基于簇的寄存器堆功耗管理方法

孙含欣, 佟 冬, 袁 鹏, 程 旭
(北京大学微处理器研究开发中心, 北京 100871)

摘 要: 本文采用软硬件协同设计技术, 提出以寄存器簇为粒度对嵌入式处理器寄存器堆进行功耗管理的方法. 在软件方面, 面向寄存器簇的编译优化使循环程序段中寄存器的编号尽可能相邻; 在硬件方面, 采用寄存器簇缓冲器过滤对寄存器堆的访问并降低其动态功耗, 采用基于寄存器簇的动态电压调节电路和门控预充电路降低存储单元和位线的泄漏功耗. 实验结果表明, 本文方法将寄存器堆的总功耗降低约 44.7%, 比传统方法达到了功耗、面积和延迟的更优折衷.

关键词: 嵌入式处理器; 寄存器堆; 寄存器簇; 动态功耗; 泄漏功耗
中图分类号: TP302.7 文献标识码: A 文章编号: 0372-2112 (2008) 02-0278-07

Cluster-Based Power Management Mechanism for Register Files

SUN Han-xin, TONG Dong, YUAN Peng, CHENG Xu
(*Microprocessor Research & Development Center of Peking University, Beijing 100871, China*)

Abstract: In this paper, we propose the cluster-based power management mechanism, which uses register cluster as the power management granularity. Specifically, the cluster-oriented compiler makes the register numbers in loops as continuous as possible to offer more opportunities for run-time power management, and the cluster-based run-time power manager employs a register cluster buffer to filter accesses to the register file for dynamic power saving. The dynamic voltage scaling and gated precharge circuits are also well utilized to reduce the leakage of bitcells and bitlines. Averagely, the total register file power is reduced by 44.7%. Compared with traditional approaches, the hardware/software co-design approach proposed in this paper achieves better power, area and delay tradeoffs for register files in embedded processors.

Key words: embedded processor; register file; register cluster; dynamic power; leakage power

1 引言

近年来, 能耗有效性已成为嵌入式处理器设计的最重要因素之一. 在嵌入式处理器中, 每个周期寄存器堆的多个端口往往同时被访问, 引起较为可观的功耗开销. 在 M•CORE 处理器中, 寄存器堆占据了处理器约 16% 的功耗和数据通路约 42% 的功耗[1]. 随着嵌入式处理器逐渐向多处理器系统芯片(MPSoC) 发展, 最新研究结果表明, MPSoC 中的每个处理器适合采用 5 到 8 级流水线的按序执行结构, 并可通过复制寄存器堆支持多线程[2], 而这很可能使寄存器堆在嵌入式处理器中的功耗比重进一步增大. 因此, 对寄存器堆进行有效的功耗管理将对提高未来嵌入式处理器的能耗有效性具有重要意义.

一种典型的嵌入式处理器流水线包含以下几个阶段: IF1, IF2 阶段进行取指和分支预测, ID 阶段进行指令译码, RF 阶段读寄存器堆, EX1, EX2, MEM 阶段进行 ALU 运算和访存操作, WB 阶段回写寄存器堆. 由于没有进行寄存器换名, 寄存器堆中每一个寄存器对应一个体系结构寄存器.

本文针对典型嵌入式处理器提出寄存器堆的功耗管理方法. 与传统的单纯从物理结构出发降低寄存器堆功耗的方法不同, 本文采用软硬件协同设计方法, 通过将编译优化、微体系结构设计与电路设计相结合来降低寄存器堆功耗. 首先, 通过对不同功耗管理粒度的分析, 选择以若干相邻寄存器的集合(寄存器簇)[3] 为粒度进行寄存器堆功耗管理. 在编译器中优化寄存器分配策略, 使循环程序段中访问的寄存器尽可能集中到少数几个寄存器簇. 在此基础上, 采用一个容量相对较小的簇缓冲器保存循环中频繁访问的簇, 以减少对寄存器堆的访问及相应的动态功耗; 当对寄存器堆的访问被簇缓冲器过滤时, 采用动态电压调节[4] 电路和门控预充电路[5] 来减少存储单元和位线的泄漏功耗.

收稿日期: 2007-08-06; 修回日期: 2007-10-23
基金项目: 国家 863 高技术研究发展计划(No. 2006AA010202)

2 相关工作

寄存器堆低功耗设计的相关技术可主要分为四类: 分体技术、缓冲技术、指令重用技术和动态电压调节技术. 文献[6]根据应用程序访问寄存器堆的剖视信息将其划分为两个或多个体, 这种方法主要适用于面向特定应用的嵌入式处理器. 文献[7]面向超标量处理器中较大的物理寄存器堆采用缓冲技术, 将最近访问的寄存器保存在缓冲器中, 从而减少对寄存器堆的访问. 文献[8]通过重用一些频繁执行的指令所需的源操作数来减少对寄存器堆的访问. 文献[9, 10]采用动态电压调节技术降低寄存器堆的泄漏功耗, 两者主要差异是动态电压调节的粒度不同: 文献[9]的粒度为单个寄存器, 而文献[10]的粒度为寄存器体.

在国内研究工作中, 文献[11]分析了低功耗编译技术并对低功耗编译的新方向作出预测, 文献[12]在算法层对处理器中存储部件的泄漏功耗进行了优化. 本文将电路设计、微体系结构设计与编译优化等技术结合使用, 使嵌入式处理器寄存器堆达到功耗、面积和延迟的较优折衷.

3 寄存器簇功耗管理粒度的提出

在选择嵌入式处理器寄存器堆的功耗管理粒度时, 不仅要考虑寄存器堆功耗节省的效果, 还需考虑功耗管理方法自身引起的功耗、面积和延迟开销. 以单个寄存器为管理粒度的方法采用一个相对较小的缓冲器保存若干最近访问的寄存器的值和地址, 用以过滤对寄存器堆的访问. 但是, 为了判断待访问的寄存器是否在缓冲器中, 需将待访问寄存器的地址与缓冲器中保存的寄存器的地址作全相联比较, 并且缓冲器容量失效时其表项需被替换, 这均会造成额外的功耗开销. 分体技术以寄存器体为功耗管理粒度, 重新划分寄存器堆物理结构. 与以单个寄存器为粒度的方法相比, 分体技术引起的额外功耗开销相对较小. 但是, 分体技术会增大寄存器堆的面积和延迟. 过度的分体虽然会降低访问每个体的功耗和延迟, 却会增大从各个体之间进行数据选择的功耗和延迟.

为了避免以上两种功耗管理粒度的不足, 本文重新考虑寄存器堆的功耗管理粒度, 选择以"寄存器簇"为粒度进行功耗管理, 使相应的功耗管理方法以更小的代价获得较大幅度的功耗节省. 寄存器簇的概念已经在基于簇的处理器设计方法[3]中提及. 一个寄存器簇(以下简称"簇")是由 N 个地址相邻寄存器组成的集合, 其中 N 为 2 的幂次. 假定处理器有 M 个体系结构寄存器($R_0, R_1, \cdots, R_{M-1}$), 则寄存器堆从逻辑上被分为 M/N 个簇, 编号分别为 0, 1, ……, ($M/N-1$), 其中编号为 i 的簇 RC_i 为:

$$RC_i = \{R_{N*i}, R_{N*i+1}, R_{N*i+2}, \cdots, R_{N*i+(N-1)}\}$$

假定指令编码中寄存器地址域宽度为 $\log_2 M$, 则其中的高 $\log_2(M/N)$ 位表示簇的编号, 低 $\log_2 N$ 位表示簇内偏移. 采用固定数值的 N 可以使寄存器地址对应的簇编号易于识别, 节省功耗管理器识别簇编号时的功耗. 在实际处理器设计中, 对 N 的选取过程需要对寄存器堆访问延迟、面积和功耗作出权衡.

以簇为粒度的寄存器堆功耗管理方法包括两部分: 面向簇的编译优化和基于簇的运行时刻功耗管理. 以下将对这两部分分别作叙述.

4 面向簇的编译优化

面向簇的编译优化的目标是在不影响原有寄存器分配算法性能的前提下, 尽可能将循环程序段内使用的变量指定到相邻寄存器中, 从而使循环运行时访问的寄存器集中到少数几个簇, 便于在运行时刻以簇为粒度进行功耗管理.

编译器对一个函数的寄存器分配过程通常包括分配和指定两个阶段[13]. 为了不影响原有寄存器分配算法的性能, 本文不改变分配阶段生成的变量组, 仅在指定阶段优化为变量组指定寄存器的方法. 由于编译器常常可以较好的预测循环在动态运行时的指令数[14], 本文根据预期动态运行的指令数将一个函数中的所有循环排序. 利用贪心算法, 首先为预期运行指令数最大的循环中的所有变量组指定相邻的寄存器. 接下来, 对于其余的所有循环中预期运行指令数最大的循环, 如果仍有未指定寄存器的变量组, 将继续为它们指定编号相邻的寄存器. 依此类推, 直至该函数中预期运行指令数最低的循环为止. 对于一个循环内部使用的变量组, 将按照在循环内的使用次数排序, 并按照使用次数由大到小的顺序依次为变量组指定寄存器.

例如, 假设一个函数中存在 3 个循环 $L1, L2, L3$, 编译器预测其运行的指令数为 $I1, I2, I3$, 且 $I1 > I2 > I3$. 则编译器首先为 $L1$ 中的变量组指定寄存器, 然后依次是 $L2$ 和 $L3$. 假定 $L1$ 中的变量组为 a, b, c, $L2$ 中的变量组为 b, d, e, $L3$ 中的变量组为 c, d, f, 且在 $L1$ 中的使用次数 $a > b > c$, 在 $L2$ 中的使用次数 $d > e$, 则编译器在为 $L1$ 中的变量组指定寄存器时, 指定顺序依次为 a, b, c. 当编译器为 $L2$ 中的变量组指定寄存器时, 由于 b 已经在 $L1$ 中指定了寄存器, $L2$ 中仅需再为 d 和 e 指定寄存器, 顺序依次为 d, e. 最后, 需为 $L3$ 中的变量组 f 指定寄存器. 于是, 相邻寄存器($Rn, Rn-1, Rn-2, Rn-3, Rn-4, Rn-5$)被依次指定到变量组 a, b, c, d, e, f 上.

如果函数中仅有一个循环, 则上述寄存器指定方法能够使循环中访问的寄存器尽可能集中到少数几个

簇. 如果函数中存在多个循环, 则上述方法偏袒预期运行指令数大的循环. 循环的预期运行指令数越大, 其中访问的寄存器编号相邻的机会越大.

5 基于簇的运行时刻功耗管理

基于簇的运行时刻功耗管理的基本思想是: 在嵌入式应用程序运行过程中动态识别循环内部若干频繁访问的簇并将其保存在簇缓冲器中. 当指令需访问的簇已被簇缓冲器保存时, 可以避免对寄存器堆的访问及相应的动态功耗, 并采用动态电压调节[4] 电路和门控预充电路[5]降低寄存器堆存储单元和位线的泄漏功耗. 当发现循环执行过程中某些簇不会被访问时, 也在动态电压调节[4] 电路中将存储单元的供电电压调低, 以降低存储单元的泄漏功耗.

5.1 功耗管理器工作原理

为支持以簇为粒度的功耗管理, 本文首先对 SEPAS-Filter[15] 作扩展, 用于在运行时刻探测循环程序段. 在文献[15]中 SEPAS-Filter 用于缓冲最近执行的分支指令的相关信息以降低分支预测器的功耗. 本文在 SEPAS-Filter 的每个表项中增加一个 Timer 域和一个 Loop 域. Timer 域为饱和计数器, 其宽度由循环探测的时间间隔阈值决定. 假设时间间隔阈值为 T, 则 Timer 域的宽度为 $\log_2 T$. Loop 域宽度为 1, 用于记录从哪条分支指令开始进入一个循环程序段.

利用扩展的 SEPAS-Filter, 功耗管理器将区分循环和非循环程序段, 其状态图如图 1 所示. 在初始时刻, 处理器运行非循环程序段, 功耗管理器处于顺序执行状态. 如果在该状态下发现 SEPAS-Filter 中某个表项的 Taken 域饱和, 就说明可能由此分支指令开始执行一个循环. 这时, 功耗管理器转换到循环探测状态.

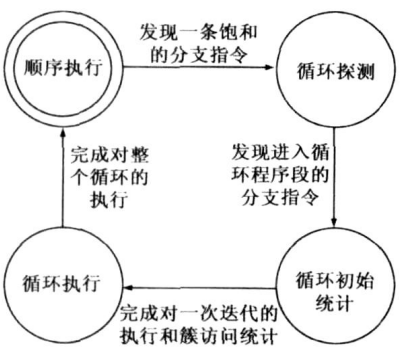

图 1 基于簇的寄存器堆功耗管理器状态转换

在循环探测状态下, 功耗管理器使用 SEPAS-Filter 中的 Timer 域统计分支指令连续两次饱和之间的时间间隔. 当发现某个分支指令所在表项中的 Taken 域为饱和时, 将 Timer 域清零并开始计时, 直至下次发现该分支指令的 Taken 域为饱和时停止计时. 在对各个分支指令计时的过程中, 若发现某个分支指令所在表项中的 Timer 域的计时结果小于阈值时间 T, 则功耗管理器推断由此分支指令进入一个循环执行的程序段, 并且该循环的每次迭代时间小于 T. 于是, 功耗管理器将此分支指令所在表项的 Loop 域设置为 1. 如果所有分支指令所在表项中的 Timer 域的计时结果均不小于 T, 则功耗管理器选择由 Timer 域计时结果最先达到或超过 T 的分支指令进入的循环程序段, 并将此分支指令所在表项的 Loop 域设置为 1. 在探测到进入循环的分支指令后, 功耗管理器立即转换到循环初始统计状态.

在循环初始统计状态下, 由于已经将 SEPAS-Filter 中某条分支指令所在表项的 Loop 域设置为 1, 到下次遇到该分支指令时, 处理器恰好完成了对循环的一次迭代的执行. 在处理器执行这一次迭代过程中, SEPAS-Filter 中的 Timer 域用于记录各个簇的访问次数.

在完成一次迭代的执行和簇访问统计后, 处理器继续执行循环, 功耗管理器从循环初始统计状态转换到循环执行状态, 并根据循环初始统计状态下统计的簇访问次数, 将一次迭代中少数最频繁访问的簇保存在簇缓冲器中. 这样, 当处理器继续执行循环时, 在每次迭代中频繁访问的簇始终被保存在簇缓冲器中.

当发现 SEPAS-Filter 中 Loop 域标记为 1 的表项中的分支指令实际执行结果为 not taken 或者该表项被替换时, 功耗管理器推测处理器很可能已完成对相应循环程序段的执行. 于是, 功耗管理器返回顺序执行状态.

嵌入式应用程序中循环体的静态指令数往往较少[16]. 因此, 循环探测的时间间隔阈值 T 可以较小, 相应的 Timer 域宽度不大. 由于在 SEPAS-Filter 中 Timer 和 Loop 域的宽度比其他域的宽度要小很多, 扩展后的 SEPAS-Filter 在降低分支预测器功耗的同时, 自身功耗变化较小. 又由于 Timer 域仅在循环探测和循环初始统计状态下各计数一次, 对整个循环执行过程中寄存器堆的功耗影响较小.

5.2 降低寄存器堆动态功耗

为降低寄存器堆动态功耗, 簇缓冲器保存少数频繁访问的簇, 以减少对寄存器堆的访问. 如图 2 所示, 在 ID 阶段对指令作译码得到待访问的簇编号, 如果该簇已被簇缓冲器保存, 就不必在 RF 阶段访问寄存器堆, 从而降低寄存器堆的动态功耗. 如果在 ID 阶段发现该簇尚未被簇缓冲器保存, 则可在 RF 阶段访问寄存器堆, 而不会由于簇缓冲器失效造成性能损失. 写寄存器的缓冲过程与读寄存器基本相似, 差别是当簇缓冲器表项替换时需将其中的寄存器值回写寄存器堆. 当功耗管理器处于顺序执行状态、循环探测状态和循环初始统计状态时, 簇缓冲器中保存程序调用和返回时使用的寄存器簇, 以减少函数调用和返回过程中对寄存

器堆的访问. 当功耗管理器处于循环执行状态时, 由于在循环初始统计状态下已统计出一次迭代中少数频繁访问的簇, 簇缓冲器将固定保存这些簇, 直至处理器完成对整个循环的执行为止.

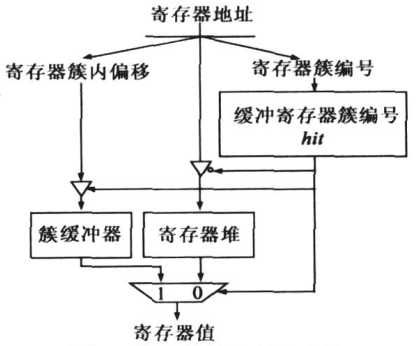

图 2　基于簇的读缓冲方法

5.3　降低寄存器堆泄漏功耗

降低寄存器堆泄漏功耗包括降低存储单元的泄漏功耗和降低位线的泄漏功耗两方面.

为降低存储单元的泄漏功耗, 在寄存器堆的每个簇中采用动态电压调节[4]电路, 通过调节每个存储单元的供电电压来减小泄漏电流. 为描述简单起见, 假设寄存器堆具有 1 个读端口和 1 个写端口, 则其中一个簇的动态电压调节电路如图3 所示. 簇中所有存储单元采用相同的供电电压. 在工作模式下, 簇中每个存储单元的供电电压均为正常电压 V_{DD}, 可随时接受读写操作. 若该簇已被簇缓冲器保存, 或者在循环初始统计状态下发现其在循环中的访问次数为 0, 则将其转换到 *Drowsy* 模式, 其供电电压被调低为 $V_{DD}Low$, 以降低簇中每个存储单元的泄漏电流及相应的泄漏功耗开销.

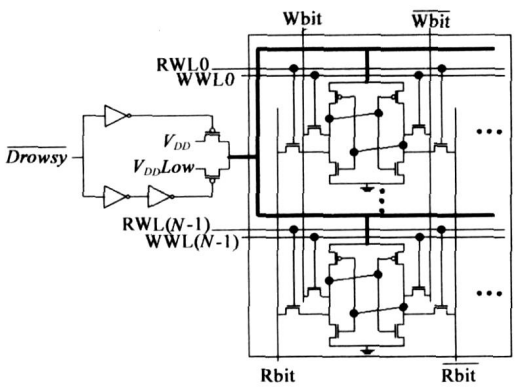

图 3　以簇为单位的动态电压调节电路

为降低寄存器堆位线的泄漏功耗, 对寄存器堆的每条位线采用门控预充电路[5]. 若在 ID 阶段发现指令对寄存器堆的访问可被簇缓冲器过滤, 则在 RF 阶段通过门控信号将位线与供电电压分离. 由于停止了位线的预充电过程, 位线的泄漏功耗减少. 采用 HotLeakage 1.0[17]的评测结果表明, 位线的泄漏功耗占整个寄存器堆泄漏功耗的 65% 以上. 因此, 降低位线的泄漏功耗对于节省整个寄存器堆的泄漏功耗具有重要作用.

6　效果评估

为评估基于簇的功耗管理方法的效果, 实验环境采用 SimpleScalar[18] 模拟器, 集成 Wattch1.02[19] 和 HotLeakage1.0[17] 的功耗模型, 运行嵌入式基准程序 Dhrystone 和 MiBench. 采用 CACTI3.0[20]评测电路面积和延迟, 工艺参数为 70nm. 基础嵌入式处理器配置如表 1 所示.

在实验中每个簇被设置为包含 4 个相邻的体系结构寄存器. 簇缓冲器表项数为 8, 可保存 2 个簇. SEPAS-Filter 表项数为 16. 循环探测的时间间隔阈值 T 设置为 256, SEPAS-Filter 中 Timer 域宽度为 8. 动态电压调节电路中正常供电电压 V_{DD} 为 0.9V, *Drowsy* 模式供电电压 $V_{DD}Low$ 为 0.3V. 环境温度为 80 摄氏度, N 管和 P 管的阈值电压分别为 0.19V 和 0.21V.

表 1　基础处理器配置参数

流水线	8 级, 按序执行
译码宽度	1
分支方向预测器	1024 表项
目标地址缓冲器	128 表项
指令 Cache	16KB, 4 路组相联
数据 Cache	16KB, 4 路组相联
指令 TLB	8 表项全相联一级TLB, 64 表项 4 路组相联二级 TLB
数据 TLB	8 表项全相联一级TLB, 64 表项 4 路组相联二级 TLB
寄存器堆大小	32* 4 字节
总线宽度	4 字节
内存访问延迟	40 周期

6.1　簇缓冲器命中率

当指令需访问的寄存器属于簇缓冲器保存的簇时, 称为簇缓冲器命中. 簇缓冲器的命中率可表示为: $Rate_{hit} = N_{hit}/N_{Reg\_Access}$. 其中 N_{hit} 表示簇缓冲器命中次数, N_{Reg_Access} 表示程序对寄存器的访问次数. 实验结果表明, 在平均情况下, 未经过编译优化的程序使簇缓冲器命中率达到 47.2%, 而经过编译优化后, 簇缓冲器命中率达到 60.9%. 因此, 面向簇的编译优化有助于提高簇缓冲器的命中率, 使基于簇的运行时刻功耗管理达到更好的效果.

6.2　功耗

6.2.1　动态功耗

传统方法与基于簇的功耗管理方法对寄存器堆动态功耗的节省情况如图 4 所示. 图中“4-bank”和“8-bank”分别表示将寄存器堆分为 4 个或 8 个体. 为避免体冲突, 每个体采用与原寄存器堆相同的读端口数和写端口数. “CAM”表示以单个寄存器为功耗管理粒度,

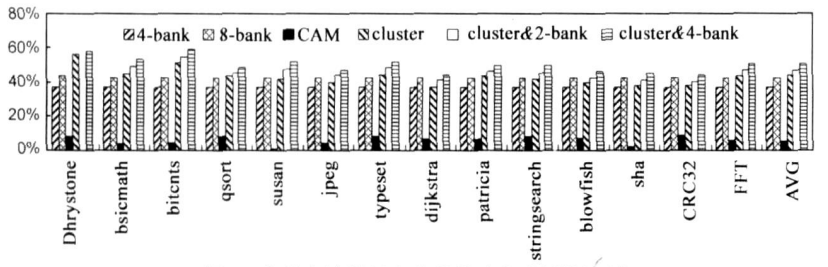

图 4 各种方法降低寄存器堆动态功耗的比例

用 8 个表项来缓冲程序最近访问的 8 个寄存器, 当指令需访问寄存器时, 首先将寄存器地址与缓冲器中的地址 CAM 作全相联比较. "cluster" 表示将连续 4 个寄存器划分为 1 个簇, 并用一个 8 表项的簇缓冲器保存 2 个簇. "cluster&2-bank" 和 "cluster&4-bank" 在 "cluster" 方法基础上分别将寄存器堆划分为 2 个或 4 个存储体. 在统计各种方法的动态功耗结果时, 不仅统计寄存器堆的动态功耗, 还计入了为降低寄存器堆功耗而加入的缓冲器和多路选择器的动态功耗.

实验结果表明, "CAM" 方法对嵌入式处理器寄存器堆动态功耗的节省很有限. 如第三节所述, 这主要由于全相联地址比较和表项替换引起较大的额外功耗.

在平均情况下, 基于簇的管理方法 "cluster" 可降低寄存器堆约 44% 的动态功耗, 与分体方法 "8-bank" 的功耗降低效果(约 43%) 相近, 比分体方法 "4-bank" 达到更好的效果. 如果将基于簇的管理方法与分体方法结合使用, 则动态功耗节省效果更加显著. 平均情况下, "cluster&2-bank" 和 "cluster&4-bank" 可分别将寄存器堆动态功耗降低约 47% 和 51%. "cluster&4-bank" 在 "4-bank" 基础上进一步降低了约 22.3% 的动态功耗. 因此, 不论是对未分体寄存器堆还是分体寄存器堆, 采用基于簇的功耗管理方法均可有效降低动态功耗.

6.2.2 泄漏功耗

如第 5.3 节所述, 寄存器堆的泄漏功耗包括存储单元的泄漏功耗和位线的泄漏功耗两方面. 根据 Hotleakage1.0[17] 的评测结果, 位线的泄漏功耗占整个寄存器堆泄漏功耗的 65% 以上. 由于文献[9, 10] 中的寄存器堆泄漏功耗节省方法仅节省寄存器堆中每个存储单元的泄漏功耗而不对位线预充电过程加以控制, 这两种方法没有降低位线的泄漏功耗.

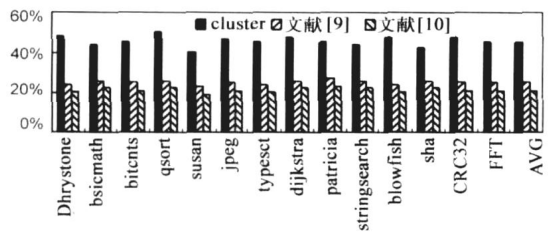

图 5 寄存器堆泄漏功耗降低比例

基于簇的功耗管理方法与传统方法对寄存器堆泄漏功耗的节省情况如图 5 所示. 由于同时降低了位线和存储单元的泄漏功耗, 在平均情况下, 基于簇的功耗管理方法可降低寄存器堆约 45% 的泄漏功耗. 由于仅降低了存储单元的泄漏功耗, 文献[9] 和[10] 中的静态功耗管理方法可分别降低寄存器堆约 25% 和 21% 的泄漏功耗. 因此, 本文方法比文献[9, 10] 中方法能够更有效降低寄存器堆的泄漏功耗.

6.2.3 寄存器堆总功耗和处理器总功耗

实验结果表明, 平均情况下寄存器堆的动态功耗占寄存器堆总功耗的比重约为 32%, 泄漏功耗占寄存器堆总功耗的比重约为 68%. 由于寄存器堆的动态功耗和泄漏功耗均被降低, 寄存器堆的总功耗被减少. 在平均情况下, 寄存器堆总功耗减少 44.7%. 其中, 寄存器堆总功耗减少的 31.5% 来自寄存器堆动态功耗的减少, 寄存器堆总功耗减少的 68.5% 来自寄存器堆泄漏功耗的减少. 同时, 实验结果表明, 在平均情况下寄存器堆的功耗占处理器总功耗的 8.2%. 寄存器堆功耗的减少引起相应比例的处理器总功耗减少. 在平均情况下, 处理器的总功耗减少 3.5%.

6.3 面积与延迟

以原有寄存器堆的面积和延迟为基准作规格化, 采用各种方法后寄存器堆的面积和延迟如图 6 所示. 实验结果表明, 当把寄存器堆划分为两个体时, 由于需要增加存储体间的数据选择逻辑, 读操作的延迟增大约 30%. 当分体数目不断增大时, 虽然读操作的延迟增长不明显, 但是面积呈明显增大趋势. 例如, 划分成 8 个体的寄存器堆的面积已经增大到原来的约 4.5 倍. 因此, 分体技术对寄存器堆面积有较明显的影响. 与分体技术相比, 基于簇的功耗管理方法对寄存器堆面积和延迟影响均不大.

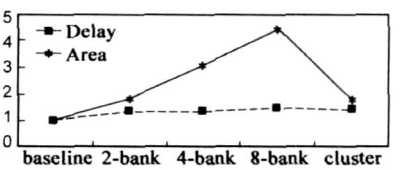

图 6 采用低功耗设计方法后寄存器堆的面积与延迟(以原寄存器堆面积和延迟为基准值)

6.4 实验结果总结

综合以上功耗、面积和延迟的结果, 可以得出以下结论:

与传统的分体技术相比, 本文方法可达到更好的动态功耗节省效果, 并且对寄存器堆面积的影响相对较小. 如果将本文方法与分体技术结合使用, 则寄存器

堆面积会明显增大, 但是会带来更好的动态功耗节省效果.

与以单个寄存器为粒度的功耗管理方法相比, 本文方法能够更有效地降低寄存器堆的动态功耗和泄漏功耗.

在实际处理器设计中选择寄存器堆功耗管理方法时, 如果面积和延迟约束较小, 可以将本文方法与分体技术相结合, 从而达到更好的功耗优化效果; 如果面积和延迟约束较大, 则选用本文方法可以比传统方法达到功耗、面积和延迟的更优折衷.

7 结论

传统的寄存器堆功耗管理方法主要从寄存器堆物理结构出发, 在降低寄存器堆功耗的同时带来不容忽视的功耗、面积和延迟影响. 本文采用软硬件协同设计技术, 提出以簇为粒度的寄存器堆功耗管理方法. 与传统方法相比, 本文方法能够更有效的降低寄存器堆功耗, 达到功耗、面积和延迟的更优折衷.

参考文献:

[1] J Scott, L H Lee, et al. Designing the low-power M• CORETM architecture[A]. IEEE Power Driven Microarchitecture Workshop[C]. Haifa, Israel: IEEE Computer Society, 1998. 29- 33.

[2] K Asanoví c, R Bodik, et al. The landscape of parallel computing research: A view from Berkeley[DB/OL]. http://www.eecs. berkeley. edu/Pubs/TechRpts/2006/EECS-2006-183.pdf, 2006-12.

[3] P Faraboschi, G Desoli, J A Fisher. Clustered instruction-level parallel processors[DB/OL]. http://www. hpl. hp. com/techreports/98/HPL-98-204. pdf, 1998-12.

[4] T Pering, T Burd, R Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms[A]. Int' l Symposium on Low Power Electronics and Design[C]. Monterey, Canada: IEEE Computer Society, 1998. 76- 81.

[5] N S Kim, K Flautner, et al. Single-VDD and single-VT superdrowsy techniques for low-leakage high-performance instruction caches[A]. Int' l Symposium on Low Power Electronics and Design[C]. Newport, California, USA: IEEE Computer Society, 2004. 54- 57.

[6] R Nalluri, R Garg, et al. Customization of register file banking architecture for low power[A]. Int' l Conference on VLSI Design and Embedded Systems[C]. Bangalore, India: IEEE Computer Society, 2007. 239- 244.

[7] R Balasubramonian, S Dwarkadas, et al. Reducing the complexity of the register file in dynamic superscalar processors[A]. Int' l Symposium on Microarchitecture[C]. Austin, Texas, USA: IEEE Computer Society, 2001. 237- 248.

[8] C Yang, A Orailoglu. Power efficient instruction delivery through trace reuse[A]. Int' l Conference on Parallel Architecture and Compilation Techniques[C]. Seattle, Washington, USA: IEEE Computer Society, 2006. 192- 201.

[9] J L Ayala, M L Vallejo, et al. Energy aware register file implementation through instruction predecode[A]. Int' l Conference on Application-Specific Systems, Architectures and Processors[C]. Hague, Netherland: IEEE Computer Society, 2003. 86- 96.

[10] S T Khasawneh, K Ghose. An adaptive technique for reducing leakage and dynamic power in register files and reorder buffers[A]. Int' l Workshop on Power and Timing Modeling Optimization and Simulation[C]. Leuven, Belgium: IEEE Computer Society, 2005. 498- 507.

[11] 胡定磊, 陈书明. 低功耗编译技术综述[J]. 电子学报, 2005, 33(4): 676- 682.
Hu Dinglei, Chen Shuming. Low power/energy compilation technology[J]. Acta Electronica Sinica, 2005, 33(4): 676- 682. (in Chinese)

[12] 张承义, 张民选, 等. LRU—Assist: 一种高效的 Cache 漏流功耗控制算法[J]. 电子学报, 2006, 34(9): 1626- 1630.
Zhang Chengyi, Zhang Minxuan, et al. LRU-Assist: An efficient algorithm for cache leakage power controlling[J]. Acta Electronica Sinica, 2006, 34(9): 1626- 1630. (in Chinese)

[13] Steven S Muchnick. Advanced Compiler Design and Implementation[M]. Morgan Kaufmann. 1997.

[14] Y Wu, J R Larus. Static branch frequency and program profile analysis[A]. Int' l Symposium on Microarchitecture[C]. San Jose, California: IEEE Computer Society, 1994. 1- 11.

[15] A Baniasadi, A Moshovos. SEPAS: A highly accurate and energy-efficient branch predictor[A]. Int' l Symposium on Low Power Electronics and Design[C]. Newport, California, USA: IEEE Computer Society, 2004. 38- 43.

[16] L H Lee, Jeff, et al. Low-cost branch folding for embedded applications with small tight loops[A]. Int' l Symposium on Microarchitecture[C]. Haifa, Israel: IEEE Computer Society, 1999. 103- 111.

[17] Y Zhang, D Parikh, et al. Hotleakage: An architectural, temperature-aware model of subthreshold and gate leakage[R]. University of Virginia. Department of Computer Sciences and Technology. CS-2003-05. 2003.

[18] D C Burger, T M Austin. The SimpleScalar tool set, Version 2.0[J]. Computer Architecture News, 1997, 25(3): 13- 25.

[19] D Brooks, V Tiwari, M Martonosi. Wattch: A framework for architectural power analysis and optimizations[A]. Int' l Symposium on Computer Architecture[C]. Vancouver, British Columbia, Canada: IEEE Computer Society, 2000. 83- 94.

[20] P Shivakumar, N Jouppi. CACTI 3.0: An integrated cache timing, power, and area model[R]. Palo Alto, CA, Compaq, 2001.

作者简介:

孙含欣 男, 1980 年生于黑龙江鸡西, 北京大学信息科学技术学院博士研究生. 主要研究方向为计算机体系结构和低功耗集成电路设计. sunhanxin@mprc. pku. edu. cn

佟 冬 男, 1971 年生于吉林长春, 北京大学信息科学技术学院教授. 主要研究方向为计算机体系结构、超大规模集成电路设计、系统芯片及软硬件协同设计.

袁 鹏 男, 1981 年生于山东荣成, 北京大学信息科学技术学院博士研究生. 主要研究方向为计算机体系结构和编译优化.

程 旭 男, 1967 年生于新疆乌鲁木齐, 北京大学信息科学技术学院教授. 主要研究方向为计算机体系结构、系统芯片及软硬件协同设计.