1. 绪论
   1. 研究背景
      1. 可重构计算概述
      2. 密码算法综述
   2. 国内外研究现状
      1. 可重构密码架构综述
      2. 可重构系统PE概述

简单叙述PE的设计方案的种类，趋势等

* 1. 论文研究内容及意义

1. **第一部分是课题的思路，第二部分和第三部分是实现和验证，后两部分是服务第一部分的，所以下面主要针对第一部分来说**
2. **课题的思路是要说明是否能够具有可行性和创新性，既要保证能够实现，又要避免流于纯工程的工作。从这样的角度来看，**

**可行性：现在看不出通过这样的工作，在性能、面积上是有保障的。现在看起来这么做是做到哪算哪，既看不出真正的难点（不是指的哪一步不会做？而是要达到某个指标很难），也看不出这样的方案与最终结果如何挂钩。**

**创新性：要在难点的解决过程中，形成创新的研究点。而难点是与指标的实现密切相关的。现在诸如直通处理，PE内拓扑结构，PE位宽等，都和指标没有建立联系。所以我认为课题真正的难点你**

**课题的整体思路说明**

本文通过对大量分组加密算法进行建模分析，根据算法表现出来的需求特征，提出了一种行列异构的、功能单元组合设计的可重构密码架构PE方案。与同构PE架构相比，本文提出的方案基于算法集中算法的统计特征，消除了同构PE架构中的大量冗余单元，减少了架构的面积规模，提升整体的性能面积比。

为了验证架构的正确性和对架构进一步优化，本文提出了一种基于子图同构的算法映射的方案。对架构进行算法集映射验证；并且根据算法集在架构中的映射分布和功能使用进一步优化PE结构。

* 1. 论文组织结构

1. 分组密码算法与可重构架构
   1. 分组密码算法概述
   2. 面向分组密码算法可重构系统架构

架构的发展，需求的变化，架构的发展趋势，多种架构的PE设计方案详细描述，指出可以改进的地方

* 1. 本章小结

1. 分组加密算法图模型与算法特征提取
   1. 分组密码算法图模型

两个模型，模型中有哪些参数表征了哪些特征，

Aoe模型，从这个模型中又能分析出哪些特征，

Aov模型

* 1. 算法特征提取
     1. 算法的关键特征

**b) 算法集的特征值有哪些？为什么选择这些特征值，与面积效率是什么关系？现在选择算法集是怎样的，特征值多少？**

哪些特征，这些特征值与架构、性能的关系；

位宽：决定PE的位宽

轮数：决定阵列的大小

特殊算子：算法支持

算子组合：PE功能组合的选择和算法映射性能

算子在纵向和横向上的出现频率：异构的依据

* + 1. 算法特征提取方案
    2. 算法特征总结

算法集中的算法体现出怎样的特征

* 1. 本章小结

1. PE初始方案与架构图模型

本章将在算法特征分析的基础上，对架构中PE设计提出一个初始的方案。这个初始方案确定了可重构密码处理器的执行阵列中异构组的大小，确定了异构组中每一个位置异构PE的种类以及这些不同种类PE的结构组成。根据S盒替换、置换、有限域乘法这三种操作在算法特征分析中表现出来的明显特征，初始方案中确定了S盒替换、置换、有限域乘法这三种运算单元的个数以及在异构组中的位置，这个结果是会在第五章的映射分析中进行验证。对于加法、移位、逻辑等操作，统计特征不明显，在初始架构中的每一个PE中都包含了这些单元，进一步的优化会在第五章的映射分析之后进行。章节的最后对这个初始PE方案进行了基于图的建模，这个图模型是第五章中映射算法的目标输入。

* 1. PE初始设计方案
     1. 设计方法

对于一个通用的可重构密码处理器，密码算法的统计特征决定了处理器的设计特征，比如处理器位宽、功能单元、互连方式等特征与算法的数据分组大小、算子种类、数据加密流程等密切相关。第三章中对算法集合中的分组加密算法进行了建模分析，算法的统计特征成为本章中PE方案的依据。

课题致力于在保证架构的算法支持和映射性能的前提下尽可能地减少架构的设计面积。算法的统计分析表明传统的可重构密码架构中存在很多单元冗余设计。在传统的同构设计中，在架构的任何位置的PE都包含了算法所需的全部功能；但是对于算法而言，某一种功能单元只会出现有限的几次，架构中这些功能单元的使用率很低，甚至有些功能单元根本不会被使用。本课题从两个方面避免这种功能单元冗余；首先在确定初始架构时对于位置信息比较明显的功能单元，只在有明确需求的位置上放置这些功能单元；其次对于那些位置特征不明显的功能单元，在制定初始架构方案时不进行确定，如传统的同构架构一样，所有的PE都包含这些功能单元，进一步的删除在进行算法映射后进行，算法映射统计结果为进一步删除这些位置信息不明显的功能单元提供依据。

* + 1. 阵列拓扑结构

如图xxx所示，可重构密码处理器阵列由多个异构组重复迭代组成。每3个PE行和3个互连结构组成1个异构组，每1个PE行中包含4个PE。

对于87.2%的加密算法来说，4路并行的数据通路就可以是算法达到最大性能，只有少数的加密算法需要8路或16路的并行PE。在架构设计中，并行PE数的增加意味着阵列规模的线性增加；对于8路或16路的阵列，在大多数算法下，整个阵列的硬件利用率将非常低，因此架构中采用4路并行PE的设计方案。



图xxx 可重构密码架构阵列拓扑图

所有的分组加密算法都有一个共同的特征，算法的加解密过程都是由多个相同的轮函数迭代完成的，因此真正表征算法的特征的是这个轮函数。对于同构架构，所有的PE中都被堆砌了算法所需要的功能单元，因此它可以不用考虑算法中功能单元的位置信息。对于异构架构来说，理想的映射方式是一个算法轮函数映射到阵列的一个异构组中，阵列以异构组重复迭代单元，对应于算法的轮函数迭代。不多轮函数较复杂，无法再一个异构组中完成，那么只能用一个新的异构组去完成余下的部分，这种情况看起来比较糟糕，但是实际上只要对异构组进行合理实际就可以把大多数算法约束到一个异构组中完成。

*如表xxx所示，对算法的关键路径进行聚类分析后发现，大部分算法的轮函数的关键路径长度都在XXXX(3.156ns)范围以内。轮函数最大的延迟单元是有限域乘法Dg(1.2ns)；*

*三行选择理由：[Dr/Dg]=3*

表 延迟聚类中心算法分布情况

|  |  |
| --- | --- |
| *分组中心(ns)* | *算法：延迟(ns)* |
| *0.510667* | *RC5:0.601000 SPECK:0.460000 SIMON:0.471000* |
| *1.142235* | *AES:1.450、BLOWFISH:1.240、CAMELLIA:0.890、GOST:1.121、SM4:1.151、TEA:0.931、XTEA:0.931、LUCIFER:0.910、CLEFIA:1.440、ARIA:1.380、PRESENT:1.080、MACGUFFIN:0.911、SHARK:1.380、CS\_CIPHER:0.931、NUSH:0.892E2:1.400、KHAZAD:1.380* |
| *2.098455* | *DES:1.640、CAST128:1.721、TWOFISH:2.521、RC6:2.072、SERPENT:2.204、C2:1.782、SQUARE:1.880 M6:2.493、ICE:2.140、GRAND\_CRU:1.950* |
| *3.156* | *IDEA:3.360、SEED:3.090S、KIPJECT:2.680、Q:3.810、HIEROCRYPT\_L1:2.760、HIEROCRYPT\_3:2.760* |

* + 1. 行间互连

可重构阵列行单元中的互联单元主要由两个部分组成：行输入多路选择器和Benes网络。如图4-xx所示是所用互联单元的结构图。



图4-xx 互联单元结构图

* 行输入多路选择器

每一行输出两个128数据：out0、out1，这两个输出数据作为行间互连的输入；每一行的输入四个128位数据：in0、in1、in2、in3，行间互连单元的四个输出数据作为下一行的输入数据。

* Benes网络

在行输入MUX的输出和ALU的输入之间用Benes网络实现任意无重复置换模式，每个行输入多路选择器对应一个Benes网络，三个组合的128位操作数可以通过输入多路选择器选取来自任何一个Benes网络的输出。架构中采用16个字节的benes结构，实现一个128bit数据中的连续16个字节的置换。通过行输入多路选择器和Benes网络的组合，下一行的输入可以来自上一行中的任何一个输出，并且可以同时完成按字节的移位、置换操作。*（画出benes16的原理图）*

*配置信息：N×N的Benes需要Nlog2N-N/2个2×2开关，因此一个Benes需要56位配置码。*

* + 1. 异构组

异构组重复迭代构成处理器阵列，是整个架构设计的核心，…

如图xxx所示，每1个异构组中包含3个互连行和3个PE行。3个PE行共有12个PE，这12个PE分成5个不同的种类。



图xxx 异构组拓扑图

在确定初始PE方案时，根据算法中S盒替换、置换、有限域乘法这三种操作表现出来的统计特征，对对应的功能单元进行了位置裁剪。5类异构处理单元的不同表现在对S盒替换、置换、有限域乘法三类处理单元的包含上，具体见表xxx所示。

表 5类异构处理单元对比

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 处理单元类别 | | PE\_Type1 | PE\_Type2 | PE\_Type3 | PE\_Type4 | PE\_Type5 |
| 包含操作 | 逻辑单元(LOU) | **√** | **√** | **√** | **√** | **√** |
| 算术运算单元(ALU) | **√** | **√** | **√** | **√** | **√** |
| 移位单元(SH | **√** | **√** | **√** | **√** | **√** |
| 置换单元(PER) | **√** |  |  | **√** |  |
| S盒子替换单元(LUT) |  |  | **√** |  |  |
| 有限域乘法单元(GFM) |  |  |  | **√** | **√** |

从表xxx可以看出，5类异构PE中都包含了逻辑、算术、移位操作单元；PE\_Type1、PE\_Type4中包含了置换单元，PE\_Type3包含了查找表单元，PE\_Type4、PE\_Type5包含了有限域乘法单元。

算法集合中有6个算法的轮函数中有置换操作，这些置换操作表现出了很一致的位置特征，置换只出现在轮函数的开始或结束的位置，如表xxx所示，而且出现最大的比特级置换是64位，因此置换单元支持最大64bit置换。架构支持的最大数据位宽是128位，因此最多只会并行使用两个置换单元，因此确定在异构组的第1、3行的第1、2个PE中包含置换单元。

表xxx 置换操作在轮数中的位置信息

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法 | | Des | Present | Square | ICE | Grand\_Cru | Q |
| 置换操作出现位置 | 轮函数开始位置 | **√** |  | **√** | **√** | **√** | **√** |
| 轮函数结束位置 | **√** | **√** |  | **√** |  |  |

S盒替换单元是整个架构中的重要单元，75%的分组加密算法使用S盒替换操作来完成算法中的非线性变换，而且S盒替换单元的面积开销占据了整个架构的相当部分，因此很有必要对S盒替换单元进行位置优化。算法的统计特征表明，S盒替换操作一般出现的位置是轮函数的中间位置，77.78%的算法在查表前进行了秘钥加操作；查表后则主要进行移位、字节变换、有限域乘法等操作；没有一个算法将S盒替换操作作为轮函数的开始或者结束。因此在异构组的第1、3行放置S盒查表单元是不合理的，异构组中只有第2行中含有S盒查表单元。

有限域乘法在AES算法中引入，随后出现的加密算法广泛使用该结构；包含有限乘法的算法有AES、TWOFISH、CLEFIA 、ARIA、SQUARE、SHARK 、GRAND\_CRU、KHAZAD、HIEROCRYPT\_L1、HIEROCRYPT\_3。在这些算法中有限域乘法都紧跟在S盒查表操作之后，因此有限域乘法应该出现在紧跟在S盒查表操作行之后，对应于异构组中的第3行。

如表xxx所示，在轮函数中，各类处理单元的前后出现概率最高的操作是抑或操作，因此除了逻辑操作单元本身除外，其它的处理单元的前后都串行连接了抑或操作，这样可以有效减少算法映射时所需的行数。

表xxx 不同的算子组合概率

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 操作 | 后向组合概率分布 | | | | 前向组合概率分布 | | | |
| 组合 | 概率 | 组合 | 概率 | 组合 | 概率 | 组合 | 概率 |
| 算术运算(AU) | MAS-XOR | 75% | MAS->SH | 25% | XOR->MAS | 56% | SH->MAS | 31% |
| 移位(SH) | SH->XOR | 47% | SH->MAS | 33% | XOR->SH | 67% | AU->SH | 27% |
| 置换(PER) | PER->XOR | 83% |  |  | XOR->PER | 67% |  |  |
| S盒子替换(LUT) | LUT->XOR | 37% | LUT->SH | 15% | XOR->LUT | 78% | MAS->LUT | 11% |
| 有限域乘法(GFM) | GFM->XOR | 70% |  |  | LUT->GFM | 70% | XOR->GFM | 10% |

* + 1. 处理单元

处理单元是架构中的最小执行单元，它包含算法需求的各种功能单元：算术单元（AU）、移位单元（SH）、置换单元（PER）、S盒替换单元（LUT）、有限域乘法单元（GFM）、逻辑单元（LOU）以及这些单元串联的抑或操作。



图xxx 带全功能的处理单元

图xxx是一个带全功能的处理单元，根据4.1.4中对异构组的分析可知，异构组中共有五类不同的执行单元，这些执行单元在置换单元（PER）、S盒替换单元（LUT）和有限域乘法单元（GFM）的包含上会有所差异。从这个全功能的处理单元中删除某些功能单元就可以得到异构组中的五类处理单元。为了易于描述，下面会描述这个全功能的处理单元。

处理单元中有5个32位的输入：in0、in1、in2、in3、Reg\_in0；其中in0、in1、in2、in3这4个输入来自上一行的互连，传输上一行处理后的数据；Reg\_in0来自寄存器堆，主要用来传输寄存的中间数据以及算法的秘钥。处理单元中有4串联的数据通路最多需要四个输入，因此用4个2输入mux从5个输入中选出4个待处理数据。

处理单元有两个输出out0和out1，输出到互连单元作为下一行的输入；异构组的第三行处理单元有一个额外的输出Reg\_out0，输出到寄存器堆，用来寄存运算中间数据。每一个输出数据可以来自处理单元并行通路中任何一个，因此在输出端有2个6输入mux来选择输出。

*表xxx 处理单元控制信号分配*

|  |  |  |
| --- | --- | --- |
| *控制信号* | *位数* | *对应功能单元* |
| *control[xx:xx]* | *x* | *算术单元* |
| *control[xx:xx]* | *x* | *逻辑单元* |
| *control[xx:xx]* | *x* | *移位单元* |
| *control[xx:xx]* | *x* | *置换单元* |
| *control[xx:xx]* | *x* | *S盒替换单元* |
| *control[xx:xx]* | *x* | *有限域乘法单元* |

*对处理单元的映射分配做说明*

* + 1. 功能单元

处理单元中有6条独立的数据通路，这6条数据通路都对应着不同的功能单元：算术单元（AU）、移位单元（SH）、置换单元（PER）、S盒替换单元（LUT）、有限域乘法单元（GFM）、逻辑单元（LOU）。接下来分别介绍这这些功能单元的结构。

* + - 1. 逻辑单元

对算法分析获取的逻辑操作使用特征如表xxx所示；所有的算法都使用了抑或操作，与操作和或操作只有很少数的算法使用，没有算法使用到非操作。

表xxx 逻辑操作统计信息

|  |  |
| --- | --- |
| 逻辑操作 | 使用频率 |
| XOR | 100.00% |
| NOT | 0.00% |
| OR | 2.80% |
| AND | 8.30% |
| XOR->XOR | 30.56% |
| XOR->XOR->XOR | 17.00% |

对于逻辑操作的级联，有30.56%的算法中含有双抑或操作，有17%的算法中含有三抑或操作；为了支持多抑或操作，如图xxx所示，逻辑单元中对抑或操作进行了可选择的三级级联。



图xxx 逻辑单元结构

如图xxx所示，结构中共有7路数据，因此需要一个8输入mux，共需要3位配置位。对应的配置信息和操作功能如表xxx所示。

表xxx 逻辑单元的配置信息分配

|  |  |
| --- | --- |
| 控制信号 | 操作 |
| 000 | ir0⊕ir1 |
| 001 | ir0⊕ir1⊕ir2 |
| 010 | ir0⊕ir1⊕ir2⊕ir3 |
| 011 | ir0|ir1 |
| 100 | ir0&ir1 |
| 101 | Ir2直通 |
| 110 | Ir3直通 |

* + - 1. S盒替代单元
      2. 算术单元

算法分析表明，有47.22%的算法使用了算术单元，其中有88.2%是32位运算，对应模运算，11.2%是16位运算，对应模运算。因此在进行结构设计时同时兼容了8、16和32三种位宽的运算需求。如图xxx所示，算术单元可以根据算法需求配置成模、模和模运算。算术运算前后出现抑或操作的概率分别是56%和75%，因此与其它功能单元一样，算术单元的前后都串联了抑或操作。



图xxx 算术单元结构

算术单元中有2个输入mux，2个3输入mux和一个4输入mux，共需要8位配置信息。

表xxx 算术单元的配置信息分配（部分）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 控制信号 | | | | | 操作 |
| Mux4 | Mux3 | Mux2 | Mux1 | Mux0 |
| 00 | 00 | 00 | 0 | 0 | (ir0+ir2) modulo |
| 00 | 00 | 00 | 0 | 1 | ((ir0⊕ir1)+ir2) modulo |
| 00 | 00 | 00 | 1 | 1 | ((ir0⊕ir1)+ (ir1⊕ir2)) modulo |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

* + - 1. 置换单元

比特置换被DES、PRESENT等算法用来完成非线性变换，同时还有组合，扩展的功能。输入为2个32位数据，输出为2个32为数据或1个64位数据。置换还可以完成某些不容易实现的操作，基于比特的置换操作可以轻松完成很多不容易实现的复杂逻辑操作。

输入的每一个比特位连接到一个可配置的比特选择器（BS），对于64位的置换单元共有64个比特选择器。没一个比特选择器需要6比特位的控制信息，如表xxx所示。

表xxx 置换单元比特选择器控制信息

|  |  |
| --- | --- |
| 配置信息 | 控制对象 |
| 000000 | 输入A[0] |
| 000001 | 输入A[1] |
| 000010 | 输入A[2] |
| . | . |
| . | . |
| 000011 | 输入A[31] |
| 100000 | 输入B[0] |
| 100001 | 输入B[1] |
| . | . |
| . | . |
| 111101 | 输入B[29] |
| 111110 | 输入B[30] |
| 111111 | 输入B[31] |

* + - 1. 移位单元

在分组加密算法中使用最多的移位操作是32比特的移位和循环移位，占总共的83.3%。只有16.7%的算法的移位运算时8比特的，因此课题中提出的移位只支持32比特的操作，少量使用到8比特移位运算的算法可以使用置换单元来代替实现。

如图xxx所示，移位单元输入两个32比特的数据，其中一个是操作数，一个是移位比特位数。移位单元支持左移位、右移位、左循环移位和右循环移位。



图xxx 移位单元

整个数据通路需要4比特的控制信息，具体的含义如下表所示。

表xxx 移位单元控制信息(部分)

|  |  |  |  |
| --- | --- | --- | --- |
| 控制信号 | | | 操作 |
| mux1 | SH | mux2 |
| 0 | 00 | 0 | ir0<<ir2 |
| 0 | 00 | 1 | (ir0⊕ir1)<<ir2 |
| 0 | 01 | 1 | (ir0⊕ir1)>>ir2 |
| 0 | 11 | 1 | (ir0⊕ir1)>>>ir2 |
| 1 | 11 | 1 | ((ir0⊕ir1)>>>ir2)⊕ir3 |

* + - 1. 有限域乘法单元

有限域乘在分组加密算法中的出现频率比较高，比较常见的使用了有限域乘法的算法有AES、TWOFISH、CLEFIA、ARIA、SQUARE、SHARK等。这些算法中出现的运算都是在域上，因此本文中的有限域乘法只支持上的乘法运算。

* 有限域上的乘法

在上，我们设f(x)是不可约多项式，则f(x)被称为域多项式，中的元素可以看作是小于n次的多项式。设上的两个元素为：

定义两者的乘积为：

其中：

根据乘法定义，上的乘法本质上是多项式中对应的系数进行运算，目前实行的基本方法是采用“Shift-and-add”算法[xx,xx]。算法执行多次移位，且加法是串行操作。算法实质上是根据乘数，将被乘数不断移位后，将每个结果作模2加，得到原始乘积，再进行模运算，就可以得到最终乘法结果。在现实中，移位分为左移与右移两种方式，并且通常采用边移位边做模运算的方式进行。使用向左移的方式时，根据乘数的第i位的值来确定移位的位数，若第i位上的数为1，那么被乘数左移i位；若第i位上的数为0，那么被乘数不运算。由上述分析可知，利用逻辑左移可以很容易实现被乘数移位，同时对每次的移位结果进行模运算，最后将每个结果模2加得到最终结果。“Shift-and-add”算法描述如图xxx所示：

根据有限域的性质以及移位相加“Shift-and-add”算法，对于具有任意不可约多项式的有限域上的乘法，假设多项式x乘以a(x)有：

(xxx)

由上式（xxx）可得：

(xxx)

|  |
| --- |
| Shift-and-add 算法描述 |
| Input：二元多项式a(x)、b(x)，最高次项系数最多为(n-1)  Output：c(x)=a(x)b(x) mod f(x)  (1)c<-0  (2)for i=n-1 to 1 do  if =1 then c<-c+b  c=cx mod f(x)  (3)if =1 then c<-c+b  return (c) |

图xxx Shift-and-add 算法描述

由式（xxx）可获得任意不可约多项式的域的乘法电路。将x乘法电路依次串联起来，便可得到域上任意不可约多项式的基本乘法运算电路。将这样的多个基本乘法运算电路并联以及异或操作，便组成了乘法矩阵运算电路[xx]。

* 电路结构

域上的乘法单元电路结构如图xxx所示。电路分为两部分，分别是域矩阵乘法电路和静态配置寄存器。域矩阵乘法电路模块负责完成域上的矩阵乘法运算工作。静态配置寄存器的功能是完成域上乘数多项式及不可约多项式信息的存储工作，这也是图中静态配置寄存器的作用。

分组密码算法实际应用中，域上的乘法通常会使用在矩阵乘法形式表示，乘数多项式以及不可约多项式是较为固定的。因此，对这两种运算参数采取静态配置的形式读入到有限域乘法运算电路中。对分组算法的分析发现，在一个4X4的矩阵中，乘数为4x4x8=128比特，不可约多项式信息则为8比特。架构中有四条通路，因此静态配置寄存器大小为4x136比特。



图xxx 域乘法电路整体电路图

根据公式xxx可以得到对于任意不可约多项式域上的x乘法电路。如图xxx所示。



图xxx 域x乘法电路

（f7,…,f1,f0）是不可约多项式的低8位，将图xxx描述的电路定义为xtime，那么将7个xtime电路依次串联起来，再将结果进行三级抑或，就可以得到如图xxx所示的域上任意不可约多项式的乘法运算电路。称它为基本有限域乘法电路，记为GF28Mult。



图xxx 基本有限域乘法电路

f[7:0]即为xtime电路中的（f7,…,f1,f0），是不可约多项式的低8位，a[7:0]和b[7:0]表示域上的两个相乘的多项式。由电路结构图可以发现，a、b和f都是可以实时输入的，因此该电路结构可以完成域的任意不可约多项式的乘法。

在分组加密算法中通常将所有的8比特的乘数多项式以固定的矩阵方式给出，表示为矩阵乘法的形式。因此，如图xxx所示，四个基本有限域乘法电路并联，输出结果再进行异或操作，这样便得到了矩阵乘法中l行×l列的8比特效果。迸一步，将该电路进行4路并联，输入128比特的乘数多项式、32比特的运算数据以及8比特的不可约多项式便可以完成完整的矩阵乘法运算。



图xxx 有限域矩阵乘法电路(1行x1列)

* 1. 架构图模型
  2. 本章小结

1. PE初始方案映射分析与优化
   1. 可重构架构映射概述

应用映射是可重构设计的一个主要难题，在可重构架构上处理数据密集型应用要求较高的吞吐量和并行性，采用手动的方法将应用映射到可重构架构上不仅费时并且容易出错，当算法集合增大时，大量的映射变得非常繁琐。手工映射需要映射人员对架构有清晰的认识，不容易掌握，而且在架构进行微调时，所有的映射要重新进行，出现大量的重复性工作。另一方面，拥有高效的自动化映射工具允许架构设计人员对架构进行大规模的应用样本映射测试，根据映射结果反馈，对架构进行优化调整。因而设计高效的自动化工具是可重构架构设计的一个关键问题。

* + 1. 问题模型

应用可用数据流图（Data Flow Graph，DFG）表示，在DFG中每个节点表示应用中的某一个运算操作，有向边表示两个操作间的数据依赖关系和数据流向。应用到可重构阵列上的映射包括三个部分：DFG中的节点到可重构阵列上的PE的映射；DFG中的有向边到PE间互连的映射；数据到局部存储的映射。图xxx给出了一个应用到可重构阵列上的映射，图中的R表示该PE不进行运算，作为路由使用。



通常将应用映射到可重构阵列上需要考虑体系结构上的几点制约因素：

* 阵列

阵列的规模，即阵列上的资源数量，直接决定应用操作的并行度和吞吐量，较大的应用映射到资源数较少的阵列上时，需要根据资源数量对应用算法进行划分，在不同划分里的运算需要对可重构阵列进行配置切换才能继续执行。另一个关键因素是阵列的拓扑结构，包括阵列的横向执行单元的个数，这决定了应用可以在阵列上映射的数据宽度；阵列的纵向深度，这在应用进行流水展开时决定了可以映射的流水级数。

* 处理单元

每一个都有多个功能单元资源，功能资源包括单操作的功能单元例如移位器和支持多种操作的功能单元如一些功能单元组合操作。在异构的可重构阵列中，一些面积开销大的功能单元如查找表单元只存在阵列中的部分处理单元中，因而不是所有的操作都能映射到任意的处理单元上，部分操作只能由特定的处理单元完成。

* 局部存储器

通常上多个通过总线来共享一个局部存储器，总线的带宽有限，不能在一个周期来完成所有PE的数据获取，因此在映射时要考虑这种总线带宽限制。

* 互连网络

可重构阵列中PE间的互连资源是有限的，不是所有的PE间都可以直接互连；在某些时候还需要将某一个PE作为路由使用。互连网络结构是应用映射时通信路径选择的主要约束因素。

本文提出的可重构密码处理器上应用映射的特点

1. 可重构密码处理器阵列的互连是有导向的，数据从阵列的第一行输入，第一行的输出结果给第二行，数据经过整个阵列从最后一行输出。因此在映射时从阵列的第一行出发，数据流向和阵列的互连导向一致。
2. 可重构密码处理器阵列上的映射属于空间映射，密码算法会被循环展开，在架构上进行流水展开映射。
3. 行间的互连不是硬连线，而是数据选择。可以选择某条路径到达下一行的任何位置，但不可以同时到达所有的位置。也就是从某一个点出发的所有连线是互斥的，不可约同时存在，每一次只能互连到一个位置。
4. PE内部功能更复杂，不只是单个运算到单个PE的映；PE内部有很多组合功能设计，应用中的多个组合的功能会被同时映射到一个PE中。
5. 单个PE的输入不唯一、功能不唯一，也就是说在输入允许的情况下，横向上并行的多个功能可以被映射到同一个PE。
6. 可重构密码处理器阵列被设计成行间异构、列间异构。这样的不一致性结构在架构描述、映射算法上都有特异性。

决定方法的变化

* + 1. 研究现状

将循环映射到时，根据映射目标架构的不同，有两种常用的映射方法：时间映射（temporal mapping）和空间映射（spatial mapping）。

时间映射适用于处理单元只有一个或者一行的可重构处理器的架构。时间映射的方法将整个循环体映射到一个PE或一行PE上，每PE个或每行执行不同迭代次数的循环。PE要顺序执行多个操作，每执行完循环的一次操作就要进行一次重构，动态配置下一级的操作功能。时间映射方法的特点有：

* 由于循环的所有操作都在一个PE或一行内完成，不需要考虑PE间的数据互连；
* 可以直接利用传统的编程编译方法完成循环的映射；

这类CGRA映射问题[ V1-V3]的方案来自于VLIW架构的编译技术，它利用了VLIW中的时间流水模调度算法和VLIW中的存储共享特征。

空间映射的方法适用于可重构阵列架构，它将循环中的每个操作一次性映射到CGRA的PE阵列上，每个PE只绑定循环中的一个操作，只执行一种固定的操作，整个循环执行的过程中只用配置一次，不需要在经过重新的配置执行其他操作。空间映射的方法有以下特点：

* 能够充分利用CGRA计算资源和并行运算的能力，并行执行循环体的多次循环；
* PE只需要完成一个固定的操作，重构的开销小，整个循环的过程不需要进行重构；
* 操作节点的布局要充分考虑各个操作间的数据依赖关系和CGRA的互连资源，布局复杂，并且要使用较多的互连资源；

这类CGRA映射问题的研究更为广泛，最开始的方案灵感来自FPGA[v4-v5]综合里的布局布线技术，但是CGRA本身和FPGA有很大不同的，具体来说，CGRA中的互连线大部分是固定的，而FPGA中是可配置的；因此基于FPGA综合里的布局布线技术在固定的互连在寻找路由变得很困难。后来的一些映射方法主要来自于图论领域，[V6 ]利用子图同构算法获取DFG图到架构图之间的映射候选集；SPKM [V7] 引入分裂和外扩的方式 [V8] ，它将应用看成一个集合，每一次向外扩展一个点，余下的点作为新的扩展集合，一直到集合中的点全部外扩完毕。类似的图论方案还有如[V9]中的子图同胚[V10(134)]以及EPIMap[V10(56)]中的图满射技术。

本文中的可重构密码处理器是架构中包含PE阵列，因映射时将密码算法循环展开以流水的形式映射到阵列上，因此本文中的应用映射问题为空间映射。根据上小节总结的架构特征，本文选用适用性更好的子图同构算法作为映射的依据，子图同构算法能够在异构的不规整架构中找出所有的可能映射方案，然后再通过一定的映射规则找出资源使用最少的方案作为最终映射方案。

* 1. 基于子图同构的映射方案
     1. 子图同构基本概念
* 基本定义

**定义5.1（图）：**一个图是一个四元组，其中，

1. 称为G的顶点集，其元素称为顶点或结点；
2. 称为G的边集，其元素称为边；
3. 为图G的顶点标记函数，说明顶点与其标记的对应关系；
4. 为图G的边标记函数，说明边与其标记的对应关系。

**定义5.2（子图）：**一个图 为图的一个子图，记为，如果有：

1. ；
2. )；
3. ；
4. 。

此时，我们也称图为图的一个超图。

**定义5.3（图同构）：**图和图是同构的，记为，如果存在一个双射函数，使得：

* 1. ；
  2. 。

这样的函数也称为图与的图同构。例如，图xxx中A和B两个图同构。



图 xxx 图同构的示例

**定义5.4（子图同构）：**给定图与的一个图同构以及另一个图，如果，则为图与的一个子图同构。图xxx是一个子图同构的示例，S是G的一个子图，它包含了G中顶点集的一个子集，而且S中顶点的连接方式与G中相同，所以称S是G的一个同构的子图。



图 xxx 子图同构的示例

* 复杂度分析

在理想情况下，问题都可以在多项式时间（也称为P时间）内解决。这意味着对于解决一个含有n项的问题，存在c，z，g几个常量，算法T所需时间受以下公式下公式[Vxx]的限制：

其中常量z是我们真正感兴趣的，它通常被叫做“增长因子”。它表示当n增加时算法预期增多的额外时间的数量。例如当z =1，n加倍时，算法所需的时间也加倍。

但是子图同构被证明是一个NP完全问题，即在最坏情况下，判定两个图子图同构所需时间与图中所包含的节点数量成指数增长关系。也就是说，最坏情况下解决子图同构问题的时间复杂度为，其中 n 为规模较大的图中的节点个数。

虽然子图同构问题具有先天的复杂性，但是由于可重构密码处理器映射本身的一些特性可以使问题的复杂度降低。

1. 算法图和架构图都是有向图，而且有明确的起点和终点，这样架构图中的算法图候选集非常有限。
2. 架构图中没一个结点（PE）的输入输出很有限，而且结点只有相邻才有互连，因此图的边集很小，这也简化了算法的复杂度。
3. 在本架构中一共只有5类PE，而且边没有属性，降低了图标记的难度。
   * 1. VF2子图同构算法

VF2算法是2004年由Cordella在文献[xxx]中提出的，随后被广泛采用来解决子图同构问题。

* + 1. 基于VF2算法的映射方案

**e) 算法图和架构图的匹配，或者说如何映射，这是这几天正在讨论的问题，为什么没有涉及？**

构建了算法图和架构图后，采用下面几篇论文中使用的子图同构（或者近似子图同构）的映射方案

[1]Clark, Nathan, et al. "Scalable subgraph mapping for acyclic computation accelerators." Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems. ACM, 2006.

[2]Hamzeh, Mahdi, Aviral Shrivastava, and Sarma Vrudhula. "EPIMap: using epimorphism to map applications on CGRAs." Proceedings of the 49th Annual Design Automation Conference. ACM, 2012.

[3]Tuhin, Mohammed Ashraful Alam, and Theodore S. Norvell. "Compiling parallel applications to coarse-grained reconfigurable architectures." Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on. IEEE, 2008.

[4]Brenner, Janina A., Sándor P. Fekete, and Jan C. Van Der Veen. "A minimization version of a directed subgraph homeomorphism problem."Mathematical Methods of Operations Research 69.2 (2009): 281-296.

* 1. 映射结果分析与PE结构优化
     1. 算法映射结果分析
     2. 基于映射结果的PE优化

**c) 架构探索空间是多大？支持多大的灵活度？灵活度越大，建模（图表示）复杂度、难度越大，同时可能获得的最佳面积效率也越高，这一块如何考虑的？**

**d) 在算法集给定的前提下，在初始架构基础上逐步形成优化后的架构，这个过程是怎样的？这个过程如果说是经过多轮迭代，每轮迭代的变化是什么？如何收敛？**

功能单元：分析算法集的映射结果，统计架构中每一个PE中的功能单元的使用频率，消除没有被使用的单元

互连：在初始架构中提供高灵活度的互连，算法映射完毕后，分析算法集的映射结果，统计架构中互连线的使用频率，消除没有被使用的互连

* 1. 本章小结

1. 优化PE方案的验证与分析
   1. PE实现与性能分析

PE的RTL实现分析，各个单元的面积和延迟，关键路径等分析

* 1. 算法映射结果

使用映射工具对算法集映射的结果

* 1. 与不同架构的对比

功能单元减少，这些减少的功能单元占原来架构的百分比（也就是说这种新设计可以为原架构减多少面积）

1. 功能单元消耗对比

算法集在不同架构的映射下阵列所需要的功能单元统计

课题中的架构优化的粒度是功能单元，因此这个表能直接说明课题的优化方案达到的效果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 架构 | 算法 | 功能单元使用 | | | | | |
| AU | SH | PER | LOU | LUT | GFM |
| 本文 | AES |  |  |  |  |  |  |
| DES |  |  |  |  |  |  |
| IDEA |  |  |  |  |  |  |
| BLOWFISH |  |  |  |  |  |  |
| CAMELLIA |  |  |  |  |  |  |
| CAST128 |  |  |  |  |  |  |
| GOST |  |  |  |  |  |  |
| RC5 |  |  |  |  |  |  |
| SEED |  |  |  |  |  |  |
| TWOFISH |  |  |  |  |  |  |
| SM4 |  |  |  |  |  |  |
| RC6 |  |  |  |  |  |  |
| SERPENT |  |  |  |  |  |  |
| TEA |  |  |  |  |  |  |
| 更多算法 |  |  |  |  |  |  |
| 项目中的架构 | AES |  |  |  |  |  |  |
| DES |  |  |  |  |  |  |
| IDEA |  |  |  |  |  |  |
| BLOWFISH |  |  |  |  |  |  |
| CAMELLIA |  |  |  |  |  |  |
| CAST128 |  |  |  |  |  |  |
| GOST |  |  |  |  |  |  |
| RC5 |  |  |  |  |  |  |
| SEED |  |  |  |  |  |  |
| TWOFISH |  |  |  |  |  |  |
| SM4 |  |  |  |  |  |  |
| RC6 |  |  |  |  |  |  |
| SERPENT |  |  |  |  |  |  |
| TEA |  |  |  |  |  |  |
| 更多算法 |  |  |  |  |  |  |
| Cyptoraptor | AES |  |  |  |  |  |  |
| DES |  |  |  |  |  |  |
| IDEA |  |  |  |  |  |  |
| BLOWFISH |  |  |  |  |  |  |
| CAMELLIA |  |  |  |  |  |  |
| CAST128 |  |  |  |  |  |  |
| GOST |  |  |  |  |  |  |
| RC5 |  |  |  |  |  |  |
| SEED |  |  |  |  |  |  |
| TWOFISH |  |  |  |  |  |  |
| SM4 |  |  |  |  |  |  |
| RC6 |  |  |  |  |  |  |
| SERPENT |  |  |  |  |  |  |
| TEA |  |  |  |  |  |  |
| 更多算法 |  |  |  |  |  |  |
| 更多架构 | AES |  |  |  |  |  |  |
| DES |  |  |  |  |  |  |
| IDEA |  |  |  |  |  |  |
| BLOWFISH |  |  |  |  |  |  |
| CAMELLIA |  |  |  |  |  |  |
| CAST128 |  |  |  |  |  |  |
| GOST |  |  |  |  |  |  |
| RC5 |  |  |  |  |  |  |
| SEED |  |  |  |  |  |  |
| TWOFISH |  |  |  |  |  |  |
| SM4 |  |  |  |  |  |  |
| RC6 |  |  |  |  |  |  |
| SERPENT |  |  |  |  |  |  |
| TEA |  |  |  |  |  |  |
| 更多算法 |  |  |  |  |  |  |

1. 面积、性能对比

这个对比是一般论文都会有的对比，也是杨博在汇报时提的要求。和论文中的对比会出现工艺对齐的问题，这些论文中都给出了阵列的门数，面积的对比有一定的参考意义。但是不同的工艺下性能是没有可比性的。

杨博只要求和清华目前的PE进行对比，这个是可以做到的，到时可以使用相同的工艺库进行综合。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 架构 | 本文 | | | 项目中的架构 | | | Cyptoraptor | | |
| 工艺/nm |  | | |  | | |  | | |
| 主频/MHz |  | | |  | | |  | | |
|  | 算法映射面积/Mgates | 性能/Gbps | 性能面积比**/(Gbps/Mgates)** | 算法映射面积 | 性能 | 性能面积比 | 算法映射面积 | 性能 | 性能面积比 |
| AES |  |  |  |  |  |  |  |  |  |
| DES |  |  |  |  |  |  |  |  |  |
| IDEA |  |  |  |  |  |  |  |  |  |
| BLOWFISH |  |  |  |  |  |  |  |  |  |
| CAMELLIA |  |  |  |  |  |  |  |  |  |
| CAST128 |  |  |  |  |  |  |  |  |  |
| GOST |  |  |  |  |  |  |  |  |  |
| RC5 |  |  |  |  |  |  |  |  |  |
| SEED |  |  |  |  |  |  |  |  |  |
| TWOFISH |  |  |  |  |  |  |  |  |  |
| SM4 |  |  |  |  |  |  |  |  |  |
| RC6 |  |  |  |  |  |  |  |  |  |
| SERPENT |  |  |  |  |  |  |  |  |  |
| TEA |  |  |  |  |  |  |  |  |  |
| XTEA |  |  |  |  |  |  |  |  |  |
| SKIPJECT |  |  |  |  |  |  |  |  |  |
| SPECK |  |  |  |  |  |  |  |  |  |
| SIMON |  |  |  |  |  |  |  |  |  |
| LUCIFER |  |  |  |  |  |  |  |  |  |
| 更多算法 |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 架构 | ProDFA | | | RCPA | | | CORBA等更多架构 | | |
| 工艺/nm |  | | |  | | |  | | |
| 主频/MHz |  | | |  | | |  | | |
|  | 算法映射面积/Mgates | 性能/Gbps | 性能面积比**/(Gbps/Mgates)** | 算法映射面积 | 性能 | 性能面积比 | 算法映射面积 | 性能 | 性能面积比 |
| AES |  |  |  |  |  |  |  |  |  |
| DES |  |  |  |  |  |  |  |  |  |
| IDEA |  |  |  |  |  |  |  |  |  |
| BLOWFISH |  |  |  |  |  |  |  |  |  |
| CAMELLIA |  |  |  |  |  |  |  |  |  |
| CAST128 |  |  |  |  |  |  |  |  |  |
| GOST |  |  |  |  |  |  |  |  |  |
| RC5 |  |  |  |  |  |  |  |  |  |
| SEED |  |  |  |  |  |  |  |  |  |
| TWOFISH |  |  |  |  |  |  |  |  |  |
| SM4 |  |  |  |  |  |  |  |  |  |
| RC6 |  |  |  |  |  |  |  |  |  |
| SERPENT |  |  |  |  |  |  |  |  |  |
| TEA |  |  |  |  |  |  |  |  |  |
| XTEA |  |  |  |  |  |  |  |  |  |
| SKIPJECT |  |  |  |  |  |  |  |  |  |
| SPECK |  |  |  |  |  |  |  |  |  |
| SIMON |  |  |  |  |  |  |  |  |  |
| LUCIFER |  |  |  |  |  |  |  |  |  |
| 更多算法 |  |  |  |  |  |  |  |  |  |

* 1. 本章小结

1. 总结与展望

Hütter, Markus, Johann Großschädl, and Guy-Armand Kamendje. "A versatile and scalable digit-serial/parallel multiplier architecture for finite fields GF (2 m)."*Information Technology: Coding and Computing [Computers and Communications], 2003. Proceedings. ITCC 2003. International Conference on*. IEEE, 2003.

Cordella, Luigi P., et al. "A (sub) graph isomorphism algorithm for matching large graphs." *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26.10 (2004): 1367-1372.

[V1] Nikhil Bansal, Sumit Gupta, Nikil Dutt, and Alexandru Nicolau. Analysis of the performance of coarse-grain reconfigurable architectures withdifferent processing element configurations. In *Workshop on ApplicationSpecific Processors, held in conjunction with the International Symposiumon Microarchitecture (MICRO)*, 2003.

[V2] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei.Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In Proceedings of the 2008 ACM SIGPLAN-SIGBEDConference on Languages, Compilers, and Tools for Embedded Systems,pages 151–160. ACM, 2008.

[V3] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker,Carl Ebeling, and Scott Hauck. Spr: an architecture-adaptive cgra mapping tool. In *Proceedings of the 17th annual ACM/SIGDA InternationalSymposium on Field Programmable Gate Arrays*, pages 191–200. ACM,2009.

[V4] Rani Gnanaolivu, Theodore S Norvell, and Ramachandran Venkatesan.Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In *Proceedings of the 2010 International Conference on Soft Computing and Pattern Recognition*, pages 145–151. IEEE,2010.

[V5] Akira Hatanaka and Nader Bagherzadeh. A modulo scheduling algorithmfor a coarse-grain reconfigurable array template. In *Proceedings of 2007International Parallel and Distributed Processing Symposium*, pages 1–8.IEEE, 2007.

[V6] Bingfeng Mei, S Vernalde, D Verkest, H De Man, and R Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the 2003 Conference onDesign, Automation and Test in Europe*, pages 296–301. IEEE, 2003.