IA-64 二进制翻译中优化代码消除技术*)

苏 铭 赵荣彩 宋宗宇

(解放军信息工程大学计算机科学与技术系 郑州 450002)

摘 要 IA-64 架构为获得高性能支持许多先进体系结构的特性,例如显式指令级并行,指令判定执行,以及投机装入等,这些特性对编译器是可见的,但是为了充分利用这些体系结构的特性,编译器优化往往将程序的代码进行深度重构,使得从优化后的可执行代码中很难恢复源程序逻辑。本文提出了在 IA-64 二进制翻译中应用优化代码消除技术,提高翻译效率和生成目标机代码的质量。

关键词 IA-64,二进制翻译,判定执行,投机装入

Removing Optimized Executable Code Technique in IA-64 Binary Translation

SU Ming ZHAO Rong-Cai SONG Zong-Yu

(Department of Computer Science and Technology, Information Engineering College of PLA, Zhengzhou 450002)

Abstract IA-64 architecture supports a number of advanced architectural features designed to get around low level performance bottlenecks and improve performance. Such features include explicit instruction-level parallelism, instruction predication, and speculative loads from memory. These features are exposed to the compiler, however, that compiler optimizations to take advantage of such architectural features can profoundly restructure the program's code, making it potentially difficult to reconstruct the original program logic from an optimized executable. This paper describes several techniques to undo some of the effects of such optimizations and thereby improve the quality and efficiency of IA-64 binary translation.

Keywords IA-64, Binary translation, Predication , Speculative loads

1 引言

为了高效地利用 EPIC 架构特点,编译器采取了一些底层优化技术,特别是三种优化可以明显提高程序的性能:指令调度,谓词执行(if-转换)和投机。指令调度通过改变指令的序列,尽可能提高指令级并行;谓词执行选择性的减少条件分支指令,从而减少指令流水中的气泡;投机是指将内存操作提前执行来隐藏它们的延迟。虽然这些优化很好地挖掘底层处理器的先进特征,但是需要对程序的低级代码进行深度重构,对静态分析和修改可执行代码来说很难从优化后的代码中恢复源程序的逻辑,使工作变得异常复杂,例如逆向工程系统[1],静态二进制翻译和链接优化程序。

文章中提出了应用反汇编工具对二进制文件进行处理,在汇编代码级应用优化消除算法,消除指令调度,谓词执行和投机优化后的效果,使识别原始程序结构的问题简化。

2 消除优化代码的原因

2.1 IA-64 架构特点

在 IA-64 架构中,所有指令可以加上限定谓词(predicate)来实现判定执行的功能。它定义了 64 个 1 位的谓词寄存器,如果该寄存器值为真,则指令执行;否则,不执行。在 IA-64 的 64 个谓词寄存器中寄存器 p0 的值恒为 1。在程序中许多指令应用 p0 作为谓词寄存器,也就是无防备的(unguarded),并且约定在汇编表示中不出现。指令指定谓词寄存器(不是 p0)被称为有防备的(guarded)指令。

程序中应用指令组来表示指令的并行,每一个指令组是指不存在寄存器相关性的指令序列,可以并行发射,三条指令为一个指令束,尽可能并行执行。谓词结合指令调度可以减少明显的分支指令以及增加相当数量的指令并行性。例如,C 语句

```
if (condition)
  {instr1; instr2;}
else
    {instr3;}
```

经过编译器优化生成下面带谓词形式的机器代码:

```
cmp.eq p6,p7=r10,r11 ;;
(p6) instr1
(p6) instr2
(p7) instr3
```

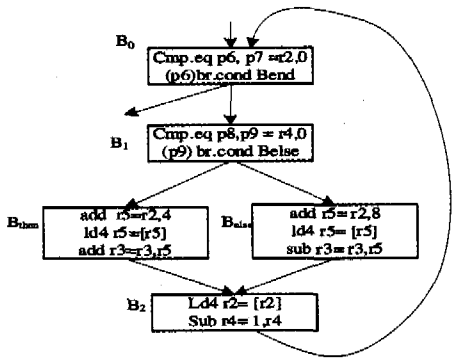
其中谓词寄存器 p6,p7 的值根据寄存器 r10 和 r11 比较结果设置为互补。经过优化后的机器代码避免了两个分支:一个跳转到 else 基本块,另一个跳转到 else 基本块以外。

在 IA-64 架构中投机机制[2]的基本特征是投机装入指令,操作码标记为"load.s"。投机装入的行为与正常装入最大的不同之处在于:如果指令产生一个异常,例如段或页错误,则不会立即处理,而是设置与装入目标寄存器相关的特殊位 NAT(Not A Thing),当程序运行到后面需要使用装入值时使用特殊的投机检验指令('chk.s')对目标寄存器进行检验。如果寄存器设置了 NAT 位,则跳转到编译器提供的恢复代码处执行,否则按正常顺序执行。NAT 位可以在寄存器之间传播,也就是说,如果指令的源寄存器设置了 NAT 位,则该指令中的目标寄存器也将设置相应的位,在投机装入后

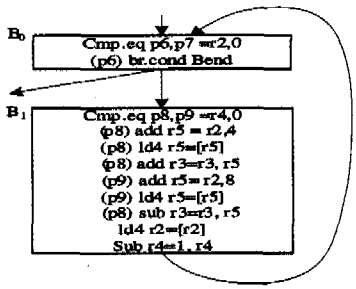
*)基金项目:国防重点科研项目资助。苏 铭 博士研究生,主要研究方向为:计算机软件与理论;赵荣彩 教授,博士生导师,主要研究方向为:计算机软件与理论;宋宗宇 讲师,硕士,主要研究方向为:计算机软件与理论。

面的一串相关的指令将在恢复代码中出现。

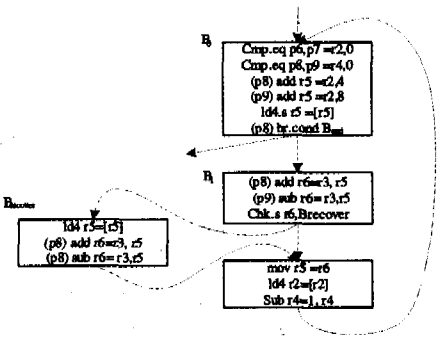
虽然编译器优化能充分应用这些架构特征，得到明显的性能提升，但是优化后代码非常模糊，对程序理解和逆向工程来说处理起来非常复杂。造成问题的原因是由于谓词、指令调度和投机改变了指令在源程序顺序和位置，导致代码在结构和操作上与源代码相差比较大。图 1 显示在不同的优化级别产生的 IA-64 代码，程序迭带计算一个值。



(a) 未优化的代码结构



(b) 经过谓词执行优化后的代码



(c) 经过投机优化后的代码

图 1

2.2 优化代码消除模块

本文的试验基础是二进制代码翻译系统 IATS，该系统是基于 UQBT[5] 的二进制翻译框架开发的，完成对 IA-64 二进制代码静态翻译，生成低级 C 代码。IATS 首先读入二进制文件，经过指令解码和语义映射生成第一级中间表示，这一级中间表示与汇编代码基本等价，为了消除代码机器相关性，需要进行代码提升和优化，生成与机器无关的带有控制结构的高级中间表示，为下一步提升到低级 C 代码做准备。优化代码消除模块的作用是去掉代码中与源机器相关的谓词执行、投机执行等特性，恢复优化前的逻辑结构，进而简化后面代码转换的工作。

IATS 的优化代码消除模块主要包含四部分工作：

1. 指令调度优化的消除，通过改变一个基本块中的指令序列，将在编译调度过程中被分离的相关指令组合在一起。

2. 谓词消除，通过数据流分析确定指令的谓词寄存器之间的关系，在控制流图中添加新基本块和边，替换带防备的指令完成指令的谓词消除，恢复原始程序控制流。

3. 投机装入指令消除，通过将投机指令下沉实现消除投机指令，并验证程序的语义一致。

4. 路径简化，在保证语义一致的前提下，对优化消除后生成的控制流图简化。

下面章节主要介绍优化消除模块中应用的谓词消除技术、投机代码消除技术和路径简化方法。

3 谓词消除算法

3.1 谓词分析

为了有效地消除谓词执行和指令调度的优化效果，需要知道指令谓词寄存器间的关系。谓词分析的目的是经过数据流分析推出谓词寄存器之间两类关系，这里 P 和 q 为两个谓词寄存器，⇒代表逻辑蕴涵。

互补性：如果 $p \Leftrightarrow \neg q$，则称 P 和 q 在程序点是互补的，也就是当控制流到达该点，两者中有一个为真。

支配：如果 $q \Rightarrow p$，则称 P 支配 q，也就是只有当 P 为真时，q 才为真。

例如下面指令设置谓词寄存器 p6 和 p7 为互补的值，依赖于 r5≤r6：

cmp. le $p6, p7 = r5, r6$

这条指令之后，$p6$ 和 $p7$ 为互补的。支配关系一般从无条件比较指令产生，例如谓词寄存器 $p6$ 和 $p7$ 在下面指令中都由 $p8$ 支配，寄存器间的支配关系与条件嵌套结构相对应 。

($p8$) cmp. unc. eq $p6, p7 = r10, r11$

谓词分析是一个向前的数据流分析，在函数的控制流图上传播谓词对(p,q)集合。考虑每个基本块 B 的两类谓词对集合，成对互补和支配/被支配的谓词。分析开始为每个基本块初始化谓词对集合，标记为 ϕ，然后再对该集合迭代验算计算不动点，直到该集合不发生任何变化。在基本块入口给定关系集合，在基本块中传播单条指令的效能，最后在基本块的结束位置确定谓词关系；在基本块的边界传播方式如下：在基本块 B 的起始获得的谓词关系信息为 B 的每一个前趋结束时谓词关系信息的交集。这样在 B 的入口获得的谓词关系与执行路径无关，具体算法的细节可参照文[3]。

3.2 算法

消除谓词的直接方法可以将每一种形式为

(p) instr

转换为下面的形式

If (! p) goto B_1

Instr

B_1 :……

这种直接转换减少了被防护的指令，但是产生了大量新基本块和控制边，结果是生成大而混乱的控制流图，大量冗余的路径模糊了程序逻辑从而增加了逆向分析的难度，下面重点讨论如何利用谓词分析的结果进行谓词消除。

定义“谓词组”为指令的谓词寄存器相关的指令序列，其形式化定义为一个基本块中连续谓词指令的最大序列，$(p_1)I_1,(p_2)I_2,\cdots,(p_n)I_n$，对于序列中控制指令执行的任何谓词寄存器对 p_i, p_j，下面等式之一成立：(i) $p_i = p_j$；(ii) p_i 支配 p_j，或反之亦然；(iii) p_i 与 p_j 为互补的。作为特殊情况，无谓

词的指令序列也形成谓词组，因为所有的谓词都是相同的(p^0)。

通过谓词分析在程序的一点推断出支配关系集合 D，($p_1, p_2, \cdots, p_k$)为最大谓词序列，于是存在：

$D \vdash p_1 \Rightarrow p_2 \Rightarrow \cdots \Rightarrow p_k$

$\vdash$表示逻辑蕴涵，换句话说从 D，可以得到 p_k 支配 p_{k-1}，$\cdots p_2$ 支配 p_1，称这种最大支配关系链为 p_1 的支配链。链表的最后一个元素 p_k 定义为 p_1 的锚。在前面提到过，程序中谓词寄存器之间的支配关系反映了原始程序控制流的嵌套条件。给定指令

$I \equiv (p)instr$

p 的支配链显式对应于影响指令 I 执行的谓词控制流嵌套。假定存在支配链($p_1, p_2, \cdots, p_k$)，支配关系的定义，意味着 $p_1, p_2, \cdots, p_k$ 的每一个谓词都为真时 I 才可以执行。描述可以表示如下形式：

$(p, p_1, \cdots, p_n)instr$

一旦支配链比较明确，则根据支配链锚之间的互补关系可以识别谓词组，然后再基于谓词组上实现谓词消除。算法如图 2。

4 投机代码消除算法

投机消除是指将包含投机装入的代码变换成语义上等价的“正常”程序，消除部分或所有的投机指令。将投机装入指令移动到代码的某一个或多个点，在该处可以被不带投机特性的装入操作所代替，称其为装入下沉。投机代码有两个特性使得装入下沉变得复杂，首先，可能根据投机装入的结果做多种操作，例如算术运算。如果投机失败，则计算结果的NAT位将通过这种操作传播。其次，投机装入和投机检验不一定一一对应：特殊的投机可能有几条相关的检验指令，一条投机检验指令可能与多条不同的投机装入指令对应。第一个特性意味着进行装入下沉时，不仅要移动投机指令，而且还要移动与投机指令相关的指令。第二个特性意味着当投机检验指令与多条投机装入指令相关时，必须确认下沉的指令集对每一条相关的投机装入指令是相同的。

算法开始将投机装入指令和投机检验指令组成投机域，投机域有以下特性：

1. 如果 x 是投机域 R 中的投机装入指令，y 是检验 x 的投机检验指令，则 y 在投机域 R 中。

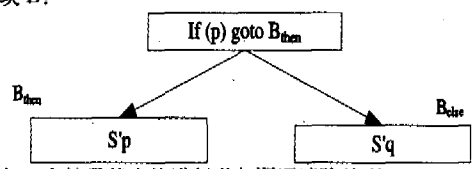
2. 如果 y 是 R 中的投机检验，x 为检验 R 的投机装入指令，则 x 在 R 中。

初始化时，投机域为投机装入指令和相关检验指令之间存在的检验者和被检验者关系闭包。

对每个投机域，需要确定无论对该域中任何一个投机装入和投机检验，必须下沉的指令集 I 是否相同。如果条件满足，则在投机域中每一个投机装入处删除指令序列 I，在每一个投机检验开始处插入指令序列 I 实现装入下沉。

输入：基本块 B，其中每一条指令的支配链明确
输出：B 经过谓词消除后对应的控制流图 GB。
算法：

GB 初始化只包含一个基本块 B。
while 当 GB 的控制流图中存在任何变化 do：
1. 查找 B 中最长的相禽指令序列 $s \equiv I_1, \cdots, I_n$，$S$ 中每条指令 I_i 支配锚中链不是 p 就是 q，并且谓词寄存器 p 和 q 为互补的。如果不存在不同的 p 和 q 的指令序列，则返回。
2. 将指令序列 S 分割为两个序列 S_p 和 S_q，S_p 包含 S 中指令的支配链的锚为 p，S_q 包含锚为 q。
3. 定义 S_p 和 S_q 为从 S_p 和 S_q，派生的指令序列，每一个 S_p 中形式如下指令：
   $(P_1, \cdots, P_K, P)^I$
   对应为 S'_p，指令格式为：
   $(P_1, \cdots, P_K)^I$
   S'_q 定义与前面类似。换句话说，S'_p 是通过删除 S_p 的每一条指令支配链的锚谓词得到，S_q 类似。
4. 应用下面的控制流结构替换 S 中的指令序列，相应的调整基本块 B：



5. 对每一个结果基本块进行递归谓词消除处理。

图 2 智能谓词消除算法

装入下沉将投机装入指令经过任何相关的条件分支移动到检验指令所在的基本块中。一般检验结果有两种可能：经过正常程序代码的“pass”路径和经过投机恢复代码的“fail”路径。为了保证消除投机不影响程序的语义，需要确定这两条路径产生等价的程序状态。在算法实现中验证使用通过Omega 算子约束解决技术[3]。

完成以上步骤后，投机消除算法最后完成以下工作：

1. 应用不带投机特性的装入指令替换投机域 R 内的投机装入指令。

2. 删除投机域 R 内的每一个投机检验指令。

其中删除投机检验指令需要把指向恢复代码的控制流边删除，这将使得相应的恢复代码变为不可达，而这些不可达的代码在后面的程序分析中会被检测和消除。

5 路径简化

谓词消除算法的基础是在基本块中发现谓词关系。从该算法得到的控制流图可以应用基本块边界之间的谓词关系进一步简化。路径简化主要考虑如下情况：当下面两个条件满足时，路径替换是安全的，也就是当存在边 $A \rightarrow B$ 和 $B \rightarrow C$，满足下面两个条件可以将 $A \rightarrow B$ 替换为 $A \rightarrow C$。

1. 执行基本块 B 不会改变任何寄存器或内存单元的值。

2. 只要控制流从 A 到 B，则必经过边 $B \rightarrow C$。

如果基本块 B 只包含一条指令，并且该指令为分支指令，则 B 满足第一个条件。

下面两条(弱)条件可以代替上面的第二个条件：

2(a)存在谓词 P；

(i)在 A 的出口 P 为真；

(ii)P 在 B 的每一个后继的入口都为假，除了 C。

如果 P 在 A 的出口总为真，则当控制流从 A 到 B 沿着边 $A \rightarrow B$，则 P 在 B 的入口处总为真。如果条件 1 满足，则 B 不改变 P 的值，P 在 B 的分支基本块开始处为真。在 B 的后继中，P 只有在 C 的入口为真，所以控制流到 C。所以条件 1 和 2(a)隐含条件 2。

在任何基本块入口处谓词值的真假信息应用直接数据流分析得到，在概念上类似于谓词分析中应用的常数传播方法。

6 实验结果

算法性能的测试通过对 SPECint2000 中 7 个测试程序 bzip2，gzip，mcf，parser，twolf，vortex，以及 vpr，经过 Intel's Icc 编译器编译(版本 8.0，优化级别为-O2)后生成的二进制代码翻译，分别统计应用优化代码消除技术后生成的控制流图中基本块个数、边数以及指令减少的情况，如图 3 所示，比较的基准是没有消除优化代码情况下产生的控制流图。从图

上可以看出应用优化代码消除技术可以明显提高二进制翻译生成的中间代码的质量：减少基本块个数 14.7%～19.8%（平均减少 16.3%），控制流边减少 10.3%～16.5%（平均减少 12.1%），指令条数减少 3.8%～10.3%（平均减少 5.8%）。

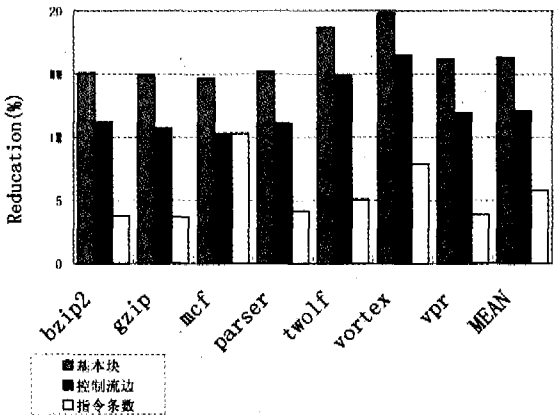


图 3　消除优化代码后控制流图简化情况

结论　编译器充分应用 IA-64 架构特征对程序进行优化，在提高程序性能的同时导致对低级代码结构的深度重构，对二进制代码翻译来说，很难恢复原程序的逻辑，增加工作的复杂性。本文提出的算法可以消除代码优化后的效果，使得识别原程序结构的工作简化，实验数据表明，该方法能够有效地缩减优化后的 IA-64 二进制代码在二进制翻译中产生的控制流图的大小和复杂性。

参 考 文 献

1　Byme E J. Software reverse engineering: a case study [M]. Software-practice and Experience, 1991, 21(12): 1349～1364
2　Intel IA-64 Architecture Software Developer's Manual [R]. Intel Corp, July 2000
3　Pugh W. The Omega test: a fast and practical integer programming algorithm for dependence analysis. [J] August, ACM, 1992, 35: 102～114
4　Debray S K, Muth R, Weippert M. Alias analysis of executable code [C]. In: 25th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL-98), January 1998. 12～24
5　Cifuentes C, van Emmerik M, Ramsey N. The Design of a Resourceable and Retargetable Binary Translator [C]. In: Proceedings of the Working Conference on Reverse Engineering, Atlanta, USA, IEEE CS Press, Oct. 1999. 280～291

（上接第 261 页）

缝合以后，系统所传输的数据量是相同的。而当客户端加倍时，由于要读取的数据总量不变，每个客户端所读数据变为原来的一半，这样经缝合以后，传输的数据量就变为原来的 2 倍，所以读数据所需时间也加倍。

5　同类工作比较

目前对不连续访问模式问题的解决方案包括：多重 I/O 请求方法和数据缝合 I/O 技术[4,14]。

大多数的并行文件系统接口只允许在一个 I/O 操作中访问一段连续的文件区域。使得一个不连续数据访问请求需要多个 I/O 操作来进行，结果带来大量的传输以及处理 I/O 请求的开销以及磁盘访问的开销。我们称这种对不连续数据访问的处理方法为多重 I/O。对于多重 I/O 来讲，最好的访问模式是仅对几个在内存和文件中都存在的数据区域进行访问的情况。

数据缝合 I/O 技术通过将文件中的一大块数据移动到内存缓冲区——数据缝合缓冲区中来处理不连续数据访问，而必要的数据移动操作都是在客户端的内存中进行的。数据缝合 I/O 方法减少了对物理上邻近的不连续数据的 I/O 请求的数目。主要缺点是要访问和通过网络传输无用的数据。另一个小不足是写操作的情况，当访问的文件数据区域不连续时，必须使用 read-modify-write 的方式。对数据缝合 I/O 模式最理想的是当访问许多不连续的数据区域且两个相邻区域之间间隔很小的情况。

我们的用户自定义文件视图结合合并 I/O 请求方法减少了一个不连续数据访问中 I/O 请求的数目，其方法是在一个单一的合并 I/O 请求中描述多个文件区域。在大多数情况下合并 I/O 请求方法要优于多重 I/O 请求方法和数据缝合方法。

结论　科学应用通常访问的是文件中许多小的不连续的数据区域。在并行文件系统中，如果使用传统的访问方法来访问这些不连续数据，无法获得理想的可扩展 I/O 性能。本文实现了一种不连续数据访问方法：用户自定义文件视图结合合并 I/O 请求方法。实验结果表明，该方法比现存的不连续 I/O 方法具有更好的 I/O 性能。

参 考 文 献

1　Schmuck F, Haskin R. GPFS: A Shared-Disk File System for Large Computing Clusters [C]. In: Proceedings of the Conference on File and Storage Technologies (FAST'02). Monterey, CA, January 2002. 231～244
2　Ligon III W B, Ross R B. PVFS: Parallel Virtual File System [J]. In: Sterling T. ed. Beowulf Cluster Computing with Linux. MIT Press, November 2001. 391～430
3　Nieuwejaar N, Kotz D. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments [C]. In: Proceedings of the Scalable High-Performance Computing Conference, 1994. 71～78
4　Thakur R, Choudhary A, Bordawekar R, et al. Passion: Optimized I/O for Parallel Applications [J]. IEEE Computer, 1996, 29(6): 70～78
5　Oldfield R, Kotz D. Armada: a parallel I/O framework for computational grids [J]. Future Generation Computer Systems, 2002, 18: 501～523
6　French J C, Pratt T W, Das M. Performance Measurement of the Concurrent File system of the Intel IPSC/2 Hypercube [J]. Journal of Parallel and Distributed Computing, Jan/Feb 1993
7　Nieuwejaar N, Kotz D. Low-level Interfaces for High-level Parallel I/O [C]. In: Workshop for I/O in Parallel and Distributed Systems, IPPS 1995. 47～62
8　Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface [EB/OL], July 1997. http://www.mpi-forum.org/docs/docs.html
9　Corbett P F, Feitelson D G. The Vesta parallel file system [J]. ACM Transactions on Computer Systems, 1996, 14(3): 225～264
10　The parallel virtual file system. http://www.parl.clemson.edu/pvfs/[EB/OL]
11　Baylor S, Wu C. Parallel I/O Workload Characteristics Using Vesta [C]. In: Jain R, Werth J, Browne J, eds. Input/Output in Parallel and Distributed Computer Systems. Kluwer Academic Publishers, 1996. 167～185
12　Crandall P, Aydt R, Chien A, et al. Input-Output Characteristics of Scalable Parallel Applications [C]. Proceedings of Supercomputing '95. ACM Press, 1995
13　Nieuwejaar N, Kotz D, Purakayastha A, et al. File-Access Characteristics of Parallel Scientific Workloads [J]. IEEE Transactions on Parallel and Distributed Systems, October 1996, 7(10): 1075～1089
14　Thakur R, Gropp W, Lusk W. Data Sieving and Col-lective I/O in ROMIO [C]. In: Proc. of the Seventh Symposium on the Frontiers of Massively Parallel Computation, 1999. 182～189
15　Thakur R, Gropp W, Lusk W. On Implementing MPI-IO Portably and with High Performance [C]. In: Proc. of the Sixth Workshop on Input/Output I Parallel Distributed Systems, 1999. 23～32
16　Schwan P. Lustre: Building a file system for 1,000-node clusters [C]. Proceedings of the Linux Symposium, Ottawa, Ontario, Canada. July 2003