面向能耗有效高性能嵌入式微处理器的VLIW调度*

管茂林,杨乾明,张春元,文　梅
(国防科技大学 计算机学院,长沙 410073)

摘　要:为了降低功耗,目前能耗有效的嵌入式微处理器一般都采用分布式与层次化的寄存器文件结构。第一层的超小寄存器文件(TORF)的极小容量使得很多数据必须存放到第二层的通用寄存器文件(GRF)中,这给编译器带来了新的挑战。通过分析程序特征,提出了新的VLIW调度算法,通过在编译时对变量进行检测,在恰当的时机插入虚拟的copy操作并进行指令与通信调度,为对寄存器需求较大的全局变量与软流水变量构建了新的包含GRF的数据传输路由,将对TORF的压力转移到GRF中。实验结果表明,新的VLIW调度算法符合处理器的设计初衷。与不使用GRF相比,在程序性能只降低约8%的情况下,降低了约51%的寄存器访问能耗,43%的处理器能耗。最关键的是避免了程序员手工分配优化的难题。

关键词:能耗有效;分布式与层次化寄存器文件;VLIW调度
中图分类号:TP393　　**文献标志码:**A　　**文章编号:**1001-2486(2012)06-0026-08

VLIW scheduling for high performance embedded energy-efficient processor

GUAN Maolin, YANG Qianming, ZHANG Chunyuan, WEN Mei
(College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: To reduce the power, the energy-efficient embedded microprocessor always adopts the distributed and hierarchical register file structure (DHRF). Many data need to be stored in the second level general register file (GRF) because of the small capacity of the tiny operand register file (TORF), and this challenges the design of compiler. A new VLIW scheduling algorithm is proposed to solve the problem through analyzing the program characteristics. The variables are detected while the compiling and virtual copy operations are inserted at the appropriate time. Through instruction scheduling and communication scheduling for the copy operations and constructing new data transfer route including GRF for the global variables and software pipelining variables, which have great demand for the register, the pressure on TORF is transferred to GRF. The experimental results show that the VLIW scheduling algorithm is consistent with the starting point of the energy-efficient microprocessor. On the condition of 8% program performance decline, the energy consumption on register accessing is reduced by about 51%, and the energy consumption of the processor is reduced by about 43%. At the same time, the burden of programmer is avoided.

Key words: energy-efficient; distributed and hierarchical register file; VLIW scheduling

目前高性能嵌入式应用对处理器提出了越来越高的性能和功耗需求。例如,对于采用100Mbps OFDM通道的4G移动通信来说,其手持设备性能则需要210~290Gops[1],基站信号处理的性能需求更是达到了1Tops[2],同时对能耗的需求也达到了100~1000Gops/W[2]。而当前主流嵌入式处理器的性能位于10Gops~100Gops之间,能耗有效性也仅在4Gops/W[3]左右,与应用的实际需求相差甚远。另一方面,VLSI工艺的发展为设计满足应用需求的嵌入式处理器提供了可能,纳米工艺将使单芯片中晶体管密度达到每平方厘米百亿至千亿个,2010年已出现集成度超10亿支晶体管的高性能CPU(Intel的Teraflops处理器[4])。然而,简单通过提高时钟频率带来的功耗瓶颈使得现有的处理器结构无法在借助工艺提升性能的同时保持相对较低的能耗[5],满足不了未来嵌入式应用发展的需求。基于大量简单小核(用于运算加速)和少量复杂大核(用于控制和IO处理)的大规模并行处理器是未来极高性能嵌入式处理器的发展方向之一[6]。为了使整个系统满足要求,最底层的微处理器的性能、功耗就成为处理器设计的一个关键问题。

* **收稿日期:**2011-06-20
基金项目:国家自然科学基金资助项目(61033008, 60903041, 61103080);国家部委资助项目;教育部博士点基金资助项目(20104307110002);湖南省研究生科研创新项目(CX2010B028);国防科学技术大学优秀研究生创新资助项目(B100603, B120605)
作者简介:管茂林(1983—),男,江苏射阳人,博士研究生,E-mail:gfkdgmlgml@163.com;
张春元(通信作者),男,教授,博士,博士生导师,E-mail:cyzhang@nudt.edu.cn

在国内,我们率先提出了面向 Tops 级嵌入式计算的高效能微处理器设计[8],采用了分布式与层次化的寄存器文件结构。通过分析寄存器文件结构的设计初衷以及程序中的数据特征,本文提出了面向能耗有效的高性能嵌入式微处理器的 VLIW 调度,解决面向分布式与层次化的寄存器文件结构的指令调度这一关键问题,使得处理器的功耗优势得以发挥,并大幅降低了应用开发人员的负担。

1　微处理器核结构

我们提出的高效能嵌入式处理器核的微体系结构如图 1(a)所示,采用 VLIW 结构,含有 5 个发射槽,其中 3 个 ALU 单元,1 个数据管理单元(DMU:Data Manage Unit),1 个执行控制单元(MC:Micro-Controller)。每个 ALU 有两个超小的操作数寄存器文件(TORF: Tiny Operand Register File)作为输入。DMU 是处理器核与外界的数据通道,负责数据的加载和保存,它含有两个较大容量的通用寄存器文件(GRF:General Register File)来捕获指令间的数据重用。MC 负责加载指令,并控制程序的执行,只支持简单的循环控制结构,不支持 if/else 结构(if/else 由程序员转化为 select 选择操作),从而最大限度地简化执行控制逻辑,降低能耗。所有的这些单元通过零延迟的全互联的交叉开关来互连。处理器核的外部是一个共享的、基于软件管理的片上存储器,多个处理器核可以通过共享存储器聚合成一个运算簇,从而构成更大规模并行的处理器。

如图 1(a)所示,与 ALU 紧密相连的 TORF 构成了第一层次的分布式寄存器文件空间,ALU 不能直接访问的 GRF 构成了另一层次的分布式寄存器文件空间。分布式与层次化寄存器文件的出发点[7]是:容量很小的 TORF 与 ALU 紧密绑定,捕捉频繁访问的数据局域性,为 ALU 的执行提供很高的带宽,而其很小的容量则可以很好地降低每次访问的能耗,从而降低整个处理器的功耗。GRF 作为下一层寄存器空间,捕捉较长时间才会重用的数据局域性,同时在 TORF 容量不够时,存放从 TORF 溢出的数据,虽然因为其容量较大,每次访问的能耗较高,但是访问频率较低,因而可以降低整体的平均功耗。然而这却使得面向分布式与层次化寄存器文件的 VLIW 调度变得更加困难。首先,面向第一层分布式寄存器文件结构的 VLIW 调度已经是一个 NP 完全问题。其次,TORF 的极小容量使得很多数据需要存放到 GRF 中以降低对 TORF 的压力。选择哪些数据在什么时候存放到 GRF 中、在什么时候将数据再存回 TORF、以何种方式驱动数据在两层寄存器文件之间的移动对编译器而言都是一个新的挑战。因此,目前面向分布式/层次化寄存器文件的指令调度主要依赖于程序员的手工分配优化。虽然大部分嵌入式应用的核心程序规模不大,但对程序员而言仍然是一个很大的挑战。

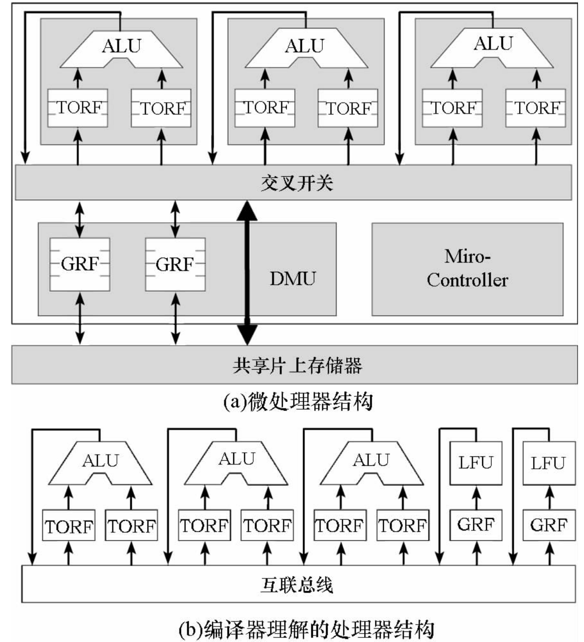

图 1　微处理器结构

Fig. 1　Microprocessor architecture

2　程序特征分析

通过分析分布式与层次化寄存器文件的设计初衷,显然编译器在进行调度时应该尽量将频繁访问和频繁重用的变量分配在 TORF 中,而将那些不经常访问或者间隔很长时间才重用的变量分配在 GRF 中,在 ALU 单元需要使用它时再将其调入 TORF。而这与程序的特征以及变量的生命周期有很大的关系。为了评估程序对寄存器的需求,本文选择了如下若干典型嵌入式应用的核心算法进行测试分析:conv3 * 3(3 * 3 二维卷积滤波);correlate(对两个序列检测其相关性);err_diff(基于错误扩散的抖动方法,可消除像素点之间的明显差异);fdct(8 * 8 行列离散余弦变换);fft2(基 2 fft,广泛用于信号处理);fir(32-tap 有限冲击响应滤波器);idct(2 维 8 * 8 离散反余弦变换);irr(无限冲击响应滤波器,用于声音合成);me_fs(全搜索运动估计);rgb2yuv(将 RGB 图像转化为 YUV 图像)。因为软件流水对于程序的性能优化有着非常重要的作用,所以对程序的最

内层循环都采用了软件流水优化。

本文使用 Iscd 编译器[11] 对这些测试程序在图 1(a)所示的处理器结构上不考虑 GRF 即只针对 ALU 与 TORF 构成的处理器结构进行了编译，并对结果进行了统计。表 1 显示了各个程序对 TORF 的寄存器需求情况。从表 1 中可以看出，在不使用 GRF 的情况下，程序对 TORF 的寄存器需求较大，大部分程序的平均值在 5 ~ 7 之间，平均值的最大值超过了 9，同时由于分布式寄存器文件之间的负载不均衡，对单个 TORF 寄存器需求的最大值一般都比平均值高出很多，最大的需求甚至达到了 14。分布式寄存器文件的特点使得只要有一个寄存器文件的需求超过其容量限制就会产生溢出，因此寄存器文件的容量要大于程序对单个寄存器文件的最大需求。这需要设计容量为 8 甚至 16 的 TORF，而这显然不符合分布式与层次化寄存器文件的设计初衷，TORF 访问频繁，过大的容量会造成系统功耗的显著上升。

表 1 TORF 寄存器需求情况

Tab. 1 Register requirement of TORF

程序	平均值	最大值
conv3 * 3	6	8
correlate	5.5	7
err_diff	7.83	10
fdct	5.67	8
fft2	4.83	6
fir	6	7
idct	9.17	14
irr	6.67	10
me_fs	8	11
rgb2yuv	3.5	6

在编译时，编译器会按照程序的控制结构将程序划分为若干个基本块。通过进一步分析程序特征以及变量的生命周期可以发现，占用 TORF 的变量可以分为两类，其划分标准主要看其生命周期是否超出一个基本块的长度。一类变量的生命周期局限于某个基本块内部，称之为局部变量。另一类变量的生命周期跨越整个基本块或者多个不同的基本块。对第二类变量进行进一步分析可以将其进一步划分为两类。一类是产生和消费该变量的操作分别处于不同的基本块，考虑程序的特点，最常见的就是产生和消费它的操作分别处于循环的外部和内部，其生命周期将会跨越整个循环体，称之为全局变量。另一类变量主要是因为编译时采用了软件流水优化而产生的，产生和消费该变量的操作都处于循环所对应的基本块内，如果不采用软件流水，其生命周期必然局限于该基本块内部。在采用软件流水后，产生和消费他的操作分别被调度到软流水的不同栈(stage)内，因此需要额外的寄存器在不同栈之间保存该变量的值，使得最终的实际效果是他的生命周期跨越整个循环，这一类变量称之为软流水变量。图 2 显示了各个测试程序中全局变量、软流水变量、局部变量对 TORF 的需求比例。从图 2 中可以看出，对 TORF 需求较大的主要是全局变量、软流水变量，局部变量一般只占很小的比例。如果能够尽量消除全局变量与软流水变量，那么 TORF 的容量就可以减少到最小，从而降低功耗。

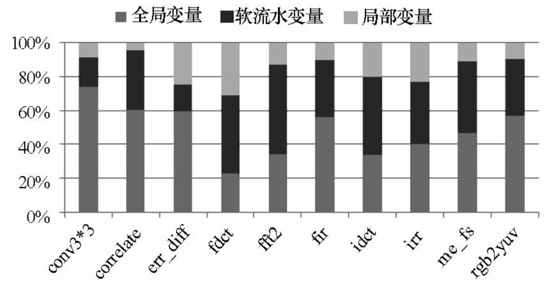

图 2 不同类型变量的比例

Fig. 2 The proportion of different kinds of variables

3 指令调度优化

基于第 2 节的分析结果，本文提出了面向分布式与层次化寄存器文件的指令调度优化算法，其主要思想是在编译时检测出对寄存器压力影响较大，即生命周期较长的全局变量以及软流水变量，并在合适的时机将其存放到 GRF 中，在其需要使用时再存回到 TORF 中。该算法如何与已有的 VLIW 调度、通信调度以及寄存器分配算法融合在一起也是一个令人头疼的问题。面向分布式寄存器文件的 VLIW 调度问题已经被证明是一个 NP 完全问题[11]，要想在此基础上在 VLIW 调度的同时就考虑如何对 TORF 甚至 GRF 进行分配太过复杂，难以找到一个时间、开销、性能都可以接受的算法。另一方面，如果在 VLIW 调度结束甚至 TORF 的分配已经确定的情况下再考虑如何利用 GRF，因为数据从 TORF 通过总线传到 GRF，再由 GRF 传回 TORF 需要一定的时间延迟，同时也要占用一定的寄存器文件端口、功能单元、总线等，必然会对原有的 VLIW 调度结果以及 TORF 的分配结果产生非常大的影响。这不仅会影响程序的性能，同时可能因为资源冲突使得根本无法找到合适的数据通路来实现数据在 GRF 与 TORF 之间的传输。因此我们的算法的原则是在 VLIW 调度的同时对各类变量进行相应的处理，同时仍

然采用 VLIW 调度与寄存器分配分开进行的两阶段算法以避免算法过于复杂、难以实现。

虽然采用 VLIW 调度与寄存器分配分开进行的两阶段算法,我们无法在 VLIW 调度时就确切地知道每一个变量会在何时存入哪个寄存器,但是根据基本的指令调度和变量生命周期的计算方法,可以对变量的生命周期进行定性的描述。例如,对于一个在循环外定义在循环内使用的全局变量而言,因为无法确切地知道该循环编译后生成的指令序列长度,因而也无法知道该变量确切的生命周期,然而我们可以确定的是,无论该循环对应的基本块长度有多长,该变量的生命周期都将跨越该循环,即在循环执行期间,他会一直占用一个寄存器。对于采用软件流水造成的寄存器需求上升,这类变量的分析也是一样的。

3.1　编译器视角变换

通信调度算法[11]要求,在一个变量的生产者与消费者之间要构成一个有效的通信路由,需要以下部分:功能单元输出端口、互联总线、寄存器文件输入端口、互联总线、功能单元输入端口等。在图 1(a)的结构下,GRF 作为 TORF 的下一层次的寄存器空间,并没有实际的功能单元与之相连接,通信调度算法无法有效地对其进行使用。为了能够利用 GRF 和算法的有效实现,我们不再把 GRF 看作第二层寄存器空间。如图 1(b)所示,在编译器中我们为每一个 GRF 增加一个逻辑上的功能单元(Logical Functional Unit, LFU),LFU 只负责从 GRF 中读取数据并将其发送到互联总线,其实际的执行通过指令码中的 GRF 地址实现。这样,GRF 与 TORF 一起组成了新的分布式寄存器文件结构,LFU 与 GRF 组成了一个与 ALU、TORF 结构一致的功能单元 - 寄存器文件组合,编译器可以将其统一对待,不需要更改复杂的通信调度算法。

3.2　全局变量优化

第 2 节已经说明,本文所述的全局变量并不是传统意义上的整个程序范围内都要使用的变量,而是指产生和消费它的操作分别处于不同的基本块的变量。因此,全局变量的检测和优化相对比较容易。图 3 给出了检测和优化全局变量的伪代码描述。在划分好基本块并建立了操作之间的依赖关系图(DAG 图)之后,编译器依次对所有基本块的所有操作进行遍历,若某操作的结果被其他基本块的操作所消费,那么这个结果变量就符合本文所述的全局变量的特征。

```
BasicBlockSepration();//划分基本块
ConstructDAG();//建立DAG图
//对所有的基本块进行遍历搜索
for(block0 = blocks.begin(); block0.valid(); block0 ++)
    //对基本块内的所有操作进行遍历搜索
    for(op0 = block->contains_ops.begin(); op0.valid(); ++op0)
        //对每个操作的输出结果进行遍历
        for(io0 = op0->ios[out].begin(); io0.valid(); ++io0)
            //对消费该结果的其他操作的输入进行遍历
            for(io1 = io0->connects.begin(); io1.valid(); io1++){
                op1 = io1->op;//op1为消费该结果的操作
                block1 = op1->block;//block1为op1所在基本块
                //若产生和消费某变量的2个操作处于不同的
                //基本块，则该变量就是本文所述的全局变量
                if(block0 != block1){
                    CreateCopyoperation(op2);//插入虚拟的copy操作
                    op2->FU = LFU; //只有LFU才能执行该copy操作
                    //copy操作包含在op1所在的基本块内
                    op2->block = block1;
                    //修改3个操作之间的依赖关系
                    DeleteConnection(io0, io1);
                    Connect(io0, op2->ios[in]);
                    Connect(op2->ios[out], io1);
                }
            }
VLIWSchedule();//进行VLIW调度
```

图 3　全局变量优化算法

Fig. 3　The optimization of global variable

对全局变量的处理是在 VLIW 调度之前进行的。在检测出某个变量为全局变量后,编译器在消费该变量的操作所在的基本块内插入一条只能在 LFU 上执行的虚拟的 copy 操作,并将原来的操作之间的数据依赖性关系删除,改为通过该 copy 操作来传递依赖性关系。在所有的全局变量都优化完毕,将新插入的 copy 操作与其他操作同样看待,进入下一步的 VLIW 调度过程,所不同的是它只能在 LFU 上执行,因此它不会对后续的 VLIW 调度算法造成影响。

3.3　软流水变量优化

与全局变量不同,对软流水变量的检测、优化只能在 VLIW 调度时进行,因为不进入 VLIW 调度的过程,无法判断某个变量是否是软流水变量。图 4 给出了能够检测和优化软流水变量的指令调度算法的伪代码描述。

按照优先级选定一个操作(op0)后,确定该操作能够流出执行的最早指令周期。然后选定一个功能单元并尝试将该操作调度到能够流出执行的最早指令周期(cycle)。因为采用了分布式寄存器文件结构,此时需要进行通信调度来确定能否找到有效的路由来实现该操作与其所依赖的操作之间的数据传输。如果通信调度成功,则该操作能够在选定的指令周期和功能单元上流出执行。若通信调度失败,则尝试将该操作调度到另一个功能单元流出执行并进行通信调度。若当前指令周期所有的功能单元都不能使得通信调度成

```
//对所有未调度的操作按照优先级进行调度
op0 = getHighestPriorityOp(block->unscheduledops);
Cycles(start); //确定该操作能够调度的最早指令周期
for(cycle0 = start; ; ++cycle0){ //选择能够最早流出的cycle
    EvaluateFUs(fus); //对所有的功能单元进行优先级排序
    for(fu0 = fus.begin(); fu0.valid(); ++fu0){
        ScheduleFU(fu0, cycle0); //将op0调度到fu0上
        if(ScheduleComm(op0, cycle0, fu0)){ //进行通信调度
            if(!pipeline) //未采用软流水，当前操作调度成功
                return success;
            else //若采用了软流水则需要检测是否会产生软流水变量
                //对该操作所消费的变量的生产者依次进行检测
                for(io0 = op0->ios[in].begin(); io0.valid(); ++io0){
                    //op1的结果被op0消费且op1已经调度完毕
                    op1 = io0->connects->op;
                    cycle1 = op1->cycle;//cycle1为op1流出执行的周期
                    //计算两个操作所在的软流水栈，除法结果向下取整
                    //II为当前调度的软流水迭代间隔（iteration interval）
                    int stage0 = cycle0/II;
                    int stage1 = cycle1/II;
                    //两个操作处于不同的软流水栈，产生软流水变量
                    if(stage0 != stage1){
                        UnSchedule(op0);//释放对当前操作的调度
                        CreateCopyoperation(op2); //插入虚拟的copy操作
                        op2->FU = LFU; //只有LFU才能执行该copy操作
                        op2->block = block;
                        //修改3个操作之间的依赖关系
                        DeleteConnection(op0, op1);
                        Connect(op1->ios[out], op2->ios[in]);
                        Connect(op2->ios[out], op0->ios[in]);
                        //对copy操作能够流出的指令周期进行限制
                        start2 = stage0 * II;
                        end2 = (stage0 + 1) * II;
                        Schedule(op2, start2, end2);//立即调度copy操作
                        Schedule(op0);//然后对op0重新进行调度
                    }
                }
        }
    }
}
```

图 4　软流水变量优化算法

Fig. 4　The optimization of software pipelining variable

功，则将该操作推迟一个指令周期流出执行，直至通信调度成功为止。

如果采用了软件流水优化，在通信调度成功后，需要检测是否会产生软流水变量。在进行 VLIW 调度时，当前调度软流水的迭代间隔(Iteration Interval，II)是确定的。假设 op0 依赖于 op1 的结果，通过将 op0 和 op1 流出的指令周期分别与 II 相除并向下取整即可得出他们所处的软流水栈数，若两个操作不处于相同的软流水栈，则需要额外的寄存器在不同栈之间保存该变量的值，产生软流水变量。此时需要释放并终止对当前操作 op0 的调度，在当前基本块内插入一条只能在 LFU 上执行的虚拟的 copy 操作，处理方式和全局变量的处理方式一致。然后立即对插入的 copy 操作进行调度。对 copy 操作进行调度时需要对其所能流出执行的指令周期进行一定的限制，它应当处于 op0 刚才调度时所处的软流水栈，这样使得 copy 操作与操作 op1 处于不同的软流水栈，在不同软流水栈之间保存变量值的任务就会落在与 LFU 相连的 GRF 中，减轻 TORF 的压力。在 copy 操作调度成功后再对原来的操作 op0 进行调度。

3.4　寄存器分配示例

VLIW 调度结束之后对寄存器进行分配。3.1 节已经说明在编译器的视角中，GRF 与 TORF 一起组成了新的分布式寄存器文件结构，两者不加区分。因此在对寄存器进行分配时，TORF、GRF 也不再区分，都统一采用经典的图着色算法进行。为了说明本文的指令调度算法对寄存器使用情况产生的影响，图 5 给出了对全局变量、软流水变量进行优化后的指令调度、通信调度以及寄存器分配的结果。因为全局变量与软流水变量具有某些相似的特性，我们将其放在一张图中描述。

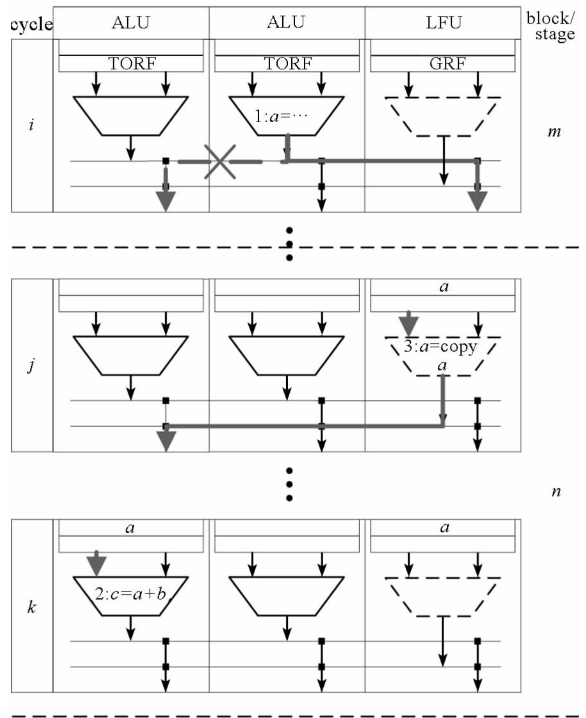

图 5　寄存器分配示例

Fig. 5　Example of register allocation

如图 5 所示，操作 2($c = a + b$)依赖于操作 1($a = \cdots$)。假设操作 1 处于基本块(block)m，操作 2 处于基本块 n(m! $= n$)，则操作 1 的结果变量 a 就是本文所述的全局变量。此时在基本块 n 中插入一条只能在 LFU 上执行的虚拟的 copy 操作——操作 3(a = copy a)，并修改 3 个操作之间的数据依赖关系，然后进行 VLIW 调度。最终的指令调度与通信调度的结果如图 5 所示。操作 1、操作 2、操作 3 分别在 cycle i、k、j 流出执行。操作 1 与操作 2 在 ALU 上执行，操作 3 在 LFU 上执行。因为插入了 copy 操作，原来的从操作 1 到操作 2 的通信路由被删除掉，通信调度的结果建立了一条新的路由。变量 a 在 cycle i 产生并被

送入到与 LFU 相连的 GRF 中,然后在执行操作 3 即 cycle j 时从 GRF 中取出并送入 TORF 供 ALU 使用。根据最基本的变量生命周期的计算方法我们可以知道,变量 a 会从 cycle i 结束开始占用 GRF 的一个寄存器,直到基本块 n 所对应的循环执行完毕才会释放该寄存器。变量 a 从 cycle j 结束开始占用 TORF 的一个寄存器,在操作 2 执行时即 cycle k 就释放该寄存器。基本块 n 的其他周期该寄存器仍可用于存放其他的变量。通过这种方式,全局变量对 TORF 的绝大部分需求都被转移到 GRF 中,大大减轻了对 TORF 的压力。对软流水变量的分析也是一样的。

4　性能评测

为了评价本文提出的指令调度算法的效果,本文对所选的测试程序在图 1(a)所示的处理器中进行了测试。因为之前采用分布式与层次化寄存器文件的微处理器的指令调度与寄存器分配主要依赖于程序员的手工分配优化[7],与本文的编译器实现没有可比性。所以我们将结果与使用 Iscd 编译器在图 1(a)所示的处理器结构上不使用 GRF 的编译结果进行对比。所有的寄存器文件都各有一个读端口,一个写端口。

4.1　程序性能与寄存器需求

表 2 显示了不同情况下各个测试程序的 IPC (Instruction Per Cycle)值。从表 2 中可以看出,在两种情况下程序都可以获得很好的性能。与不使

表 2　程序的 IPC 值

Tab. 2　IPC of the programs

	不使用 GRF	使用 GRF	降低
conv3 * 3	2. 25	2. 01	10. 67%
correlate	2. 41	2. 15	10. 78%
err_diff	2. 52	2. 41	4. 37%
fdct	2. 27	2. 08	8. 37%
fft2	2. 23	2. 17	2. 69%
fir	2. 41	2. 11	12. 44%
idct	2. 63	2. 51	4. 56%
irr	2. 38	2. 21	7. 14%
me_fs	2. 59	2. 31	10. 81%
rgb2yuv	2. 33	2. 11	9. 44%
平均值	2. 402	2. 207	8. 13%

用 GRF 相比,通过本文的指令调度算法来利用 GRF 使得程序的性能略有下降,平均下降了 8. 13% 左右。这主要是因为插入的 copy 操作除了要占用虚拟的 LFU 单元外,还要占用一定的总线及寄存器文件端口,这必然会影响原有的调度结果,影响程序的性能。另外为了消除软流水变量,在软流水循环每个 stage 起始的一两个 cycle 中,ALU 单元主要都是从 GRF 中读取数据,可能无法执行有效的操作,降低了利用率。然而即便如此,本文的指令调度算法仍可以获得很好的程序性能,平均 IPC 值达到了 2. 207,也就是说相对于 3 个 ALU 的峰值性能,其平均利用率达到了 73. 6% 以上,取得了很好的效果。

表 3 显示了在使用本文的指令调度算法后程序寄存器的需求。从表 3 中可以看出只需要将 TORF 的容量设置为 4 项,GRF 的容量设置为 16 项即可满足所有程序的需求。从表 1 与表 3 的对比中可以看出,在使用本文的指令调度算法后,程序对 TORF 的需求大幅降低,原来对 TORF 的压力现在都转移到 GRF 中,GRF 需求较大。但是因为对 GRF 的访问频率不高,程序对 GRF、TORF 的不同需求符合分布式与层次化寄存器文件结构的设计理念,可以发挥其在功耗上的优势。

表 3　新算法下的寄存器需求

Tab. 3　Register requirement under new algorithm

	TORF		GRF	
	平均值	最大值	平均值	最大值
conv3 * 3	2. 17	3	11. 5	12
correlate	2. 5	3	9	10
err_diff	3. 17	4	10. 5	11
fdct	2. 17	3	8	9
fft2	2. 5	3	6	7
fir	2. 5	3	8. 5	9
idct	3	4	12	13
irr	3. 5	4	10. 5	11
me_fs	2. 67	3	13	13
rgb2yuv	1. 33	2	5. 5	6

4.2　功耗分析

图 6 列出了各个程序在不同情况下对寄存器文件的读写频率。其中 TORF_R、TORF_W 分别代表在不使用 GRF 的情况下程序对 TORF 的读写频率,TORF_R_new、TORF_W_new 分别代表采用本文的指令调度算法程序对 TORF 的读写频率,GRF_R、GRF_W 分别代表采用本文的指令调度算法程序对 GRF 的读写频率。从图 6 中可以看出,在两种情况下对 TORF 的读写频率之和差别不大,而两种情况下 TORF 容量不同,每次访问不同的功耗则使得程序的整体功耗有较大的差别。从图 6 中还可以看出,采用本文的指令调度算法后,TORF 的访问频率较高,而 GRF 的访问频率较低,这也符合分布式与层次化寄存器文件的

设计初衷。

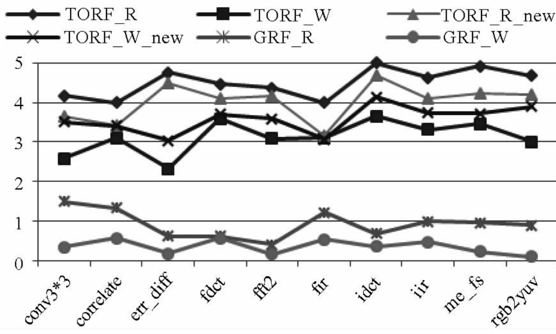

图6 寄存器文件访问频率

Fig. 6 Access frequency of register file

为了评价两种情况下处理器的功耗差别，本文利用EDA工具对图1(a)所示的处理器结构进行了初步的评估，相关参数如表4所示。所有数据都是在500MHz频率，1.0V电压下基于TSMC 65nm高性能标准单元库获得。

表4 详细参数

Tab. 4 Detailed parameter

	概要描述	能量消耗
ALU	32位操作，如加、乘、除、逻辑、移位操作等	乘法操作6.2pJ/op，其他操作3.4 pJ/op
寄存器文件	4项，读写端口各1个	1.69pJ/读 1.67pJ/写
	16项，读写端口各1个	5.33pJ/读 5.4pJ/写

本文为寄存器文件建立了如下的功耗模型。在不使用GRF的情况下，TORF的容量为16项，平均每拍寄存器访问能量消耗可由式(1)计算得出。其中N_{TORF_R}、N_{TORF_W}分别为平均每拍读、写TORF操作的数目；E_{TORF_R}、E_{TORF_W}分别为每次读、写TORF的能量消耗，对应于表4中16项寄存器文件的数据。与之相对应，在使用本文的指令调度算法来使用GRF的情况下平均每拍寄存器访问能量消耗可由式(2)计算得出。此时TORF的容量设置为4项，GRF的容量设置为16项，其中E_{TORF_R}、E_{TORF_W}分别为每次读、写TORF的能量消耗，对应于表4中4项寄存器文件的数据；N_{GRF_R}、N_{GRF_W}分别为平均每拍读、写GRF操作的数目；E_{GRF_R}、E_{GRF_W}分别为每次读、写GRF的能量消耗，对应于表4中16项寄存器文件的数据。对于两种情况下处理器平均每拍的能量消耗可以粗略地通过式(3)、式(4)计算得出。其中IPC为每拍的ALU操作数目，分别与表2中的数据相对应，E_{ALU}为每次ALU操作的能量消耗，与表4中的数据相对应。

$$E_{withoutGRF} = N_{TORF_R} \times E_{TORF_R} + N_{TORF_W} \times E_{TORF_W} \quad (1)$$

$$E_{withGRF} = N_{TORF_R} \times E_{TORF_R} + N_{TORF_W} \times E_{TORF_W} + N_{GRF_R} \times E_{GRF_R} + N_{GRF_W} \times E_{GRF_W} \quad (2)$$

$$E_{P_withoutGRF} = \mathrm{IPC} \times E_{ALU} + E_{withoutGRF} \quad (3)$$

$$E_{P_withGRF} = \mathrm{IPC} \times E_{ALU} + E_{withGRF} \quad (4)$$

根据上面4个公式，表5给出了在不同情况下平均每拍的寄存器访问能量消耗对比以及平均每拍的处理器的能量消耗对比。从表5中可以看出，与不使用GRF相比，通过本文的指令调度算法来使用GRF，寄存器文件的访问开销以及整个处理器的能量消耗都大幅降低。访问寄存器文件的能量开销降低了39.3%～59.6%，平均降低了51.4%；处理器的能量消耗降低了32.4%～49.2%，平均降低了43.2%。综合以上测试结果可以看出，本文提出的指令调度算法能够在性能略有下降的情况下，充分利用好处理器中第二层的寄存器文件空间，使得处理器的功耗优势得到很好的发挥。

表5 能量消耗对比

Tab. 5 Comparison of energy consumption

单位：pJ/cycle	寄存器访问能量消耗			处理器能量消耗		
	不使用GRF	使用GRF	减少	不使用GRF	使用GRF	减少
conv3 * 3	36.12	21.92	39.31%	45.87	31.00	32.42%
correlate	38.06	21.74	42.88%	48.46	30.69	36.67%
err_diff	37.90	17.46	53.91%	46.80	25.70	45.09%
fdct	43.14	19.55	54.69%	54.94	30.37	44.72%
fft2	39.95	16.14	59.60%	49.41	25.38	48.63%
fir	38.06	19.96	47.56%	48.46	28.22	41.77%
idct	46.39	20.50	55.81%	56.28	29.93	46.82%
irr	42.54	21.03	50.56%	53.77	31.20	41.98%
me_fs	44.82	19.79	55.85%	53.63	27.27	49.15%
rgb2yuv	41.08	19.03	53.77%	51.81	28.69	44.62%
平均值			51.39%			43.19%

5　相关工作

斯坦福大学的 EEC 项目率先提出了面向嵌入式应用的能耗优先的微处理器[7],采用了分布式与层次化的寄存器文件结构,但是其实验结果是通过执行手工优化的汇编代码得到的,这对应用开发人员是一个很大的挑战。Zalamea 等提出了面向 VLIW 处理器的两级层次化的寄存器文件组织结构[10],不过其两层寄存器文件采用的都是集中式寄存器文件。Mattson 等提出了面向分布式寄存器文件的通信调度[11],但是对于分布式与层次化的寄存器文件结构,数据如何分配则是一个新的问题。之前的调度算法主要集中在将操作调度到合适的功能单元,然后选择合适的 cycle 流出执行,并不考虑寄存器分配的问题。Barany 等提出了一种优化的合并的指令调度和寄存器分配算法[9],首先进行一定程度的指令调度,然后根据寄存器分配的结果需求对指令调度进行调整以达到对寄存器的利用效率最优化,但他不能从理论上避免最坏效果的产生,有时甚至可能产生寄存器利用率很低的结果。

6　结论

高性能嵌入式应用对处理器提出了越来越高的性能和功耗需求。为了提高性能,编译器一般都会采用循环展开、软件流水等优化措施来开发程序的 ILP,这会增加寄存器文件的压力。另一方面,为了降低功耗,目前的微处理器中采用了分布式与层次化的寄存器文件结构,这给编译器的设计带来了新的挑战。第一层 TORF 的极小容量使得除了要考虑指令调度、通信调度外,还要考虑两层寄存器文件之间的数据转移。通过分析变量的生命周期的特征,将其划分为全局变量、软流水变量、局部变量三种不同类型的变量。通过在恰当的时机插入虚拟的 copy 操作并进行指令与通信调度,为对寄存器需求较大的全局变量与软流水变量构建新的数据传输路由,将对 TORF 的寄存器压力转移到第二层的 GRF 中去。实验结果证明,相比于不使用 GRF,在程序性能平均只降低约 8% 的情况下,本文的指令调度算法使得访问寄存器的能量消耗平均降低了 51% ,使得处理器的能量消耗平均降低了 43% 。

参考文献(References)

[1] Silven O, Jyrkka K. Observations on power-efficiency trends in mobile communication devices [J]. EURASIP Journal of Embedded Systems, 2007(1):17.

[2] Who M, et al. The next generation challenge for software defined radio [C]//Proceedings of the 7th International Conference on Systems, Architectures, Modeling, and Simulation, 2007:343 –354.

[3] Halfhill T R. MIPS threads the needle [R]. Microprocessor Report, 2006, 20(2):1 –8.

[4] Rattner J R. Tera-scale computing: a parallel path to the future [EB/OL]. http://software. intel. com, 2007.

[5] Kogge P, et al. ExaScale computing study: technology challenges in achieving exascale systems [R]. DARPA IPTO, 2008.

[6] Dally W J, Balfour J, Black-Schaffer D, et al. Efficient embedded computing[J]. IEEE Computer, 2008, 41(7): 27 –32.

[7] Balfour J, Dally W J, et al. An energy-efficient processor architecture for embedded systems [J]. IEEE Computer Architecture Letters, 2008, 7(1): 29 –32.

[8] 杨乾明, 伍楠, 管茂林, 等. ET:一种能耗有效的高性能嵌入式处理器[J]. 国防科技大学学报, 2011, 33(6): 24 –30.
YANG Qianming, WU Nan, GUAN Maolin, et al, ET: an energy-efficient processor architecture for embedded tera-scale computing[J]. Journal of Nationtod University of Defense Technology, 2011, 33(6): 24 –30. (in Chinese)

[9] Barany G, Krall A. Optimistic integrated instruction scheduling and register allocation[C]//2010 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2010.

[10] Zalamea J, Llosa J, Ayguad'e E, et al. Two-level hierarchical register file organization for VLIW processors[C]// The 33rd International Symposium on Microarchitecture (MICRO), 2000: 137 –146.

[11] Mattson P. A programming system for the imagine media processor, dept. of electrical engineering [D]. Stanford University, 2002.