；二是当前大多数加密算法都使用了申威单指令多数据流（single instruction multiple data, SIMD）指令集进行向量优化，而规则处理算法的 SIMD 向量优化则缺少相关研究，因此规则处理算法的速度瓶颈表现得更加突出.

为了突破申威处理器上的规则处理速度瓶颈，进一步提升口令恢复系统的性能，增强申威处理器平台上口令恢复应用的实用性，本文提出了面向申威处理器平台的规则处理算法优化方法，主要内容和创新点包括 4 点：

1）从线程级、数据级 2 个维度分析了规则处理算法的可并行度，针对申威处理器主核执行规则处理算法速度慢的问题，提出了并行任务映射机制，实现了规则处理算法的并行任务分解和从核阵列加速.

2）针对如何高效利用从核局部数据存储器（local data memory, LDM）空间降低口令和规则数据传输开销的问题，提出了从核缓冲区配比优化机制，通过对从核数据通信量、通信时间建模，运用非线性优化理论求解了最优的口令、数据缓冲区容量配比，相比等容量分配策略使数据通信时间和通信量分别减少了 40.5% 和 44.7%；通过变长规则存储机制减缓了规则数据冗余造成的通信开销问题，使通信时间和通信量分别下降了 89.0% 和 89.3%.

3）针对从核间计算量分配不均匀导致的加速比下降问题提出了负载均衡机制，使负载不均匀情况下的规则处理整体执行时间减少了 13.5%.

4）针对现有规则函数实现不能充分利用申威 SIMD 指令集优势的问题，提出了 41 种规则函数的 SIMD 向量优化方案，结合不同类型规则函数的计算访存模式，利用申威 SIMD 指令集提供的特殊指令实现了大小写转换、字符匹配、字符移动等操作的向量优化，将部分规则函数执行的时钟周期降低了 50%.

实验结果表明，应用本文提出的优化方案后，现有口令恢复系统的规则模式口令恢复速度在不同规则集下提升了 30~101 倍.

1 研究背景与相关工作

1.1 申威处理器指令集架构

图 1 展示了神威•太湖之光超算中使用的申威系列 SW26010 处理器的核组架构[7]. 每个核组包含一个

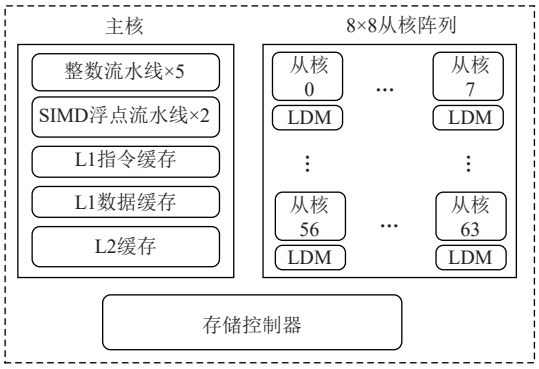


Fig. 1 Architecture of a SW26010 processor core group

图 1 SW26010 处理器核组架构

主核和一个从核阵列, 每个从核阵列由 8 行 8 列共 64 个从核组成. 核组内部, 主核与从核阵列的运行频率均为 1.5 GHz, 每个主核包含 5 条整数流水线和 2 条 SIMD 浮点流水线. 主核的 L1 指令缓存、L1 数据缓存均为 32 KB, L2 缓存大小为 512 KB.从核阵列中每个从核包含 1 条整数流水线和 1 条 SIMD 流水线. 每个从核还配备了 64 KB 的 LDM, 从核访问 LDM 的开销为 4 个时钟周期, 因此 LDM 被广泛用于缓存外部主存中的数据. 在存储器方面, 每个核组可通过本地存储控制器访问 8 GB DDR3 外部存储器. 主核与从核阵列可以离散地访问外部存储器中的数据, 也可以通过直接存储访问(direct memory access, DMA)方式批量访问从而提高访存性能.

指令集方面 SW26010 处理器采用了申威自主设计的 64 b 精简指令集体系架构, 该指令集除了支持常用的标量 8 b, 16 b, 32 b, 64 b 整数运算和单精度、双精度浮点运算外, 还支持 256 b 的 SIMD 向量指令. 利用申威 SIMD 指令集, SW26010 处理器每个从核每个时钟周期最多可以完成 8 次单精度浮点运算或 4 次双精度浮点运算或 8 次 32 b 整数运算. 此外, 申威 SIMD 指令集还支持 256 b 整数运算指令, 此时 SIMD 向量寄存器中的数据被视为 1 个 256 b 的整数, 对该整数可以进行逻辑左移、逻辑右移等操作. 合理利用此类指令可以高效地实现规则处理算法中的部分规则函数.

1.2 规则与规则函数

在口令生成领域, 规则本质上是一种形式化描述语言. 一条规则由 1 个或多个规则函数组成, 这些规则函数可以实现对口令的修改、截取、扩展、复制等操作.表 1 列举了 Hashcat 中部分常用的规则函数, 完整的规则函数列表可见 Hashcat 官网[8]. 每个规则函数都使用一个字符作为其标识符, 例如“u”“T”“*”“r”“^”. 部分规则函数在其标识符后面还跟随了 1 个或 2 个字符组成的参数, 例如“^X”的参数为 *X*, “*NM”的参数为 *N* 和 *M*. 通过分析真实的泄露口令库的特征并将其表达成各种规则函数的组合形式, 基于规则的口令恢复成为了最灵活、准确且高效的口令恢复方式[8].

Table 1 Part of the Frequently Used Rule Functions

表 1 常用的部分规则函数

函数名称	符号表示	功能描述
uppercase	u	大写所有字符
toggle@	TN	切换位置 *N* 处字符的大小写
swap@N	*NM	交换位置 *N* 和 *M* 处的字符
reverse	r	颠倒口令中字符的顺序
prepend	^X	将字符 *X* 加为前缀

1.3 基于规则处理的口令生成过程

在 Hashcat 和 John the Ripper 中, 规则模式均是常用且命中率较高的恢复模式. 如图 2 所示, 在使用规则模式进行口令恢复前, 用户需要分别指定一个字典文件和一个规则文件. 字典文件中存储了用于规则处理的基础口令; 规则文件中存储了以字符串形式表示的规则, 一条规则包含了若干规则函数. 随后, 口令恢复系统依次从字典文件和规则文件中取出 1 条输入口令和 1 条规则, 组合后送入规则变换模块生成待验证口令.

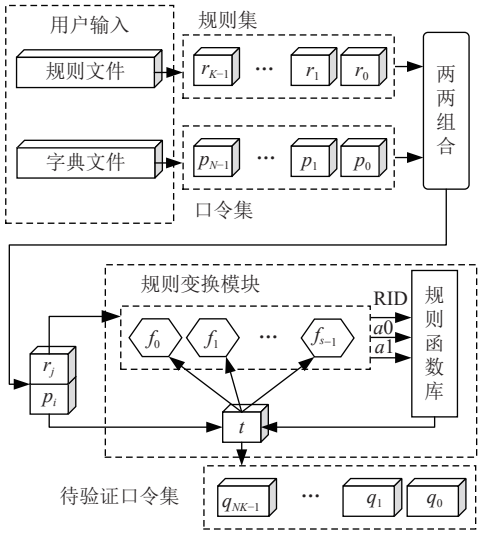


Fig. 2 Rule-based password generation process

图 2 基于规则的口令生成过程

若以集合 $P=\{p_0, p_1, \cdots, p_{N-1}\}$ 表示输入口令集, 以集合 $R=\{r_0, r_1, \cdots, r_{K-1}\}$ 表示规则集, 则规则处理过程可以用算法 1 描述, 其中对每个“口令-规则”组合 (p_i, r_j) 的具体变换过程为: 口令生成系统解析当前

的规则字符串 r_j, 获取其中的规则函数列表 $F=\{f_0, f_1, \cdots, f_{s-1}\}$, 并令输出口令 $t=p_i$, 从第 1 个规则函数 f_0 开始, 依次解析出规则函数的标识符 RID 和 2 个参数 $a0$, $a1$, 然后根据 RID 调用规则函数库中对应的规则函数并传入口令 t 和参数 $a0$, $a1$, 生成新的口令覆盖 t, 之后重复对口令 t 应用剩余规则函数, 最后一个规则函数 f_{s-1} 的输出结果 t 即为基础口令 p_i 和规则 r_j 生成的待验证口令 $q_{i\times K+j}$. 上述过程要求任意一条口令都要和每一条规则组合 1 次, 因此最终生成的待验证口令数量为基础口令数量 N 和规则数量 K 之积. 换言之, 通过规则处理后一个字典被扩充为原来的 K 倍.

为了便于区分, 后文将对规则集和口令集的处理称为“规则处理”, 对“口令-规则”组合的处理称为“规则变换”.

算法 1. 基于规则的口令生成算法.

输入: 输入口令集 P, 规则集 R;

输出: 输出口令集 Q.

① for $i=0\rightarrow N$ do

② for $j=0\rightarrow K-1$ do

③ $q_{i\times K+j}=ApplyRule(p_i, r_j)$;

④ end for

⑤ end for

⑥ $Q\leftarrow\{q_0, q_1, \cdots, q_{N\times K-1}\}$;

⑦ procedure $ApplyRule(p, r)$;

⑧ $t\leftarrow p$;

⑨ $\{f_0, f_1, \cdots, f_{s-1}\}\leftarrow ParseRule(r)$;

⑩ for $s=0\rightarrow S-1$ do

⑪ $\{\text{RID}, a0, a1\}\leftarrow ParseRuleFunc(f_s)$;

⑫ $t\leftarrow ApplyRuleFunc(t, \text{RID}, a0, a1)$;

⑬ end for

⑭ return t;

⑮ end procedure

1.4 国内外相关工作

由于申威系列众核处理器主要应用于国产超级计算机系统, 所以目前基于申威众核处理器的口令恢复系统研究主要集中在国内. 湖南大学陈玥丹[9]提出了一种神威•太湖之光上的 AES 算法的异构并行加速方案, 该方案综合利用 DMA 技术与并行流水技术实现了高效的 AES 算法加解密, 相较于串行 AES 算法减少了 89% 的执行时间; 郑州大学任必晋[1]在太湖之光上实现了 PDF, WinZip, NTLM 这 3 种算法的口令恢复加速, 该方案采取了 MPI+Athread 的 2 级并行实现, 并使用 DMA 加载切片后的口令, 但该方案仅使用了主核生成口令且只实现了掩码模式, 因此存在规则模式的性能瓶颈; 中原工学院董本松等人[3]基于 SW26010 处理器实现了 Office 加密算法的优化方案, 优化技术包括从核并行、DMA 访存优化、SIMD 向量化等, 但该方案中的口令生成部分仍旧为简单的掩码模式; 张恒等人[10]研究了 SW26010 处理器上 MD5 算法的优化技术, 但同样仅使用了掩码模式生成口令. 可以看到, 当前基于申威系列众核处理器的口令恢复研究均未涉及诸如规则处理等更复杂的口令生成方法, 因此现有口令恢复系统在规则模式下的性能难以满足需求, 亟需对规则处理进行优化.

在其他处理器平台方面, Hashcat[5]是目前主流的基于 GPU 的口令恢复工具, 为了解决 CPU 口令生成速度瓶颈问题, Hashcat 在 GPU 上实现了掩码和规则 2 种口令生成算法内核, 使得口令生成和口令验证均在 GPU 核心上完成, 从而避免了大量的口令数据传输并且提高了口令生成速度, 但 Hashcat 的规则函数实现专门针对 GPU, 在申威处理器上的执行效率不高; 谢鑫君等人[11]提出了一种基于 GPU 的对口令进行穷举的技术, 该技术本质上与基于掩码的口令生成类似, 通过 GPU 多线程加速, 将口令生成速度相较于 CPU 提升了 4 个数量级; 郑州大学董婉莹[12]提出了一种基于口令字典树的口令生成方法, 该方法通过统计口令字符的频率构建字典树, 然后基于字典树生成掩码, 最后利用掩码生成口令, 该方法还采用 FPGA 对掩码生成口令的过程进行了加速, 实现了平均 12.85 倍的加速比.Zhang 等人[13]基于 FPGA 提出了一种专用规则处理加速器(RUPA), 解决了 CPU-FPGA 异构口令恢复系统的口令生成瓶颈, 使系统性能提升了 245.4 倍.

在使用规则进行口令生成的研究中, Weir 等人[14]较早地提出使用概率上下文无关文法 (probabilistic context-free grammars, PCFG) 学习公开口令集的分布特征从而生成规则集, 消除了传统规则集设计需要依赖人工专家的弊端; Marechal[15]提出的 rulesfinder 工具和 Kacherginsky[16]提出的 PACK 工具, 能够利用机器学习自动产生规则集; Nam 等人[17]利用对抗生成网络方法产生了 REDPACK 规则集, 使规则处理的猜测命中率提升了至多 26%; Li 等人[18]提出了一种基于密度聚类的规则生成方法, 所生成的规则集的猜测命中率提升了 104%. 经过不断的研究和探索, 目前规则处理已经成为 Hashcat 和 John the Ripper 中最高效的口令恢复模式, 其猜测命中率接近

神经网络的同时，生成速度可达到神经网络的 10 倍以上[19-20].

2 规则处理算法的可并行度分析

2.1 线程级可并行度

线程级并行优化的前提是将计算任务分解为若干相对独立的子任务，通过图 2 可知，在规则处理过程中，一对“口令-规则”组合是其最小的处理单元，且任意“口令-规则”组合的规则变换过程互不干涉，因此可以将一次规则变换作为一个子任务. 尽管一次规则变换中还包括了若干次规则函数的调用，但这些调用之间存在次序关系，因此子任务不能继续细分到规则函数层次. 结合图 2 分析，若口令集 P 和规则集 R 的大小分别为 N 和 K，则规则处理算法的总任务数量为 $N\times K$，假设不同子任务具有相同的运算时间 w，则规则处理算法在串行处理器上的执行时间为 $T_s=N\times K\times w$. 由于规则处理算法的各个子任务互相独立可以完全并行执行，根据 Amdahl 定律[21]，其在并行处理器上的执行时间为 $T_p=N\times K\times w/C$，其中 C 为并行处理器的最大线程数，因此规则处理算法的线程级可并行度为 $T_s/T_p=C$，在 SW26010 处理器单个核组上的最大加速比为 64.

2.2 数据级可并行度

规则处理算法经过分解成为规则变换子任务后，每个处理器核心需要依次完成属于自己的子任务. 在单个处理器核心中，规则变换子任务之间已经无法并行. 而规则变换本质上是对口令应用规则函数，因此规则函数是规则变换的核心与关键，规则函数的性能也决定了规则变换的性能. 为了提升规则变换子任务的执行效率，还需要探索规则函数的数据级可并行度. 在 Hashcat 和 John the Ripper 中常用的规则函数有 41 种，这些规则函数数量众多且类型繁杂，有必要针对其计算特点进行分类讨论.

根据 1.3 节的分析，规则函数的输入为口令 p 和 2 个参数 $a0$, $a1$，输出为口令 q，其 C 语言应用程序编程接口(application programming interface, API)如图 3 所示. 在现有实现中，输入口令 p 通过长度为 N 的字符数组 $p[N]$ 储存，输出口令通过字符数组 $q[N]$ 存储，N 的取值决定了口令恢复系统能够处理的最大口令长度，通常取 N=32.此外，口令恢复系统从字典文件中加载口令时会单独计算出口令的长度 L，和数组 $p[N]$ 一起存放于口令结构体 *pwd_t* 中，为了提高口令的传输效率，口令结构体中定义了额外的 3 个占位

```
#define N 32
typedef unsigned char u8;
typedef struct { /*口令结构体*/
        u8 p[N];
        u8 L;
        u8 pos0, pos1, pos2;  /*占位符*/
}  pwd_t;
u8 rule_function_xxx(u8 p[N], u8 q[N],
    u8 a0, u8 a1, u8 L) {
        {...}       /*对p[N]进行处理*/
        return L_new;  /*返回新生成的口令长度*/
};
```

Fig. 3 The unified C-language API of rule functions

图 3 规则函数的 C 语言统一应用程序编程接口

符以保证其内存占用为 4 B 的整数倍. 参数 $a0$, $a1$ 主要为部分带有可变参数的规则函数提供额外的信息，根据规则函数的不同，其意义也会产生变化，部分规则函数没有可变参数，此时 $a0$, $a1$ 的取值对函数执行结果无影响.

根据函数内部对字符数组 $p[N]$ 的计算模式，规则函数可分为 5 类：

1) 全字符遍历变换

全字符遍历变换(operation on all characters, OAC)函数的计算模式伪代码如图 4 所示.OAC 函数中输出口令与输入口令长度相同，输出字符 $q[i]$ 是与其位置相同的输入字符 $p[i]$, $a0$ 和 $a1$ 的函数 $ROP(p[i], a0, a1)$，这里函数 *ROP* 泛指包括大小写转换、数值加减、移位等操作在内的各种变换函数.OAC 类型函数包括 *lowercase*, *uppercase*, *toggle* 等.

```
u8 L_new = L;
for(int i = 0; i< L_new; i++) {
    q[i] = ROP(p[i], a0, a1);
}
```

Fig. 4 The computing pattern of OAC-type functions

图 4 OAC 类型函数的计算模式

2) 指定字符变换

指定字符变换(operation on specified characters, OSC)函数的计算模式伪代码如图 5 所示. 与 OAC 函数类似，OSC 函数的输出口令长度与输入口令长度相同，但只有输出字符 $q[a0]$ 是输入字符 $p[a0]$ 的 *ROP*

```
u8 L_new = L;
for(int i = 0; i < L_new; i++) {
    if(i == a0) q[i] = ROP(p[i], a1);
    else q[i] = p[i];
}
```

Fig. 5 The computing pattern of OSC-type functions

图 5 OSC 类型函数的计算模式

变换，其余位置的输出字符与输入字符相同.OSC 类型函数包括 *overwrite*, *toggle*@等.

3）全字符遍历移动

全字符遍历移动（move across all chacracters, MAC）函数的计算模式伪代码如图 6 所示.MAC 函数的输出口令长度 *L_new* 与输入口令长度 *L* 不一定相同，*L_new* 的值由 *L*, *a*0, *a*1 共同决定，每一个输出字符 *q*[*i*] 由某个输入字符 *p*[*idx*] 直接移动得到，*p*[*idx*] 的位置坐标 *idx* 通过 *CalcIndex*(*i*, *L*)计算得到.MAC 函数包括 *reverse*, *duplicate* 等.

```
u8 L_new = CalcLength(L, a0, a1);
for(int i = 0;  i< L_new;  i++) {
    u8 idx = CalcIndex(i, L);
    q[i] = p[idx];
}
```

Fig. 6　The computing pattern of MAC-type functions

图 6　MAC 类型函数的计算模式

4）指定字符移动

指定字符移动（move across specified characters, MSC）函数的计算模式伪代码如图 7 所示. MSC 函数的输出口令长度 *L_new* 同样取决于 *L*, *a*0, *a*1，但只有输出字符 *q*[*a*0] 和 *q*[*a*1] 由其他位置的输入字符 *p*[*idx*] 直接移动得到，其余位置的输出字符与输入字符相同. MSC 函数包括 *swap_front*, *swap_back*, *replace_N*+1 等.

```
u8 L_new = CalcLength(L, a0, a1);
for(int i= 0;  i< L_new;  i++) {
      if(i == a0 || i == a1) {
          u8 idx= CalcIndex(i, L);
              q[i] = p[idx];
      }
      else q[i] = p[i];
}
```

Fig. 7　The computing pattern of MSC-type functions

图 7　MSC 类型函数的计算模式

5）指定字符清除

指定字符清除函数 *purge* 是单独成类的一个规则函数，其计算模式与前面 4 种函数有较大差异，如图 8 所示. 函数 *purge* 的功能是去除口令 *p*[*N*] 中 ASCII 值等于 *a*0 的所有字符，留下其余的字符作为输出字符. 这种情况下，尽管输出字符 *q*[*i*] 仍由某个输入字符 *p*[*idx*] 直接移动得到，但 *p*[*idx*] 的位置坐标 *idx* 无法通过 *i* 和 *L* 直接计算，而是与输入字符的 ASCII 值 *p*[0], *p*[1], …, *p*[*N*−1] 有关，因此必须依次读取并比较所有输入字符的值才能最终确定输出字符.

```
u8 L_new = 0;
for(int i= 0;   i< L; i++) {
    if(p[i] ≠ a0) {
        q[L_new] = p[i];
        L_new++;
    }
}
```

Fig. 8　The computing pattern of the *purge* function

图 8　函数 *purge* 的计算模式

上述 5 类规则函数的主体部分均为一个 for 循环，循环次数为输入口令长度 *L* 或输出口令长度 *L_new*，除函数 *purge* 外，OAC, OSC, MAC, MSC 函数中的 for 循环体之间均无依赖关系，因此各循环体理论上可以完全并行执行，其数据级可并行度为 *L* 或 *L_new*. 然而现有规则函数实现方案无法发挥 SW26010 处理器从核核心的数据级并行能力，这是因为每个从核核心只具有 1 条整数流水线和 1 条 256 b 的 SIMD 流水线，现有规则函数实现方案只能利用其中的整数流水线达成每条指令处理 4 个字符，而无法利用 256 b 的 SIMD 流水线.

3　线程级并行优化技术

3.1　主从核任务分配机制

SW26010 处理器的一个核组主要由 1 个主核与 64 个从核构成，其中主核适合复杂逻辑运算和 I/O 操作，从核适合相对简单但可并行的运算，主核、从核需要合理的分工才能发挥各自的性能优势. 图 9 展示了本文提出的规则处理系统的主从核任务分配机制，图中的处理器系统被简化为主核、从核阵列和主存 3 个部分. 其中，主核负责解析输入参数，根据输入参数从硬盘中读取字典文件和规则文件，并对字典文件和规则文件进行预处理，生成口令数组、规则数组

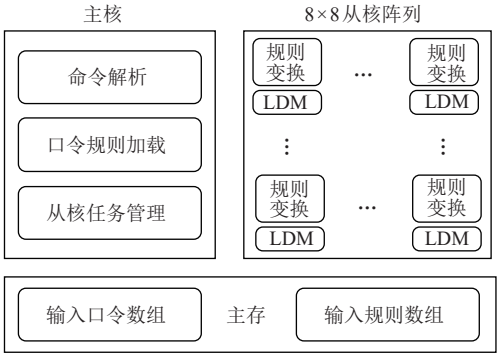


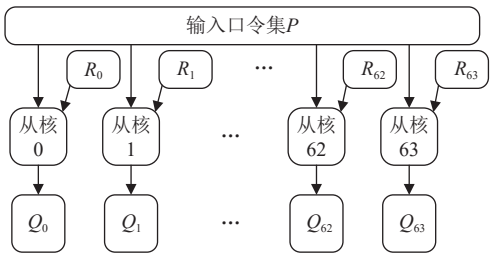
Fig. 9　Task assignment policy of the master core and the slave core

图 9　主从核任务分配机制

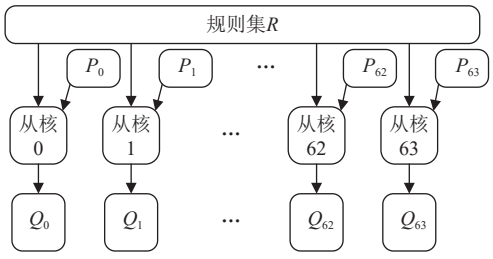
以及口令数量、规则数量等参数,然后启动从核阵列进行规则变换;从核阵列主要负责处理规则变换子任务,根据主核生成的参数,每个从核找到自己对应的子任务集然后执行规则变换,规则变换生成的输出口令集存储于LDM中,供后续口令验证任务使用.

3.2 子任务映射机制

在线程级并行优化中,如何将规则变换子任务映射到众核处理器的各个从核是首先要解决的问题.根据2.1节所述,大小分别为N和K的口令集和规则集组合成的子任务集大小为$N\times K$,此处暂时假定所有子任务的运算量相同,按照规则处理的流程,需要将子任务集划分为64等份分配给64个从核,其中每份的子任务数量为$N\times K/64$.为实现子任务集的划分,一般有共用口令映射(share passwords mapping, SPM)和共用规则映射(share rules mapping, SRM)两种方案,如图10所示.在共用口令映射方案中,规则集R被均匀划分为64个子集$R_0, R_1, \cdots, R_{63}$,分别分配给从核0、从核1、…、从核63,而口令集则不经划分直接分配给所有从核.与之相反,共用规则映射方案中,口令集被均匀划分并分配给64个从核,规则集则不划分.



(a) 共用口令映射



(b) 共用规则映射

Fig. 10 Sub-task mapping scheme of slave cores

图10 从核子任务映射方案

在规则处理的过程中,由于从核的局存不能存储完整的口令集和规则集,因此必须以分块的方式进行加载.口令和规则的加载也有2种方案:一种是口令外循环-规则内循环(password-out-rule-in, PORI),即在外部循环中读取口令分块,在内部循环中读取规则分块;另一种则是口令内循环-规则外循环(password-in-rule-out, PIRO).任务映射方式和数据加载方式两两组合能够得到4种方案,分别为SRM-PORI, SRM-PIRO, SPM-PORI, SPM-PIRO.

任务映射方式和口令规则加载方式主要影响了从核阵列整体与主存间的数据通信量,若每条口令数据大小为u字节,每个口令块包含x条口令,每条规则数据大小为v字节,每个规则块包含y条规则,则以上4种方案下核组的总数据通信量可由式(1)近似计算:

$$\begin{cases} W_{\text{SRM-PORI}} \approx u\times N + v\times \dfrac{KN}{x}, \\ W_{\text{SPM-PORI}} \approx u\times 64N + v\times \dfrac{KN}{x}, \\ W_{\text{SRM-PIRO}} \approx v\times 64K + u\times \dfrac{KN}{y}, \\ W_{\text{SPM-PIRO}} \approx v\times K + u\times \dfrac{KN}{y}, \end{cases} \tag{1}$$

由式(1)可以看出

$$\begin{cases} W_{\text{SRM-PORI}} < W_{\text{SPM-PORI}}, \\ W_{\text{SPM-PIRO}} < W_{\text{SRM-PIRO}}. \end{cases} \tag{2}$$

因此在PORI方案下使用SRM映射的数据通信量更小,在PIRO方案下则应该使用SPM机制,而$W_{\text{SRM-PORI}}$与$W_{\text{SPM-PIRO}}$的大小关系则与N, K, u, v, x, y的具体取值有关,后文若无特别说明,默认基于SRM-PORI方案展开讨论.

3.3 从核缓冲区配比优化机制

SW26010处理器中每个从核具有64 KB的LDM,利用好局部缓存提升从核的数据访问效率同样是并行优化的关键.在规则处理算法中,输入口令集P、规则集R,以及生成的输出口令集Q都需要一定的存储空间.通常情况下,上述3个集合占用的空间均远大于从核的局部缓存空间,因此必须对输入口令集P、规则集R、输出口令集Q进行分块读取和写入,这里不妨将它们对应的局部缓冲区称为P缓冲、R缓冲和Q缓冲,优化P, R, Q这3种缓冲区的容量配置能够减少从核对主存的数据访问量,提升规则处理访存性能和降低访存能量消耗.

为了分析有限LDM空间下P, R, Q这3种缓冲区容量的最佳配比,首先需要介绍本文提出的从核规则处理任务执行机制,如算法2所示.首先每个从核使用系统调用函数*GetSlaveCoreID*获得自己的从核ID号cid($cid=0, 1, \cdots, 63$),然后根据cid和任务映射机制M计算从核需要处理的口令集和规则集的起始位置和大小.共用口令映射时口令集的起始位置$p_idx=0$,口令集大小$n=N$,规则集的起始位置$r_idx=K\times cid/64$,规则集大小$k=K/64$;共用规则映射时$p_idx=N\times cid/64$,$n=N/64$,$r\_idx=0$,$k=K$.确定从核需要处理的口令集和规则集后,从核进入一个双

重循环，在外循环（口令循环）中从核每次加载 x 条口令到 P 缓冲中，然后执行内循环（规则循环），在内循环中，从核每次加载 y 条规则到 R 缓冲中，然后对 P 缓冲和 R 缓冲中的口令子集和规则子集进行处理，生成的输出口令子集存储于 Q 缓冲中，用于后续口令验证过程，不写回主存.

算法 2. 从核规则处理任务执行机制.

输入：输入口令集 P，规则集 R，映射机制 M；

输出：是否破解成功 *cracked*.

① $cid \leftarrow GetSlaveCoreID()$；
② $p\_idx, n \leftarrow GetMyPasswordSet(M, cid)$；
③ $r\_idx, k \leftarrow GetMyRuleSet(M, cid)$；
④ $i \leftarrow 0$；
⑤ while $i < n$ do
⑥ $Pbuf \leftarrow DMALoadPassword(P, p\_idx+i, x)$；
⑦ $j \leftarrow 0$；
⑧ while $j < k$ do
⑨ $Rbuf \leftarrow DMALoadRule(R, r\_idx+j, y)$；
⑩ $cracked \leftarrow RunRuleProcess(Pbuf, Rbuf)$；
⑪ $j \leftarrow j + y$；
⑫ end while
⑬ $i \leftarrow i + x$；
⑭ end while
⑮ return *cracked*；
⑯ procedure $RunRuleProcess(Pbuf, Rbuf)$
⑰ $cracked \leftarrow 0$；
⑱ $s \leftarrow 0$；
⑲ for $i = 1 \rightarrow x$ do
⑳ for $j = 1 \rightarrow y$ do
㉑ $Qbuf[s] \leftarrow ApplyRule(Pbuf[i], Rbuf[j])$；
㉒ $s \leftarrow s + 1$；
㉓ if $s = z$ then
㉔ $cracked \leftarrow PasswordValidate(Qbuf)$；
㉕ $s \leftarrow 0$；
㉖ end if
㉗ end for
㉘ end for
㉙ if $s > 0$ then
㉚ $cracked \leftarrow PasswordValidate(Qbuf)$；
㉛ end if
㉜ return *cracked*；
㉝ end procedure

下面首先考虑算法 2 中单个从核与主存间的数据通信量和通信时间. 设每条口令占用的存储空间为 u 字节，每条规则占用的存储空间为 v 字节，于是口令循环的循环次数为 $\lceil n/x \rceil$，每次循环读取的口令数据量为 ux；规则循环的循环次数为 $\lceil k/y \rceil$，每次循环读取的规则数据量为 vy；因此算法 2 中口令数据总读取量为 $W_{\mathrm{P}} = ux \times \lceil n/x \rceil$、规则数据总读取量为 $W_{\mathrm{R}} = vy \times \lceil k/y \rceil \times \lceil n/x \rceil$. 由于输出口令集 Q 不会被写回主存，Q 缓冲的大小不影响数据通信量和通信时间，所以参数 z 未出现在 W_{P} 和 W_{R} 的表达式中. 尽管 W_{P} 和 W_{R} 的表达式中所有变量均为自然数，为了简化分析，后文中将其取值范围定为 $[1, +\infty)$，即不小于 1 的实数.

在算法 2 中，从核使用 DMA 技术与主存进行数据传输，DMA 读写数据的开销可以分为两部分：一部分是 DMA 启动的固定开销 D；另一部分是随传输数据量线性增加的数据传输开销 g. 因此算法 2 中数据传输的总时间 T 可以用式（3）计算. 由于大多数情况下有 $n \gg x$ 和 $k \gg y$，并且程序会自动调整最后一轮循环加载的数据量，因此式（3）中的向上取整符号可以近似忽略.

$$\begin{aligned} T =& g \times (W_{\mathrm{P}} + W_{\mathrm{R}}) + D \times \left(\left\lceil \frac{n}{x} \right\rceil + \left\lceil \frac{k}{y} \right\rceil \times \left\lceil \frac{n}{x} \right\rceil \right) \approx \\ & g \times \left(un + \frac{vkn}{x} \right) + D \times \left(\frac{n}{x} + \frac{kn}{xy} \right) = \\ & n \times \left(gu + \frac{kgv}{x} + \frac{D}{x} + \frac{Dk}{xy} \right). \end{aligned} \tag{3}$$

根据需要分析的参数不同，式（3）有不同的诠释方法，显而易见的有：

1）口令集和规则集越大，则 n 和 k 越大，通信时间 T 越大；

2）DMA 的传输速率越大、固定开销越小，则 g 和 D 越小，通信时间 T 越小；

3）单条口令和规则的存储空间越小，则 u 和 v 越小，通信时间 T 越小.

当研究给定口令集和规则集下的最佳 P 缓冲和 R 缓冲配比（即 x 和 y 的取值）时，式（3）中 u, v, k, n, g, D 可以视为常数，并且 P 缓冲、R 缓冲、Q 缓冲的总容量大小受从核 LDM 大小限制，x、y 还应满足 $ux + vy \leqslant S$，其中 S 为除去 Q 缓冲外可用于缓冲的 LDM 空间大小. 因此数据通信时间 T 的最小值计算可转化为一个非线性优化问题，如式（4）所示. 对式（4）求解可得式（5）.

$$\begin{cases} \min\limits_{x,y} F(x,y) = \dfrac{kgv}{x} + \dfrac{D}{x} + \dfrac{Dk}{xy}, \\ \text{s.t.:} \\ ux + xy \leqslant S, \\ u, v, x, y \in \mathbb{R} \cap [1, +\infty), \end{cases} \tag{4}$$

$$\begin{cases} x^* = \dfrac{ac+b-\sqrt{b^2+abc}}{a} = \\ \quad c+\dfrac{b}{a}-\sqrt{\left(\dfrac{b}{a}\right)^2+c\times\dfrac{b}{a}}, \\ y^* = \dfrac{S}{v}-\dfrac{u}{v}\times\dfrac{ac+b-\sqrt{b^2+abc}}{a}, \\ a = gvk+D, \\ b = \dfrac{Dkv}{u}, \\ c = \dfrac{S}{u}. \end{cases} \tag{5}$$

通过实际测试 SW26010 中从核 DMA 的性能, 测得式(5)中 $g \approx 2.069\times10^{-9}$ s/B, $D \approx 5.172\times10^{-7}$ s. 此外, 通常单条规则数据大小$4 \leqslant v \leqslant 128$, 规则数量 $k > 100$, 据此可计算出式(5)中

$$\frac{b}{a} = u\times\left(\frac{g}{D}+\frac{1}{kv}\right) \approx u\times\left(\frac{1}{40}+\frac{1}{kv}\right) \approx \frac{u}{40}, \tag{6}$$

由于 $1/kv \ll 1/40$, 该项可忽略不计, 因此式(5)中 x^*和 y^*几乎不随 k 的取值发生变化. 换言之, P 缓冲、R 缓冲大小的最佳配比与口令集大小 n 和规则集大小 k 均无关. 根据这一结论, 设计程序时可以结合实际可用的 LDM 空间计算最佳的 P 缓冲、R 缓冲大小, 从而减少从核阵列与主存的通信时间和通信量.

3.4 口令规则数据结构优化技术

式(3)揭示了从核数据通信时间 T 与口令数据大小 u 和规则数据大小 v 正相关, 因此减小口令数据大小和规则数据大小能够降低从核数据通信开销. 在现有实现中, 每条口令采用 36 B 存储, 每条规则采用 128 B 存储, 其 C 语言结构体如图 11 所示. 其中每个规则结构体 *rule_t* 由一个数组 *rfs*[32] 构成, 数组 *rfs*[32] 中的每个元素为一个规则函数结构体, 每个规则函数结构体包括了函数名称(*name*)、参数 0(*a*0)、参数 1(*a*1)和占位符(*pos*), 每个规则结构体可存储的规则函数最多为 32 个.

图 11 中口令和规则的结构体的定义方式不能充分利用从核的 LDM 空间. 首先口令结构体为了保持

```
#define N 32
typedef struct {
    u8 p[N], L, pos0, pos1, pos2;
} pwd_t;
typedef struct {
    u8 name, a0, a1, pos;
} rf_t;
typedef struct {
    rf_t rfs[32];
} rule_t;
```

Fig. 11 Data structure of the password and the rule

图 11 口令与规则的数据结构体

4 B 对齐以提高传输效率, 浪费了 3 B 的空间作为占位符, 对此, 本文将口令结构体中的最大口令长度 N 设置为 31, 并去除了占位符, 从而在保持 4 B 对齐的情况下将口令结构体缩小为 32 B, 考虑到绝大部分正确口令长度小于 16 B, 将最大口令长度减少 1 B 的影响可以忽略不计.

另一方面, 规则结构体中固定包含 32 个规则函数结构体, 然而实际使用的规则中包含的规则函数数量平均为 3 个左右, 因此按照图 11 中的定义方式平均浪费了 90.6% 的空间. 为了解决这一问题, 本文提出了变长规则存储机制, 如图 12 所示展示了 2 条相邻的规则“uT2”和“Cp2$AsAB”, 其中规则“uT2”由规则函数“u”和“T2”组成, 规则“Cp2$AsAB”由规则函数“C”“p2”“$A”“sAB”组成. 注意到每个规则函数使用 4B 存储, 在原定义方式下第 4 个字节为占位符, 而在变长规则存储机制中, 第 4 个字符为规则结束指示符. 当指示符为 0 时, 表示当前规则后面还有其他规则函数, 当指示符为 1 时表示当前规则结束. 通过变长规则存储机制, 图 12 中 2 条规则需要的存储空间由 256 B 降至 24 B.

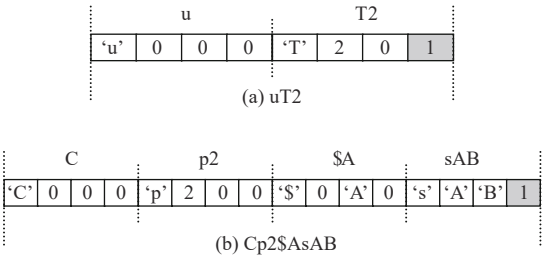


Fig. 12 Variable length rule storage policy

图 12 变长规则存储机制

3.5 从核间负载均衡机制

3.1~3.4 节的分析假定了规则处理算法的各个子任务具有相同的运算复杂度, 然而在实际中, 不同子任务的运算复杂度往往是不同的, 差异来源主要有 3 个方面: 一是不同规则包含不同数量的规则函数, 一般而言规则函数越多运算复杂度越大; 二是不同规则函数的运算复杂度不同, 例如函数 *append*(“$X”)只需在口令最后添加一个字符 *X*, 而函数 *lowercase* 则需要遍历口令中的所有字符, 判断其大小写状态并对其修改; 三是同一规则函数在处理不同口令时运算复杂度不同, 例如函数 *lowercase* 在处理 8 B 长度的口令时需要至少 8 次运算, 而处理 16 B 长度的口令时需要至少 16 次运算.

在当前规则处理方案中, 子任务在从核间按照数量均匀分配, 由于并行处理器每个核心的运算能

力相同，若各个核心分配到的子任务总运算复杂度差异较大则会导致部分核心提前于其他核心完成任务，从而处于空闲状态，浪费了算力导致加速比下降. 对此一种朴素的解决方案是提前计算出各个子任务的执行时间，进而依照子任务的执行时间在核心间均匀分配，但是口令集和规则集的随机性以及计算子任务时间的开销使得这种方案不可行.

为了解决这一问题，本文设计一种从核间负载均衡机制，其原理如图 13 所示. 主核从字典文件中加载口令时计算出输入口令的总数量，并将输入口令集分为大小均为 m 的子集，然后在主存中设置一个信号量 s 记录待处理的子集总数量. 处理开始后，每个从核依次读取主存中的信号量 s，并将其减 1 然后更新 s. 这一过程采用原子操作进行，避免多个从核同时更新 s 而发生错误. 之后根据 s 的值计算需要处理的口令的起始地址并对其执行规则处理.

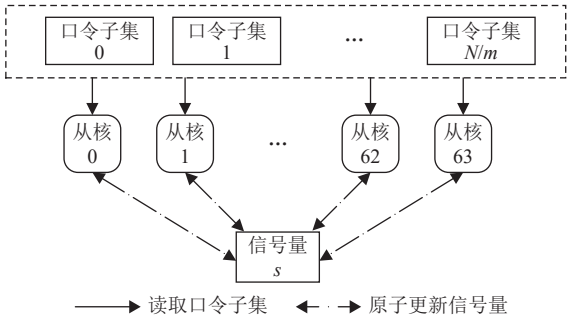


Fig. 13 Workload balance policy between slave cores

图 13 从核负载均衡机制

图 13 所述的负载均衡机制可以使不同从核的处理时间的极差不超过 m 条口令的处理时间. 因此若要尽可能减小不同从核的处理时间差异，则需使 m 的取值尽可能小. 但是当 m 很小时，从核会频繁地对信号量 s 进行原子更新操作，考虑到从核对主存的访问开销，整体的规则处理时间反而会增大. 实践中应综合考虑上述因素的影响选取最佳的参数设置.

4 数据级并行优化技术

数据级并行优化技术主要针对单次规则函数的执行过程，2.2 节以字符为单位分析了各类规则函数的数据级可并行度，对绝大部分规则函数，理想情况下每个字符的处理都可以完全并行，然而以字符为单位进行处理意味着从核中每条指令最多处理 1 个字符，而 SW26010 中从核最多可以同时执行 2 条指令，因此以字符为单位的处理方式在从核上的执行时可并行度最多为 2.尽管在 Hashcat 中对规则函数进行了优化，将 4 个字符当作一个 32 b 的无符号整型数据进行了处理，使得部分规则函数的性能得到了提升，但这种优化仍然不能充分发挥 SW26010 处理器从核的性能. 一方面每条指令最多只能处理 4 个字符，不能充分利用从核中向量寄存器 256 b 的位宽；另一方面部分规则函数的实现未能高效利用申威 SIMD 指令集中的特殊指令，无法达到最佳优化效果.

为了进一步提升规则函数的执行效率，本文提出了基于申威 SIMD 指令集的规则函数优化方案，该方案的核心是通过合理使用 SIMD 指令，降低执行规则函数需要的指令数量，进而降低规则函数的执行时间. 使用 SIMD 指令实现规则函数的基本思路是将输入口令存储于 SIMD 向量寄存器中，利用 SIMD 指令对口令中的多个字符同时进行相同的操作，最后将 SIMD 向量寄存器中的输出口令写回主存. 根据 2.2 节中的分析，现有的 41 种规则函数可以分为 OAC, OSC, MAC, MSC 这 4 种类型和函数 *purge*. 除单独的规则函数 *purge* 较为特殊外，其余 4 类规则函数均有明显的计算访存特征，因此相应的 SIMD 实现也围绕其计算访存特征展开.

1）对于 OAC 类型规则函数，其特点为对所有输入口令字符的操作完全相同且互不依赖. 互不依赖的特性决定了对每个字符的操作可以完全并行，对所有输入口令字符的操作完全相同决定了这些操作可以用完全相同的指令完成，这一特点非常契合 SIMD 指令的设计思路，所以对 OAC 类型函数直接使用 SIMD 指令将其并行化即可得到很好的加速效果.

2）对于 OSC 类型规则函数，其特点在于只对特定位置的 1 个字符采取操作，其余位置字符保持不变. 由于申威 SIMD 指令集不支持单独对 SIMD 向量中的 1B 进行操作，所以 OSC 类函数必须结合掩码实现，即首先生成一个待处理字节位置全为“1”、其余位置全为“0”的掩码，然后对原口令向量中所有字符进行相同操作生成新口令向量，最后根据掩码组合的原口令向量与新口令向量得到输出口令向量. 组合方式为：掩码为“1”的位置使用新口令向量的值，掩码为“0”的位置使用原口令向量的值.

3）对于 MAC 类型规则函数，其特点为对所有字符采用特定的方式进行位置变换. 在以字符为单位的处理方式中，对每个字符直接进行位置变换都会产生 1 次读操作和 1 次写操作，要生成长度为 L 的输出口令则会产生 $2L$ 次读/写操作，带来可观的时间开销. 要提升此类函数的性能就必须尽量减小读/写操

作的数量，申威 SIMD 指令集中提供了向量移位指令和向量混洗指令，利用这 2 类指令可以在 SIMD 向量中同时变换多个字符的位置，并且无需读写存储器.

4）对于 MSC 类型规则函数，其特点为对特定的 1 个或 2 个字符进行位置变换. 由于申威 SIMD 指令集不支持对 SIMD 向量中的单个字节进行移动，所以要实现 MSC 类型函数同样需要结合掩码，即首先生成目标字节位置处的掩码，然后对原口令向量中的所有字符进行移动生成新口令向量，最后通过掩码组合原口令向量与新口令向量得到输出口令向量.

5）对于函数 *purge*，由于对其各个字符的操作之间存在数据依赖，因此难以使用 SIMD 指令实现，本文仍旧采用普通标量指令集对其进行实现.

在上述 SIMD 优化思路的基础上，本文提出了 4 种 SIMD 优化方案.

4.1 口令数据向量化

在 2.2 节规则函数实现中，口令数据以结构体方式存储，每条口令占据 32 B 存储空间. 为了使用 SIMD 指令进行规则函数实现，必须首先将口令数据加载到 SIMD 向量寄存器中，由于 SW26010 的向量寄存器位宽为 256，因此恰好能够存放 1 条口令. 将口令数据加载到向量寄存器时，位置靠前的字符被存储于向量寄存器的低位，例如 *p*[0] 存储于向量寄存器的第 0~7 位，*p*[1] 存储于向量寄存器的第 8~15 位，以此类推. 口令数据向量化的优势和劣势并存. 其优势在于对口令中字符的所有操作均在寄存器中进行，避免了存储读写；其劣势在于将口令数据加载到向量寄存器以及将向量寄存器中的口令数据写回 LDM 的过程会带来额外的开销. 口令数据的加载开销可以通过多次复用输入口令进行均摊，而口令数据的写回开销则无法避免.

4.2 大小写转换向量优化

大小写转换是规则函数中出现频率最高的一类 *ROP* 变换操作，在 41 种常用的规则函数中，使用到大小写转换的规则函数共有 8 种，分别为 *lowercase*，*uppercase*，*capitalize*，*invert_capitalize*，*toggle*，*toggle@N*，*title*，*title_w/separator*. 通常大小写转换的实现采用条件分支语句，如图 14 所示，首先判断字符 *c* 为大写字母还是小写字母，若为大写字母则将其 ASCII 值加 32，若为小写字母则将其 ASCII 值减 32.这种实现方式存在 2 个弊端，首先条件运算会导致同一口令中的不同字符可能具有不同的执行分支，这将破坏 SIMD 指令的应用条件，使其无法使用 SIMD 指令加速；其次申威 SIMD 指令集中不包含 8b 向量加/减法指令，若以 32b 向量加/减法指令实现“*c*+32”和“*c*−32”语句则每条指令最多只能同时处理 8 个字符，降低了执行效率.

```
u8 normal_toggle_case(u8 c){
    if(c ≥ 'A' && c ≤ 'Z')
        return c + 32;
    else if(c ≥ 'a' && c ≤ 'z')
        return c – 32;
}
u8 opt_toggle_case(u8 c){
    /*若c ≥ 64，则a = 0x20 */
    u8 a = (c & 0x40) >> 1;
    /* 若c ≥ 128，则b = 0x20 */
    u8 b = (c & 0x80) >> 2;
    /* 若c的低5位 ≥ 26，则x & 0x20 = 0x20*/
    u8 x = (c & 0x1f) + 0x05;
    /* 若c的低5位≥ 1，则y & 0x20 = 0x20*/
    u8 y= (c & 0x1f) + 0x1f;
    /* 若c处于[64,128)区间，且c的低5位处于
[1,26]区间(满足此条件的c是字母)，则m = 0x20，
否则m = 0x00 */
    u8 m = (a & ~b) & (~x) & y;
    /* 任意字母与0x20异或可改变其大小写*/
    return c ^ m;
}
u8v opt_toggle_case_vector(u8v vc) {
    /* VEC0x40是由32个0x40组成的256 b向量，下同*/
    u8v va = (vc & VEC0x40) >> 1;
    u8v vb = (vc & VEC0x80) >> 2;
    u8v vx = (vc & VEC0x1f) + VEC0x05;
    u8v vy = (vc & VEC0x1f) + VEC0x1f;
    u8v vm = (va & ~vb) & (~vx) & vy;
    return vc ^ vm;
}
```

Fig. 14 Toggle case function

图 14 大小写转换函数

本文对大小写转换函数的优化实现如图 14 中的函数 *opt_toggle_case* 所示. 其基本原理为互相对应的大写字母与小写字母的 ASCII 值相差固定为 32.在二进制角度，大写字母与小写字母的 ASCII 值仅第 5 位不同，大写字母的第 5 位为“0”，小写字母的第 5 位为“1”，因此将任何小写字母的 ASCII 值与“0x20”异或即可得到其对应的大写字母的 ASCII 值，反之亦然. 但在此之前，必须先保证输入字符 *c* 是大写字母或小写字母，函数 *opt_toggle_case* 中展示的方法可以筛选出 ASCII 码在 [64, 128）区间内且低 5 位在 [1, 26] 区间内的字符，满足此条件的字符即为大写字母或小写字母. 将满足条件的字符与“0x20”异或即可完成大小写转换. 此方法不会产生条件分支语句，因此可以直接向量化，如函数 *opt_toggle_case_vector* 所示.

4.3 字符匹配向量化

在 OAC 类型规则函数中，*replace*（“sXY”），*title*（“E”）和 *title_w/separator*（“eX”）是较为特殊的 3

种函数. 其他 OAC 类型规则函数(例如 *lowercase*)对所有字符执行同样的 *ROP* 操作, 而这 3 种函数则只对特定 ASCII 值(假设为 *X*)的字符执行 *ROP* 操作, 因此必须首先找到 ASCII 值为 *X* 的字符的位置. 在传统实现方案中, 这一字符匹配的过程只能通过依次遍历输入口令中的每个字符实现. 本文使用 SIMD 指令对字符匹配进行了向量优化, 如图 15 所示为函数 *replace* 向量优化前后的代码. 由于申威 SIMD 指令集中不存在以 8 b 字符为基本元素的比较指令, 因此必须将一个 32 b 整型数的 4 B 分开比较. 经过 simd_veqvw 指令同或操作后, 口令中所有 ASCII 值等于 *X* 的字符均会变为“0xff”. 然后保留值为“0xff”的字节并将其他字节置为 0 形成掩码, 最后通过此掩码进行字符替换.

4.4 基于向量移位指令的优化

在 MAC 和 MSC 类型规则函数中存在很多向输入口令中增加或减少字符的函数, 例如函数 *delete* 删除口令中的第 1 个字符, 函数 *insert* 向口令的第 *N* 个字符位置插入字符 *M*, 这些函数均需要对输入口令中的全部或部分字符进行移动. 在传统的以单个字符为基本操作单元的实现中, 移动一个字符需要 4 个步骤: 计算源字符坐标、计算目的字符坐标、读取源字符、写入目的字符. 对多个字符进行这 4 个步骤会产生大量的读写指令. 值得注意的是这类函数中对字符的移动存在规律性, 即所有字符的移动方向和步长是相同的, 因此可以通过整体移动进行优化.

SW26010 处理器提供了一类向量移位指令, 该类指令可以同时移动 256b 向量中的 32 个字符, 以函数 *truncate_left* 为例, 图 16 展示了其 SIMD 优化前后的代码. 原本需要 *L_new* 次的字符移动操作利用 *simd_srlow* 指令只需要 1 条指令即可完成, 提高了函数执行效率. 其他类似的规则函数也可以利用向量移位指令实现并行优化, 例如函数 *rotate_left*, *rotate_right*, *insert*, *prepend* 等, 篇幅所限此处不再赘述.

4.5 基于向量混洗的优化

在 MAC 和 MSC 类型函数中还存在若干改变输入口令字符位置的函数, 例如函数 *reverse*, *reflect*, *duplicate_all*, 这些函数不同于 4.4 节中增加或减少字符的函数, 其字符移动的方向和步长不统一, 因此无法单纯通过向量移位指令实现. 对这些规则函数的向量优化利用了 SW26010 处理器中的向量混洗指令 *simd_vshuffle*, 该指令能够将向量寄存器中的 8 个 32 b 整型数据以任意顺序重新排列, 以函数 *reverse* 为例, 图 17 展示了使用向量混洗指令优化前后的代码.

```
u8 rule_function_replace(u8 p[N], u8 q[N],
u8 a0, u8 a1, u8 L) {
    u8 L_new = L;
    for(int i = 0; i < L_new; i++) {
        /*将ASCII值等于a0的字符替换为a1 */
        if(p[i] == a0) q[i] = a1;
    }
    return L_new;
}
u8 rule_function_replace_vector(u8 p[N], u8
q[N], u8 a0, u8 a1, u8 L) {
    u8 L_new = L;
    u8v vpwd; simd_load(vpwd, p); /*加载口令*/
    u8v va0 = VECTOR_EXT(a0); /*扩展a0为向量*/
    /*按位同或vpwd和va0，ASCII值等于a0的字节
将被替换为0xff*/
    u8v vpwd_eq = simd_veqvw(vpwd, va0);
    u8v vm0 = 0x000000ff; /* 生成4个字节掩码*/
    u8v vm1 = 0x0000ff00;
    u8v vm2 = 0x00ff0000;
    u8v vm3 = 0xff000000;
    /*分别提取4个字节并移动到最低字节位*/
    u8v tmp0 = (vpwd_eq& vm0);
    u8v tmp1 = (vpwd_eq& vm1) >> 8;
    u8v tmp2 = (vpwd_eq& vm2) >> 16;
    u8v tmp3 = (vpwd_eq& vm3) >> 24;
    /*保持值为0xff的字节不变，其余置为0*/
    tmp0 = VZERO − (tmp0 + VONE) >> 8;
    tmp1 = VZERO − (tmp1 + VONE) >> 8;
    tmp2 = VZERO − (tmp2 + VONE) >> 8;
    tmp3 = VZERO − (tmp3 + VONE) >> 8;
    /*将字节移回原位并组合为vmask, vpwd中ASCII
值等于a0的位置在vmask中为0xff，其余为0x00*/

    tmp0 = (vpwd_eq& vm0);
    tmp1 = (vpwd_eq& vm1) << 8;
    tmp2 = (vpwd_eq& vm2) << 16;
    tmp3 = (vpwd_eq& vm3) << 24;
    u8v vmask = tmp0 | tmp1 | tmp2 | tmp3;
    u8v va1 = VECTOR_EXT(a1); /*扩展a1为向量*/
    /*完成a0到a1的替换*/
    vpwd = (vpwd & ~vmask) | (va1 & vmask);
    simd _store(vpwd, p); /*写回口令*/
    return L_new;
}
```

Fig. 15 Vector optimization of the *replace* function

图 15 函数 *replace* 的向量优化

5 实验评估

为了评估各项优化技术的实际效果, 本文基于神威•太湖之光超级计算机设计了实验, 实验中使用了单个核组硬件资源, 其参数如表 2 所示.

在表 2 所示的硬件平台上, 本文测试了不同口令恢复系统架构和规则函数实现版本下的系统性能指标, 各种系统架构和规则函数实现的可选项如表 3 所示. 系统架构方面, 本文测试了主核处理(MASTER)

```
u8 rule_function_truncate_left(u8 p[N], u8 q[N],
u8 a0, u8 a1, u8 L) {
    u8 L_new= L – 1;
    for(int i= 0; i< L_new; i++){
        q[i] = q[i+1];
    }
}
u8 rule_function_truncate_left_vector(u8 p[N],
u8 q[N], u8 a0, u8 a1, u8 L) {
    u8 L_new = L – 1;
    u8v vpwd; simd_load(vpwd, p);
    /*将vpwd整体右移8 b*/
    vpwd = simd_srlow(vpwd, 8);
    simd_store (vpwd, p);
    return L_new;
}
```

Fig. 16 Vector optimization of the *truncate_left* function

图 16 函数 *truncate_left* 的向量优化

```
u8 rule_function_reverse(u8 p[N], u8 q[N],
u8 a0, u8 a1, u8 L) {
    u8 L_new = L;
    for(int i = 0; i< L_new; i++) {
      q[i] = p[L_new – 1 – i];
    }
    return L_new;
}
u8 rule_function_reverse(u8 p[N], u8 q[N],
u8 a0, u8 a1, u8 L) {
    u8 L_new = L;
    u8v vpwd; simd_load(vpwd, p);
    /*将vpwd整体左移32 – L_new个字符*/
    vpwd = simd_sllow(vpwd, (32 – L_new)*8);
    /*颠倒vpwd中的8个32 b整数的顺序*/
    vpwd = simd_vshuffle(vpwd, vpwd, 0x01234567);
    /*颠倒每个32 b整数中字节的顺序*/
    vpwd = VECTOR_SWAP32(vpwd);
    simd_store (vpwd, p);
    return L_new;
}
```

Fig. 17 Vector optimization of the *reverse* function

图 17 函数 *reverse* 的向量优化

Table 2 Hardware Resource Configuration of a Single Core Group in SW26010 Processor

表 2 SW26010 处理器单核组硬件资源配置

硬件资源	参数配置
主核数量	1
从核数量	64
主存容量/GB	8
局存容量/GB	64 × 64
核心频率/GHz	1.5

和从核加速(SLAVE)这 2 种规则处理方案; 规则函数实现方面, 本文测试了不同版本的规则函数实现代码, 其中 CWISE 每条指令处理 1 个字符, OPT32 为 Hashcat 中实现的版本, 每条指令处理 4 个字符. 为了表述方便, 后文以“口令恢复系统架构-规则函数实现版本”组合表示一种具体的配置, 例如“SLAVE-SWV”表示采用从核进行口令生成, 规则函数实现针对申威 SIMD 指令集进行了优化. 测试程序使用 sw5cc 进行编译, 版本号为“5.421-sw-500”, 编译优化级别为“O2”. 实验中的性能指标通过读取 SW26010 处理器中的实时时钟计数器并换算得到.

Table 3 Configurable Arguments of the Benchmark Program

表 3 测试程序可配置参数

配置名称	可选项	说明
口令恢复系统架构	MASTER	主核负责口令生成
	SLAVE	从核负责口令生成
规则函数实现版本	CWISE	单字符处理
	OPT32	32 b 整型优化
	SWV	申威 SIMD 优化

实验选取了 5 个具有代表性的 Hashcat 规则集作为测试规则集, 分别为 best64, d3ad0ne, dive, rockyou-30000, unix-ninja-leetspeak.其中 best64 是最常用的规则集, d3ad0ne 中包含了多个函数 *purge*, dive 是最大的规则集, rockyou-30000 不包含函数 *purge*, unix-ninja-leekspeak 只包含 OAC 和 OSC 类型规则函数, 且规则长度较大.

5.1 并行任务映射机制

表 4 列出了 4 种映射机制方案下单个从核的数据通信时间、数据通信量和 DMA 次数. 实验中 P 缓冲、R 缓冲、Q 缓冲的大小分别为 16 KB, 16 KB, 4 KB, 口令集数量为 64 万条, 规则集数量为 16 384 条, 单条规则采用 128 B 存储. 由表 4 中数据可以看出, 在当前实验设置下, SPM-PIRO 方案具有最小的数据通信量和最小的数据通信时间, 但与 SRM-PORI 和 SRM-PIRO 这 2 种方案差距不大, 而 SPM-PORI 方案

Table 4 Comparison of Communication Overhead of a Single Slave Core under Different Task Mapping Policies

表 4 不同任务映射机制下单个从核的通信开销比较

映射机制	通信时间/ms	通信量/MB	DMA 次数	
			口令	规则
SRM-PORI	91.0	42.3	20	2 560
SPM-PORI	132.3	61.4	1 250	2 500
SRM-PIRO	92.5	43.1	2 560	128
SPM-PIRO	86.8	41.0	2 500	2

的数据通信量和通信时间最大，这是因为 SPM 映射下从核需要处理的口令数量最多，使得口令外循环的次数增多，进一步导致规则内循环的次数增加，通过口令和规则分块的 DMA 传输次数可以验证这一结论，因此在口令数量远多于规则数量的情况下，应该避免采用 SPM-PORI 方案.

5.2 从核缓冲区配比优化机制

为了验证从核缓冲区配比优化机制的效果，本文实验测试了最优策略、等容量策略、等数量策略这 3 种不同的缓冲区分配策略下的数据通信时间与通信量，其中最优策略依据式(5)计算得出，等容量策略将 P 缓冲和 R 缓冲设置为相同容量，等数量策略令 P 缓冲和 R 缓冲中存储的口令和规则数量相等. 实验中使用的口令集大小为 64 万条，规则集大小为 16 384 条. 在测试最优策略的性能时，实验还选取了与理论最优缓冲区配置相近的其他配置以验证理论最优是否为实测最优，实验结果如图 18 所示.

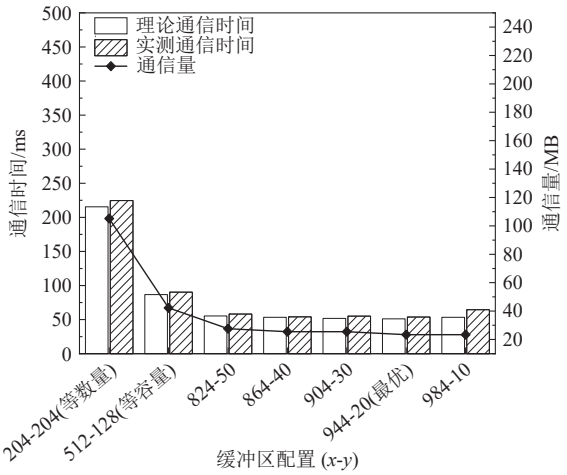


Fig. 18 Communication overhead under different buffer sizes

图 18 不同缓冲区大小下的通信开销

图 18 中使用的缓冲区总容量 S 为 32 KB，单条规则数据 v 大小为 128 B，此时式(5)计算得到的最优 P 缓冲口令条数 x^*约为 944，最优 R 缓冲规则条数 y^*约为 20，而在“944-20”配置下的实测通信时间同样也取得了最小值，说明理论结果与实测结果吻合，验证了 3.3 节中数据通信时间模型的准确性. 相比于等容量策略和等数量策略，最优策略对应的缓冲区配置将数据通信时间分别缩短了 40.5% 和 76.1%，将通信量分别减少了 44.7% 和 77.8%.

5.3 变长规则存储机制

为了验证变长规则存储机制的优化效果，本文实验比较了应用变长规则存储机制前后的数据通信时间和通信量，如表 5 所示. 由于统计上每条规则包含的规则函数数量约等于 3，因此应用变长规则存储机制后，平均每条规则数据的大小为 12 B，此时由于每条规则数据占用的局存空间变小，从核每次 DMA 可以加载更多的规则到局存中，从而大幅度减少了加载规则的 DMA 次数. 同时由于规则集的总大小减小，从核的数据通信时间和通信量分别减少了 89.0% 和 89.3%.

Table 5 Communication Time and Data Traffic Under Different Rule Sizes

表 5 不同规则数据大小下的通信时间与通信量

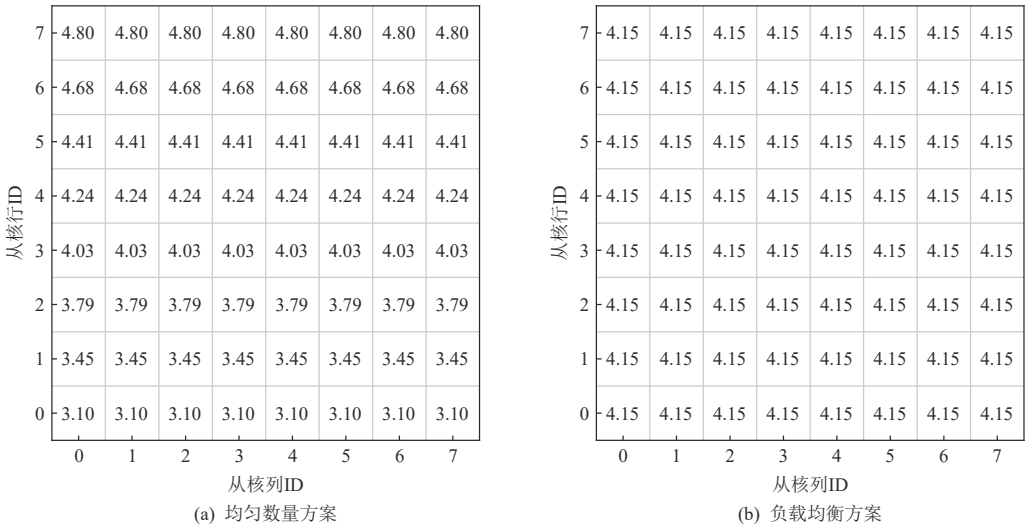
规则大小/B	x^*-y^*	通信时间/ms		通信量/MB
		理论值	实测值	
128	944-20	51.1	53.7	23.4
12	944-218	5.4	5.9	2.5

注：x^*和 y^*分别表示最优的口令条数和最优的规则条数.

5.4 负载均衡机制

为了测试负载均衡机制的优化效果，本文实验构造了一个包含 1 280 万条口令的字典，字典中的口令按照长度由短到长排序，最短的口令长度为 1，最长的口令长度为 28，然后比较了均匀数量方案与负载均衡方案下 64 个从核的规则变换执行时间与执行时间的标准差，结果如图 19 所示. 实验中使用了 Hashcat 中使用频率最高的规则集 best64 作为测试规则集，规则函数的实现版本为 CWISE，每个工作集包含的口令数量为 256.从图 19 中可以发现，在均匀数量方案下各个从核执行规则变换最短时间为 3.10 s，最长为 4.80 s，二者相差 1.70 s，这意味着在这 1.70 s 内，很多从核处于空闲状态，没有得到充分利用. 而在负载均衡方案下，每个从核的规则变换执行时间均为 4.15 s，所有从核自始至终均处于忙碌状态，因此负载均衡方案下从核阵列的规则变换执行总时间比均匀数量方案少了 13.5%.

为了验证工作集大小对负载均衡机制优化效果的影响，本文实验还测试了不同工作集大小下的从核阵列规则变换执行时间，结果如表 6 所示. 观察表 6 可以发现，负载均衡方案下各个从核执行时间的均值大于均匀数量方案，这是因为负载均衡方案需要访问主存中的信号量，因此产生了额外的开销. 工作集太小时(例如 64)，从核阵列需要重新获取工作集的次数最多，并且不能充分利用局存中的 P 缓冲，因此开销也越大. 随着工作集的大小增加，从核阵列重新获取工作集的次数减少，但这也导致不同



(a) 均匀数量方案 (b) 负载均衡方案

Fig. 19 Rule mangling execution time of the slave cores under different policies

图 19 不同方案下从核的规则变换执行时间

Table 6 Rule Mangling Execution Time of the Slave Cores Under Different Workset Sizes

表 6 不同工作集大小下从核的规则变换执行时间

分配方案	工作集大小	总时间/s	极差/s	均值/s
均匀数量	*	4.803	1.704	4.065
负载均衡	64	4.157	0.001	4.157
	256	4.153	0.006	4.149
	1 024	4.165	0.025	4.149

注：*表示均匀数量方案下没有工作集的概念.

从核间的规则变换执行时间极差变大. 综合考虑以上因素, 将工作集大小设置为与 P 缓冲同等大小较为合适.

5.5 SIMD 优化效果对比

本文实验测试了 CWISE, OPT32, SWV 这 3 种规则函数实现版本中执行单次规则函数需要的平均时钟周期数. 为了充分展现并行优化的效果, 实验中大部分规则函数使用长度为 24 的口令进行测试, 部分会使输出口令长度加倍的规则函数则使用长度为 15 的口令进行测试以避免输出口令超出最大口令长度.

图 20 比较了 OAC 类型规则函数的优化效果, 8 种规则函数中除了函数 *replace* 外其余函数均包含了大小写转换操作. 由图 20 可以看出, OPT32 版本的时钟周期数约为 CWISE 版本的 1/3, SWV 版本的时钟周期数约为 CWISE 版本的 1/6、OPT32 版本的 1/2, 这说明提升单条指令同时处理的字符数能够有效减少 OAC 类型规则函数消耗的时钟周期. 从图 20 中还可以发现, CWISE 版本的函数 *toggle_case* 消耗的时钟周期数约为函数 *lowercase* 的 2 倍, 这是因为 CWISE 版本函数中既要判断一个字符是否为小写字母, 也

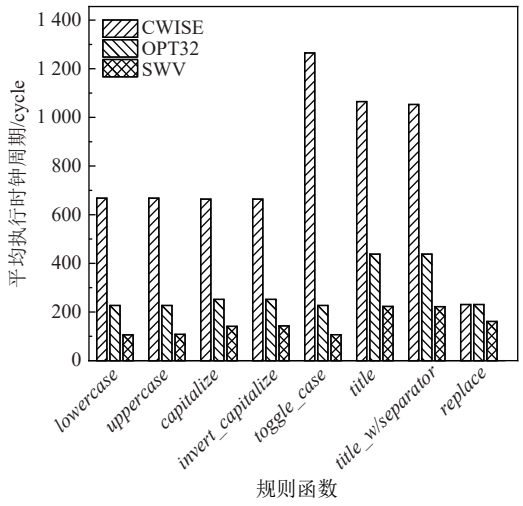


Fig. 20 Optimization results of the OAC-type rule functions

图 20 OAC 类型规则函数的优化效果

要判断其是否为大写字母, 这导致执行时间加倍, OPT32 版本和 SWV 版本因为采用了优化后的大小写判断机制没有发生此现象.OAC 类型规则函数中还包括 3 种使用了字符匹配的规则函数 *title*, *title_w/separator*, *replace*. 比较函数 *replace* 的性能可以发现, 由于 CWISE 版本和 OPT32 版本均采用了逐个字符匹配替换方案, 因此二者性能相同. 而 SWV 版本使用了 SIMD 并行字符匹配方案, 不仅提升了字符匹配的速度, 而且便利了匹配完成后的字符并行替换操作, 因此提升了性能.

图 21 比较了 OSC 类型规则函数的优化效果, 由于此类规则函数实质上只对单个字符进行了操作, 所以 CWISE 版本和 OPT32 版本执行时间非常短, 而在申威 SIMD 指令集中由于没有对向量中单个字节

进行操作的指令，要实现对单个字节的修改，必须首先生成掩码，然后对所有字符执行同样修改，最后通过掩码将单个字符修改合并回原向量，这一过程使得 SIMD 优化后的规则函数性能反而下降.

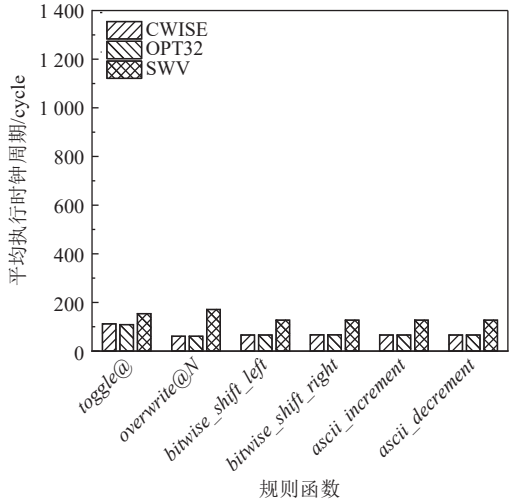


Fig. 21 Optimization results of the OSC-type rule funtions

图 21 OSC 类型规则函数的优化效果

注意到 SWV 版本的相邻 2 次规则函数调用之间使用 SIMD 向量传递口令，而 CWISE 和 OPT32 版本则使用局存传递口令，因此无法将 SWV 版本函数与 CWISE 或 OPT32 版本函数混合使用，这会带来更大的口令数据转换开销，所以实际应用中若 OSC 类型规则函数占比过多，反而会导致整体性能下降.

图 22 比较了 MSC 类型规则函数以及函数 *purge* 和函数 *nothing* 的优化效果，其中函数 *nothing* 实质上不做任何操作，因此函数 *nothing* 的执行周期数可以作为衡量函数调用开销的指标，即单次规则函数开销约为 60 个时钟周期.MSC 类型规则函数包含了 1 或 2 次字符移动，这种情况下 SIMD 指令难以发挥其并行能力，并且需要多条指令组合才能实现移动，因此导致性能下降. 函数 *purge* 由于其特殊性，难以使用 SIMD 指令实现相同功能，所以 SWV 版本中先将向量寄存器中的口令存储于局存空间，然后使用普通标量指令对其进行操作，最后再将结果存回向量寄存器，这导致 SWV 版本的函数 *purge* 实现包含了一次 SIMD 向量存储开销和一次 SIMD 向量加载开销，其性能变为 CWISE 版本和 OPT32 版本的 3 倍. 由于函数 *purge* 在实际应用中占比约为 0.2%，所以 SWV 版本的性能下降不会对整体性能产生明显影响.

图 23 比较了 MAC 类型规则函数的优化效果，此类函数大多包含了字符移动操作，以 *truncate_left* 函数为例，要对长度为 24 的口令应用 *truncate_left* 操作，需要将 23 个字符各向左移动 1 B，CWISE 版本和 OPT32 版本中均不能一次完成，而 SWV 版本则可以通过向量整体移位用 1 条指令完成. 其他函数中也或多或少用到了向量整体移位指令，由于缺少相关指令，申威 SIMD 指令集不擅长在口令末尾添加、修改字符的操作，因此在少量函数上 SWV 版本的性能出现了下降，例如 *append_character*，*duplicate_last_N*，*truncate* 等，这一点类似于 OSC 类型规则函数. 在函数 *reverse*，*reflect*，*duplicate_all* 中，SWV 版本使用了向量混洗指令，取得了 1 倍以上的性能提升.

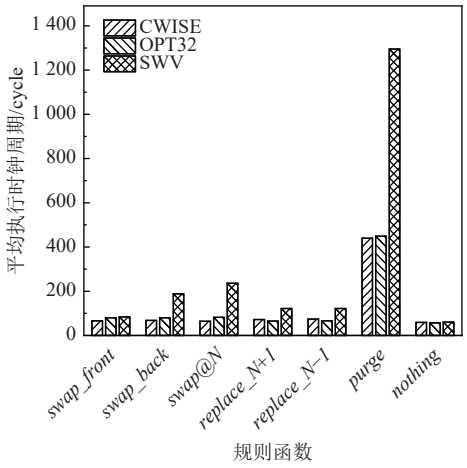


Fig. 22 Optimization results of the MSC-type rule functions and *purge*, *nothing* rule funtions

图 22 MSC 类型规则函数以及规则函数 *purge*，*nothing* 的优化效果

5.6 整体优化效果

图 24~26 展示了本文提出的优化方案的整体优化效果，实验测试了不同口令恢复系统实现方案下 NTLM, MD5, SHA1 这 3 种加密算法的口令恢复速度. 在系统架构方面，本文实验测试了纯主核口令生成方案（MASTER）和从核口令生成方案（SLAVE），其中 MASTER 方案为主核负责口令生成、从核负责口令验证，SLAVE 方案为本文提出的方案，口令生成与口令验证均由从核负责. 在规则函数实现方面，本文实验测试了 CWISE, OPT32, SWV 这 3 种版本，其中 CWISE 版本和 OPT32 版本代码均来自于 Hashcat，SWV 版本为本文提出的 SIMD 优化版本. 组合上述方案一共可以得到 6 种具体实现方案，但由于 SW26010 主核不支持部分 SIMD 指令，因此无法测试 MASTER-SWV 方案的性能. 因为不同规则集包含的规则数量、规则函数的分布均有所差异，相应的口令恢复速度亦有所不同，本文实验选取了具有代表性的 5 个常用规则集进行测试，它们分别为 best64, rockyou-30000, unix-ninja-leetspeak, dive, d3ad0ne.

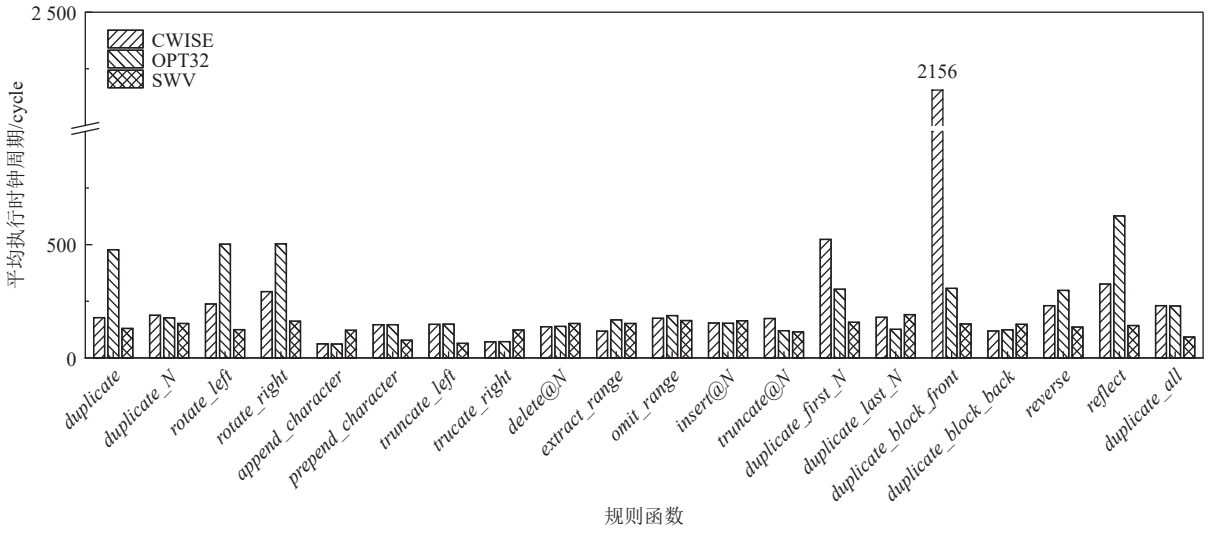


Fig. 23 Optimization results of the MAC-type rule functions

图 23 MAC 类型规则函数的优化效果

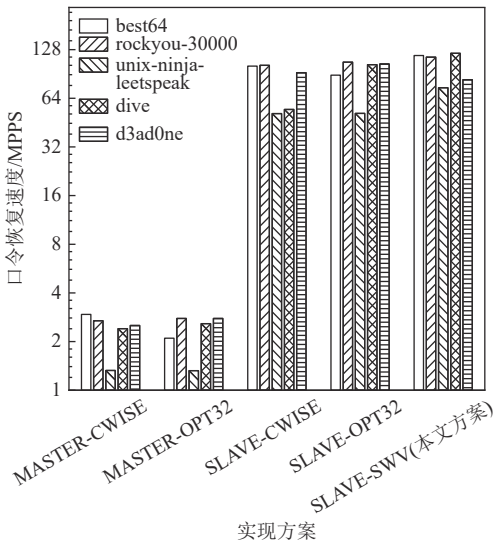


Fig. 24 Password recovery speed of NTLM algorithm under different implementations

图 24 不同实现方案下 NTLM 算法的口令恢复速度

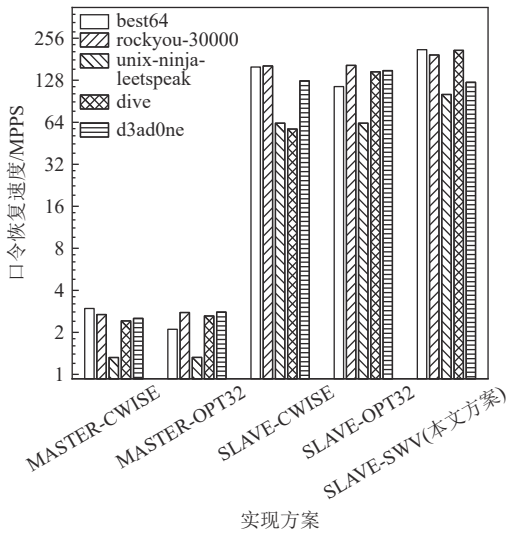


Fig. 25 Password recovery speed of MD5 algorithm under different implementations

图 25 不同实现方案下 MD5 算法的口令恢复速度

以 NTLM 算法为例分析，对比 MASTER-CWISE 和 SLAVE-CWISE 这 2 种方案，可以发现前者的速度仅为后者的 2.9% ~ 4.4%，显然这是因为主核的规则处理速度远小于从核的口令验证速度，因此成为了性能瓶颈，而将规则处理映射到从核后速度大幅度提升，进而使整体性能得到了提升. 对于 NTLM, MD5, SHA1 这 3 种算法而言，将规则处理映射到从核阵列带来的性能提升分别为 34.1, 41.7, 36.5 倍.

对比SLAVE-CWISE, SLAVE-OPT32, SLAVE-SWV 这 3 种方案，可以发现三者在不同规则集上的速度表现有所差异. 在 best64 规则集上，OPT32 版本的性能低于 CWISE 版本和 SWV 版本，这是因为 best64 规则集包含大量的规则函数 *rotate_left*, *rotate_right*, 对此类规则函数，OPT32 版本的性能表现不佳. 在 unix-ninja-leetspeak 规则集中，所有的规则函数均为函数 *replace*, 因此 CWISE 版本，OPT32 版本的速度相同，而 SWV 版本的速度提升了 44.5%，这一结果与图 20 中单次函数 *replace* 的执行周期数相符. 总体来看，在 best64, rockyou-30000, unix-ninja-leetspeek, dive 规则集上，SWV 版本的规则函数实现相较于 CWISE 版本使系统速度提升了 12.4% ~ 266.2%，相较于 OPT32 版本使系统速度提升了 7.4% ~ 113.3%. 在 d3ad0ne 规则集上，SWV 版本的性能劣于 CWISE 版本和 OPT32 版本，原因在于该规则集包含了大量 SWV 版本不占

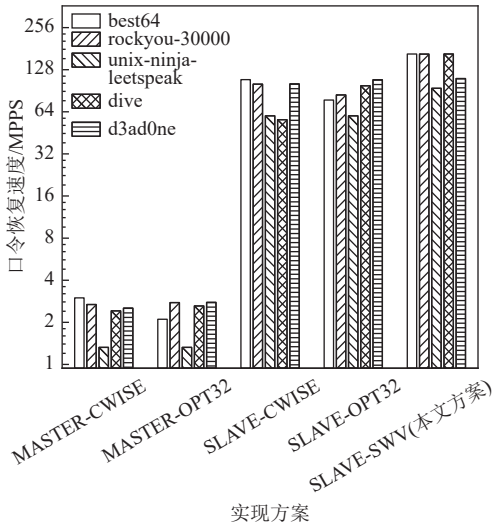


Fig. 26 Password recovery speed of SHA1 algorithm under different implementations

图 26 不同实现方案下 SHA1 算法的口令恢复速度

优势的规则函数 *swap@N*, *overwrite@N*, *purge* 等.

5.7 与其他工作对比

本文实验测试了应用本文优化方案后 NTLM, MD5, SHA1 这 3 种算法在完整的 SW26010 处理器上的性能, 并与业界主流的 Hashcat 系统进行了对比. Hashcat 系统运行于台式工作站, CPU 型号为 Intel Core i7-10 700, GPU 型号为 AMD Radeon Pro WX3200.由于 Hashcat 能够在纯 CPU 模式和 GPU 加速模式下运行, 本文实验分别将这 2 种模式命名为 Hashcat-CPU 方案和 Hashcat-GPU 方案. 测试结果如图 27~29 所示.

测试结果表明, 经过本文方案优化后的 SW26010 单节点口令恢复系统性能大幅度超越运行于 Intel 和 AMD 处理器上的 Hashcat 系统, 在 NTLM, MD5, SHA1 这 3 种算法上分别提升了 1.94, 2.74, 3.04 倍以上, 充

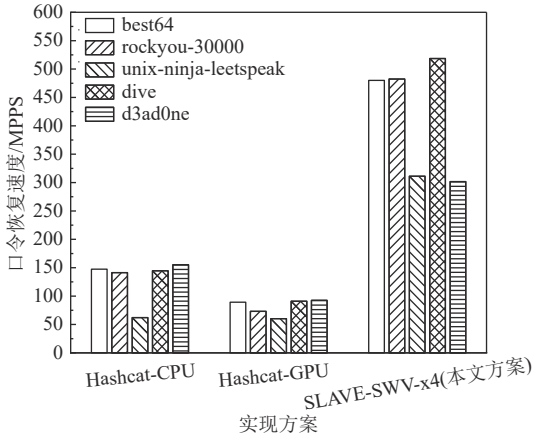


Fig. 27 Password recovery speed comparison of NTLM algorithm

图 27 NTLM 算法的口令恢复速度对比

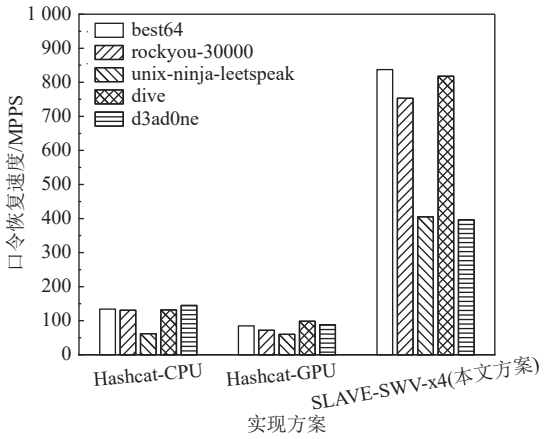


Fig. 28 Password recovery speed comparison of MD5 algorithm

图 28 MD5 算法的口令恢复速度对比

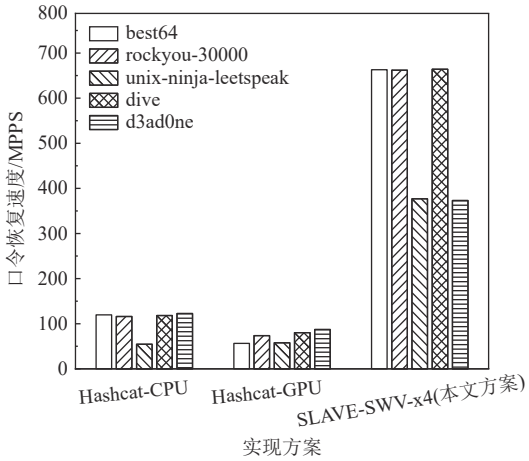


Fig. 29 Password recovery speed comparison of SHA1 algorithm

图 29 SHA1 算法的口令恢复速度对比

分体现了国产申威众核处理器在口令恢复领域相较于国外处理器的性能优势.

6 结 论

本文的研究目标是申威众核处理器平台上口令恢复系统的性能优化问题. 针对规则模式下口令生成速度性能瓶颈, 本文首先从线程级和数据级 2 个维度分析了规则处理算法的可并行度, 然后结合 SW26010 处理器的硬件架构, 提出了切实可行的优化方法, 其中主从核任务映射和从核任务分配机制解决了规则处理算法的并行分解问题, 使其能够通过从核阵列进行加速; 从核缓冲区配比优化机制解决了如何将 LDM 空间高效分配给口令数据和规则数据的问题, 降低了从核阵列的数据通信时间和通

信量；变长规则存储机制消除了规则数据的存储冗余，减少了数据通信量；负载均衡机制提高了从核的利用率，降低了整体执行时间；最后，基于申威 SIMD 指令集实现的规则函数向量优化减少了规则函数执行消耗的时钟周期，进一步提升了规则处理的速度.

本文基于神威•太湖之光超算节点进行了实验评估，测试了应用本文优化方案前后的 NTLM，MD5，SHA1 这 3 种算法的规则模式口令恢复速度. 实验结果表明，本文提出的优化方案将原有系统的规则模式口令恢复速度提升了 30～101 倍，从而解决了原有系统中基于规则的口令生成速度瓶颈问题，增强了申威处理器平台上口令恢复应用的实用性.

同时，本文实验也表明，申威处理器在执行规则处理任务时仍存在一些不足，首先 SW26010 处理器的 SIMD 指令集不支持以 8b(1B)为基本操作单元的指令，导致指定字符变换(OSC)和指定字符移动(MSC)类型规则函数的 SIMD 实现效果不佳，性能甚至低于原有实现；其次，口令数据与 SIMD 向量的互相转化开销较大，这也导致了函数 *purge* 的 SIMD 向量化实现速度较慢；最后，SW26010 处理器本质上仍是冯•诺依曼架构的通用处理器，由于通用处理器依赖指令操作数据且数据必须经由存储器在不同指令间传递，在口令恢复这种数据密集型应用中会产生大量的存储器读写开销，增加了能耗. 目前看来，突破冯•诺依曼架构探索更加专用的领域架构，是未来提升口令恢复系统的性能和能效的解决方案.

作者贡献声明：张振东提出了研究思路和实验方案，并完成主要实验和论文撰写；王彤负责了部分实验；刘鹏提出研究思路并修改论文.

参 考 文 献

[1] Ren Bijin. Implementation and optimization of password recovery on Sunway Taihu Light [D]. Zhengzhou: Zhengzhou University, 2020(in Chinese)
(任必晋. 口令恢复在神威•太湖之光上的优化与实现 [D]. 郑州：郑州大学，2020)

[2] Fu Haohuan, Liao Junfeng, Yang Jinzhe, et al. The Sunway taihulight supercomputer: System and applications[J]. Science China Information Sciences, 2016, 59(7): 1−16

[3] Dong Bensong, Zhao Rongcai, Zhang Heng. Office password recovery technology based on Sunway many-core processor[J]. Journal of Computer Technology and Development, 2021, 31(5): 137−142(in Chinese)
(董本松，赵荣彩，张恒. 基于申威众核处理器的 Office 口令恢复技术 [J]. 计算机技术与发展，2021，31(5)：137−142)

[4] Tatlı E S. Cracking more password hashes with patterns[J]. IEEE Transactions on Information Forensics and Security, 2015, 10(8): 1656−1665

[5] Hashcat. Hashcat: Advanced password recovery [EB/OL]. [2022-07-17].https://hashcat.net/hashcat/

[6] Openwall. John the Ripper password cracker [EB/OL]. [2022-07-17]. http://www.openwall.com/john/

[7] Hu Xiangdong, Ke Ximing, Yin Fei, et al. Shenwei-26010: A high-performance many-core processor[J]. Journal of Computer Research and Development, 2021, 58(6): 1155−1165(in Chinese)
(胡向东，柯希明，尹飞，等. 高性能众核处理器申威 26010[J]. 计算机研究与发展，2021，58(6)：1155−1165)

[8] Hashcat. Rule-based attack [EB/OL]. [2022-07-17].https://hashcat.net/wiki/doku.php?id=rule_based_attack

[9] Chen Yuedan. Design and implementation of heterogenous parallel algorithms on the Sunway Taihulight [D]. Changsha: Hunan University, 2020(in Chinese)
(陈玥丹. 面向“神威·太湖之光”的异构并行算法设计与实现 [D]. 长沙：湖南大学，2020)

[10] Zhang Heng, Zhao Rongcai, Dong Bensong. Optimization of MD5 decryption algorithm based on Sunway many-core processor[J]. Computer and Modernization, 2022(2): 13−18(in Chinese)
(张恒，赵荣彩，董本松. 基于申威众核处理器的 MD5 解密算法优化 [J]. 计算机与现代化，2022(2)：13−18)

[11] Xie Xinjun, Luo Shun, Yang Shihua. An efficient improvement on gpu-hardware decryption through password's self-manufacturing[J]. Information Security and Communications Privacy, 2013(3): 82−84(in Chinese)
(谢鑫君，罗顺，杨士华. 基于口令自生成的 GPU 暴力破解优化技术 [J]. 信息安全与通信保密，2013(3)：82−84)

[12] Dong Wanying. Research on efficient dictionary generation method based on password features [D]. Zhengzhou: Zhengzhou University, 2021(in Chinese)
(董婉莹. 基于口令特征的高效字典生成方法研究 [D]. 郑州：郑州大学，2021)

[13] Zhang Zhendong, Liu Peng, Wang Weidong, et al. RUPA: A high performance, energy efficient accelerator for rule-based password generation in heterogenous password recovery system [J]. IEEE Transactions on Computers, 2023, 72(4): 900−913

[14] Weir M, Aggarwal S, Medeiros B, et al. Password cracking using probabilistic context-free grammars[C]//Proc of the 30th IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2009: 391−405

[15] Marechal S. Automatic mangling rules generation [EB/OL]. [2022-12-30].https://www.openwall.com/presentations/Passwords12-Mangling-Rules-Generation/

[16] Kacherginsky P. Password analysis and cracking kit [EB/OL]. [2022-12-30].https://github.com/iphelix/pack

[17] Nam S, Jeon S, Moon J. Generating optimized guessing candidates toward better password cracking from multi-dictionaries using relativistic gan[J]. Applied Sciences, 2020, 10(20): 7306

[18] Li Shunbin, Wang Zhiyu, Zhang Ruyun, et al. Mangling rules generation with density-based clustering for password guessing [J] IEEE Transactions on Dependable and Secure Computing, 2023, 20(5): 3588−3600

[19] Ur B, Segreti S M, Bauer L, et al. Measuring real-world accuracies

and biases in modeling password guessability[C]//Proc of the 24th USENIX Conf on Security Symp. Berkeley, CA: USENIX Association, 2015: 463−481

[20] Lundberg T. Comparison of automated password guessing strategies [D]. Linköping: Linköping University, 2019

[21] Amdahl G M. Validity of the single processor approach to achieving large scale computing capabilities[C]//Proc of the Spring Joint Computer Conf. New York: ACM, 1967: 483–485

Zhang Zhendong, born in 1996. PhD. His main research interests include computer architecture, domain-specific accelerator, and high-performance computing.

张振东, 1996 年生. 博士. 主要研究方向为计算机体系结构、领域专用加速器、高性能计算.

Wang Tong, born in 1998. Master. Her main research interests include computer architecture, compiler optimization, and AI software and hardware co-design.

王　彤, 1998 年生. 硕士. 主要研究方向为计算机体系结构、编译优化、人工智能软硬件协同设计.

Liu Peng, born in 1970. PhD, professor. His main research interests include computer architecture, parallel computer architecture, VLSI design, and hardware security.

刘　鹏, 1970 年生. 博士, 教授. 主要研究方向为计算机体系结构、并行计算机体系结构、大规模集成电路设计、硬件安全.