面向 BW104x 软流水框架①

洪立涛, 郑启龙

(中国科学与技术大学 计算机科学与技术学院, 合肥 230039)

摘 要: 现代高性能数字信号处理器大多数采用超长指令字体系结构, 通过在同一时钟周期发射多条指令以便获得更高的运算性能来发掘目标机器指令级别并行性. 介绍了 BW104x 目标体系特征, BWDSP104X 是一款针对高性能计算领域设计的处理器, 采用 16 发射、单指令流, 多数据流架构. 为了充分利用多簇及簇内硬件资源, 基于 open64 编译基础设施提出了后端软流水优化, 其中包括循环选择, 资源依赖数据依赖计算, 采用经典的模调度方法进行软流水调度, 为解决不同迭代变量冲突引入模变量拓展模块. 实验结果证明流水后性能相对流水前有了很好的提升.

关键词: 编译器; 软流水; 迭代间隔; 模调度; 模变量拓展; 代码生成

Software Pipelining Framework for BW104x

HONG Li-Tao, ZHENG Qi-Long

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230039, China)

Abstract: The digital signal processor (Digital Signal Processing, DSP) is widely used in the field of signal processing, digital communication. The majority of modern high-performance DSP use long instruction word architecture, by exploiting instruction-level parallelism to launch multiple instructions at the same clock cycle out for a higher level of calculating performance. The article describes target system characteristics on BWDSP104x, BWDSP104x is designed in the light of high performance computing and processor, uses 16 launch, single instruction stream and multiple data stream architecture.in order to make full use of multi-cluster hardware resources, this paper proposes the back-end optimization about software pipelining based on the open-source compiler named Open64. Including the early stage of cycle options, resource constraints and precedence constraints computing, the classic Module-Scheduling algorithm is used in SWP-Scheduling, module variable expansion is for the conflict of registers using in different iteration. The experimental results show that program has better performance after software pipelining optimization.

Key words: compiler; software pipelining; iteration interval; module-scheduling; module variable expansion; code generation

1 概述

软流水[1]是编译后端 CGIR 阶段优化中较为重要且有效的一项优化调度技术, 它能改善任何允许指令级并行的系统的循环执行性能, 包括 VLIW 和超标量系统, 通过允许同时处理循环的若干迭代来利用循环体潜在的并行性. DSP(数字信号处理器)和一些嵌入式设备中存在很多经典循环体, 如快速傅里叶变换(Fast Fourier Transform, FFT), 在 FFT 程序中, 由于计算任务的相互独立性, 存在一些具有潜在的可并行性的程序段, 以及没有依赖的迭代循环程序段. 编译器应该能够识别出最内层循环中的这些程序段, 通过优化的软流水和分簇算法, 将相互没有依赖的计算任务分配到独立的运算单元上并行执行, 将不同的循环迭代通过循环展开软流水技术重叠执行, 从而缩短计算任务

① 基金项目:核高基重大专项(2012ZX01034-001-001)

收稿时间:2016-01-17;收到修改稿时间:2016-03-22 [doi:10.15888/j.cnki.csa.005353]

的执行时间.

目前国内外已经有很多实现软件流水的算法, Javier Zalarnea, Josep Llos 等人提出了 Integrated Register Spilling[2]的模调度算法. 该算法为减少寄存器压力, 分配和寄存器溢出以及簇间传输的提出相应的指令调度策略, 在执行簇数量, 簇间总线数量, 簇间传输延迟等方便具有较好的扩放性. Jesus Sanchez 和 Antonio Gonzalez 等人提出面向多簇的基于循环展开的模调度算法[3], 该算法在循环展开时同时进行分簇和指令调度, 主要就是尽可能隐藏传输的延迟. 这样在不同簇上执行不同迭代时, 簇间的传输延迟可以忽略.

Open64 是一个拥有 GUN 许可证的开源编译器, 代码结构清晰, 对后端的优化分析全面透彻, 是非常理想的编译器研究平台. 目前 Open64 已经实现针对 IA64 目标体系的软流水优化, 这一块整体的设计结构流程很清晰, 对工作的开展有很好的参考价值.

本文主要讲解在编译基础设施 Open64 下, 研究并实现针对 BW104x 体系的软流水后端优化整体的框架结构. 剩下的部分组织如下所示. 第二部分阐述目标体系 BWDSP 的结构, 第三部分阐述软流水的概念原理以及具体的实现过程, 第四部分进行了实验的测试结果的阐述与分析, 第五部分对全文以及对自己工作进行进行了总结以及目前工作的不足之处及下一步的工作展望.

2 目标体系结构

本文目标体系是基于 BWDSP 平台的, 该 BWDSP 系列的处理器可以被广泛运用于各种高性能计算的领域, 如雷达、电子对抗、精确制导、通信保障、图像处理等信号处理领域等等. 采用 VLIW 架构, 具有强大的并行处理能力. 该 BWDSP 处理器是一款 32bit 静态超标量处理器, 是一款 32 位的浮点 DSP, 但是能够同时兼容 16 位和 32 位定点数据格式; 该处理器采用 16 发射、SIMD(单指令流, 多数据流)的架构, 并且具有强大的并行处理的能力, 能够比较好的满足对高速实时的处理信号的应用要求. 该处理器内部包含 4 执行单元(簇), 每一个执行单元由 8 个算术逻辑单元(ALU)、4 个乘法器(MUL)、2 个移位器(SHF)、1 个超算(SPU)和 1 个通用寄存组组成.

3 BW104x软流水的理论原理与实现

软流水的概念可以用下面的例子验证: 处理一个简单的向量加的程序, 未用软流水优化时循环的不同迭代串行执行, 效率显然不高. 为了充分利用并行性以及硬件资源, 不同的迭代需要并发地进行. 下面这个例子中, 每次循环都有一次迭代被初始化.

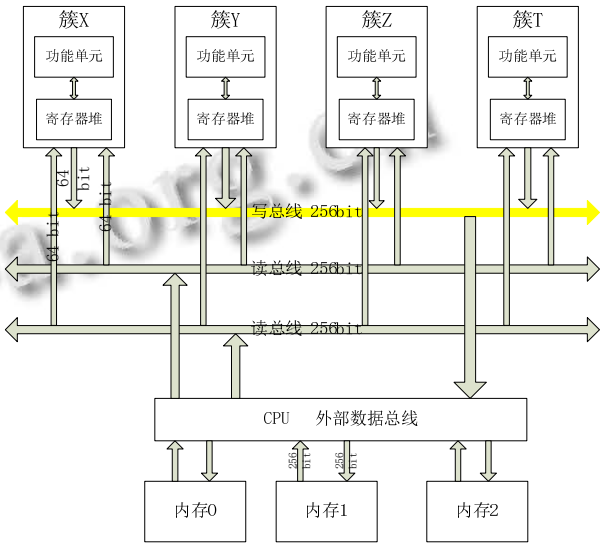


图 1 BWDSP 处理器结构图

```
int s;
for(int   i = 0;i < n ; i++)
{
    a[i] = a[i] * s
}
```

(a) C 代码

```
        %t0   ← 0%
        %t1   ← (n - 1) %
        %t2   ← s%
i0:     t3    ← load a(t0)
i1:     t4    ← t2 * t3
i2:     a(t0) ← t4
i3:     t0    ← t0 + 4
i4:     t1    ← t1 – 1
i5:     if (t1 >=  1)  goto i0
```

(b) 汇编伪代码

图 2 测试小程序以及它的伪汇编代码

下图中 0 到 3 被称为序言部分: 每个周期有一次迭代被初始化, 该阶段每个指令周期启动一个新的迭代. 第 4, 5 个周期到达软流水的核心阶段, 这个阶段

会一直重复直到所有的迭代初始化完毕, 图中稳定阶段 3 次迭代同时进行处理, 一次迭代开始一次迭代输出结果结束. 离开稳定阶段后软流水进入尾声阶段(6 到 9), 这个阶段用于流出软流水的核心阶段的运算结果. BW104x 下每个簇上有 8 个加法器, 同时支持 8 个 load 和 8 个 store, 所以应用软流水理论将程序在 CGIR 阶段对核心循环基本块进行流水化后生成的目标汇编, 循环核心执行需要的周期数相对于流水前串行执行少了很多, 硬件资源利用率也有很大提高.

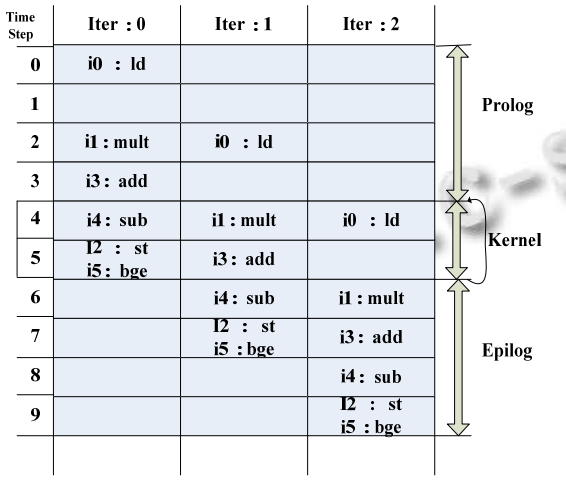


图 3　软流水后的三个阶段

3.1 计算迭代间隔 II

软流水优化中最核心的一个限制因素就是不同迭代的启动间隔, 软流水的调度依赖于间隔周期, 目标在于找到这样一个最小最优启动间隔. 通常启动间隔的计算和资源约束和数据依赖有关.

3.1.1 资源约束 ResMII

在一个经软流水优化的循环中, 循环单次迭代需要的资源是指循环体执行中所有指令需要的各种硬件资源总和, 其中临界资源指使用数目最多的资源, 该类资源对 ResMII 的确定有着核心的影响. 如果不同迭代启动间隔 k 个周期, 那么循环不同迭代在依次间隔 k 个周期初始化相似地执行, 且不同迭代之间保证不能有任何资源冲突. 在流水稳定阶段, 资源的使用情况通过维护一个模资源约束表[4]进行查看.

$$\text{ResMII} = Max\frac{\text{使用资源R的操作个数总和}}{\text{处理器中资源R的总数}}(\forall\text{资源R})$$

3.1.2 数据依赖约束 RecMII

上面图 2 例子中循环体的不同迭代之间明显存在依赖相关, a[i]变量的值更新存储需要在 a[i]的获取和乘法操作进行后进行, 迭代体内指令的操作顺序存在数据依赖. 程序中的数据依赖通常用带环有向图 DDG(V, E ,P ,D)表示, V 表示顶点, 每个顶点代表循环体内一条指令, E 表示依赖边的集合, 即相关指令之间的相互依赖. 顶点 v1 和 v2 之间的依赖边为(v1, v2), 他们之间的最小迭代距离为 p, 指令延迟体差为 d, 令 F: $V \rightarrow N$ 为节点的调度函数, 则必须满足以下条件: F(v2) – (F(v1) – s*p) > d, 意指指令 v2 必须从 v1 执行点 p 个迭代之前点开始计算, 至少过 d 个周期才能执行. 其中 s 是迭代启动间隔, 因为每条指令都不可能依赖于还未执行的迭代, 所以最小迭代距离都是非负数, 在迭代内部指令间依赖的迭代距离是 0. 有向循环图中一条路径的最小迭代距离和最小延迟定义为经过该路径的所有边的最小迭代距离和延迟之和. 现在令 c 为图中的一个回路. 由 F(v2) – F(v1) >= s*p(e)得到的的 d (c) – s*p(c) <= 0, 因此

$$\text{RecMII} = Max\frac{\text{回路c上所有依赖延迟之和}}{\text{回路c上所有依赖体差之和}}(\forall\text{回路c})$$

由以上两点得出:

迭代间隔 II 的下界 MII = Max{ResMII, RecMII}

3.2 BW104x 框架下多簇软流水优化

为了充分利用软件流水发掘出细粒度的指令级并行性(ILP), 分簇体系结构应运而生. 在分簇的架构中, BW104x 由一系列功能簇构成, 每个功能簇由一定数目的功能部件和寄存器文件构成. 在多簇软流水优化时要考虑簇间指令传输带来的延迟.

3.2.1 传统的模调度算法[5]

传统模调度是软流水调度的核心算法, 获取核心循环体基本块信息, 结合目标体系硬件资源生成模资源约束表 MRT 并计算出 ResII, 且根据指令相关依赖图计算出的 RecII, 模调度的最小启动 II 从 Min(ResII, RecII)开始进行调度, 如果调度不成功则增大 II 再次进行循环调度, 直到找到一个满足资源和依赖约束的合适的 II. 松弛度 Slack 是模调度算法中较为核心的概念, 顾名思义在不影响调度情况下指令最晚须发射的周期 Lstart 和最早可发射周期 Estart 的间隔, 调度过程即根据指令的松弛度优先级进行调度.

3.2.2 改进的 BW104x 多簇流水优化

基于多簇的软件流水调度的主要意义就是在充分利用多簇处理器功能单元运算资源的情况下同时最小

化簇间数据传输. 在设计针对多簇架构软件流水调度框架时, 需考虑分簇算法, 可以先分簇再进行流水优化, 也可以尝试分簇与流水同时进行, 先流水调度再进行分簇的方法. 若循环体比较合适软件流水优化的开展, 就根据模调度算法回朔迭代计算出的启动间隔 II, 展开次数 N, 对核心循环体代码进行循环展开, 并根据迭代次数将不同迭代划分到不同计算簇上. 如果循环体过小或者过大不适合进行流水优化, 则只进行基于寄存器压力的分簇[6]不进行流水优化调度.

对适合软件流水核心循环代码面向 BW104X 分簇流水: 根据迭代次数进行分簇, 首先根据簇内计算单元, 存储单元, 寄存器数目计算每个簇的最小迭代次数阈值 M, 当低于该阈值 M 时尽量将不同迭代分到较少簇上执行, 这样可以充分利用簇内硬件资源增大并行度, 当迭代次数大于阈值 M 时, 将循环体迭代分配到不同计算簇上执行, 这样可以充分利用 BW104x 多簇的硬件资源, 维护簇间运算负载相对均衡.

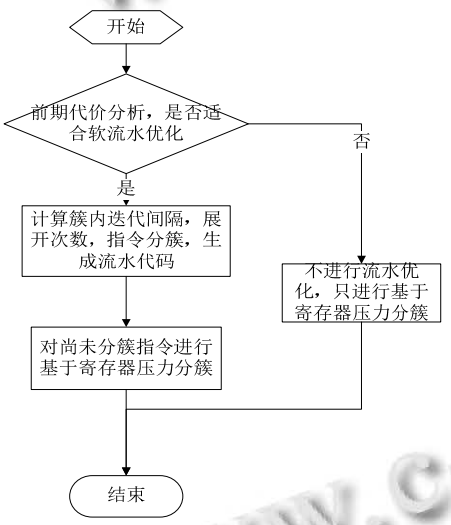


图 4 改进的多簇流水算法流程图

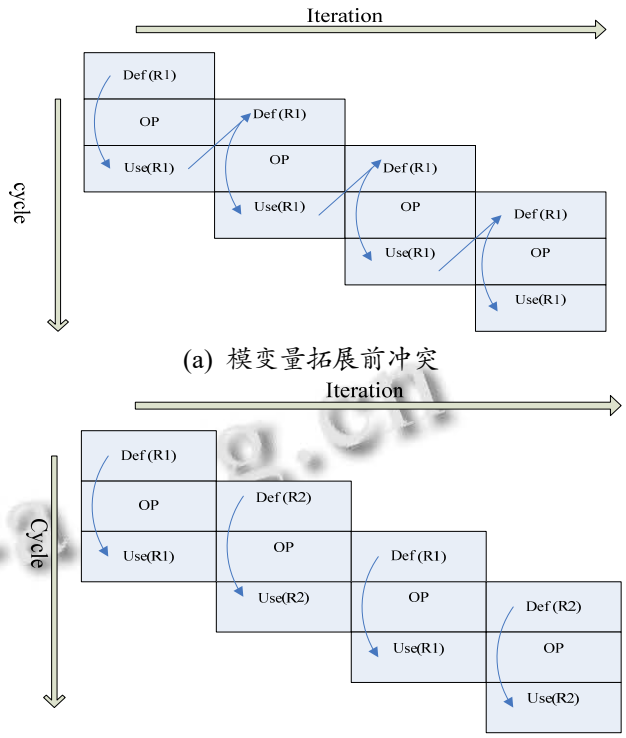
目标体系 BWDSP104x, 它是 4 簇架构, X、Y、Z、T. 这四个簇之间通过簇间总线进行传输数据, 每个簇可以同时向其他三个簇发射 64bit 数据, 但每个簇每一次只能从一个簇中读入数据. 每个簇上有 8 个算术逻辑运算单元(ALU), 4 个乘法器 MUL, 64 个寄存器. 如果某个核心循环的一次迭代需要 1 个 MUL, 2 个 ALU, 16 个寄存器, 则每个簇的最小迭代间隔 N 为 4.

1) 若循环执行次数为 8 时, 如果模调度根据计算启动间隔确定的 II 为 1, 展开次数为 8. 则将核心循环进行 8 遍展开后, 分到 2 个运算簇中, 这样可以充分发挥每个簇内的硬件资源的并行性, 减少由于不同簇之间数据传输指令带来的延迟.

2) 若循环执行次数为 64 时, 假定模调度后确定的的 II 为 1, 展开次数为 16. 则将核心循环进行 16 词展开后, 分到 4 个运算簇中, 这样既能充分利用每个簇的硬件资源, 又维护簇间负载相对均衡.

3.3 模变量拓展[7]

软流水的核心思想在于重叠执行不同的循环体迭代, 充分利用硬件资源. 这里带来一个问题, 如果每次迭代中对同一个变量用相同的寄存器变量存放, 那迭代之间就会相互有影响, 可以通过模变量拓展来有效解决上述问题, 不同循环体的迭代, 用循环展开的方法, 对使用的寄存器进行重命名. 从下图可以看出:



(a) 模变量拓展前冲突

(b) 模变量拓展后

图 5 模变量拓展示例

变量拓展转换通过识别那些在每次迭代开始处重新定义的变量, 通过寄存器重命名技术(这里使用的仍然是伪寄存器), 使得每次迭代对同个变量的引用指向一个不同的位置, 彼此之间不冲突.

4 实验测试结果分析

为了验证实现软流水模块在目标体系上运行生成

的目标汇编是否为预期结果, 进行了如下的测试过程:

实验环境:

程序在 Rethat Enterprise5.0 + Open64 编译器 + occ + Inter Core i5 CPU + 2G 内存. 其中 occ 为 open64 面向 BWDSP 体系移植的编译工具, 用于编译生成 BWDSP 体系下的目标汇编. 在虚拟机上运行 Rethat, 将面向 BWDSP 体系移植好的 open64 代码进行编译生成可执行编译工具 occ, 用 occ 和 gdb 工具对测试程序进行编译调试.

测试命令:

occ -show -O2 -keep -S loop.c : 打开循环优化将优化等级设为 O2, 会打开后端谓词以及软流水优化.

occ -show -O1 -keep -S loop.c : 优化等级设为 O1, 关闭软流水优化, 用于和流水后生成的目标汇编作对比.

输入循环测试程序:

```
int a[64],b[64],c[64];
int main()
{
    int i;
    for(i = 0;i < 64; i++)
    {
     c[i] = a[i] + b[i];
    }
    return c[i-1];
}
```

关闭软流水优化生成的目标汇编核心代码:

```
u7=__c    u5=__a   u6=__b   xr15=1
xr10=64
xr11=[u5+0,0]
u5=u5+1
xr12=[u6+0,0]
u6=u6+1
xr11=r11+r12
[u7+0,0]=xr11
u7=u7+1
xr14=r14+r15
If xr10!=r14 B _Lt_0_2050
```

软流水优化后目标汇 编核心代码:

```
u7=__c
u5=__a
u6=__b
xr5 = 0
```

核心循环进行软流水:

```
xr7 = 64
xr3 = 4   //展开次数
_Lt_0_2050:
u14 = 0
xr10 = [u5 + u14,0] || u3 = u14 + 1
xr13 = [u6 + u14,0] || xr15 = [u5 + u3,0] || u8 = u14 + 2
xr14 = xr10 + xr13 || xr17 = [u6 + u3,0] || xr18 = [u5 + u8,0] || u2 = u14 + 3
[u7 + u14,0] = xr14 || xr8 = xr15 + xr17 || xr9 = [u6 + u8,0] || xr21 = [u5 + u2,0]
[u7 + u3,0] = xr8 || xr11 = xr9 + xr18 || xr25 = [u6 + u2,0]
[u7 + u8,0] = xr11 || xr23 = xr25 + xr21
[u7 + u2,0] = xr23
u14 = u14 + 4
xr5 = xr5 + xr3
```

循环结束条件:

```
IF xr5 != xr7 B _Lt_0_2050
```

实验结果分析:

实验使用了简单的向量加循环, 执行 64 次加法操作. 如果不进行软流水操作, 那么每个迭代将串行执行数组 a 和 b 的元素取数, 下标自增, 循环条件判断, 数相加, 最后再进行存数操作, 每次迭代需要 9 个周期, 那么循环执行 64 次, 则需要 64*9 = 576 个周期才能完成这个循环, 软流水后, 循环展开 4 次, 重叠执行不同的迭代, 以上核心汇编模块执行需要 13 个周期, 则总共需要(64 / 4)* 13 = 208 个周期即可以完成循环的执行, 可见充分并行利用硬件资源后程序的执行周期执行效率有了很大程度的提高.

5 结语

本文针对目标体系 BWDSP104x、基于 Open64 开源编译框架提出后端软流水优化方案. 旨在充分利用目标体系硬件资源, 增加循环程序调度的并行度, 提高程序执行的效率. 本文采用经典的模调度算法进行调度, 为了充分利用 BW104x 多簇体系, 对传统算法进行了多簇分配优化. 在分配和寄存器溢出方面提出

了有效调度策略, 根据先决条件进行回朔为模调度算法带来较低的溢出代码和较高的吞吐率. 最小启动间隔 II 的计算影响着软流水的优化效率及速度, 针对资源约束和数据依赖, 利用模资源约束表和数据依赖图来计算. 为了消除不同迭代变量引用和定义的冲突, 使用了模变量拓展的方法有效解决了问题.

5.1 不足之处

目前实现的仅仅支持简单的非嵌套循环程序的测试, 不支持复杂嵌套循环的执行. 且仅仅对 for 程序, do-loop 循环程序支持, 不支持 while 循环的优化. 对于数据依赖程度较高的程序暂且没有实现相应的处理方法.

5.2 下一步研究方向

对已完成的部分进一步进行优化, 在简单循环能够实现软流水优化的基础上再研究嵌套复杂的循环的实现. 深入研究 open64 后端寄存器分配模块, 以及调度算法. 结合谓词, SIMD 技术和软流水优化, 对后端优化整体的流程有个清晰的认知.

参考文献

1 Ferreira R, Denver W, Pereira M, et al. A dynamic modulo scheduling with binary translation: Loop optimization with software compatibility. Journal of Signal Processing Systems, 2015: 1–22.

2 Zalamea J, Llosa J, Ayguadé E, et al. Register constrained modulo scheduling. Parallel & Distributed Systems IEEE Trans. on, 2004, 15(5): 417–430.

3 Sanchez FJ, Gonzalez A. Cache sensitive modulo scheduling Proc. of the 30th Annual ACM/IEEE International Symposium on Microarchitecture. IEEE Computer Society. 1997. 338–348.

4 Scheuch M, Höper D, Beer M. RIEMS: A software pipeline for sensitive and comprehensive taxonomic classification of reads from metagenomics datasets. BMC Bioinformatics, 2015, 16(1): 69.

5 Cheng X, Tan M, et al. Compiling method and device for realizing loop instruction scheduling based on modulo scheduling. WIPO Patent Application, WO/2012/155442. 2012.

6 雷一鸣,洪一,徐云,姜海涛.一种基于寄存器压力的 VLIW DSP 分簇算法.计算机应用,2010,30(1).

7 Cho SW, Kim JK, Park SJ, et al. 4X framer/deframer module for PCI-express and PCI-express framer/deframer device using the same. United States Patent Application US, 20080162767, 2008.