多媒体处理器的SIMD代码生成*)

吴圣宁　李思昆
(国防科技大学计算机学院　长沙 410073)

摘　要　通用处理器的SIMD(Single Instruction Multiple Data)多媒体扩展,为提高多媒体应用的性能提供了新的体系结构支持。但目前编译技术对这类指令不能提供很好的支持。本文提出了一个新的SIMD指令生成算法,基于把编译器前端的程序分析和编译器后端的机器信息相结合的思想,采用扩展的tree parsing技术,有效识别程序中的并行操作以生成SIMD指令。基于SUIF(Stanford University Intermediate Format)[1]编译器框架的实验表明,针对一组多媒体kernel,本文提出的算法可平均减少其非SIMD代码47%的cycles。

关键词　多媒体处理器,SIMD,编译技术

SIMD Code Generation for Multimedia Processors

WU Sheng-Ning　LI Si-Kun
(School of Computer Science, National University of Defense Technology, Changsha 410073)

Abstract　The SIMD (Single Instruction Multiple Data) extensions appeared in general-purpose microprocessors provide new architectural support for improving performance of multimedia applications, but current compiler techniques cannot exploit them well. Based on the idea of integrating program analysis of the compiler front-end and machine information of the back-end, this paper proposes a new algorithm for SIMD code generation, which can identifies program parallelism effectively to generate SIMD code adopting enhanced tree parsing techniques. The algorithm has been implemented in SUIF (Stanford University Intermediate Format) compiler infrastructure, and experimental results for a group of multimedia kernels show that the algorithm proposed can reduce 47% cycles on average compared to non-SIMD code.

Keywords　Multimedia processors, SIMD, Compiler techniques

1　引言

在多媒体应用中常需对长的数据流进行计算密集型操作,因此,很多嵌入式处理器甚至通用处理器都增加了扩展指令集,以对这类应用提供体系结构支持,例如,Intel的MMX/SSE/SSE2, IBM/Motorola的VMX/AltiVec, AMD的3DNow!,HP的MAX1/2,SUN的VIS,DEC的DVI,MIPS的MDMX/MIPS-3D。尽管不同的处理器提供不同类型和数量的扩展指令,但其核心是一组SIMD操作,一条指令就能并行地对一组数据执行同样的运算。

SIMD指令的实现简单且容易扩展,能利用程序基本块内的并行性,显著提高了多媒体应用的性能。但当前的编译技术不能对SIMD指令给予很好的支持。目前主要通过两条途径解决此问题:一是使用编译器intrinsics,即调用编译器已知的函数,每个这样的函数直接对应一条特定的SIMD指令;二是程序员用汇编语言编写程序。这两种方法都导致程序员利用SIMD指令的负担,而且编写的程序是机器相关的,移植困难。更好的办法是让程序员仅关心高级语言编程,处理器的指令集对程序员透明,而由编译器来自动选择SIMD指令。

指令选择是把源程序的机器无关的Intermediate Representation (IR)映射为处理器指令模式。传统的指令选择技术不能检测利用SIMD指令的机会,因而难以生成高质量代码。目前对SIMD指令选择的研究,或者局限于循环级的向量化,假定SIMD并行性已经在向量程序中显示地表达,或者没有考虑扩展指令集中的permutation类指令可对SIMD操作的一组数据重新排序,因而可创造更多利用SIMD指令的机会,或者过于复杂,难以实现。

本文提出了一个新的SIMD指令生成算法,基于把编译器前端的程序分析和编译器后端的机器信息相结合的思想,采用扩展的tree parsing技术,有效识别程序中的并行操作以生成SIMD指令。

本文第2节介绍相关工作;第3节描述本文提出的SIMD指令生成算法;第4节是试验结果和分析;最后是结论。

2　相关研究

传统的指令选择基于tree parsing,这种高效的编译技术通过tree pattern matching和dynamic programming实现[2]。但tree parsing要求程序IR是Data Flow Tree (DFT),而SIMD指令的生成需要同时考虑多个DFT,因此SIMD指令生成应基于Data Flow Graph (DFG)。

SIMD编译技术的一个重要方面是memory address alignment。当前体系结构一般仅支持aligned存取,否则需要permutation类指令重新组织数据,降低了程序性能。如何组织数据,使得permutation类指令数最小,是重要的研究方向[3,4]。但这些研究假定SIMD并行性已经在向量化程序中显示地表达。

SIMD编译技术的另一个重要方面是如何选择并行操作

*)本课题得到国家自然科学基金(90207019)和国家863计划(2002AA1Z1480)的资助。吴圣宁　博士生,主要研究领域为嵌入式系统编译器、操作系统、人工智能;李思昆　教授,博士生导师,研究方向为电子CAD、系统芯片设计方法学、虚拟现实。

以生成 SIMD 指令。Larsen[5,6] 提出了 Superword Level Parallelism (SLP) 的概念以及识别基本块内 SLP 的技术,但忽略了 permutation 类指令的生成。Leupers[7] 提出了基于 DFG 的 SIMD 指令识别技术,但也没有考虑 permutation 类指令的生成。

Kudriavtsev[8] 的方法同时考虑到上述 SIMD 编译技术的两个方面,但复杂度高,只能适用于小规模的 DFG。

3 基于 DFG 的指令选择

本文的 SIMD 指令生成算法是对 Leupers 算法[7] 的扩展:首先,增加了 permutation 指令的生成机制,以获得更多的并行操作;第二,把编译器前端的程序分析结果有选择地传递给编译器后端,而不完全依赖于后端局部信息,简化了 SIMD 指令的生成。

下面介绍一下 Leupers 算法,然后说明本文扩展的方法和算法流程。

Leupers 算法分为 DFG 覆盖和代码选择两个阶段。在 DFG 覆盖阶段中,通过给每个共同子表达式 CSE 分配一个符号寄存器来把 DFG 分解为 DFT,对此 CSE 的引用替换为对此寄存器的读操作。然后,基于扩展的 tree parsing 技术,各 DFT 分别被目标机器指令模式覆盖。在传统的 tree parsing 技术中,DFT 的每个节点所能规约到的每个非终结符只对应一条总开销最小的派生规则,如果对应某个非终结符有多个派生规则都有最小总开销,则任意保留其中一条规则。Leupers 算法中,DFT 节点对应的非终结符的所有最小派生规则都得到保留,具体的 SIMD 指令选择推迟到整个 DFG 全局代码选择阶段。

在第二阶段进行实际的代码选择。此时各 DFT 的每个节点都对应一组派生规则,各 DFT 中若干同构的操作可能形成一条 SIMD 指令,但需要满足一定的约束;以 SIMD 指令使用的最大化为目标函数,利用整数线性规划(ILP, Integer Linear Programming)求得满足约束的解,使得每个节点仅对应一条派生规则,则最终形成 SIMD 指令和非 SIMD 指令混合的目标代码。

本文的算法由多个阶段组成。进行循环展开以获得更多的并行操作,alignment 分析并且利用 SUIF 的 annotation 机制记录分析结果以传递给编译器后端;编译器后端利用扩展的 tree parsing 生成机器相关代码,把同构操作组合成 SIMD 指令。图 1 给出了总体流程,具体如下。

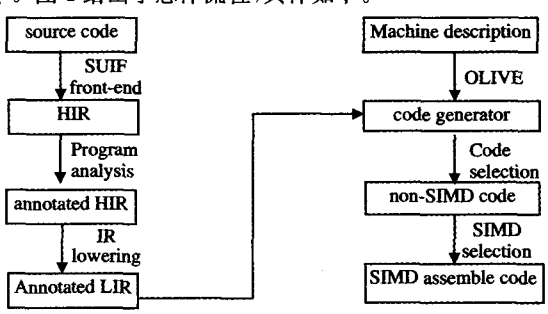

图 1 SIMD 代码生成的编译器流程

3.1 循环展开

由于多媒体应用中存在大量的循环结构,且循环展开有利于获得更多的并行操作,本文在编译流程的早期进行循环展开。循环展开的两个重要方面是,展开哪些循环以及 unrolling factor 的选择。由于本文主要考虑 SIMD 代码的生成,因此,仅展开最内层循环,且由简单启发式确定 unrolling factor,即为 SIMD 寄存器位宽和循环内数据类型位宽的商。假如循环内数据类型为 16 位,SIMD 寄存器 64 位,则循环需展开 4 次。

3.2 Alignment 分析

Alignment 分析确定存储操作相对于 SIMD 数据通路宽度的对齐状况。本文采用 Larsen[5] 的静态数据流算法来获取 alignment 信息。此算法把指针变量表示为 stride * n + offset,其中 n 是非负整数。在过程中的每个点,都给涉及到地址计算的任何变量附上一个 stride 和 offset。算法把所有基本块放入于一个 worklist 中,然后将其迭代地取出进行分析。每当一个基本块修改了其数据流信息,此基本块的所有前驱插入到 worklist 中。迭代过程到 worklist 为空时终止。

在一个基本块内,alignment 分析通过 transfer 函数传播数据流信息。首先用 meet 操作符合并变量来自其所有前驱基本块的数据流值,然后依次处理当前基本块内的操作。地址计算的主要操作是加、减、乘,用相应的 transfer 函数计算涉及到地址的变量的数据流信息。

这些数据流信息用 SUIF 的 annotation 机制记录在变量的相应域中,传递给编译器后端以辅助 SIMD 指令的生成。

3.3 DFG 覆盖

传统的 tree parsing 用机器指令模式覆盖 DFT,每个 DFT 的节点所能规约到的每个非终结符只对应一条总开销最小的派生规则,如果由 dynamic programming 为节点的某个非终结符产生多个代价同样小的派生规则,则任选其一。但在有 SIMD 指令的情况下,多个 DFT 中同构的操作有可能构成 SIMD 指令,因此,节点的同一个非终结符的所有最小派生规则应保留,而在后面的 SIMD 指令生成阶段再做最后的决定。

为表达可能的 SIMD 操作,本文引入专门的非终结符。例如,一条 ADD 指令把两个 32 位寄存器的值相加,结果写入一个 32 位的目标寄存器。利用 tree parsing 技术,这可以用如下树文法规则表示:

reg: PLUS(reg, reg)

其中,非总结符 reg 表示 32 位的寄存器。但一条同时执行两个 16 位加的 SIMD 指令可以实现同样的运算:

Reg16: PLUS(reg16, reg16)

其中,非终结符 reg16 分别表示一个 32 位寄存器的低 16 位子寄存器或高 16 位子寄存器。这样,执行 PLUS 操作的节点就可能有多个候选的优化覆盖。

3.4 SIMD 代码选择

SIMD 代码选择从程序的 IR 中识别并行操作,用这些并行操作生成 SIMD 指令并替换原来的并行操作,最终生成 SIMD 指令和非 SIMD 指令混合的目标代码。

SIMD 代码选择分为两个步骤。首先,我们识别程序 IR 中的并行操作,并把它们分为 SIMD 组,每个 SIMD 组对应目标机器的一条 SIMD 指令。然后,需将每个 SIMD 组中的每个操作对应到 SIMD 指令各子操作的相对位置。

3.4.1 SIMD 组的构造和冲突消除

由于采用扩展的 tree parsing 生成目标代码,树文法规则中的 SIMD 非终结符可以方便地表示可能形成 SIMD 指令的操作。

一条 SIMD 指令可以同时实现多个存储操作,显著提高

性能，因此首先识别存储操作的 SIMD 组。通过 SUIF 的 annotation 机制传递的编译器前端指针分析结果和 alignment 信息，识别相邻存储单元的存取操作。把这些操作根据操作的数据类型和 SIMD 数据宽度划分为 SIMD 组。同一个 SIMD 组的各操作必须存取相邻的存储单元，而且存取的总的存储块需满足 SIMD 指令的对齐要求。SIMD 组的操作按照有效地址排序，以对应到 SIMD 指令的各子操作。

下一步，以这些存储操作 SIMD 组为基础，通过数据流分析的 def-use 链和 use-def 链构造其它操作的 SIMD 组。以某个 SIMD 组的结果操作数为源操作数的各操作有可能构成新的 SIMD 组，或者结果操作数被某个 SIMD 组使用的各操作有可能构成新的 SIMD 组。各操作需满足以下条件，才能形成新的 SIMD 组：这些操作必须是同构的，即以相同的次序执行同样的操作；必须是独立的，即相互间不存在依赖关系；alignment 信息一致。

满足以上条件的操作可能存在多种构造 SIMD 组的方式，这时由编译器前端传递的依赖信息消除多余的构造方式。构造以 SIMD 组为节点的 DFG，根据各组操作之间的依赖关系，构造 SIMD 组的依赖图。如果依赖图存在 cycle，则移除 cycle 中收益值最小的组，使其中操作按照标量执行，直到没有 cycle 为止。

3.4.2 SIMD 操作排序和置换

SIMD 组中操作的排序对应它们在 SIMD 数据路径上的相对位置。在 SUIF 编译器的前端得到各变量的 alignment 信息，通过 SUIF 的 annotation 机制，这些 alignment 信息随着编译流程向后端传递。变量的 alignment 信息决定了对此变量的操作在 SIMD 数据路径上的相对位置。

如果某个 SIMD 组的操作数来自多个其它 SIMD 组的结果，或者源 SIMD 组的各子操作数和当前 SIMD 指令子操作数不一致，则需插入置换指令(permutation)以正确组合和排列源 SIMD 组中子操作数。

4 实验结果和分析

我们的实验采用 Intel 体系结构的 MMX 多媒体技术，Linux 操作系统。MMX 技术引入了 57 条新指令和 4 种新数据类型，每种数据类型都表示 64 位数据，但分别解释为 8 个 byte、4 个 word、2 个 doubleword、1 个 quadword。寄存器文件包括 8 个 64 位 SIMD 寄存器，但实际上是把它们映射到浮点部件。由于没有独立的 SIMD 部件，MMX 代码和浮点指令不能并发执行。

我们的 benchmark 程序基于 Berkeley Multimedia Workload (BMW)[9]。BMW 由一组 C 程序构成，包括了音频、视频、3D 图形、图像等广泛的多媒体应用。多媒体应用的一个典型特征是，小部分核心函数占用了程序大部分的运行时间。因此，本文优化 BMW 的核心函数，称为 kernels，可更好地提高目标代码性能。

实验结果如表 1 所示。采用三种方法生成目标代码：不采用 SIMD 指令，Leupers 的算法，本文提出的算法。机器主频为 750MHZ，表中给出的是机器周期数。

由表 1 可见，采用 SIMD 指令可以显著提高程序性能。和不采用 SIMD 指令相比，Leupers 算法可减少平均 32% 的 cycles，本文提出的算法可减少平均 47% 的 cycles。Leupers 算法在 OLIVE 机器描述中，用 reg_up 和 reg_low 非终结符分别表示 32 位 SIMD 寄存器的高 16 位和低 16 位子寄存器；由两个 DFG 节点组成的一个 SIMD pair (v_i, v_j)要构成一条 SIMD 指令，除了需满足其它的条件外，v_i 需规约到 reg_up，v_j 需规约到 reg_low。本文的算法在构成 SIMD 组时没有此类约束，而是可通过 permutaiton 类指令使得 SIMD 组中各操作满足 alignment 约束。这样可更好地利用生成 SIMD 指令的机会，因此性能优于 Leupers 算法。

表 1 各核心函数三种目标代码(不采用 SIMD 指令，Leupers 算法，本文算法)的机器周期数比较，机器主频为 750MHz

Kernel Name	Data Type	No SIMD	Leupers SIMD	Proposed SIMD
Add Block	8-bit unsighed	815	254	169
Block Match	8-bit unsighed	1350	578	385
Color Space Conversion	8-bit unsighed	91,577,891	29,226,986	19,484,657
Inverse DCT	16-bit signed	2,249	1,874	1249
Max Value	32-bit signed	1,805	1,353	902
Mix	16-bit signed	717	551	512
Short Term Analysis Filter	16-bit signed	11,451	9,542	7,415
Short Term Synthesis Filter	16-bit signed	16,386	12,604	11,300
Subsample Horizontal	8-bit unsighed	11,983,376	10,893,978	8,502,939
Subsample Vertical	8-bit unsighed	19,138,228	17,398,389	15,948,523

结论 基于通用处理器的 SIMD 扩展，为提高多媒体应用性能提供了良好的体系结构支持，但目前的编译技术还不能充分发挥这种潜能。编译优化依赖于程序分析。本文提出了一种从高级语言程序生成 SIMD 指令的算法，把编译器前端对程序的分析和编译器后端的具体机器信息相结合，减少了 SIMD 指令生成的复杂性，考虑了 permutation 类指令对 SIMD 指令生成的重要性；对一组多媒体 kernel 的实验结果表明可减少非 SIMD 目标代码 47% 的机器周期。

为保证编译优化保持程序语义的正确，指针分析常采用保守的估计。如何在程序分析的粒度和时间之间折衷，以更好地利用 SIMD 指令生成的机会，是将来的工作之一。

参考文献

1 DeVries D J. A Vectorizing SUIF Compiler: Implementation and Performance. Master's thesis, University of Toronto, June 1997

2 Aho A, Ganapathi M, Tijang S. Code Generation Using Tree Matching and Dynamic Programming. ACM Transactions on Programming Languages and Systems, 1989, 11(4): 491～516

3 Ren G, Wu P, Padua D. Optimizing Data Permutations for SIMD Devices. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, Canada, 2006

4 Nuzman D, Rosen I, Zaks A. Auto-Vectorization of Interleaved Data for SIMD. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, Canada, 2006

5 Larsen S, Amarasinghe S. Exploiting superword level parallelism. In: International Conference on Programming Language Design and Implementation, Canada, 2000

6 Larsen S, Witchel E, Amarasinghe S. Increasing and detecting memory address congruence. In: International Conference on Parallel Architectures and Compilation Techniques, USA, 2002

7 Leupers R. Code Optimization Techniques for Embedded Processors. Kluwer Academic Publishers, 2000

8 Kudriavtsev A, Kogge P. Generation of permutations for SIMD processors. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2005. 147～156

9 Slingerland N T, Smith A J. Design and Characterization of the Berkeley Multimedia Workload. Multimedia Systems, 2002, 8(4): 315～327