维普资讯 http://www.cqvip.com

ORC的代码生成的关键技术*)

薛丽萍　张兆庆
(中科院计算技术研究所先进编译技术研究组　北京100080)

摘　要　处理机的指令级并行能力只有通过编译器才得以发挥。代码生成是编译器后端中直接针对目标处理机的优化阶段的集合，因而它是提高目标处理机指令级并行的重要组成部分。ORC(Open Research Compiler)是一款针对IA-64体系结构的开放源码编译器，它具有丰富的优化阶段和良好的性能。本文介绍了ORC的代码生成的关键技术，这些技术包括区域构造、谓词相关优化、全局指令调度、参数化的机器描述以及微调度。实测结果表明，这些技术取得了显著的优化结果，ORC相对于Pro64能取得平均23%的加速比。

关键词　ORC，IA-64，Pro64，代码生成

The Key Phase of ORC Code Generator

XUE Li-Ping　ZHANG Zhao-Qing
(Research Groop for Advanced Compile Technology, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080)

Abstract　The potential instruction level parallelism (ILP) of modern processors can only be delivered by compiler. Code generator is a suite of target processor specific optimization in a compiler's back-end. And hence, it is a very important component of a compiler in exploiting ILP. ORC (Open Research Compiler) is an optimizing open source compiler targeted for IA-64 architecture. It features rich optimization and very good performance. This paper introduces some key phases of ORC code generator, including region formation, optimization regarding predicate, global code motion, parameterized machine model. Our result indicates that these optimizations are very effective. The experiment result indicates the code generator of ORC outperforms Pro64's by as much as 23% in average.

Keywords　ORC, IA-64, Pro64, Code generator

1　引言

ORC[1]是一款以IA-64[2]为目标体系结构的开放源码编译器。IA-64是显式并行的体系结构。类似于传统的VLIW体系结构，IA-64不在运行时动态调度指令，能够在一拍内并行执行的指令由编译器指定。显式并行简化了硬件的设计但同时对编译器的设计提出了很大的挑战。ORC采用的一系列优化技术充分发挥了IA-64体系结构的指令并行的能力。ORC派生于SGI公司的开放源码编译器Pro64，而后者是从SGI的产品编译器MIPSpro移植来的。如图1所示，ORC主要由5个部分构成：前端(FE)、过程间分析与优化(IPA)、循环嵌套优化(LNO)、全局标量优化(WOPT)和代码生成(CG)。基于符号表的中间表示WHIRL作为各个部分通讯的"载体"。WHIRL是层次结构的，共计6层。主要的5个部分所使用的中间表示或其输出结果的中间表示如图1所示。

代码生成是一个将中间表示转化成目标处理机的汇编代码并充分挖掘目标处理机的指令级并行能力的阶段。ORC代码生成的主要阶段(phase)及其时序关系如图2所示。以下简单描述各个阶段的功能。

·Profiling 通过对目标代码插桩(instrumentation)获取目标代码的动态运行时的信息并将这些信息通过反馈文件标注(Annotation)在目标代码的中间表示上以指导编译优化。

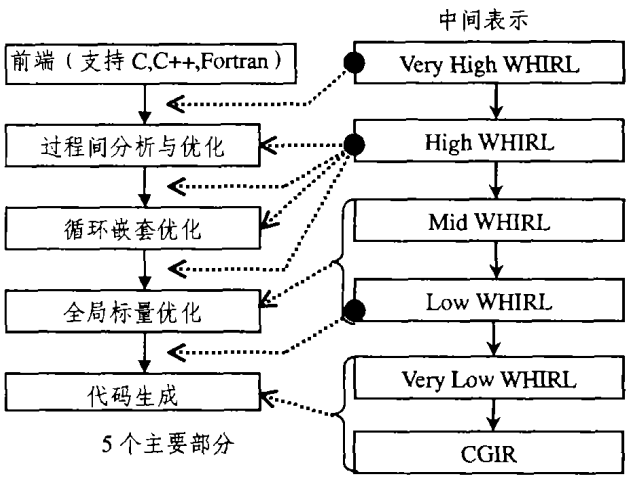


图1　ORC的主要部分与中间表示

·区域构造阶段以循环为边界将整个控制流图抽象为若干层次组合的单入口多出口控制流子图，即区域。区域是ORC代码生成中许多阶段的优化边界，例如if-conversion、全局指令调度以及谓词(predicate)关系分析等。

·If-conversion[3]利用IA-64硬件上的谓词支持，将程序中的控制依赖转化成数据依赖(直接效果是将一个控制流子图复合成一个谓词化的基本块)。它有效地消除了部分分枝指令及其预测失败(mis-prediction)所带来的开销。If-conversion

*)本研究受国家836计划(合同号2001AA11061)和国家自然科学基金(批准号69933020)的资助。

维普资讯 http://www.cqvip.com

还利用并行比较的硬件支持将串行的比较序列转化成并行的比较集合，从而缩短了关键路径的长度，减小了目标代码的执行时间。

·循环优化包括循环展开和软流水(software pipelining)，这两种优化阶段目前只施用于单个基本块(可以是通过 if-conversion 谓词化的基本块)的循环。软流水主要基于文[4]。由于软流水循环的寄存器分配的特殊性(例如要分配 rotating register)，软流水阶段本身包含了寄存器分配器(算法原型详见文[5])并且在对一个循环软流水结束之后立即对其分配寄存器。经过的软流水循环将不再进行图2所示的"全局指令调度"、"寄存器分配"和局部指令调度。

·全局指令调度的调度范围是单入口的无环区域。全局调度器充分利用 IA-64的大寄存器组、功能部件多等特点，通过调度指令跨越基本块的边界以增强指令级并行度。为了减小全局指令调度的复杂性，全局指令调度的一些功能由相对独立的工具模块负责。

·寄存器分配采用的算法[6]，具体划分为全局寄存器分配和局部寄存器分配。它们为跨越基本块边界的活跃区间和局限于基本块的活跃区间分配寄存器。

·局部指令调度器对如下基本块作局部指令调度：①在全局调度阶段没有被调度过的基本块；②在全局调度之后，指令有被增加或删除的基本块。

·代码发射阶段将中间表示转变成汇编文件。

在 ORC 中，Profiling、区域构造、if-conversion、全局以及局部指令调度是 Pro64没有的或取代其对应的阶段。本文主要介绍这些阶段。此外，ORC 还(相对于 Pro64)为这些优化阶段提供了一些方便的"工具模块"(例如机器模型、PRDB 等)。在本文中，"工具模块"也被视为优化阶段并予以介绍。

本文第2节介绍区域构造；第3节描述谓词相关的优化，包括 if-conversion 和谓词查询数据库；第4节介绍关于全局指令调度及其子模块；第5节描述 ORC 对目标处理机抽象的两个主要模块：参数化的机器模型和微调度；第6节给出了实验结果；最后总结全文。

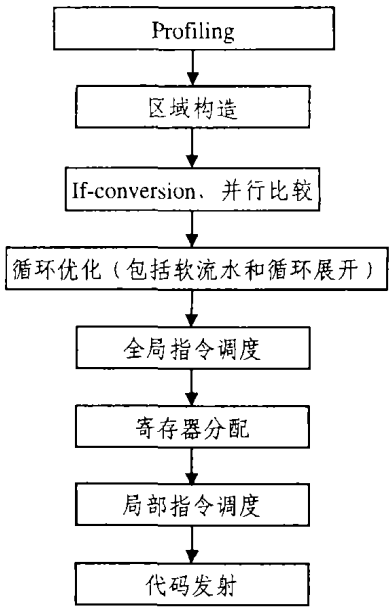


图2　代码生成的主要阶段及其时序

2　区域构造

在编译器后端中，许多优化只能在特定的控制流上才能进行。例如，有些软流水算法仅仅作用于内层循环，无环指令调度通常不能调度指令跨越循环的边界。因此，我们需要将适宜于优化的全局控制流子图用相对简捷的表示描述出来。此外，如果优化阶段一次处理的输入集过大，编译的时间和空间开销将会很大(许多算法的复杂度是非线性的)。在 ORC 中，区域这个数据结构就是为了上述两个目的而设计的。在代码生成中，基于区域的优化阶段有全局指令调度、If-conversion 以及谓词查询。对于这些阶段来说：区域划定了一个优化的边界并且控制了编译时间和空间的开销。

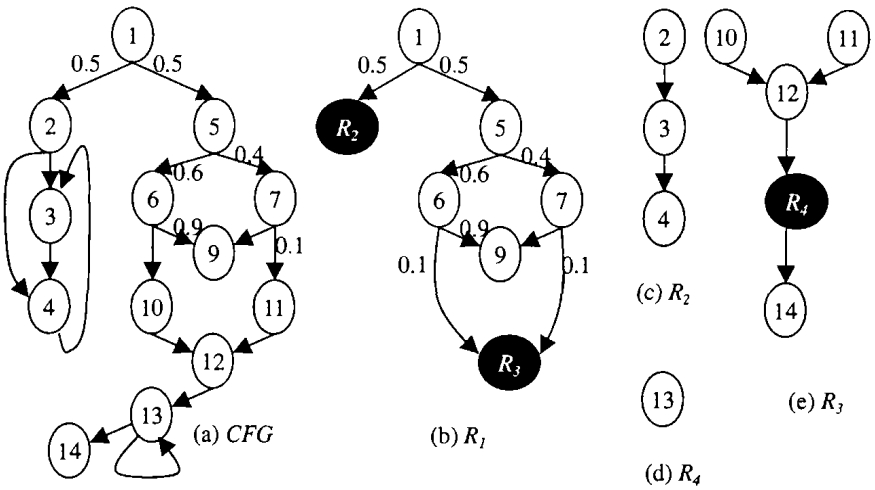


图3　区域构造

整个全局控制流被抽象成一个区域，任何循环(为说明方便，在本文中"循环"仅指自然循环而不包括不可规约循环)被表示成一个区域，内层循环对应的区域在外层循环中表示为一个抽象结点。因为，依赖于区域的阶段的优化范围不跨越循环迭代的边界以及循环的边界，因此，在循环的区域表示中，回边被忽略。为了控制编译时空开销，太大的区域要被分割成若干小区域并且在外层区域中用抽象结点表示出来。分割区域应该让相关性强(在控制流上，如果一个基本块到另一个基本块的概率大，我们称这两个基本块的相关性强)的基本块尽量聚集在同一个区域中。此外，分割的过程中还应该注意到尽量不要构造多入口区域(即控制流可以从多个结点进入该区域)，因为有些算法只作用于单入口的控制流上。对于不可归约循环，我们也用区域表示它。由于不可归约循环出现的可能性很小(在结构化程序中不可能出现)，许多算法也不适合于

维普资讯 http://www.cqvip.com

不可归约循环，因此对于基于区域的阶段来说，不可归约循环区域的意义在于它划定了一个不作优化的范围。

综上所述，区域共有4种：①Loop 区域，对应于一个循环；②SEME 区域，对应于一个单入口多出口非循环的控制流子图(Loop 区域可能嵌套 SEME 区域)；③Improper 区域，对应于一个不可归约循环；④MEME 区域，对应于一个多入口多出口的非循环控制流子图。

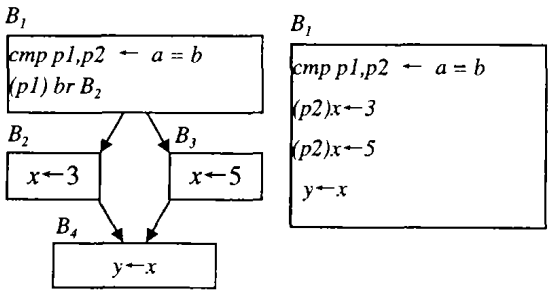
在这里，我们用一个例子说明区域的构造过程。在图3中，基本块用白圈表示，嵌套区域用黑圈表示。有向边表示控制流，旁边的标注表示它的执行概率。编译单元的控制流图如图1(a)所示，其中循环{13}被抽象为 Loop 区域 R_4，Improper 区域 R_2代表了不可归约循环{2,3,4}，剩下的控制流部分{1,5,6,7,9,10,11,12,14}用区域 R_1表示，在这里我们假设大小合适的区域不应该包含多于5个基本块，于是，我们要把 R_1分割成一些小区域：由于{1,5,6,7,9}的相关性较大，我们把它继续保留在 R_1中，而其余的都划分给另外一个区域 R_3。R_1现在成为 R_3外层区域，所以在 R_1中，R_3用一个抽象结点表示。在分割一个大基本块时，尽管我们希望尽量构造 SEME 区域，但由于区域大小的所限，在这里我们只能构造一个 MEME 区域 R_3，它有2个入口，即块10和11。

文[8]详细介绍了区域构造的算法及其启发性方法。

3 谓词相关优化

3.1 If-conversion

谓词(或条件执行)是 IA-64体系结构的一个重要特色。编译器可以通过 If-conversion 将不同执行路径上代码谓词化后复合到一个基本块中(如图4所示)，从而消除了部分分枝指令由于预测失败带来的开销。谓词还给指令调度带来方便，调度器可以以非投机的方式向上调度指令跨越分枝指令。If-conversion 的意义还在于能够使一些只用作于单个基本块的算法应用于多个基本块的控制流中。例如在 ORC 中，循环展开和软流水只应用于单个基本块，If-conversion 将一些相邻的基本块复合成一个谓词化的基本块后，这两个优化算法就能够事实上应用于一组相邻的基本块中。



(a) 原控制流图　　(b) 复合基本块

图4 If-conversion

类似于指令调度和谓词查询处据库，If-conversion 的作用域为 SEME 区域。If-conversion 阶段以自顶向下的拓扑顺序在区域中寻找能够且适合于进行 If-conversion 的控制流子图。假如找到这样的控制流子图，If-conversion 阶段把控制流子图中的基本块复合成一个基本块，随后，将上述的控制流子图从控制流中删除并把复合基本块加入到控制流中。上述过程迭代地进行直到区域中不再出现"能够且适合于进行 If-conversion 的控制流子图"。"能够进行 If-conversion"的控制流子图局限于优化阶段的能力。目前，下列3种结构的控制流子图属于"能够"之列：①串行结构；②If-then 结构；③if-then-else 结构。"适合于进行 If-conversion"是指通过 If-conversion 能够提高目标代码性能。If-conversion 阶段需要考虑这些因素判断一个控制流子图是否"适合"于 If-conversion：不同路径上关键路径的长度、硬件资源的使用情况、分枝指令预测失败率和预测失败代价等。

除了"复合"一个控制流子图为一个谓词化的基本块外，If-conversion 还充分利用并行比较的硬件支持将一个串行的比较序列转化成一组并行比较指令，从而缩短了关键路径的长度，减小了目标代码的执行时间。

3.2 谓词查询数据库

谓词查询数据库(PRDB)顾名思义是用来查询谓词之间关系的数据库。我们关心的谓词之间的"关系"包括：在任何时刻两个谓词是否都是互补的，在任何时刻一个谓词为真值是否蕴含另一个谓词也为真值等等。尽管 PRDB 与 If-conversion 阶段都是针对谓词的优化，但是两者之间没有任何关系。PRDB 可以作为一个工具模块被其它谓词敏感的阶段独立地创建和删除，而且这些优化阶段可以在对作优化变换的同时对 PRDB(通过它的高度抽象的接口)增量式地维护。类似于 If-conversion 和全局指令调度，PRDB 建立的谓词关系局限于单入口多出口的无环区域。

目前，PRDB 在 ORC 中被应用于指令依赖关系图的创建和指令调度上。它还可以应用于数据流分析等谓词敏感的分析和优化上。PRDB 主要的算法原型是文[7]。与文[7]不同的是 PRDB 并不假定中间代码满足 SSA 式(SSA form)，因为 ORC 的代码生成的中间表示是不满足这个假定的。

4 全局指令调度

4.1 全局指令调度框架

全局指令调度(GLOS)能够在任何一个单入口的无环区域(即 loop 区域或 SEME 区域)内调度指令。GLOS 主要采用算法[9]，在结合 IA-64体系结构特点的同时，GLOS 还集成了 P-ready 指令调度[10]并且能够调度指令跨越嵌套循环。由于全局指令调度在寄存器分配之前进行，调度器可以通过 renaming 和 forward substitution[11]使得更多的指令能够被投机。目前，GLOS 并不支持向下的指令调度。向下指令调度对调度质量影响很小，而向下调度却又十分复杂。

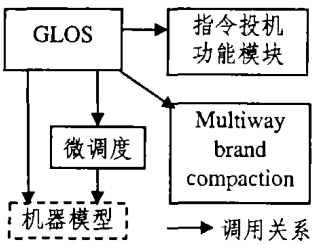


图5 从 GLOS 中分离的三个模块

为了减小调度器的复杂性增强它的可维护性。三个逻辑上属于指令调度但功能上相对独立的模块从 GLOS 中分离出来，它们是"指令投机功能模块"、"微调度"和 Multiway-branch-compaction，如图5所示("机器模型"不属于 GLOS)。"指令投机功能模块"主要功能是跟踪投机链(speculative train)并以此生成恢复代码(recovery code)。"微调度"管理一个指令组中资源占用情况、选择"最好"的 template 等、向调度器抽象处理机细节等。由于"微调度"高度机器相关，我们将

维普资讯 http://www.cqvip.com

在目标机抽象一节中予以介绍。Multiway-branch-compaction的功能是利用Multiway-branch硬件支持将相邻的分枝指令合并到一拍中执行。这个阶段在寄存器分配之后进行。

调度器通过调度编译单元(通常是一个函数)的区域层次结构表示中的每一个区域(被软流水调度过的循环和不可规约循环例外)构成对该编译单元的调度。调度器从区域层次结构最内层到它最外层的顺序调度区域,优先频率高的区域。

在调度一个区域时,调度器以自顶向下的拓扑顺序遍历区域中的每一个结点(基本块或嵌套区域),优先频率高的结点。在访问一个基本块 B 时,调度器调度 B 中的指令同时还将其它基本块中的指令调度到 B 中,同时做代码补偿(code compensation)。例如在图6所示的区域 R 中,基本块 B_1 是 R 唯一的入口结点。调度器遍历 R 的顺序是先访问入口结点 B_1,此后,按照拓扑顺序,B_2 或 B_3 是下一个访问结点。由于,B_2 的频率比 B_3 高(在图6中,B_1 到 B_2 的概率是0.9),调度器选 B_2 作为下一个访问结点。接着,调度器依次访问 B_3 和 B_4。

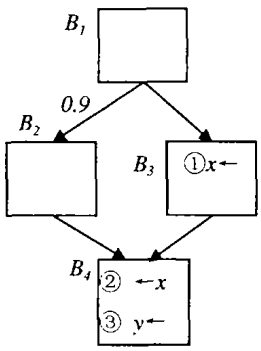


图6

在访问一个基本块(设为块 B)时,调度器不仅对该基本块作指令调度,任何 B 能够到达的基本块(局限于当前区域)中的指令都有可能被调度到 B 中。至于不在 B 中的指令是否真的被调度到 B 中取决于这些指令在整个区域中的优先级。

调度器既可以调度M-ready候选指令[9]也可以调度P-ready候选指令[10]。在图6中,由于从 B_1 到指令③的任何一条路径上都没有③所依赖的指令,因此指令③相对于 B_1 M-ready,指令③可以直接被调度到 B_1 中而不需要代码补偿。而指令②则不然,它相对于 B_1 P-ready(由于它依赖于指令①),若②调度到 B_1 中,调度器需要复制③-作为补偿代码-插到 B_3 的末尾。与文[10]不同的是,ORC的P-ready指令调度在调度的同时生成补偿代码,而文[10]滞后代码补偿直到一个恰当的时间。

为了调度指令跨越嵌套区域,调度器结束一个区域的调度后需要收集这个区域的数据流信息并且用简捷的数据结构表示。这个数据结构称为区域的概要信息。在调度外层区域时,调度器根据嵌套区域的概要信息判断跨越这个嵌套区域的指令调度是否合法。

全局指令调度对控制流有很大的改变:在指令调度之前,调度器需要通过edge-splitting消除critical-edge;在调度的过程中由于对load的控制和数据投机需要插入恢复块(见下一节);在指令调度之后一些基本块中的指令全都被调度到其它基本块中。这些改变给控制流优化和code layout以很大的空间。在整个编译单元的调度结束之后,ORC通过调用控制流优化阶段(未在图2中画出)对控制流进新优化。

4.2 控制投机和数据投机

控制投机是指将load提前到尚不知道这条指令是否会被执行的地方执行。数据执行是指将load提前到与它alias的store之前执行。投机使得关键路径上的指令被提前执行从而缩短了目标代码的执行时间。IA-64为这两种投机提供了硬件支持。如果一条load指令被数据投机或一条unsafe load指令被控制投机,调度器要在程序中插入chk指令以检查投机是否成功。如果投机是成功的,处理机继续执行chk之后的指令,否则,chk指令跳转向由编译器事先生成的恢复块重新执行被投机load对应的投机链(见下文)上的指令。

调度器权衡指令投机的利弊,而"指令投机功能模块"负责如下工作:

·确保失败的投机是可恢复的。一条load指令被投机后,依赖于这条指令的其它指令随后也会被投机。恢复失败的投机需要重新执行被投机的load和相应的chk指令直接或间接依赖于这条load指令的指令(包括load本身)。这条需要重新执行的指令序列我们称作投机链。因为投机链需要被重新执行,所以不是所有的指令都能够出现在投机链上。例如,在投机链上upward exposed use变量不能够在投机链上被赋值。"指令投机功能模块"在通过在调度的同时建立某些依赖关系防止一些指令出现在越过chk指令出现在投机链上。

·生成恢复块。在整个编译单元结束之后,"指令投机功能模块"根据在指令调度过程中记录下来的投机链信息创建恢复块并把这些恢复块加入到全局控制流中。

关于控制投机和数据投机及恢复块的生成详见文[12]。

5 目标机抽象

5.1 参数化机器模型

机器相关优化在现代编译技术中具有重要的作用。由于后端优化对机器高度依赖,使得编译器的可移植性受到了制约。为了提高编译器的灵活性和可移植性,需要通过机器模型把机器的硬件信息进行封装,与机器相关的优化可以通过机器模型提供的接口来获得所需的机器参数和机器状态。

过去的编译器把各种机器信息分散在编译优化的各个阶段,现代编译器为了提高灵活性和可移植性,将各种游离的机器信息有条理地组织起来单独形成一个部分,在ORC中,这个部分被称做机器描述,它是机器模型的一部分。机器描述模块描述和封装了指令集和微体系结构的信息,例如:功能部件、寄存器、延迟、和内存层次结构,它同时还提供了外部接口使得编译器的其它部分能够获得所需的机器参数和硬件信息。为了提高编译器的可移植性,机器描述部分还实现了机器描述生成器,它可以通过输入的内部机器描述文件(在ORC中,我们使用Knobsfile[13]作为内部机器描述文件)生成外部机器描述。编译器的其它相关模块在编译时刻可以通过机器描述查询接口查询外部机器描述来获得它们所需要的机器参数和机器状态。图7描述了机器描述部分的工作原理以及它与其它模块的交互方式。

5.2 微调度

由于指令调度的复杂性,我们希望把指令调度中与机器相关的部分尽可能地隔离出来,从而提高编译器的灵活性和可移植性。在ORC中,我们把这一部分称作微调度,它也是机器模型的一部分。

微调度主要完成两个方面的任务:(1)和指令调度模块进行交互,完成功能部件的检查和分配,并为指令选择相应的模板;(2)微调度需要能够重排一个周期中的指令顺序来消除指

维普资讯 http://www.cqvip.com

令序列中的停顿。微调度使得指令调度模块在执行时不用考虑与机器相关的一些约束条件，如功能部件的分配，指令模板的选择和机器的宽度等等。

从图7可以看出，微调度是如何与指令调度协同工作的：当指令调度选定一个待调度的指令后，它通过调用微调度来判断该指令是否能够合法地调度到指定的周期。微调度在判断这一调度是否合法需要考虑三个方面的因素：功能部件的分配、指令模板的选择以及依赖关系的检查。微调度首先为这个指令分配相应的功能部件，分配成功就形成一个合法的预留表，我们根据预留表查询自动机的状态（自动机的状态中保存了对应于这个预留表的所有合法指令格式）来选择合适的模板，在查询的同时也要兼顾到满足依赖关系。最后，微调度给出是否能在本周期中调度给定的指令。当指令调度最终决定要调度这一条指令时，指令调度调用微调度来实现指令的重排和拼装。

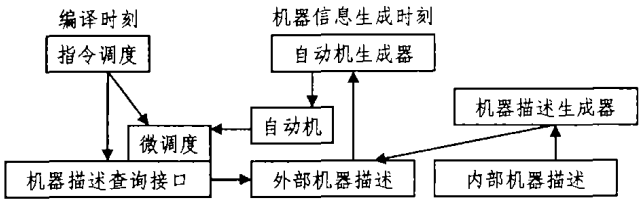


图7 参数化机器模型与微调度

关于微调度的具体实现详见文[14]。

6 试验数据

在这一节我们评估ORC代码生成的性能。用于试验的ORC版本是2.0[11]。测试例是SPEC CPU2000[15]的整型基准测试程序。运行测试例的机器是Itanium-1工作站（Itanium-1是IA-64体系结构的一个具体实现），频率为733Mhz，具有2MB L3 cache和1GB的内存。运行的操作系统是RedHat 7.2 for IA-64。

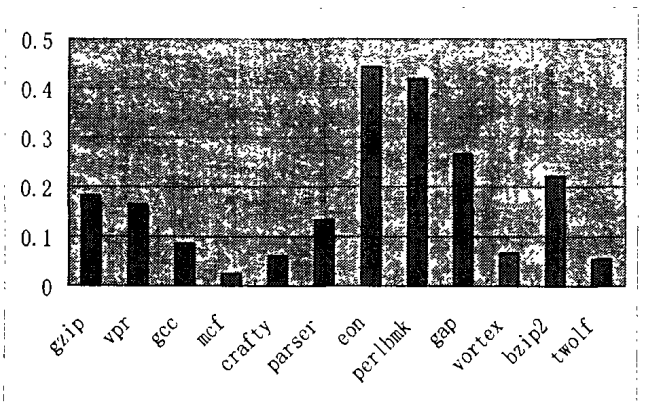


图8 ORC相对于Pro64的加速比

图8给出了ORC相对于Pro64的加速比。两者的编译选项均为O3。我们不打开IPA选项，这样使得两者的性能差别只和代码生成有关。从图8可以看出，ORC相对于Pro64对12个测试例子均有不同程度的提高，其中eon和perlbmk的加速比最为明显，分别达到了44%和42%。12个例子的平均加速比为23%。

总结 本文主要介绍了ORC的代码生成部分的关键技术。ORC充分利用了IA-64体系结构的特点，如谓词、控制投机和数据投机等硬件支持，来挖掘指令级并行的潜力。实验结果表明，ORC的代码生成充分发挥了IA-64体系结构的特点，相对于Pro64有23%的性能提高。IA-64作为一种新的体系结构，有很多的特性可以加以利用。ORC良好的性能为研究IA-64体系结构和基于区域的编译优化技术提供了良好的编译平台。

参考文献

1 ORC team, ORC 2.1 suite. http://sourceforge.net/projects/ipf-orc, 2001-2003
2 Intel Inc., http://www.intel.com/design/itanium/family/, 2003
3 Allen J R, et al. Conversion of control dependence to data dependence. In: Proc. 10th ACM Symp. Principles of Programming Languages, Jan. 1983. 177～189
4 Huff R. Lifetime-Sensitive Modulo Scheduling. In: Proc. of the Int. Conf. on Programming Languages, Design and Implementation, 1993. 318～328
5 Rau B, et al. Register allocation for software pipelined loops. In: Proc. of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, June 1992. 283～299
6 Chow F, Hennessy J. Register allocation by priority-based coloring. In: Proc. of the ACM SIGPLAN Symposium on Compiler Constructions, June 1984
7 Gillies D M, Ju D R, Johnson R, Schlansker M. Global predicate analysis and its application to register allocation. In: Proc. 29th Int. Symposium. Microarchitecture, Dec. 1996. 114～125
8 刘旸，张兆庆，乔如良. 基于区域的编译框架. 计算机学报，2003，26(2)：188～194
9 Bernstein D, et al. Code duplication: An assist for global instruction scheduling. In: Proc. of the 24th Annual Intl. Symposium on Microarchitecture, Dec. 1991
10 Bharadwaj J, et al. Wavefront Scheduling: Path-Based Data Representation and Scheduling of Subgraphs. In: Proc. of the 32nd Annual Intl. Symposium on Microarchitecture (MICRO32), Dec. 1999
11 Moon S-M, Ebcioglu K. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining. ACM Transactions on Programming Languages and Systems, 1997. 853～898
12 Roy, et al. A unified compiler framework for control and data speculation. In: Intl. Conf. on parallel architectures and compilation techniques, Oct. 2000
13 Intel Inc., KAPI Programmer's Guide. http://developer.intel.com, June 2000
14 Chen D-Y, et al. Efficient Resource Management during Instruction Scheduling for the EPIC Architecture. In: Intl. Conf. on parallel architectures and compilation techniques, Sept. 2003
15 Stan Performance Evaluation Corporation. http://www.spec.org/cpu2000, 2003

（上接第158页）

的规约叫复合规约。

这样一个任意复杂的代数就可以由循环递归的嵌套的标准代数的规约来表示，形成一种自然的层次结构，它使我们联想到面向对象设计中的类层次。因此这种模式规约的构成对于解决面向对象的框架应用是很有利的。

参考文献

1 Buschmann F, et al. Pattern-Oriented Software Architecture Volume: A system of Patterns. Publisher Wiley, John & Sons, Incorporated, Nov. 1995
2 Gamma E, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
3 周之英. 现代软件工程（下）新技术篇. 北京：科学出版社，1999
4 Loeckx J. Algorithmic specification. A constructive specification method for abstract Data Types. New York, ACM Transactions on Programming Languages and System, 1987, 9(4): 646～685
5 Caijia-mei. The Sequence Modeling Method Based on ECC in Developing Program Specifications. Journal of Computer Science and Technology, 1999, 14(7): 337～348