

基于 MIPS 体系的扩展指令融合技术

陈文智 姜振宇 吴 帆

(浙江大学计算机科学与技术学院 杭州 310027)

摘 要 MIPS 作为 RISC 体系的典型代表,不能避免代码密度不高和指令域的有效利用程度低的缺陷,使得程序体积膨胀.文中将 MIPS 指令集扩展为 exMIPS ISA,并提出一种基于 MIPS 体系的指令融合技术.它在解码阶段对预取指令扫描并转换成 exMIPS ISA,将符合融合条件的相邻两条或多条 exMIPS ISA 指令压缩合并.一条“融合指令”的执行,等效于多条被融合的指令同时发射执行,不仅提升了 CPU 性能,也提升了指令域的有效利用率和代码密度. SimpleScalar 模拟平台的实验结果显示可获得较大的性能提升.

关键词 指令融合;代码压缩;MIPS 指令集扩展;指令级并行;SimpleScalar

中图法分类号 TP314 **DOI 号**: 10.3724/SP.J.1016.2008.01888

Instruction Fusion Technology for the MIPS

CHEN Wen-Zhi JIANG Zhen-Yu WU Fan

(College of Computer Science and Technology, Zhejiang University, Hangzhou 310027)

Abstract As a typical representative of RISC architecture, MIPS has the problem of low code density and ineffective utilization of instruction fields, making procedure volume expansion. The authors made some extensions to the existing MIPS Instruction Set, called exMIPS ISA. The authors propose an instruction fusion technology for the MIPS architecture. The pre-fetched instructions was converted into exMIPS ISA, and multiple sequential instructions was compressed and merged into a single instruction when meeting the fusion-condition. A fused instruction's execution is equivalent to multiple instructions running at the same time, and will gain extra CPU performance. The process of instruction fusion also enhances the effective utilization of the instruction fields and improves code density. Experimental results from SimpleScalar simulation platform show that great improvement can be achieved.

Keywords instruction fusion; code compression; MIPS instruction set extention; ILP; SimpleScalar

1 引 言

MIPS 作为 RISC 体系的典型代表,有着这样两个不足:(1)代码密度不高,与 CISC 体系相比,完成相同功能,程序规模通常更大,程序在静态存储时浪费存储器资源,动态加载到内存和高速缓存(cache)中运行时又增加了功耗以及 CPU 取指

令带宽;(2)指令域的有效利用程度不高,例如一条 add \$3, \$2, \$0,完成的功能仅仅是将寄存器 \$2 中的内容转移到 \$3,却占用着一整条 MIPS 指令的 32 个 bit(以 32 位 MIPS 为例),既不直观也不高效,浪费了指令域的有效利用程度,使得 CPU 花费更多的时钟周期完成相同功能.

本文以“Make the Common Case Fast”为原则,针对 MIPS 指令集进行优化和扩展,设计出

exMIPS ISA(扩展 MIPS 指令集),进而提出一种基于 MIPS 架构的指令融合技术.所谓指令融合,是指将符合融合条件的相邻两条或多条 exMIPS ISA 指令压缩合并,一条“融合指令”的执行,等效于多条被融合的指令同时发射执行,使 CPU 花费更少的时钟周期完成相同的工作;一个指令序列被压缩合并成单独一条“融合指令”,也大大提升了指令域的有效利用率和代码密度.

通过在处理器中引入冗余功能部件,例如多个整型处理部件、多个加载/存储部件等,可以更好地利用指令融合技术带来性能提升,为处理器提供并发利用这些功能部件执行多条指令的能力.

本文第 2 节介绍相关工作;第 3 节是 MIPS 体系的宏融合技术设计和实现;第 4 节介绍如何编译软件支持 exMIPS ISA,并产生优化的融合指令;第 5 节在对模拟器进行修改后,引入扩展指令融合技术,以 SPEC95 和 SPEC2000 的部分测试程序作为模拟输入,给出实验结果,结果显示平均有 8% 的性能提高;最后是结论和未来的工作.

2 相关工作

为了方便流水线设计,各种 CISC 体系 CPU 纷纷引入微操作(Micro-operations)指令^[1]这一概念,以 x86 为代表,它在流水线的预取和预解码阶段,将一条普通的 x86 指令转化为一条或多条类似 RISC 的长短相同的微操作指令^[2].微操作指令提出,使得 CISC 架构的流水线设计得到极大简化,各种新的 ILP 提升技术也应运而生.

在 Intel Pentium M 微架构^[3]中提到一种微指令融合(Micro-ops fusion)技术,它将内存存取操作(memory store 和 memory load)与地址算术运算融合,变为原来的 x86 指令,以减少微操作指令运算的数目,但是每一对融合的微操作指令仍然单独调度.这种融合实际上将分解后的指令又还原成了原来的 x86 指令,做了额外的无用功,然而为了保持流水线结构的整体一致性,又不得不这样做.相比之下,MIPS 指令本来就属于 RISC 指令集,不需要在预取阶段将指令转化为微操作指令,自然避免了 x86 面临的这一窘境.

Intel 的 Core 架构^[4](核心代号 Conroe,命名 Core 2 Duo/Extreme 系列)处理器,在继承了 Pentium M 处理器微指令融合的基础上,应用了另一项类似的新技术:宏融合(Macro-fusion)^[5].它与

微指令融合技术的区别在于,宏融合是直接可在可变长 x86 指令基础上(转化为微操作指令之前)对相邻指令的融合技术.但是为了向前兼容,已有的 x86 指令不能改动,并且在 x86 体系各指令域的有效利用率已经较高,使得宏融合应用范围很窄.

文献[1]中提到一种协同设计 x86 虚拟机(co-designed x86 virtual machine),具体研究了将 x86 指令动态转换成类似 RISC 的微操作,然后将微操作融合成一个指令对(pair),但这个指令对需要在相邻的时钟周期里顺序执行,仅仅提升指令调度(dispatch)的效率.而我们的基于 MIPS 体系的扩展指令融合的目标是并行执行各指令对,有本质上的区别.

Transmeta Crusoe 处理器^[6]和 IBM 的 DAISY^[7]的内部指令集是 VLIW 风格的指令集,由类似 RISC 的操作组成.VLIW 为了记录指令,需要做相当多的软件优化工作,特别是当指令集中含有预测指令时.相比之下,基于 MIPS 体系的扩展指令融合架构,仅用硬件就可以动态判断并生成融合指令,不会有 VLIW 那样复杂的软件转换工作.

文献[8-9]介绍了一种指令压缩技术:如果两条 16 位 Thumb 指令的功能可以由一条单独的 ARM 指令完成,则将它们压缩,这种指令压缩技术限制条件太紧.我们的 exMIPS ISA 的设计目标是将任意相邻的两条扩展 MIPS 指令融合在一起,只要满足融合条件,而不必对融合后的指令是否存在有任何限制.

3 MIPS 体系扩展指令融合技术的设计和实现

3.1 Macro-fusion 原理简介

为了实现宏融合,fetch 阶段预取到指令后,对指令进行扫描,如果连续两条指令的全部有效指令域,经优化可以由单独一条指令容纳,则将这两条指令融合成一条“融合指令”.在判断指令是否能够融合时会碰到两个问题:(1)已有指令的空闲指令域不多甚至没有(见图 1 MIPS 基本指令列表);(2)相邻的两条指令之间有数据依赖关系.针对第一个问题,我们提出 exMIPS ISA 扩展.

3.2 exMIPS ISA 扩展

以相同指令格式为划分原则,我们以 Add 为代表,表示 add、addu、sub、subu、slt、sltu、and、or、xor、nor、sllv、srlv、srav 等 R-type 类型的指令,它们是 rs+rt+rd 组合;以 Sll 为代表,表示 sll、srl、sra 这

Bit #	[31..26]	[25..21]	[20..16]	[15..11]	[10..06]	[05..00]	Operations
R-type	op	rs	rt	rd	sa	func	
add	000000	rs	rt	rd	00000	100000	$rd \leftarrow rs + rt$
sub	000000	rs	rt	rd	00000	100010	$rd \leftarrow rs - rt$
and	000000	rs	rt	rd	00000	100100	$rd \leftarrow rs \& rt$
or	000000	rs	rt	rd	00000	100101	$rd \leftarrow rs rt$
sll	000000	00000	rt	rd	sa	000000	$rd \leftarrow rt \ll sa$
srl	000000	00000	rt	rd	sa	000010	$rd \leftarrow rt \gg sa$ (logical)
sra	000000	00000	rt	rd	sa	000011	$rd \leftarrow rt \gg sa$ (arithmetic)
I-type	op	rs	rt	immediate			
addi	001000	rs	rt	immediate			$rt \leftarrow rs + (\text{sign_extend})\text{immediate}$
andi	001100	rs	rt	immediate			$rt \leftarrow rs \& (\text{zero_extend})\text{immediate}$
ori	001101	rs	rt	immediate			$rt \leftarrow rs (\text{zero_extend})\text{immediate}$
lw	100011	rs	rt	immediate			$rt \leftarrow \text{memory}[rs + \text{sign_extend}]\text{immediate}$
sw	101011	rs	rt	immediate			$\text{memory}[rs + (\text{sign_extend})\text{immediate}] \leftarrow rt$
beq	000100	rs	rt	immediate			
bne	000101	rs	rt	immediate			
J-type	op	address					
j	000010	address					$PC \leftarrow (PC + 4)[31..28], \text{address} \ll 2$
	op	rs				func	
jr	000000	rs	0			001000	$PC \leftarrow rs$

图 1 MIPS 基本指令及其指令域

类有 sa(shift-amount)域的移位指令,它们是 rt+rd+sa 组合;以 Lw 为代表,表示 lw、lb、lbu、lh、lhu、lwl、lwr、sw、sb、sh、swl、swr 这类内存存取操作,它们是 rs+rt+imm(immediate)组合;以 Addi 为代表,表示 addi、addiu、stli、stliu、andi、ori、xori 这类含有一个立即数的操作,它们也是 rs+rt+imm 组合.本文后面出现的 Add(首字母大写)指令,如若无特殊说明,均表示以 Add 为代表的那一系列指令,对于 Sll、Lw、Addi 也是如此.

通过对 MIPS 指令的分析^[10],很多指令并没有耗尽指令的每一个域,如图 1 所示,add、sub、and、or 没有使用 sa(shift-amount)域,而 sll、srl、sra 的 rs 域都是空闲.

根据宏融合的基本思想,如果某条指令的空闲指令域足够多,足以容纳另一条指令的所有有效指令域,则可以将这两条或多条相邻的指令融合.例如,add 的有效寄存器索引是 rs、rt 和 rd,sa 闲置,而 jr 只有 rs 有效,因此可以将这两条指令融合.

然而 MIPS 指令中拥有空闲指令域的指令毕竟有限,以下例子说明了融合过程碰到的难题.对于 add 和 sll,add 的 sa 域空闲,sll 的 rs 域空闲,只有当 rt 域与 rd 域完全一样时才能融合这样两类指令.融合后的指令如图 2 所示.

Bit #	31..26	25..21	20..16	15..11	10..06	05..00
R-type	op	rs	rt	rd	sa	func
addsl	000000	rs	rt	rd	sa	110000
slladd	000000	rs	rt	rd	sa	110001

图 2 add 与 sll 融合指令(v1)

这是第一个版本的融合指令,称之为 addsl_v1,为了与其他已有的 MIPS 指令区分,func 域被修改为 110000(具体数值任选,与其他指令不冲突即

可),图 3 显示了它所完成的功能.

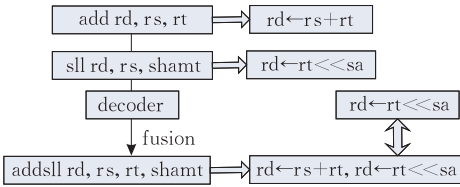


图 3 addsl_v1 指令融合过程(v1)

从图 3 可以看到,存在 WAW(Write After Write)冲突,并且融合条件要求 rt 与 rd 一样,限制太紧,实际意义不大.为此,针对 MIPS 指令集做优化和扩展,提出 exMIPS ISA.

3.2.1 扩展 1: Add2

一条指令所涉及到的寄存器个数,MIPS 通常是 3 个,Intel X86 通常是 2 个:

MIPS: add \$r1, \$r2, \$r3; r3=r1+r2

X86: add \$r1, \$r2; r1=r1+r2

各有好处,但是当 MIPS 指令涉及到相同寄存器时,如 add, \$r1, \$r1, \$r1,就严重浪费用于寄存器索引的指令域了,而指令融合关键在于提高指令域的利用率,为此对 MIPS 指令作第一次扩展.

从图 4 中可以看到,以 add2 和 sll2 为代表,添加了一部分新指令,它们的共同特点在于复用了指令域用于寄存器索引,从而空闲出一个指令域.

Bit #	31..26	25..21	20..16	15..11	10..06	05..00	Operations
R-type	op	rs	rt	rd	sa	func	
add2	000000	rs	rt	00000	00000	101000	$rs \leftarrow rs + rt$
sub2	000000	rs	rt	00000	00000	101010	$rs \leftarrow rs - rt$
and2	000000	rs	rt	00000	00000	101100	$rs \leftarrow rs \& rt$
or2	000000	rs	rt	00000	00000	101101	$rs \leftarrow rs rt$
sll2	000000	00000	00000	rd	sa	001000	$rd \leftarrow rd \ll sa$
srl2	000000	00000	00000	rd	sa	001010	$rd \leftarrow rd \gg sa$ (logic)
sra2	000000	00000	00000	rd	sa	001011	$rd \leftarrow rd \gg sa$ (arith)

图 4 Add2 系列扩展指令

指令扩展后,我们再次尝试指令融合. 仍然以 add 和 sll 为代表,add2 操作完成动作 $rs \leftarrow rs + rt$, sll2 操作完成动作 $rd \leftarrow rd \ll sa$,它们的指令域刚好完全错开,可以随意融合,并且指令顺序对融合没有任何影响,如图 5 所示.

Bit #	[31..26]	[25..21]	[20..16]	[15..11]	[10..06]	[05..00]	Operations
R-type	op	rs	rt	rd	sa	func	
addsl2	000000	rs	rt	rd	sa	111000	$rs \leftarrow rs + rt$ $rd \leftarrow rd \ll sa$
subsl2	000000	rs	rt	rd	sa	111010	$rs \leftarrow rs - rt$ $rd \leftarrow rd \ll sa$
andsl2	000000	rs	rt	rd	sa	111100	$rs \leftarrow rs \& rt$ $rd \leftarrow rd \ll sa$
orsl2	000000	rs	rt	rd	sa	111101	$rs \leftarrow rs rt$ $rd \leftarrow rd \ll sa$

图 5 add 与 sll 融合指令(v2)

这是第 2 个版本的融合指令,称之为 addsl_v2,同样 func 域做了一定修改. 它所完成的功能如图 6 所示.

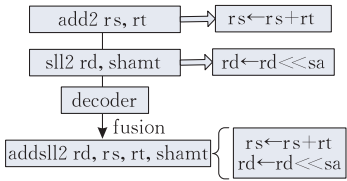


图 6 addsl_v2 指令融合过程(v2)

addsl_v2 实现了融合指令并行执行. 但仍需注意,当 rs 和 rd 为同一寄存器时,会有 WAW 冲突. 对于数据冲突问题,我们在后面会讨论.

我们把这类通过复用寄存器索引而获得额外空闲指令域的扩展标记为 Add2 扩展,这类扩展指令只用到两个操作数,它适用于 Add 系列指令.

3.2.2 扩展 2: Lw2

lw 和 sw 指令的格式一致. 它们的各个指令域都被使用,没有空闲指令域用于融合.

lw 的 immediate 域是一个 16 位的偏移,和 rs 一起用于计算物理地址, $rt \leftarrow \text{memory}[rs + (\text{sign_extend})\text{immediate}]$. 这对于数组操作特别有利,immediate 相当于数组内偏移. 然而对于大多数内存访问,变量直接取址操作更普遍,而且对于数组元素,也可以事先将地址+偏移计算出来,存放到 rs 寄存器中. 这样,在多数情况下, lw 的 immediate 域实际上为 0. 通过实验环境用 gcc 反汇编出的汇编指令,也可以看到这一点.

基于以上的考虑,对 lw 和 sw 做如下扩展,见图 7.

扩展指令 lw2 和 sw2 针对 imm 域为零的情况

进行优化,这一优化使得我们获得额外的 16-bit 空闲指令域. 这样, lw2、sw2 可以和前面提到的 Add2 系列扩展指令随意融合, Lw2 与 Add2 的指令顺序对融合没有影响. 图 8 显示了一部分指令融合后的情况.

Bit #	[31..26]	[25..21]	[20..16]	[15..00]	Operations
l-type	op	rs	rt	immediate	
lw2	110011	rs	rt	00000000 00000000	$rt \leftarrow \text{memory}[rs]$
sw2	111011	rs	rt	00000000 00000000	$\text{memory}[rs] \leftarrow rt$

图 7 Lw2 系列扩展指令(immediate 域空闲)

Bit #	[31..26]	[25..21]	[20..16]	[15..11]	[10..06]	[05..00]	Operations
R-type	op	rs	rt	rd	sa	func	
lwsll2	110011	rs	rt	rd	sa	011000	$rs \leftarrow \text{memory}[rt]$ $rd \leftarrow rd \ll sa$
swsll2	111011	rs	rt	rd	sa	011000	$\text{memory}[rs] \leftarrow rt$ $rd \leftarrow rd \ll sa$
lwsrl2	110011	rs	rt	rd	sa	011010	$rs \leftarrow \text{memory}[rt]$ $rd \leftarrow rd \gg sa(\text{logic})$
swsrl2	111011	rs	rt	rd	sa	011010	$\text{memory}[rs] \leftarrow rt$ $rd \leftarrow rd \gg sa(\text{logic})$
lwsra2	110011	rs	rt	rd	sa	011011	$rs \leftarrow \text{memory}[rt]$ $rd \leftarrow rd \gg sa(\text{arith})$
swsra2	111011	rs	rt	rd	sa	011011	$\text{memory}[rs] \leftarrow rt$ $rd \leftarrow rd \gg sa(\text{arith})$

图 8 lw、sw 与 sll、srl、sra 融合指令

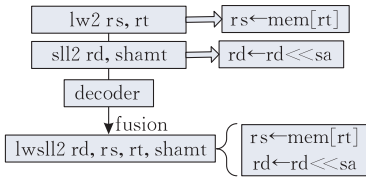


图 9 lwsll2 指令融合过程

同样要注意当 rs 和 rd 为同一寄存器时的 WAW 冲突问题.

我们把这类通过重用指令的 imm 域而获得额外空闲指令域的扩展标记为 Lw2 扩展,这类扩展指令只用到两个操作数,或者说第 3 个 imm 立即数默认为 0,它适用于 Lw 系列指令.

3.2.3 扩展 3: Addi2 和 Addi2_s

对于 Addi 系列指令,我们作如下扩展:若 rt 与 rs 相同,则将下面这条指令

addi \$rt, \$rs, imm; $rt = rs + (\text{signed extended})\text{imm}$ 扩展成

addi2 \$rt, \$rt, imm; $rt = rt + (\text{signed extended})\text{imm}$.

这一扩展与 Add2 的做法一致,但仍然只有一个空闲指令域. 我们进一步考虑对 imm 域进行扩展. 通过对具体实验环境用到的 gcc 反汇编出的汇编指令分析,我们发现 90% 以上的 addi 指令的 imm 域数值在 -128~127 之间,这意味着仅用一个字节即可有效表示 imm 域.

为此,我们将 addi2 进一步扩展成 imm 域仅占 8bit 的 addi2_s(其中的 s 是 short 的缩写,表示 short immediate),如图 10 所示.

Bit#	31..26	25..21	20..16	15..11	10..03	02..00	Operations
	op	rs	rt	rd	imm_s	func	
addi2_s	opcode	00000	00000	rd	imm_s	func	$rd \leftarrow rd + (\text{signed extended})imm_s$
sliui_s	opcode	00000	00000	rd	imm_s	func	if ($rd < imm_s$) $rd = 1$ else $rd = 0$

图 10 Addi2_s 系列扩展指令

我们把这类指令扩展标记为 Addi2 扩展和 Addi2_s 扩展,这类扩展指令通过指令域复用和 imm 域压缩,提供了更多的空闲指令域,适用于 Addi 系列指令.

3.2.4 扩展 4: Mov 和 Movi_s

通过实验分析,我们还发现了大量这样的指令:
addu \$rd, \$0, \$rt ; rd=rt

这类指令的实际作用只不过是 将一个寄存器的内容赋予目的寄存器,可是却白白占用了一个指令域用于表示 0 号寄存器(内容永远为 0). 我们将这类指令扩展成为

```
mov $rd, $rt ; rd=rt
```

对于 addiu \$rd, \$0, imm 这类指令,我们将其扩展为 movi \$rd, imm, 并且考虑对 imm 域的压缩,可进一步扩展成 movi_s \$rd, imm, 与 movi 的区别在于其中的 s 是 short 的缩写,表示 short immediate,如图 11 所示.

Bit#	31..26	25..21	20..16	15..11	10..03	02..00	Operations
	op	rs	rt	rd	imm_s	func	
movi_s	opcode	00000	00000	rd	imm_s	func	$rd \leftarrow (\text{signed extended})imm_s$

图 11 movi_s 系列扩展指令

我们把这类指令扩展标记为 Mov 扩展和 Movi_s 扩展,这类扩展指令通过重用 0 号寄存器的指令域以及 imm 域压缩,提供了更多的空闲指令域.

特别值得说明的是:以上所有的扩充指令都作为内部指令处理,可由处理器运行时动态生成.

3.3 扩展指令融合

前面介绍了 exMIPS ISA,下面将对指令序列的融合条件作具体介绍.

3.3.1 直接融合

Add2、Lw2、Addi2_s、Mov 和 Movi_s 扩展系列,使用两个指令域用于寄存器索引,空闲两个指令域,这一扩展系列的任意两条指令都可直接融合,如表 1 所示.

表 1 直接指令融合

指令格式		操作
add2_add2	\$rs, \$rt, \$rd, \$rx	$rs \leftarrow rs + rt$ $rd \leftarrow rd + rx$
add2_sub2	\$rs, \$rt, \$rd, \$rx	$rs \leftarrow rs + rt$ $rd \leftarrow rd - rx$
lw2_lw2	\$rs, \$rt, \$rd, \$rx	$rt \leftarrow \text{memory}[rs]$ $rx \leftarrow \text{memory}[rd]$
add2_sw2	\$rs, \$rt, \$rd, \$rx	$rs \leftarrow rs + rt$ $\text{memory}[rd] \leftarrow rx$
add2_addi2_s	\$rs, \$rt, \$rd, imm_s	$rs \leftarrow rs + rt$ $rd \leftarrow rd + (\text{signed extended})imm_s$
mov_movi_s	\$rs, \$rt, \$rd, imm_s	$rs \leftarrow rt$ $rd \leftarrow (\text{signed extended})imm_s$

这类扩展指令的融合需要注意数据依赖的问题,如下面两条指令就不能融合:

```
add2 $3, $4 ; $3 ← $3 + $4
add2 $6, $3 ; $6 ← $6 + $3
```

前一条 add2 的目的寄存器是 \$3,正好是后一条 add2 的源寄存器之一,在前一条指令得到运算结果之前,无法执行后一条指令,因此不能将它们融合后并行执行.

遵循前一条指令的目的操作数不能为后一条指令的源操作数的原则,表 2 对各类型指令融合的数据依赖关系总结.

表 2 指令融合数据依赖限制

指令序列及类型		限制条件
Add2_Add2	\$rs, \$rt, \$rd, \$rx	$Rs \neq Rd \ \& \& \ Rs \neq Rx$
Lw2_Lw2	\$rs, \$rt, \$rd, \$rx	$Rt \neq Rd$
Sw2_Sw2	\$rs, \$rt, \$rd, \$rx	No limitation
Add2_Lw2	\$rs, \$rt, \$rd, \$rx	$Rs \neq Rd$
Lw2_Add2	\$rs, \$rt, \$rd, \$rx	$Rt \neq Rd \ \& \& \ Rt \neq Rx$
Sw2_Add2	\$rs, \$rt, \$rd, \$rx	No limitation
Add2_Sw2	\$rs, \$rt, \$rd, \$rx	$Rs \neq Rd \ \& \& \ Rs \neq Rx$ ①
Addi2_s_Sw2	\$rs, imm_s, \$rd, \$rx	$Rs \neq Rd \ \& \& \ Rs \neq Rx$ ②
Sw2_Addi2_s	\$rs, \$rt, \$rd, imm_s	No limitation
Mov_Mov	\$rs, \$rt, \$rd, \$rx	$Rs \neq Rx$
Movi_s_Movi_s	\$rs, imm_s, \$rd, imm_s	No limitation
Mov_Movi_s	\$rs, \$rt, \$rd, imm_s	No limitation
Movi_s_Mov	\$rs, imm_s, \$rd, \$rx	$Rs \neq Rx$
Mov_Add2	\$rs, \$rt, \$rd, \$rx	$Rs \neq Rd \ \& \& \ Rs \neq Rx$
Mov_Add2_s	\$rs, \$rt, \$rd, imm_s	$Rs \neq Rd$
Movi_s_Add2	\$rs, imm_s, \$rd, \$rx	$Rs \neq Rd \ \& \& \ Rs \neq Rx$
Movi_s_Add2_s	\$rs, imm_s, \$rd, imm_s	$Rs \neq Rd$

对表 2 的说明:①在实际的实验环境中,sw 指令会被分解成两条内部指令,一条计算地址,另一条则完成真正的 sw 存储功能. 对于 Add2_Sw2,当 $Rs = Rx$ 时,表明 Sw2 所要存储的数值正是 Add2

的运算结果,但是 Sw2 在地址计算时还不需要 Add2 的计算结果,因此可以去除 $\$rs \neq \rx 的限制;其次还可以去除 $\$rs \neq \rd 的限制,当 $\$rs = \rd 时表明 Sw2 需要 Add2 的运算结果进一步计算存储目标地址,但是 Sw2 的 imm 域为 0,其地址完全由 $\$rd$ 决定,因此只需对硬件稍作改进,让 Sw2 直接将 Add2 的运算结果作为目标地址,就可以避免 Sw2 在流水线 Ex 阶段的计算,从而可以和 Add2 并行执行.②与①类似,经过改进,这两个限制条件实际上可以去除.

除了以上扩展系列指令之间可以直接融合之外,它们都还可以和 jr 指令融合,并且未扩展 Add 系列指令也可以和 jr 融合,针对 jr,除了前一条指令的目的操作数不能为后一条指令的源操作数这一限制外,jr 指令还必须是指令序列中的最后一条指令.

3.3.2 特殊融合

除了前面较为普遍的融合外,还有一些特殊的指令序列也可以融合.

a. $lw + lw \rightarrow dlw$

对于形如以下的连续两条 lw 指令,将其融合成一条 dlw(double lw)指令.

```
lw $rt, imm($rs)
lw $rt', imm'($rs)
```

其中 $rt' = rt + 1$, $imm' = imm + 4$ 或 $rt = rt' + 1$, $imm = imm' + 4$,即两条 lw 指令的目的寄存器索引是连续的,内存地址也是连续的两个 word. 我们将其融合成一条 dlw 指令:

```
dlw $rt, imm'($rs)
```

它完成的操作为

```
rt ← memory[rs + (signed extended)imm]
rt + 1 ← memory[rs + (signed extended)(imm + 4)]
```

同理,对 sw 也可以融合成 dsw.

b. $slt + bne$

对于形如下面的指令序列:

```
sltu $2, $2, $3
bne $2, $0, offset
```

通过 sltu 的扩展指令可以变成如下形式:

```
sltui $2, $3
bne $2, $0, offset
```

bne 的一个寄存器为 0,另一个正好是 sltu 的目的寄存器.根据此特殊点,我们将这样的指令序列融合成如下形式:

```
sltui_bne $2, $3, offset
```

它节省了一次 bne 的 ALU 运算,根据 sltui 的运算结果,直接判断 zero 标志即可达到 bne 比较 \$2 是否为 0 的效果.

还需要调整融合后的指令的 offset 域,指令融合的实际效果相当于将 bne 向前移了一条指令的位置,因此要将 offset 变成 $offset + sizeof(\text{一条指令字的长度})$.

4 编译器扩展

编译器支持前述 exMIPS 扩展指令,将降低硬件解码时的开销.编译器层的实现很灵活,可以尝试多种优化算法,扩展性很强,有利于进一步扩大可融合指令集范围.我们将基于 Linux 系统下的 gcc-binutils 编译工具链进行扩展.

gcc 后端处理完成各种优化、寄存器分配和汇编代码的生成.为了把 GCC 应用到一个新的目标平台,需要提供一个目标机器描述文件 TARGET.md,用于描述目标机器的特征.我们将在 md 文件中使用 RTL 表达式描述 exMIPS 指令.

以 add \$1, \$1, \$2 为例,将通过如下 RTL 表达式生成 add2 \$1, \$2 的 exMIPS 指令.

```
(define_insn "addsi3_internal"
  [(set (match_operand:SI 0 "register_operand" "=d")
        (plus:SI (match_operand:SI 1 "register_operand" "0")
                  (match_operand:SI 2 "register_operand" "d")))]
  ""
  "add2\\t%0,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "SI")
   (set_attr "length" "1")])
```

对于生成的新的 add2 指令需要分配新的 opcode,这是在 binutils(包含链接工具)中进行的,如下所示,结构体中前 3 个域分别表示指令名称、操作数类型、操作码.

```
{ "add2", "v,t", 0x000000F1, 0x0000ffff, 0 },
{ "sub2", "v,t", 0x000000F2, 0x0000ffff, 0 },
{ "addi2", "t,j", 0x000000F3, 0x0000ffff, 0 },
{ "subi2", "t,j", 0x000000F4, 0x0000ffff, 0 }.
```

当编译器可产生多种可选指令序列时,根据数据相关性确定指令区块范围,以前述 exMIPS 指令集融合条件为指导,调整指令顺序,产生融合指令,图 12 显示了这一过程.

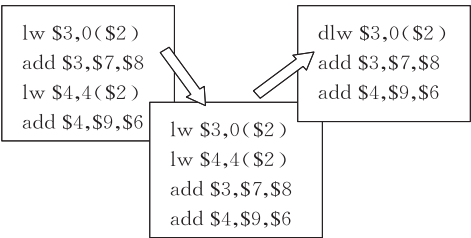


图 12

5 实验环境及结果

5.1 实验环境

本节以 SimpleScalar^① 为实验平台进行验证测试。SimpleScalar 是一款极好的体系设计模拟和验证平台,许多体系结构设计都是采用它进行验证,例如龙芯 CPU 模拟器 Sim-Godson^[11]。

SimpleScalar-3.0 包含 PISA 体系,指令长度设计成 64-bit,如图 13 所示。

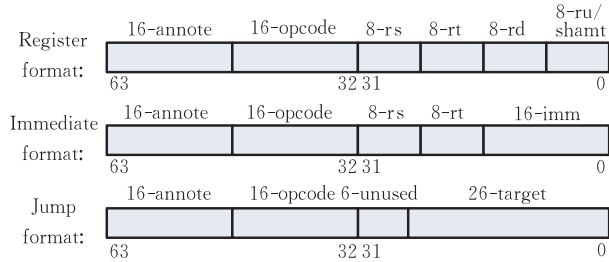


图 13 SimpleScalar 体系指令格式

除了位数的区别,SimpleScalar 的 PISA 体系指令格式与 MIPS 体系实际上是一致的。SimpleScalar 的流水线模拟器 sim-outorder^[12] 包含 5 个流水阶段:fetch、dispatch、issue、writeback、commit,内含保留站和重构序缓存,支持乱序执行。SimpleScalar3.0 的 sim-outorder 模拟器使用的默认参数如表 3 所示,所有的模拟都是 little-endian 模式下运行的。

表 3 sim-outorder 默认模拟参数

SimpleScalar 参数	值
fetch queue size	4
fetch speed	1
decode width	4
issue width	4 out-of-order, wrong-path issue included
commit width	4
RUU (window) size	16
LSQ	8
FUs	alu: 4, mul: 1, memport: 2, fpalu: 4, fpmult: 1
branch prediction	2048-entry table of 2-bit counters, 4-way 512-set BTB, 3 cycle extra mispredict latency, non-speculative update, 8-entry return address stack

(续 表)

SimpleScalar 参数	值
L1 D-cache	128-set, 4-way, 32-byte lines, LRU, 1-cycle hit, total of 16KB
L1 I-cache	512-set, direct-mapped 32-byte line, LRU, 1-cycle hit, total of 16KB
L2 unified cache	1024-set, 4-way, 64-byte line, 6-cycle hit, total of 256KB
memory latency	18 cycles for first chunk, 2 thereafter
SimpleScalar parameter	Value
memory width	8 bytes
Instruction TLB	16-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty
Data TLB	32-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty

通过扩展 sim-outorder 模拟器,我们加入了实现指令融合的硬件模拟模块,在 sim-outorder 的指令预取阶段,判断并转化 exMIPS 指令。

由于预取指令队列一次可容纳多条指令,在单条指令发射执行过程指令预取模块处于空闲状态,指令融合工作利用这段空闲,完成对指令的融合工作,因此指令融合模块不会对指令融合后整体性能有太大影响,也不会影响到我们对性能提升比的测试结果。

5.2 测试结果

我们在 sim-outorder 模拟器上实现了基于 PISA 体系的宏融合技术,测试程序使用 SPEC95 整数测试程序集以及几个 SPEC2000 测试程序^②,测试程序都是在 SimpleScalar 环境下由 gcc-2.7.2.3 (移植到 PISA 指令集)编译。测试程序集的数据输入如表 4 所示,其中有些数据集做了适当的修改^[13],使得 sim-outorder 能够在合理的时间内完成模拟运行。

表 4 测试程序的输入说明

Category	Benchmark	输入
SPECint95	compress	25000 e 2231
	gcc	1stmt.i(包含 4800 多行 C 源代码)
	go	50 9 2stone9.in
	jpeg	-image_file ijpeg.ppm
		-compression.quality 50
SPECint2000	li95	-compression.smoothing_factor 25
		-verbose 1
	perl	queens2.lsp(八皇后游戏)
	perl	perl-tests.pl
	parser	dict.tiny.3.0 <test-2.0.batch

① SimpleScalar LLC, SimpleScalar 3.0, available at <http://www.simplescalar.com>
② The SPEC95 and SPEC2000 PISA binaries, contained in the MIRV Benchmarks Suite, available at <http://www.eecs.umich.edu/mirv/benchmarks/benchmarks.html>

表 5 显示了测试结果,测试程序名字中含有 O0、O1、O2 的表示分别在 gcc 编译时开启了-O0、-O1、-O2 编译选项. n_inst 是测试程序运行的指令总和,n_macrofused 表示运行的 n_inst 条指令中被融合的指令数目. old_CPI 为指令融合之前 CPI, new_CPI 表示实施指令融合技术后的 CPI, improved 表示宏融合技术带来的效率提升百分比.

表 5 测试结果

文件	n_macrofused	n_inst	old_CPI	new_CPI	improved/%
compress95.O0.ss	12179151	130827770	0.5251	0.4762	9.31
compress95.O1.ss	3719495	82434997	0.59	0.5634	4.51
compress95.O2.ss	3420560	80415059	0.5834	0.5586	4.25
compress95.ss	3423396	80432368	0.5833	0.5585	4.26
gcc00.O0.ss	108984274	1079038567	1.0783	0.9694	10.10
gcc00.O1.ss	100527239	752995009	1.0435	0.9042	13.35
gcc00.O2.ss	102450609	734787866	1.0476	0.9015	13.94
gcc95.O0.ss	13316901	165153738	1.1225	1.032	8.06
gcc95.O1.ss	12511815	120557551	1.1004	0.9862	10.38
gcc95.O2.ss	12384629	119154001	1.1493	1.0298	10.39
gcc95.ss	12361599	119260008	1.1136	0.9982	10.37
go.O0.ss	70209990	1111158745	1.0318	0.9666	6.32
go.O1.ss	50283299	599237244	1.081	0.9903	8.39
go.O2.ss	45189926	554937499	1.0547	0.9688	8.14
go.ss	45183534	548130577	1.1376	1.0438	8.24
jpeg.O0.ss	33707620	501734216	0.5364	0.5004	6.72
jpeg.O1.ss	20193529	267940149	0.481	0.4447	7.54
jpeg.O2.ss	14089438	262373165	0.4676	0.4425	5.37
jpeg.ss	17031880	262007628	0.4695	0.439	6.50
li95.O0.ss	2147949	24397985	0.7322	0.6677	8.80
li95.O1.ss	1708515	16453937	0.7379	0.6613	10.38
li95.O2.ss	1649676	16257985	0.7705	0.6923	10.15
li95.ss	1702328	15992993	0.6767	0.6047	10.64
perl.O0.ss	32251748	274420368	1.2391	1.0935	11.75
perl.O1.ss	24571905	229794097	1.2205	1.09	10.69
perl.O2.ss	24980530	229290815	1.1993	1.0686	10.89
perl.ss	23923724	227850706	1.1979	1.0721	10.50
parser00.O0.ss	12793790	125028236	0.6694	0.6009	10.23
parser00.O1.ss	11103218	83931832	0.5338	0.4632	13.23
parser00.O2.ss	9530793	81942325	0.5496	0.4857	11.63
gzip00.O0.ss	737203	672178840	0.4245	0.424	0.11
gzip00.O1.ss	2364632	373221246	0.5163	0.513	0.63
gzip00.O2.ss	2428745	349655515	0.5232	0.5196	0.69
				average:	8.38

从图 14 可以看出,实现指令融合后,大部分测试程序的 CPI 都有所下降,与之对应的是运行时间的缩短,效率提升图(图 15)显示了大部分测试程序都有 8% 以上的效率提升,平均值为 8.38%.

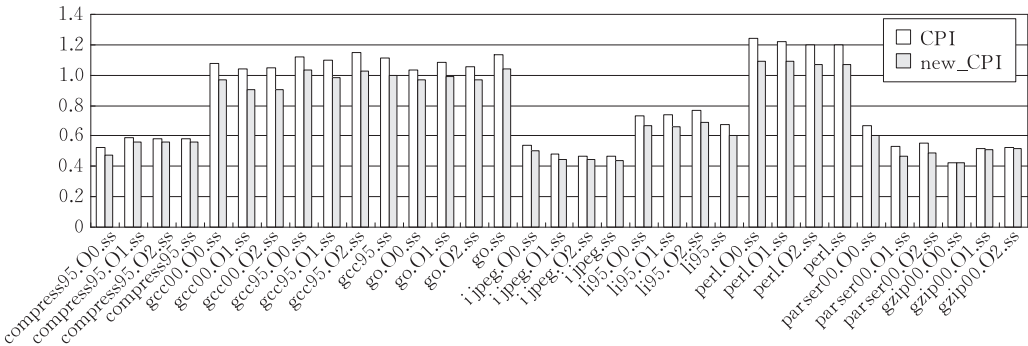


图 14 扩展指令融合前后 CPI 对比图

从图 15 可以看出, gcc 目前的优化(-O1, -O2)对指令融合没有太大的影响. 指令融合可由硬件单独承担,在程序缺乏源代码时,根据其二进制代码中指令融合的静态特性,可

以得出程序的“融合指令”数目,这一数目与融合前指令总数的比例就是代码压缩率(图 16 虚线).图 16 还显示了程序实际运行时的性能提升比(实线),从两线的对比来看,有较大代码压缩率的,不一定能在运行时有很大的性能提升(见 gzip);反之亦

然.主要原因是,代码整体的可融合指令比率不能代表一个代码块里的可融合指令比率,虽然整体的比率不高,但是程序有可能 90%的时间都运行在某一段融合指令较多的代码块.

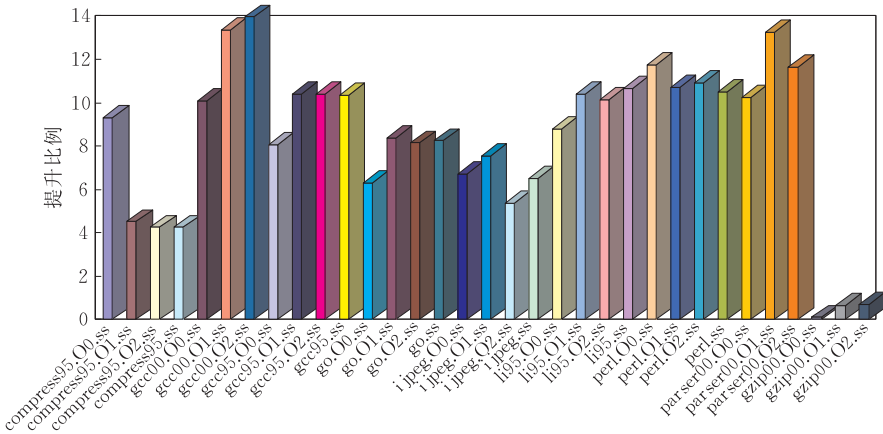


图 15 扩展指令融合的效率提升比例

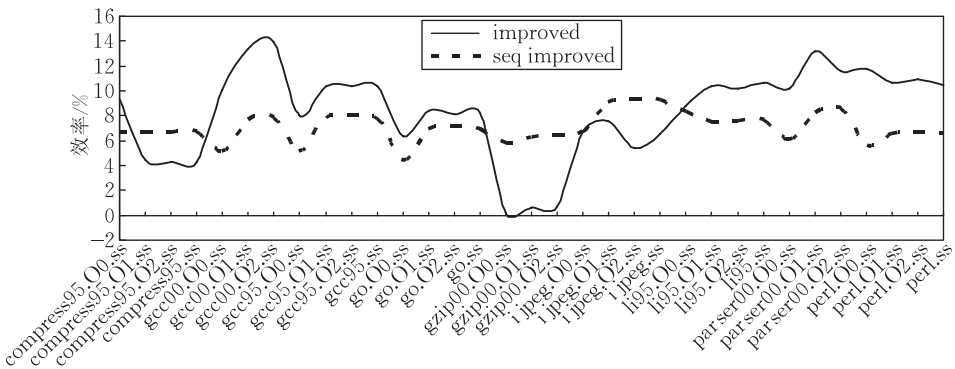


图 16 效率对比

6 结论和未来工作

本文设计和实现了基于 MIPS 体系结构的扩展指令融合技术,提出 exMIPS ISA 扩展指令集,对 MIPS 指令作了扩展和优化,增加了指令可融合比率.在硬件层对指令融合条件进行判断,CPU 运行时对指令动态转化和融合.在 SimpleScalar 实验环境,通过对 sim-outorder 模拟器的修改,验证了该技术.从测试结果来看,平均可以有 8%的效率提升.

文章对 gcc 编译器进行了扩充,从软件角度来支持 exMIPS ISA,目前这方面的工作仅仅完成一些简单的优化,比如一段代码可以转换成多种指令序列时,编译器尽量产生可融合的指令序列.由于这部分扩展性很强,对于更复杂的情况可以在后继工作中不断实现和验证.

另外,目前的 exMIPS ISA 没有尝试压缩指令中用于寄存器索引的域,未来工作将尝试类似 ARM 的 16bit 的 Thumb 指令集的设计思路,对寄存器索引域也进行压缩.相信会有性能上的进一步提升.

参 考 文 献

[1] Hu Shi-Liang, Smith James E. Using dynamic binary translation to fuse dependent instructions//Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. Palo Alto, California, 2004: 213

[2] Marr Deborah T, Binns Frank, Hill David L, Hinton Glenn, Koufaty David A, Miller J Alan, Upton Michael. Hyper-threading technology architecture and microarchitecture. Intel Technology Journal, 2002, 6(1): 4-15

[3] Gochman S et al. The Intel Pentium M processor: Microarchitecture and performance. Intel Technology Journal, 2003, 7(2): 21-36

[4] Gochman Simcha, Mendelson Avi, Naveh Alon. Introduction to Intel® Core™duo processor architecture. Intel Technology Journal, 2006, 10(2): 89-97

[5] Nguyen Khang. Preparing applications for Intel® Core™ microarchitecture. Technology@Intel Magazine, 2006, 1-9

[6] Klaiber A. The technology behind crusoe processors. Transmeta Technical Brief, 2000

[7] Ebcioglu K et al. Dynamic binary translation and optimization. IEEE Transactions on Computers, 2001, 50(6): 529-548

[8] Krishnaswamy Arvind, Gupta Rajiv. Enhancing the performance of 16-bit code using augmenting instructions//Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems. San Diego, California, USA, 2003: 254-264

[9] Krishnaswamy Arvind, Gupta Rajiv. Dynamic coalescing for 16-bit instructions. ACM Transactions on Embedded Computing Systems (TECS), 2005, 4(1): 3-37

[10] Cmelik R F, Kong S I, Ditzel D R, Kelly E J. An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks. SIGOPS Operating Systems Review, 1991, 25 (Special Issue): 290-302

[11] Zhang Fu-Xin, Zhang Long-Bing, Hu Wei-Wu. Sim-Godson: A godson processor simulator based on SimpleScalar. Chinese Journal of Computers, 2007, 30(1): 68-73(in Chinese)
(张福新,章隆兵,胡伟武. 基于 SimpleScalar 的龙芯 CPU 模拟器 Sim-Godson. 计算机学报, 2007, 30(1): 68-73)

[12] Chung Sung Woo, Park Gi Ho, Suh Hyo-Joong, Kim Han Jong, Im Jung Bin, Park Jung Wook, Park Sung Bae. Sim-ARM1136: A case study on the accuracy of the cycle-accurate simulator. Microprocessors and Microsystems, 2006, 30(3): 137-144

[13] KleinOsowski A, Flynn J et al. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research//Proceedings of the International Conference on Computer Design. Austin, Texas, USA, 2000: 73-82



CHEN Wen-Zhi, born in 1969, Ph. D. , associate professor. His main research interests include high performance computer architecture, trustable operating system and embedded system.

JIANG Zhen-Yu, born in 1984, master. His main research interests include micro-processor design and operating system.

WU Fan, born in 1984, master. His main research interests include operating system and embedded system.