可重构数据流 SPJ 查询处理器的研究*

周茂春, 陈叶芳†, 钱江波, 王志杰, 董一鸿, 陈华辉
(宁波大学 信息科学与工程学院, 浙江 宁波 315211)

摘 要: 数据流的实时处理需要很高的处理速度,一种解决方法是使用协处理器。然而协处理器硬布线是不变的,查询不断变化使其一定时间内综合性能达不到最优。为提高数据流处理速度和资源利用率,采用了可重构的数据流 SPJ 查询处理器,在具备选择、投影和连接三种查询模块及相应指令集的基础上,根据输入查询的查询树调用相应的模块自适应对 FPGA 编程,改变自身的硬布线,实现数据流的处理。通过大量实验验证了处理器不仅正确,而且具备高速度和灵活性。

关键词: 数据流; 可重构; SPJ; FPGA
中图分类号: TP332,TP311 **文献标志码**: A **文章编号**: 1001-3695(2012)05-1781-06
doi:10.3969/j.issn.1001-3695.2012.05.048

Research on reconfigurable SPJ query processor for data streams

ZHOU Mao-chun, CHEN Ye-fang†, QIAN Jiang-bo, WANG Zhi-jie, DONG Yi-hong, CHEN Hua-hui
(*School of Information Science & Engineering*, *Ningbo University*, *Ningbo Zhejiang* 315211, *China*)

Abstract: Real-time processing for data streams requires a high performance system. One of the candidate solutions was using a co-processor. However, normal circuit co-processor generally maintains the same circuit, which makes the performance of the co-processor is not always optimal. In order to improve resource utilization and promote processing speed, this paper proposed a novel reconfigurable SPJ (select, projection and join) query processor, which was based on the above three query modules and corresponding instruction set. It called the corresponding module according to the query tree translated from input queries, and then self-adaptively programed with the FPGA. By changing hard wired circuit, the processor achieves the optimal processing performance. A large number of experiments verify that the processor is not only correct, but also with high-speed and flexibility.

Key words: data stream; reconfigurable; SPJ(select, projection and join)query; FPGA

数据流是实时、连续、有序、时变、无限的元组序列。数据流的处理以查询为中心,用户向系统注册查询并期望系统返回他们想要的查询结果。连续查询[1]是对数据流不停地、连续地执行查询,对新到来的元组执行操作,从而增量式产生新的查询结果。

由于用户一般只关心最近的数据,因此可以使用窗口对其进行限定。尽管操作被限制在窗口内,其处理还是非常耗时的。降载策略[2,3]可以通过丢弃一部分数据来减少处理时间,但降载必然会导致连接结果的偏差。由于存储容量有限,用户又要求数据流不间断无延迟的实时处理结果,因此提高数据流的处理速度非常重要。

随着硬件技术的发展,新硬件特性已经成为数据库技术的主要研究领域。目前研究人员已经认识到采用硬件可以大幅度提高数据流的处理速度,近年已经有一些专家研究采用专用处理器如片内多重处理器、图形处理器以及 FPGA 等来加速数据处理,减少降载操作,提高数据的处理速度。

本文提出适合高速数据流 SPJ 查询处理的若干算法和技术,设计实现可重构数据流 SPJ 查询处理器,可较大程度地提高数据流查询处理速度,为数据流处理领域提供更先进实用的解决方案。

1 相关工作

随着硬件技术的发展,如乱序执行的超标量多 CPU、并发多线程、多级的存储层次等,这些新硬件特性已经成为数据库技术的重要研究领域。现在的研究者可能涉及很多层次,从操作系统、文件到具体的 CPU、cache、机器指令甚至逻辑门电路 FPGA。为了促进跨领域研究,VLDB2010 设立了 Databases on Modern Hardware 专题,SIGMOD2011 设立了 Databases on new hardware 等专题,讨论用于加速数据库及数据流处理的硬件技术。

低延迟、高吞吐量是数据流管理系统需要解决的关键问题,Gedik 等人[4]利用 IBM 片内多重处理器自身的并行性来提

收稿日期: 2011-10-20; **修回日期**: 2011-11-22 **基金项目**: 国家自然科学基金资助项目(60803021,60973047);浙江省公益技术应用研究项目(2010C33149,2011C21076);浙江省自然科学基金资助项目(Y1091189);宁波市自然科学基金资助项目(2010A610115,2010A610098);浙江省教育厅科研项目(Y201120729)

作者简介: 周茂春(1986-),女,山东临沂人,硕士,主要研究方向为数据库技术(zhoumc_87@163.com); 陈叶芳(1973-),女(通信作者),讲师,硕士,主要研究方向为数据库管理、数据挖掘、人工智能;钱江波(1974-),男,副教授,博士,主要研究方向为数据库、数据流及硬件技术研究与开发;王志杰(1986-),男,硕士,主要研究方向为数据库技术;董一鸿(1969-),男,教授,博士,主要研究方向为移动数据库、数据挖掘和人工智能;陈华辉(1964-),男,副教授,博士,主要研究方向为数据挖掘和数据流处理.

高数据流连接操作的处理速度,指出将元组按属性存储能够减少连接时数据的移动,SIMD 能够最大限度地提高数据的并行性。Cieslewicz 等人[5]讨论了采用片内多重处理器来提高数据库聚集操作的方法。Blanas 等人[6]讨论了多核处理器下基于主存的最高效的哈希连接算法。

在数据库操作中,GPU 在某些操作上能比 CPU 处理速度快。He 等人[7]利用新一代 GPU 功能,如随机内存寻址、高效内核通信、通用编程模型等,设计实现数据并行处理原语,并使用原语实现四种常见连接如嵌套连接等。丁鹏[8]设计和实现了一个使用 GPU 进行通用计算的函数库,将几种常用算法映射到 GPU 上。利用 GPU 加速技术,Bandi 等人[9]讨论了空间数据库查询,Govindaraju 等人[10,11]讨论了数据库排序及数据流上挖掘的技术和方法。

利用不均匀的多核硬件来进行数据处理是很困难的。Mueller 等人[12,13]利用 FPGA 来作为多核处理结构进行数据流处理的协处理器,讨论作为模块的选择、聚集、分组以及窗口操作;他们还研究了在 FPGA 上编程设计数据处理操作,以加速数据处理。Teubner 等人[14]研究了如何用 FPGA 来加速频繁项问题,还提出用 FPGA 来并行实现握手连接[15]。

2 预备知识

本研究中对于数据流处理的可重构硬布线思想是:根据用户注册的查询,分析当前的查询语句,将此查询语句转换为可处理的查询树,此时将查询树的每个节点看做是一个原子操作,整个查询被分为多个原子操作,这些原子操作的处理核模块被预先定义完成后,通过一定的方式将对应查询树中各个模块组合起来,按顺序调用,进行编译执行,并下载到 FPGA 开发板上。当查询更新时,重新转换生成新的查询树,并根据新查询树调用对应的模块,编译下载到开发板上,改变硬布线得出查询结果,从而实现 FPGA 的可重构编程。图 1 显示了硬布线的重构过程。

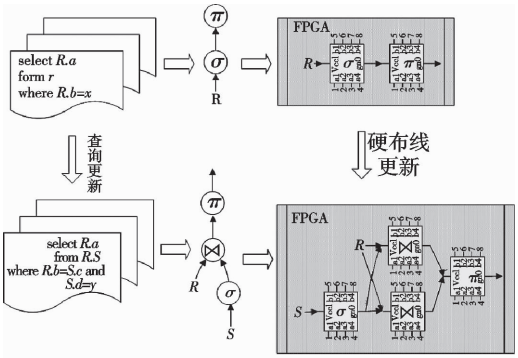

图1 硬布线的重构过程

3 SPJ 处理器模块的实现

多数据流处理的 SPJ 模块主要包括投影、选择和连接三种基本操作的实现,它们分别通过三个处理模块来实现,目前原型系统连接模块最多可以处理六条流之间的连接。为了满足可扩展的目的,本文采用元组的可扩展存储方式,如图 2(处理前的元组存储)所示。其中,阴影部分表示一个元组,元组头部一般存储在最前面,接下来则是其他的属性。本文处理的数据属性大小是 64 bit,连接后最多占用 31 个存储单元。元组头部的设计如图 3 所示,第一个是用于连接的总头格式,第二个是元组本身的头部格式。总头格式包括最大时间戳、最小时间戳、所占的全部空间大小、路由标记和六个偏移地址。其中这六个偏移地址表示六条流的元组相对于总头地址的偏移量,便于直接定位到其中某条流的元组头部。元组的头部格式包括元组的流号、元组到来的时间戳、元组所占的空间大小和四个属性的偏移地址,其中的四个偏移地址表示四个属性相对于元组头部的偏移量,便于直接定位到其中某个属性。另外,由于数据属性大小是 64 bit,根据相应的流偏移量大小等条件,得出可以处理的最大数据流连接数是六条,每条流元组最多是四个属性。

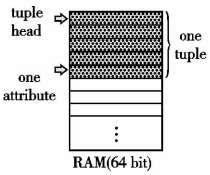

图2 元组的可扩展存储方式

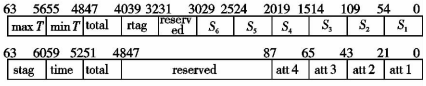

图3 处理模块中的数据格式

3.1 投影模块

投影查询基本形式是:select 列名 from 表名。其中,列名指出了要检索的列的名称,表示属性,可以是一列或多列;from 子句指出了从什么表中提取数据。

投影模块的逻辑设计如图 4 所示。

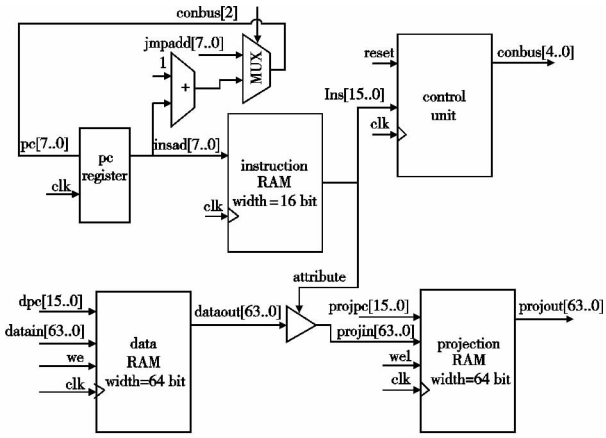

图4 投影模块的逻辑设计图

图 4 中,数据存储在 data RAM 中,指令存储在 instruction RAM 中,通过 pc 取出每条指令,通过控制单元控制在 data RAM 中取出相应的属性,其中 Dpc 可以指向需要的属性,结果输入到 projection RAM 中。

投影模块的指令格式如图 5 所示。投影模块包含 Init、Next、Sel、Jmp 和 Nop 五条指令。每条指令长度均为 16 bit。Init 是初始化指令,它需要将元组头部的数据分别保存,以便后面的指令使用;Next 指令对流进行判断,如果是要处理的流,则继续,否则跳转到下一个元组的头部;Sel 指令要完成对 select 语句中相应属性的选择;Jmp 是跳转指令,即完成一个元组的处理后跳转到 Init 指令开始对下一个元组进行处理;Nop 是延时指令,由于存储器的输出延迟,需要用 Nop 指令来保证输出数据的正确性。

3.2 选择模块

选择查询就是指定查询条件,只从表提取或显示满足条件的查询。基本形式是:select 列名 from 表名 where 查询条件。

其中,where 子句的查询条件是一个逻辑表达式,它可以是多个关系表达式通过逻辑运算符连接而成。

选择模块的逻辑设计如图 6 所示。

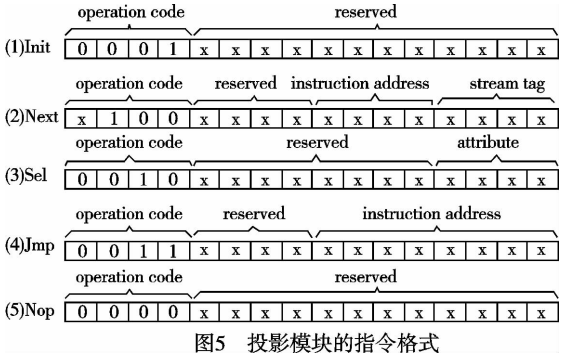

图5　投影模块的指令格式

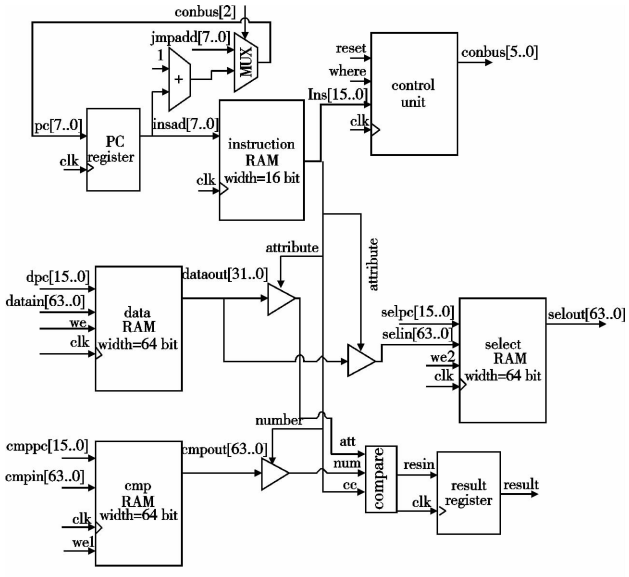

图6　选择模块的逻辑设计图

图 6 中,数据存储在 data RAM 中,指令存储在 instruction RAM 中,通过 pc 取出每条指令,通过控制单元控制在 data RAM 和 cmp RAM 中分别取出相应的属性与数据,然后进行比较,将比较结果存入到寄存器中,where 语句中全部属性比较完成后,通过寄存器中的结果来判断是否满足 where 条件,并将满足条件的元组根据 select 语句进行属性的选择,最后结果输入到 select RAM 中。

选择模块的指令格式如图 7 所示。

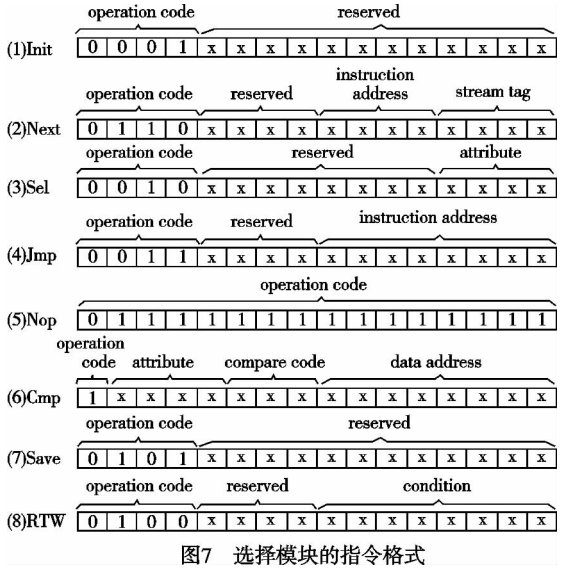

图7　选择模块的指令格式

选择模块包含 Init、Next、Sel、Jmp、Nop、Cmp、Save 和 RTW 八条指令。每条指令长度均为 16 bit。其中,前五条指令与投影模块中的指令名相同,其相应的功能也类似,因此不再赘述。Cmp 指令将 dpc 和 cmppc 指向要进行判断的属性地址和数据地址,但不进行比较,要经过 Nop 指令的延迟,由下一条指令 Save 来比较;Save 指令进行比较操作,并保存结果位到寄存器中;RTW 指令要判断 where 条件是否满足,即对寄存器中的结果位进行与操作。

3.3　连接模块

连接查询基本形式是:select 列名 1,列名 2,…from 表 1,表 2,…where 连接条件。对于连接的多个表通常存在公共列,为了区别是哪个表中的列,在连接条件中通过表名前缀指定连接条件。

连接操作比前面两种操作复杂,因此首先通过一个例子来介绍多流连接的路由。设 $A(a,b,c,d)$、$B(c,d,e,f)$、$C(a,g,e,h)$、$D(h,i,j,k)$、$E(e,i,k,m)$、$F(j,n,o,p)$六条数据流,同时注册七条并发连接请求。

Q1:select * from A,B where $A.a=B.d$;

Q2:select * from D,E,F where $D.k=E.e$ and $E.e=F.n$;

Q3:select * from B,C where $B.d=C.e$;

Q4:select * from A,F where $A.a=F.n$;

Q5:select * from C,E where $C.e=E.e$;

Q6:select * from A,B,C,D,E,F where $A.a=B.d$ and $B.d=C.e$ and $C.e=D.k$ and $D.k=E.e$ and $E.e=F.n$;

Q7:select * from C,D where $C.e=D.i$;

根据两路嵌套循环的连接步骤,如果 A 和 B 是要进行连接的两条数据流,则对于每一个新到来的 A 元组,它要执行三个步骤:a)探测 B 窗口内的元组,然后返回满足条件的连接后的元组;b)要将新元组插入到 A 的窗口中;c)删除 A 窗口中的过期元组。

流数据进来后,首先要对其进行一个处理,即加连接总头,其中总头中的路由标记是根据流号得到的。根据连接总头中的路由标记,进入到路由表中进行查找。查找这条流要进入哪些连接区的探测区及其连接区的窗口区并将其插入。在连接区内,探测区的元组与窗口区的元组一一进行比较,如果满足条件则进行连接。连接完毕的元组,其路由标记改变了,要再次查询路由表,继续进行上述过程。如果连接后路由表中相应的路由标记对应了某个查询,则可以将此连接后的元组输入到连接结果区中。另外,为了保证连接的正确性,在探测区中的元组最多只能是一个。

图 8 是针对这七条查询的连接路由。查询提交之后,根据这些查询生成相应的路由表 route table 以及相应的连接区中的连接条件 condition 和下一条路由 Next。其中六条流 A、B、C、D、E 和 F 对应的路由标记分别是 0、1、2、3、4 和 5。图中,路由标记为 5 的元组进来之后,要进入路由表中查找(箭头 0)。路由表(箭头 1)指示了路由标记是 5 的元组的去向:insert = F,probe = A、E,queries = 0。因此它要插入到 F 的窗口区(箭头 2),A 和 E 的探测区(箭头 3 和 4),此时不对应任何查询。进到 JA 连接区后,它与窗口区中 A 的元组进行比较,其条件在 condition 中显示的是(2)$F.n=A.a$。如果有满足此条件的则进行连接。连接后,连接总头的各项需要进行修改,其中路由标记可以在 Next 中进行查找。其上一个路由表示 pre = 5,连

接项 comb = (2),下一个路由标记 join = 11。因此,从 JA 中输出连接结果(箭头 5)后,需要再次查找路由表(箭头 6),此时去向:insert = 0,probe = 0,queries = Q4。因此连接后的这条元组不需要再进行连接操作,并被输入到第四个查询的结果中(箭头 7)。而相对的,插入到 JE 连接区中的元组(箭头 4),进行连接后,根据 condition 和 Next,得出下一个路由标记是 21,并输出(箭头 8)。查找路由表(箭头 9)后,发现还需要插入到 *D* 的探测区(箭头 10),再进行连接并输出(箭头 11),此时的路由标记是 22(箭头 12),再查找路由表,不需要再进行连接操作,并被输入到第二个查询的结果中(箭头 13)。

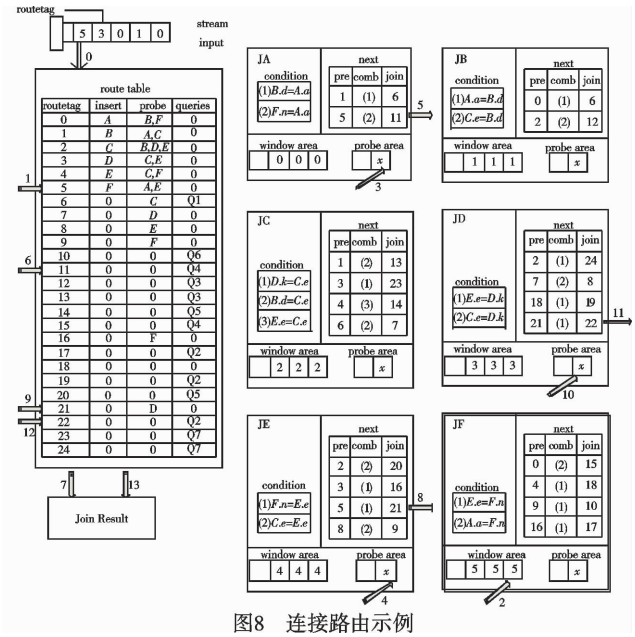

图8 连接路由示例

连接模块的逻辑设计如图 9 所示。

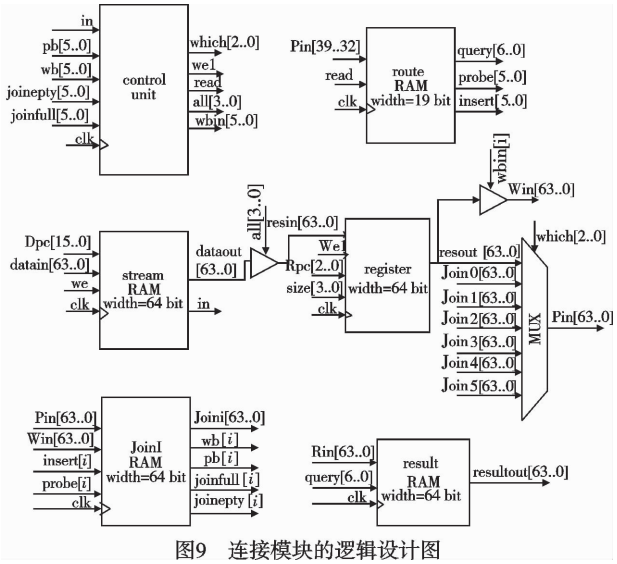

图9 连接模块的逻辑设计图

图 9 的逻辑设计是图 8 路由图的硬件实现。由于访问的互斥,定义了一些空或满的标志位进行控制。首先添加连接总头,这在寄存器中实现,然后根据路由表决定其去向。途中的 Join I 是六个连接区的一个接口示例。各个连接区的具体逻辑设计如图 10 所示。其中探测区的元组要与窗口区的元组进行比较,比较后的结果存入到比较结果寄存器中,然后判断 where 条件是否满足。满足则进行连接,结果输入到 join RAM 中。

连接模块的指令格式如图 11 所示。

连接模块包含 Init、Cmp、Jnext、Jcond、Join、Head、Hdin、Nop 和 Out 九条 指令。每条指令长度均为 16 bit。其中,Init、Cmp 与 Nop 指令功能与前面两个模块类似。Cmp 指令仍是指向要比较的探测区和窗口区相应的属性。Jnext 指令进行比较操作并判断是否满足条件,若不满足,直接跳到下一个窗口区元组进行比较;满足,则继续下一条指令。Jcond 指令对于满足条件的元组,查看其路由标记,并给出下一个路由标记。Join 指令给出连接后的路由标记。Head 指令是生成连接后新的头部。Hdin 指令是将新头部输出到结果区。Out 指令是将执行连接的窗口区元组和探测区元组分别输出到结果区,即接在新头部的后面输出。

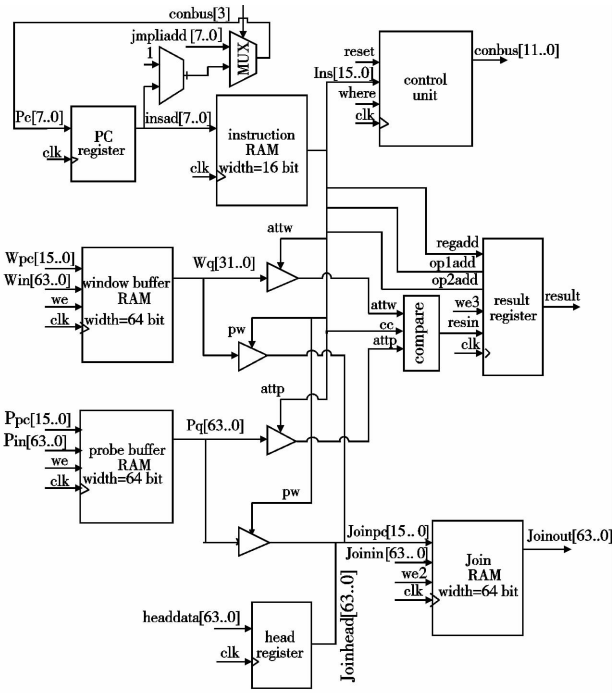

图10 连接区的逻辑设计图

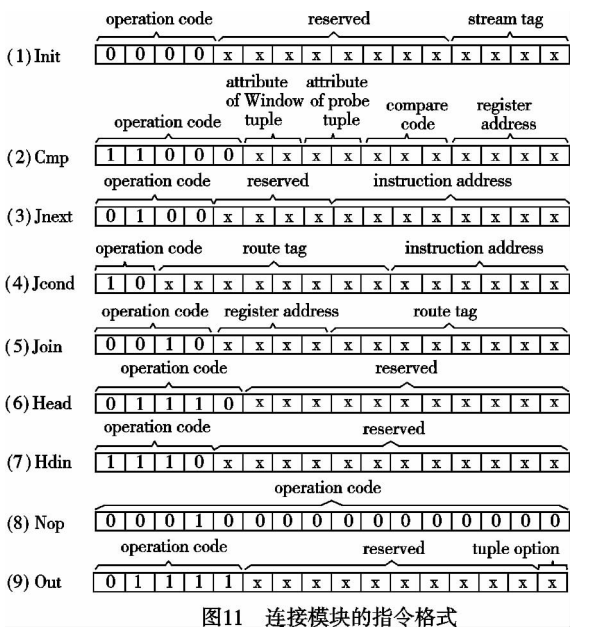

图11 连接模块的指令格式

4 实验与分析

本文是在 quartus Ⅱ 8.1 环境下使用 verilog 语言实现的 SPJ 三个模块;在 VC ++ 环境下使用的 C ++ 语言实现的查询

树的转换、调用模块以及下载;最终下载到 DE2 开发板上实现硬布线的重构。

4.1　SPJ 模块的仿真结果

首先使用实验数据来验证三个模块的正确性。

1)投影模块　使用的 select 语句如下,其中涉及的数据流格式为 $B(c,d,e,f)$。

```
select B. c,B. d,B. e from B
```

图 12 显示了其中部分仿真图。头部和其他属性本文规定以(头部,属性 1,属性 2,属性 3,属性 4)形式表示。

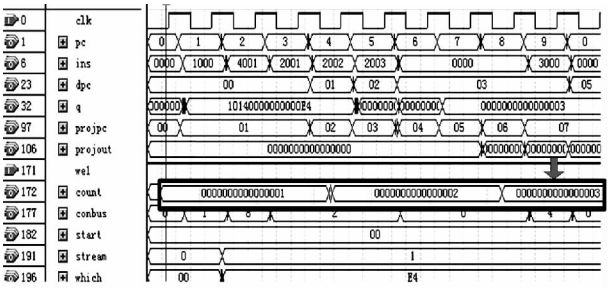

图12　投影仿真示例图

从图 12 可看出,其中处理的数据是 B 数据流的一个元组,此元组的十六进制表示为(10140000000000E4, 0000000000000001, 0000000000000002, 0000000000000003, 0000000000000004),那么 select 语句执行后,得到的对应元组为(0000000000000001, 0000000000000002, 0000000000000003),即图 12 中黑框显示的数据。显然,当处理的元组是其他流的元组,则元组会被跳过并不被输出。

此投影操作的指令如表 1 所示。

表 1　投影操作的指令

Seq	Ins	Code(hex)	Seq	Ins	Code(hex)
0	Init	1000	3	Sel att2	2002
1	Next	4000	4	Sel att3	2003
2	Sel att1	2001	5	Jmp to Init	3000

2)选择模块　使用的 select 语句如下,其中涉及的数据流格式为 $B(c,d,e,f)$。

```
select * from B
where B. c = x and B. d < = y
```

其中,$x=0x1$,$y=0x2$。

图 13、14 显示了其中部分仿真图。同样,处理的元组与上面相同。图 13 显示的是 where 语句的比较,图 14 显示的是输出元组。select 语句执行后,得到的对应元组应该是(10140000000000e4,0000000000000001,00000000000000 02,0000000000000003,0000000000000004),即图 14 中黑框显示的数据。显然,当处理的元组是其他流的元组,则元组会被跳过并不被输出;如果处理的 B 数据流的元组不满足条件,也不会被输出。

此选择操作的指令如表 2 所示。

表 2　选择操作的指令

Seq	Ins	Code(hex)	Seq	Ins	Code(hex)
0	Init	1000	5	Save	5000
1	Next	6001	6	RTW	4006
2	Cmp x	8B01	8	Sel all	2009
3	Save	5000	9	Jmp	3000
4	Cmp y	9502			

3)连接模块　这里给出的是其中 JA 连接区内的当前指令操作,使用的 select 语句如下,其中涉及的数据流格式为 $A(a,b,c,d)$,$B(c,d,e,f)$。

```
select * from A,B
where A. a = B. d
```

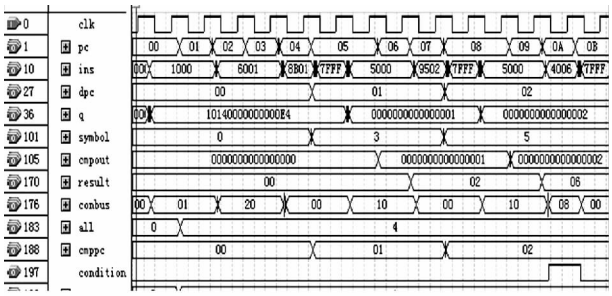

图13　选择仿真示例图1

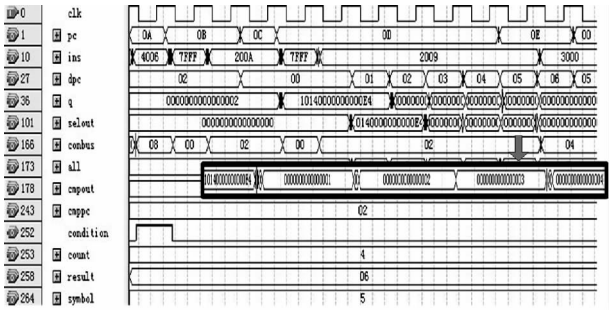

图14　选择仿真示例图2

图 15 中显示了其中部分仿真图。

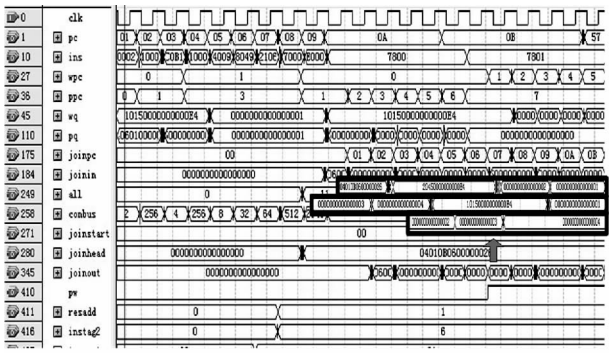

图15　连接仿真示例图

图 15 中,窗口区中的一个 A 元组为(101500000000000E4, 0000000000000001, 00000000000000 02, 0000000000000003, 0000000000000004);探测区中 B 元组(已加连接总头)为(0404060100000020, 204500000000000E4, 00000000000000 02, 0000000000000001, 0000000000000003, 0000000000000004),那么 select 语句执行后,得到的对应元组为(04010B0600000026, 204500000000000E4, 00000000000000 02, 0000000000000001, 0000000000000003, 0000000000000004, 101500000000000e4, 0000000000000001, 0000000000000002, 0000000000000003, 0000000000000004)即图 15 中黑框显示的数据。显然,当窗口区的元组连接条件不满足时,它会被跳过并不会进行连接。

此连接操作的指令如表 3 所示。

表 3　连接操作的指令

Seq	Ins	Code(hex)	Seq	Ins	Code(hex)
0	Init	0002	5	Head	7000
1	Cmp	C0B1	6	Hdin	E000
2	Jnext	4009	8	Out B	7800
3	Jcond	8049	9	Out A	7801
4	Join	2106			

4.2 整体重构性能分析

本文实现三种模块的 FPGA 芯片是 Cyclone Ⅱ EP2C35F672C6L。由 C++语言实现查询树的转换,根据所得查询树调用相应模块并最终下载到 DE2 开发板上,以实现硬布线重构。

实现模块调用的核心操作如下:

```
system("quartus_sh --flow compile xx");
//compile module xx
system("quartus_pgm.exe -m jtag - c
USB-Blaster[USB-0] -o \"p; xx.sof\"");
//download module xx to FPGA using JTAG mode
```

图 16 记录了可重构数据流 SPJ 查询处理器根据不同查询、编译及变换 FPGA 所需要的时间,这里变换 FPGA 的时间中包含编译时间。其中,本文使用了四个查询组:a)投影查询;b)选择查询;c)连接查询;d)投影、选择与连接查询。可以看到,对于查询的变换时间,前三组是比较接近的。第四组由于包含前三组的全部查询,相对比较慢。

下面来比较可重构的流处理器与软件方法的处理性能。实验平台是 Intel Core2 Quad CPU,2.66 GHz,3 GB 内存,Win XP 操作系统。硬件使用 VC++ 6.0 和 Quartus Ⅱ 8.1(其中 FPGA 芯片是 Cyclone Ⅱ EP2C35F672C6L),软件使用 VC++ 6.0 与 Oracle 9i 数据库。使用的比较数据是随机生成的实验数据,均包含一个头部和四个属性,数据头部一部分是固定的,其他是随机生成的。本文分别对数据总量是 1024 bit、2048 bit 和 4096 bit 的三组数据的处理时间进行了比较,其结果如图 17 所示。

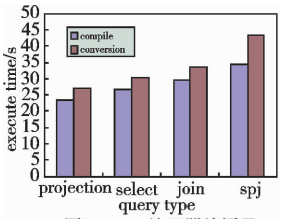

图16 SPJ处理器编译及变换FPGA时间比较

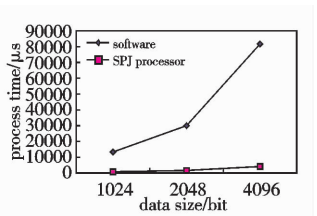

图17 SPJ处理器与软件处理速度比较

本研究注册了八条查询,包括两条投影查询、两条选择查询和四条连接查询。硬件方法是通过 C++程序把注册的查询转换成查询树,然后根据查询树调用相应的模块进行处理。软件方法是连接到 Oracle 数据库上,直接用 select 语句处理注册的查询。其中各模块的频率分别是 118.39 MHz(投影模块)、101.46 MHz(选择模块)和 65.85 MHz(连接模块)。这两种方法处理三组数据的总体时间相差较大,在硬件上的执行非常快速(μs 级),软件的执行时间在 ms 级。

5 结束语

本文研究了可重构的数据流 SPJ 查询处理器的实现,它能够根据输入的查询来进行自适应编程,重构自身的硬布线达到高性能处理。通过分析输入的查询语句,将查询转换为相应的查询树,即分成多个原子操作,然后用 Verilog 语言设计实现了三种查询模块(投影模块、查询模块和连接模块)及各自指令集。根据输入查询的查询树调用相应的模块编译下载到开发板上,改变硬布线得出查询结果,从而实现 FPGA 的可重构编程。通过大量实验验证了此处理器的正确性、快速性以及灵活性。

参考文献:

[1] GOLAB L, OZSU M T. Issues in data stream management[J]. ACM SIGMOD Record,2003,32(2): 5-14.

[2] CHENG R, KALASHNIKOV D, PRABHAKAR S. Evaluating probabilistic queries over imprecise data[C]/Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2003:551-562.

[3] BENJELLOUN O, SARMA A D, HALEVY A, *et al*. ULDBs: databases with uncertainty and lineage[C]//Proc of the 32nd International Conference on VLDB. 2006: 953-964.

[4] GEDIK B, YU P S, BORDAWEKAR R. Executing stream joins on the cell processor[C]//Proc of the 33rd International Conference on VLDB. 2007: 363-374.

[5] CIESLEWICZ J, ROSS K A. Adaptive aggregation on chip multiprocessors[C]//Proc of the 33rd International Conference on VLDB. 2007: 341-350.

[6] BLANAS S, LI Yi-nan, PATEL J M. Design and evaluation of main memory hash join algorithms for multi-core CPUs[C]//Proc of International Conference on Management of Data. New York: ACM Press, 2011:37-48.

[7] HE Bing-sheng, YANG Ke, FANG Rui, *et al*. Relational joins on graphics processors[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2008: 511-524.

[8] 丁鹏. 基于 GPU 的通用并行计算库的设计与研究[D]. 成都:西南石油大学, 2007.

[9] SUN C, AGRAWAL D, ABBADI A E. Hardware acceleration for spatial selections and joins[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2003: 455-466.

[10] GOVINDARAJU N K, GRAY J, KUMAR R, *et al*. GPUTeraSort: high performance graphics coprocessor sorting for large database management[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2006: 325-336.

[11] GOVINDARAJU N K, RAGHUVANSHI N, MANOCHA A D. Fast and approximate stream mining of quantiles and frequencies using graphics processors[C]//Proc of SIGMOD. New York: ACM Press, 2005: 611-622.

[12] MUELLER R, TEUBNER J, ALONSO G. Data processing on FPGAs [J]. Proceedings of the VLDB Endowment, 2009,2(1):910-921.

[13] MUELLER R, TEUBNER J, ALONSO G. Streams on wires: a query compiler for FPGAS[J]. Proceedings of the VLDB Endowment, 2009,2(1):229-240.

[14] TEUBNER J, MUELLER R, ALONSO G. FPGA acceleration for the frequent item problem[C]//Proc of ICDE. 2010:669-680.

[15] TEUBNER J, MUELLER R. How soccer players would do stream joins[C]//Proc of International Conference on Management of Data. New York: Acm Press, 2011: 625-636.

[16] WOODS L, TEUBNER J, ALONSO G. Complex event detection at wire speed with FPGAS[J]. Proceedings of the VLDB Endowment, 2010,3(1-2):660-669.

[17] GOLD B, AILAMAKI A, HUSTON L, *et al*. Acclerating database operators using a network processor[C]//Proc of the 1st International Workshop on Data Management on New Hardware. New York: ACM Press, 2005.