第 34 卷 第 6 期 计 算 机 工 程 2008 年 3 月
Vol.34 No.6 Computer Engineering March 2008
·软件技术与数据库· 文章编号：1000—3428(2008)06—0088—02 文献标识码：A 中图分类号：TP311

基于IA64 二进制翻译的解码技术研究

杨 欣，李 崇
(解放军信息工程大学信息工程学院，郑州 450002)

摘 要：作为 64 位处理器架构，IA64 提供了更高的指令级并行性，代表了一种新型微处理器的发展方向。该文介绍了基于 SLED 对 IA64 指令的描述和利用 MLTK 生成反向工具，讨论了 IA64 中指令解码器的设计与实现。这些技术对 IA64 二进制指令代码流的自动分析和变换，以及实现 IA64 二进制自动翻译具有重要的意义。

关键词：EPIC 操作系统；指令级并行；指令槽

Research of Decoding Technology Based on IA64 Binary Translation

YANG Xin, LI Chong
(College of Information Engineering, PLA University of Information Engineering, Zhengzhou 450002)

【Abstract】IA64, as a processor of 64-bit, supplies more ILP, and it stands for the developing orientation of new microprocessor. This paper introduces the representation of IA-64 instructions based on SLED and the reverse tool which produces by MLTK, discusses the design and implementation of IA64 instruction decoder. It is very significant for automatic analysis and transform of IA64 binary instruction streams in the realization of automatic translation of binary and reverse direction engineering based on the descriptions of computer and operating system.

【Key words】EPIC; Instruction Level Paralysation(ILP); slot

IA64 作为 64 位处理器架构，是Intel自推出 80386 微处理器IA32 架构以来处理器发展中最重要的进展[1-2]。但随着计算机微处理器体系结构的不断发展，大量在旧的体系结构上开发的可执行代码应用程序如果不能在新的体系结构上运行，就会被淘汰。二进制代码翻译将会有效地解决这一问题，是一种在新体系结构上运行旧体系结构代码的技术，将一种体系结构的二进制指令代码翻译成另一种体系结构的指令，这一过程通常涉及不同的机器体系结构(Mi)、不同的操作系统 (OSi) 和不同的二进制文件格式 (BFFi)，即由以(M1,OS1,BFF1) 表示的二进制可执行程序变换为由(M2,OS2,BFF2)所表示的二进制可执行程序。它对不同体系结构平台上二进制应用程序的兼容运行和移植也具有很大的实用价值[3-5]。

1 指令解码模块概述

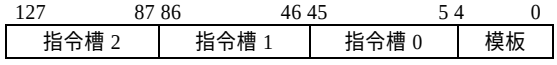
基于机器和操作系统的描述可以建立通用的、与平台无关的二进制翻译系统，用户可以选择源和目标机，实现新的二进制代码翻译器。指令解码模块处于二进制代码翻译系统的前端，是IA64 二进制代码翻译课题中的重要模块。它从输入的二进制文件中读入指令流，然后使用机器指令的语法描述将该指令流解析成源机器指令序列[6-8]。

2 IA64 指令解码模块的设计及实现

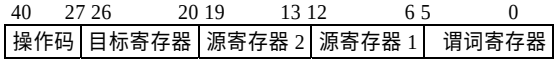
IA64 突破了传统 IA32 架构的许多限制，与其他的体系结构相比，基于 IA64 的处理器提供了更高的指令级并行性(Instruction level Paralysation, ILP)。

2.1 IA64 指令结构

Intel 的 IA64 指令系统具有 EPIC 特性，这是一种兼并了 RISC 和超长指令字各自优势的技术。指令以指令包(bundle)的形式存在，每个指令包为 128 位，包含 3 个指令槽(slot)和一个模板字段：



每个指令槽包含一条指令，每条指令 41 位，形式类似于 RISC 指令：



指令槽是一个具有独立功能的操作，模板字段负责把指令映射到执行部件。3 条无冲突的指令可以同时在不同执行单元内处理[2,4]。

指令集可以分为 6 种类型：A, I, M, F, B, L＋X[2-3]。

2.2 指令解码器的设计

解码模块从主入口点分解(文本/代码)指令流。该分解将指令的二进制表示与等价的汇编指令相匹配，由机器 SLED 文件指定，主要通过写 IA64 的指令系统说明语言 SLED 来实现。解码模块使用的算法是标准的到达算法，即从入口点开始，沿着程序的路径直到遇到控制转移，当一个控制转移隐含着多个目标地址，沿着一个地址执行，将其他的目标地址放到处理队列中，当一条路径结束，则转到处理队列的其他路径，直到处理队列中为空。

2.3 指令解码器的实现

指令解码可以抽象为一个匹配语句，语法类似于一个 C 的 switch 语句，语义相当于匹配指令中一系列的指位，返回指令变量域的值。MLTK 支持对机器指令解码和编码的抽象。指令解码器的设计和构造过程如下：

(1)建立指令集划分树，扩展指令位，为选择的体系结构

基金项目：河南省杰出人才创新基金资助项目(0521000200)
作者简介：杨 欣(1982－)，女，博士研究生，主研方向：计算机软件与理论；李 崇，讲师
收稿日期：2007-03-30 **E-mail：**yangxin_gg@sohu.com

写file.spec描述文件[4]。

(2)编写与之相应的基于 spec 的匹配文件，即 file.m 文件。

(3)使用 MLTK 工具，并对结果进行必要的修改。

1)对程序头的部分修改，使解码程序能够正确处理 41 位指令和 82 位长指令。

2)为提交正确立即操作数对程序的修改。

3)为区分部分相似指令和伪指令对程序的修改。

4)对程序尾的部分修改。

(4)对加长指令的处理。在 IA64 指令中，有一类 L+X 指令，它占据一个 bundle 的 2 个 slot，即一条指令是 82 位。L+X 类型指令的存在及其指令特点，使得在指令解码时需要加以特殊考虑。

1)判断是否存在 L+X 指令

在解码指令程序之前，先对 bundle 的值进行判断。

```
unsigned long long int _te = getDDword(MATCH_p);
int __t =(    _te & 0x1f ) ;
 /*获取前 5 位 template 的值*/
if    (( __t == 4 || __t == 5 ) && (gInstrNum%3 == 1))
{
gInstrNum++ ;
_GET_INSTRUCTION_(MATCH_w_64_0,MATCH_p,
gInstrNum%3 );
gInstrNum=gInstrNum+2 ;
    }
else { /*非超常指令正常处理*/
_GET_INSTRUCTION_(MATCH_w_64_0,MATCH_p,
InstrNum%3);      }
```

要注意指令与指令 slot 之间的对应关系，以 movl 为例，继 5 位 template 之后，slot-0 对应一个 M 类型的指令；slot-1 对应的是 L 部件，即 imm41，而不是指令 movl；slot-2 对应的是 L+X 指令类型的 X 部件即 movl。因此在解码时，如果满足条件((__t == 4 || __t == 5) && (gInstrNum%3 == 1))，则执行 gInstrNum++，对 slot-2 解码。这样避免了对 imm41 的二进制码进行解码，提交多余的指令。之后执行 gInstrNum= gInstrNum+2，使 slot 恢复正确的取值。

2)对程序中变量的影响

在解码指令时，需要用到很多变量，如 instr 表示读取的字符串；slot 表示读取的一个指令 bundle 中指令的序号，取值为 0, 1, 2；__template 表示一个 bundle 前 5 位模板的值。由于加长指令的存在，并且一条这样的指令占据两个指令槽，势必会对一些变量产生影响，如变量 slot，会影响到后续指令的正确解码。因此，需要对程序中受到影响的变量进行处理，使得程序能够达到正确解码的目的。

在头文件“ia64.pat.h”中，通过定义宏_GET_INSTRUCTION_(instr, uAddr, slot)，从指令流中得到每次待解码的一条指令的二进制表示，并对其进行处理，增加指令的 ty 信息，使每条指令由原来的 41 位转化为符合所需的 44 位。

```
#define _GET_INSTRUCTION_(instr, uAddr, slot)
ADDRESS64 MATCH_P_0;
int;      bitSize          /* bitSize 表示偏移量*/
int index;
unsigned int _temp = getDword(uAddr);
/*从指令流中读取 64 位*/
int = _temp & 0x1f;
index = slot + __template*4;
if ( slot == 0 )
  { MATCH_P_0 = uAddr; bitSize = 5; }
else if ( slot == 1 )
{ MATCH_P_0 = uAddr + 4;
 bitSize = 14; }
else {
MATCH_P_0 = uAddr + 8; bitSize =23; }/*根据 slot 取值不同，
确定偏移量，目的是从指令流中正确的读取 bundle 中的每条指令*/
instr = getDDword(MATCH_P_0)>>bitSize;
unsigned lowAddrInstr = (int)instr;
unsigned highAddrInstr = (int)(instr>>32);
unsigned temp = (highAddrInstr<<23)>>23;
int type = ty[index];
/*通过 index，从 ty 数组中获取指令的 type 值*/
int temp1 = type << 9;
unsigned endhigh = temp + temp1;
ADDRESS64 highInstr = ((ADDRESS64)endhigh)<<32;
instr = highInstr + (ADDRESS64)lowAddrInstr;
```

由于 slot 的取值以偏移量的形式决定了所读得的二进制代码，因此当指令流中存在超长指令时，准确的 slot 取值对正确解码至关重要。在解码程序中，用 gInstrNum 表示已解码指令的个数，它从 0 开始计数，每解码一条指令其值加 1，gInstrNum%3 对应 slot 的取值。因此，需要在解码完含有加长指令 bundle 中的 slot-0 后，对 gInstrNum 加 1，去解码 slot-2 的指令。这样避免了解码器对 slot-1 进行解码而产生原程序多余的指令。而后，执行 gInstrNum=gInstrNum+2，以保证解码对应的 slot 值正确。

3)对加长指令的正确解码

识别指令 movl 的二进制指令流：

```
case 86:
if ((MATCH_w_64_0 >> 20 & 0x1)
/* vc at 0 */ == 0)/*扩展位 vc 判断*/
{
unsigned long long i = (MATCH_w_64_0 >> 36 & 0x1) ;
unsigned ic = (MATCH_w_64_0 >> 21 & 0x1) /* ic at 0 */;
unsigned imm5c=(MATCH_w_64_0 >>22 & 0x1f)/* imm5c at 0 */;
unsigned imm7b=(MATCH_w_64_0 >>13 & 0x7f)/*imm7b at 0 */;
unsigned imm9d=(MATCH_w_64_0>>27 & 0x1ff)/*imm9d at 0 */;
unsigned r1 = (MATCH_w_64_0 >> 6 & 0x7f) /* r1 at 0 */;
_GET_INSTRUCTION_(MATCH_w_64_0,MATCH_p, 1);
/*由于加长指令的存在，略去 slot-1，识别 slot-2*/
unsigned long long    imm41a =
(MATCH_w_64_0 & 0x1fffff) ;
        /*取 imm41 中的低 21 位*/
unsigned long long    imm41b = (MATCH_w_64_0 >> 21 &
0xfffff) ;
 /*取 imm41 中的高 20 位*/
unsigned    long long imm41 = (imm41b <<21)|(imm41a) ;
 /*对 slot-1 中的 imm41 正确组合*/
unsigned long long imm64 =
(i<<63)|(imm41<<22)|(ic<<21)|(imm5c<<16)|(imm9d<<7)|imm7b ;
/*通过立即数之间的组合，生成加长指令 64 位立即数操作数*/
nextPC = 8 + MATCH_p;
RTs = instantiate(pc, "movl",dis_Reg(r64_names[r1]),
dis_Num(imm64));
/*提交 L+X 类型指令 movl*/
} /*opt-block*//*opt-block+*/
```

(5)运行并调试程序。

(下转第 92 页)