一种具有快速条件断点的并行程序调试器*

刘 建+, 王 皓, 沈美明, 郑纬民

(清华大学 计算机科学与技术系 高性能计算技术研究所,北京 100084)

A Parallel Debugger with Fast Conditional Breakpoint

LIU Jian+, WANG Hao, SHEN Mei-Ming, ZHENG Wei-Min

(High Performance Computing Institute, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

+ Corresponding author: Phn: 86-10-62783505, Fax: 86-10-62771138, E-mail: liujian98@mails.tsinghua.edu.cn

http://www.hpc.cs.tsinghua.edu.cn

Received 2002-09-25; Accepted 2003-03-04

Liu J, Wang H, Shen MM, Zheng WM. A parallel debugger with fast conditional breakpoint. *Journal of Software*, 2003,14(11):1827~1833.

http://www.jos.org.cn/1000-9825/14/1827.htm

Abstract: A fast conditional breakpoint is considered to be an essential process control capability of advanced high-performance parallel debuggers. In this paper, the fundamental principles for finding and implementing fast conditional breakpoints are introduced and the design problems associated with their implementations such as code generation, code instrumentation and source program mapping are discussed. Dynamic code instrumentation, combined with hybrid code generation and source breakpoint identification, provides a well-suited solution to those problems. Using an enhanced version of the Dyninst run-time code patching library, a fast conditional breakpoint in a parallel debugger, XBUSTER, is implemented. Compared with GDB experimental results show that XBUSTER can debug a program with a higher efficiency. Compared with other fast conditional breakpoint implementations based on dynamic instrumentation technique, XBUSTER is more portable and functional.

Key words: debugger; conditional breakpoint; dynamic instrumentation; run-time code patching

摘 要: 快速条件断点是高性能并行程序调试器中进程控制必备的功能之一.分析了快速条件断点基本原理以及快速条件断点设计中需要考虑的代码生成、代码插装和源程序对应等问题.并针对上述问题,提出了预编译与运行时编译结合的代码生成方式、动态代码插装、源断点标识技术等解决方案.利用经过改进的动态补丁码工具 Dyninst,设计实现了并行程序调试器 XBUSTER,并实现了快速条件断点这一重要功能.测试表明,XBUSTER 的执行效率比传统条件断点高.与现有的基于动态插装的系统,如 Ceder,ldb 和 Dyner 相比,XBUSTER 具有功能性强、可移植性好等鲜明特点.

关键词: 调试器;条件断点;动态插装;动态补丁码

中图法分类号: TP311 **文献标识码**: A

* Supported by the National Natural Science Foundation of China under Grant No.69933020 (国家自然科学基金)

第一作者简介: 刘建(1962－),男,湖北宜昌人,博士,高级工程师,主要研究领域为并行程序调试环境,IT 项目管理.

© 中国科学院软件研究所 http://www.jos.org.cn

条件断点是一种常用的软件调试手段.使用条件断点时,调试者选择被调试程序控制流中的某一点作为断点位置,提供一个布尔表达式作为断点条件,指定一系列操作作为断点动作.当被调试程序执行到断点位置并且满足断点条件时,断点动作就会被执行,称为触发断点.

为了明确本文讨论的问题,有必要指出控制断点与数据断点的区别.控制断点有固定的断点位置,只有在被调试程序执行到断点位置并且满足断点条件的情况下才会被触发.数据断点没有固定的断点位置,在被调试程序执行过程中,一旦断点条件满足,就会被触发.本文所讨论的条件断点是一种控制断点.

条件断点常用于检查程序在特定条件下的执行情况,特别适用于在多次循环中寻找满足特定条件的循环.灵活使用条件断点可以提高调试效率,缩短调试时间.在这种交互式调试中,快速条件断点可以加快被调试程序的执行速度,缩短调试时间.

条件断点还可用于程序性能测量,例如函数执行时间统计或者语句执行计数.很多性能指标的测量都要用到条件断点.这时,断点动作通常是修改计数器或计时器.在这种非交互式调试中,我们希望条件断点的执行速度越快越好,因为程序性能测量中的一个关键问题就是减少测量代码对程序的干扰.普通条件断点的执行速度太慢,不能满足性能测量的要求.因此,快速条件断点在非交互式调试中同样具有很重要的意义.

并行程序具有不确定性,运行环境的干扰是造成并行程序不确定的主要原因,快速条件断点可以降低调试器对并行程序的干扰.

基于上述原因,快速条件断点被认为是高性能并行调试器必须具备的特性之一[1].

清华大学计算机科学与技术系高性能计算技术研究所建造了一个基于 LINUX 系统的高性能集群计算机系统.针对并行调试的实际需要,我们实现了一个能够调试 PVM 或 MPI 程序、用户界面友好并具有较好移植性的并行程序调试器 XBUSTER[2].在 XBUSTER 中实现快速条件断点,既可以提高正确性调试的效率,又可以为性能测量打下基础.

1 快速条件断点

传统条件断点一般利用无条件断点实现:在断点位置设置一条无条件断点指令,当被调试程序执行到该指令时,产生操作系统陷入,控制转移到调试器,调试器检查断点条件,如果条件不满足,调试器就继续运行被调试程序;如果条件满足,调试器就执行断点动作.

从断点执行过程可以清楚地看到这一点:控制从被调试程序转移至调试器,需要一次进程切换.如果断点条件不满足,恢复被调试程序的执行又需要一次进程切换.调试器检查断点条件需要访问被调试程序的进程空间,这要通过操作系统提供的调试接口来完成,而这些接口通常是系统调用.进程切换和系统调用的开销通常都很大,远远超过了检查断点条件和执行断点动作的开销,严重影响了断点的执行速度.导致进程切换和系统调用的原因是断点条件代码在调试器中执行.

简单地说,快速条件断点的原理就是直接在被调试程序中执行断点代码.在快速条件断点的执行过程中,由被调试程序中的代码检查断点条件,而不需要切换到调试器.此外,检查代码可以直接存取被调试程序的数据,不需要通过操作系统调试接口.仅当断点被触发而且断点动作要求唤醒调试器时,才会产生一次进程切换;如果断点动作不需要唤醒调试器,例如用于非交互式调试,则可以完全避免进程切换.可见,采用快速条件断点的被调试程序可以在没有调试器干预的情况下运行,直到断点被触发.因此,快速条件断点的执行速度远远快于传统条件断点.

快速条件断点的设计通常包括 3 个方面:代码生成(code generation)、插装(code instrumentation)和源程序对应(source program mapping).

本节将从以上 3 个方面,讨论快速条件断点设计的一般问题和实现技术,介绍并行调试器 XBUSTER 中快速条件断点的设计.

1.1 代码生成

代码生成要解决两个问题,即断点描述和断点代码生成.

© 中国科学院软件研究所 http://www.jos.org.cn

断点描述包括断点条件描述和断点动作描述.断点条件是一个布尔表达式,可以采用类似某种高级语言的布尔表达式来描述.断点条件描述应该允许使用复杂的布尔表达式.大多数断点动作可以用程序语句来描述.此外,还应该提供对一些特殊断点动作的描述,例如:挂起被调试程序的执行,核心内存转储等等.

断点代码生成是对断点描述进行编译的过程.编译得到的断点代码,既可以是目标机器的机器指令序列,也可以是汇编语言或高级语言程序片断.断点代码生成通常有两种方式:运行时编译和预编译.在运行时编译方式下,调试器带有一个运行时编译器,接收断点描述,编译生成断点代码;在预编译方式下,由编译器预先编译好一些条件断点代码模板,使用时调试器根据断点描述选取相应的模板生成断点代码.

运行时编译使用起来灵活方便,并且可以针对断点的具体位置对断点代码进行全局优化.但在实际应用中,受实现复杂性的限制,调试器中的运行时编译器通常没有很强的代码优化功能.预编译方式对断点代码可以做很好的局部优化,但却缺乏灵活性.

XBUSTER 采用两者相结合的方式生成断点代码:对用户自定义的断点条件和断点动作,采用运行时编译方式,由一个小型运行时编译器生成断点代码,为用户提供灵活性;对一些常用或具有固定模板的断点条件和断点动作,采用预编译的方式,提高断点代码效率.

1.2 代码插装

设计代码插装方案时需要考虑两个关键问题:待插装代码的形式和代码插装的时机.

待插装代码的形式由代码生成决定,可以是机器指令序列(称为二进制代码插装),也可以是汇编语言或高级语言片断(称为源代码插装).二进制代码插装不需要对源程序重新编译连接,可以采用动态插装技术.但是,二进制代码插装要受到很多限制,并需要对目标程序原有指令序列进行修改,保证程序正确执行,因此实现起来比较复杂[3,4].源代码插装实现简单,但要求重新编译连接目标程序,因而限制了它的使用范围,不能采用动态插装技术.

代码插装的时机既可以在程序开始运行之前(称为静态插装),也可以在程序开始运行之后而不需要重新启动程序(称为动态插装,又称为动态补丁码(run-time code patching)).动态插装更加灵活方便,尤其是完成代码插装后,可以继续调试,不需要重新启动程序.但是由于目前还没有办法编译连接一个运行中的程序,动态插装的代码只能是二进制代码.

用户往往需要在调试过程中设置条件断点,并且希望调试过程能够继续进行,而不是重新开始.因此,动态插装更符合调试器对条件断点代码插装的要求.XBUSTER 采用动态二进制代码插装方式,利用动态补丁码工具 Dyninst 实现.

1.3 源程序对应

源程序对应的作用是实现源断点,即断点被触发后,调试器能够准确地向用户报告断点在源程序中的位置,同时正确地处理用户对断点处程序状态的操作请求.精确地实现源断点,是源级调试器设计的主要目标之一,特别是在支持优化代码的调试器中[5].

源断点的一般实现方法是代码地址映射(code location mapping)[5],即调试器将一个源断点映射至目标程序中一条指令的地址,这条指令称为目标断点(object breakpoint).目标断点一般是设置断点的源程序语句所对应的第一条机器指令.从目标程序调试信息中可以得到目标断点的地址.调试器在目标断点处停止被调试程序的执行,向用户报告源断点,并用目标断点处的程序状态处理用户的操作请求.

代码插装方式决定了源断点的实现难度.在采用源代码插装时,目标程序调试信息包括对断点代码的描述,可以从中获得目标断点地址.调试器使用代码地址映射就可以很容易地实现源断点.在采用二进制代码插装时,目标程序调试信息不包括对断点代码的描述,不能从中获得目标断点地址,也就无法使用代码地址映射.一个解决办法是,在代码插装过程中获取目标断点的地址,进而使用代码地址映射实现源断点.

XBUSTER 使用动态补丁码工具 Dyninst 完成动态二进制代码插装.Dyninst 不能提供插装代码地址,这使我们无法确定目标断点地址,也就无法采用代码地址映射.为了解决这个问题,我们设计了一种“断点标识”技术,在 XBUSTER 中实现了源断点.

© 中国科学院软件研究所 http://www.jos.org.cn

2 Dyninst

动态补丁码工具 Dyninst[6]是一套应用程序编程接口(API),可以用来将一段代码插装到一个正在运行的程序中去.被插装的程序可以继续运行,不需要重新编译连接,也不需要重新启动.当程序执行到被修改处时,不仅会执行原有代码,还会执行被插装的代码.Dyninst 是前面提到的动态插装技术的一个具体实现[7,8].

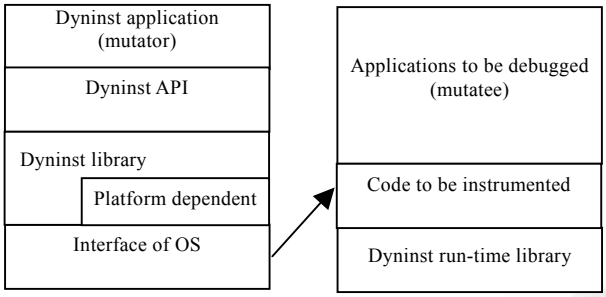

Fig.1 Program structure of Dyninst

图 1 Dyninst 程序结构

Dyninst 是平台无关的,它建立在程序及其状态的抽象上,例如进程(线程)、映像、表达式等.API 的实现屏蔽了与目标机器和操作系统相关的细节.因此,使用 Dyninst API 的程序具有很好的可移植性.

动态插装和平台无关使 Dyninst 成为我们实现快速条件断点的一个有力工具.使用 Dyninst API 的程序结构如图 1 所示.

Dyninst 应用程序(mutator)调用 Dyninst API,控制和修改受控应用程序(mutatee).进程控制和代码插装由 Dyninst 库(dyninst library)通过操作系统提供的调试接口(例如 UNIX 下的 ptrace 系统调用或/proc 文件系统)进行.Dyninst 运行库(run-time library)是在创建受控应用程序 Mutatee 时动态加载的,包含了插装代码运行所需的支持代码.

3 XBUSTER 系统

3.1 系统结构

并行调试器 XBUSTER 采用主/从控制器模式,系统结构如图 2 所示.

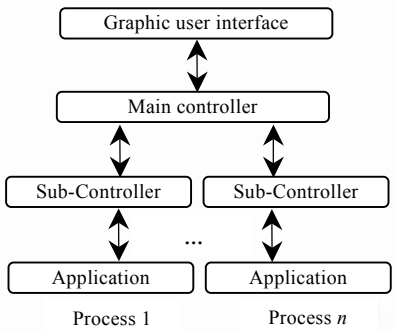

Fig.2 Architecture of XBUSTER

图 2 XBUSTER 系统结构

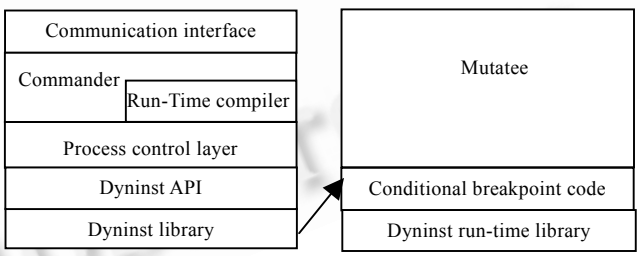

Fig.3 Structure of sub-controller XBUSTER

图 3 XBUSTER 从控制器结构

图形用户界面(graphic user interface)是一个基于 MOTIF 的多窗口界面.主控制器(main controller)负责全局控制,它根据用户选择的操作,向相应的一个或一组从控制器(sub-controller)发出命令.若从控制器有结果返回,它将负责结果的接收并通过用户界面反馈给用户.从控制器负责控制单个进程,它接收主控制器发来的命令,执行相应的操作,并将结果返回给主控制器.主从控制器通过 socket 通信.

在 XBUSTER 中使用 Dyninst 实现快速条件断点,首先要考虑 XBUSTER 与 Dyninst 的结合方式.Dyninst 有一套完整的进程控制接口,在原有的 XBUSTER 中,进程控制功能是由从控制器通过操作系统调试接口实现的.现在我们必须用 Dyninst 提供的 API 来完成进程控制功能.

Dyninst 是为性能调试设计的,其进程控制 API 与正确性调试的要求有较大的差距.为了弥补这种差距,我们在从控制器中设计了一个进程控制层 PCL(process control layer).PCL 使用 Dyninst API 实现调试器所需的各种进程控制功能.这样的设计方案可以使原有 XBUSTER 系统与 Dyninst 之间的接口最小化,同时简化了命令处理

© 中国科学院软件研究所 http://www.jos.org.cn

器的控制逻辑.

综合 Dyninst 程序结构(如图 1 所示)和 XBUSTER 系统结构(如图 2 所示),可以得到从控制器结构,如图 3 所示.图 3 中,网络通信接口(communication interface)负责从控制器和主控制器之间的通信;命令处理器(commander)负责处理来自主控制器的调试命令,并向主控制器返回操作结果;命令处理器调用运行时编译器(run-time compiler)生成条件断点代码;进程控制层(process control layer)使用 Dyninst API 实现正确性调试所需的进程控制功能.

3.2 Dyninst的修改与扩展

Dyninst 是为性能调试设计的,对正确性调试的支持并不完善,需要进行一些修改与扩展,以满足正确性调试的需要.

单步执行是每次执行被调试程序的一条机器指令的一种执行方式,是实现正确性调试器各种执行方式的基础.由于性能调试很少使用单步执行,所以 Dyninst 没有提供这一执行方式.为了使用 Dyninst 实现 XBUSTER 的各种执行方式,我们扩展了 Dyninst,设计了一个新的 API(singleStepExecution),用于实现单步执行方式.

源级调试器通常都提供以语句为单位的跟踪执行方式.为此,我们需要知道一条指令是否为所属源程序行的第 1 条指令.Dyninst 提供了一个 API(getLineAndFile),可以根据指令地址查找所属的源文件和行号.我们增强了该 API 的功能,增加了一个布尔标志 exactMatch;当 exactMatch 为 false 时,API 的功能不变;当 exactMatch 为 true 时,只有当所给地址是源文件某一行的起始地址的时候,才返回源文件名和行号.

正在调试中的程序可能因为各种异常原因而退出,调试器应该能够向用户报告导致程序异常退出的原因.Dyninst 原有的实现无法获取导致进程终止的信号.我们修改了相应的 API(stopSignal),实现了报告进程终止信号的功能.

我们在修改与扩展后的 Dyninst 上运行了附带的测试程序集,结果全部获得通过,表明我们的修改与扩展并未影响 Dyninst 原有的功能.

进程控制层使用修改与扩展后的 Dyninst API,可以容易地实现正确性调试器所需的各种控制功能.

3.3 快速条件断点的实现

XBUSTER 的快速条件断点代码生成采用运行时编译与预编译结合的方式.

对于用户自定义的断点条件和断点动作,XBUSTER 采用运行时编译方式生成断点代码.Dyninst 使用抽象语法树(abstract syntax tree,简称 AST)描述代码生成过程[6],为编译器的构造提供了方便.从控制器带有一个小型的运行时编译器,该编译器使用 YACC 编写,接受 C 语法描述的断点条件和断点动作,采用 LR 分析方法对断点描述进行编译,得到断点代码.

对于一些常用或具有固定模板的断点条件和断点动作,XBUSTER 采用预编译的方式.这些断点代码被作为子程序编译在一个动态连接库里,并在创建被调试进程时被加载.在快速条件断点中只需要简单地调用这些子程序,就可以使用这些功能.采用预编译的一个典型断点动作就是挂起被调试程序并唤醒调试器,这一例程直接由 Dyninst 通过其运行库加载到被调试程序中.此外,一些常用性能指标的测量代码也将采用预编译的方式.

得到断点代码以后,XBUSTER 根据用户指定的断点位置,创建插装点,然后插装断点代码.

XBUSTER 使用“断点标识”技术实现源断点,该技术要点如下:

(1) 利用 Dyninst 在被调试程序中创建一个整型变量 bpNumber 作为当前被触发断点标识,初始值为 0,代表当前没有条件断点被触发;

(2) 创建断点时,为每个断点生成一个惟一的非零整数作为断点标识;

(3) 设一个快速条件断点的断点标识为 id,断点条件为 cond,断点动作为 action,则为该断点生成的代码相当于如下的条件语句:if (cond){bpNumber=id; action};

(4) 调试器维护一张断点标识与断点相关信息的对应表 BPTable,特别地,表中对每个断点都有以下两项内容:① 断点在源程序中的位置,例如源文件名和行号;② 目标断点地址,这个目标断点地址是从插装前的目标程序调试信息中得到的,一般来说并不等于断点触发后指令指针寄存器的当前值.

© 中国科学院软件研究所 http://www.jos.org.cn

(5) 断点触发后,调试器读取 bpNumber 变量的值,得到当前被触发断点的断点标识;

(6) 调试器查询 BPTable 得到断点的相关信息,向用户报告源断点,并使用保存的目标断点地址处理用户对程序状态的操作请求.

完成对断点的处理以后,在恢复程序运行之前,调试器重置 bpNumber 为 0.

4 与相关工作的比较

Kessler[3]在 SUN SPARC 工作站上实现了 Ceder 调试器.Ceder 可以动态修改程序,插装断点代码,但它需要用户提供预先编译成机器指令的断点代码,并且缺乏可移植性.严格地讲,Kessler 只是在 Ceder 中实现了二进制代码插装技术.

Los Alamos Debugger(ldb)是 Brown[9]为 Los Alamos 实验室的 Cray 计算机开发的一个调试器.Brown 在 ldb 中构造了一个小型编译器,提供运行时代码生成,同时采用动态插装技术.但是 ldb 的设计仍然与机器平台相关,缺乏可移植性.

Dyner[10]是 Dyninst 附带的一个示例性质的调试器,它也使用 Dyninst 实现快速条件断点,具有运行时断点代码编译功能,并且在一定程度上实现了源断点.但是 Dyner 受 Dyninst 原有实现的限制,不支持单步跟踪调试等正确性调试器常用的功能,而且没有图形界面,在操作界面上不如 XBUSTER 方便.

在国内,目前尚未见到相关工作的报道.

我们测量了 XBUSTER,GDB 和 Dyner 条件断点的性能.

需要说明的是,XBUSTRE 是专为调试并行程序而设计的,但也可以调试串行程序.快速条件断点在并行程序调试中的作用主要体现在以下几方面:① 提高条件断点的执行效率;② 减少条件断点对被调试程序的干扰;③ 为性能调试提供更友好的 API.本文主要测试执行效率,由于 GDB 和 Dyner 都是串行的调试器,因此我们没有选择并行程序而是选择串行程序来测试.而且就执行效率而言,串行程序和并行程序的测试结果是一样的.对另外两点有兴趣的同行可参考文献[12].

我们从 SPEC CPU2000 测试程序集中选择了 UNIX 压缩程序 gzip 进行测试.测试在清华大学机群系统的一个结点上进行,该结点的配置为:4 个 Xeon 700MHz PIII,共有 1G 内存,每个 CPU 有 1M 高速缓存,运行 Redhat Linux 7.2;GDB 版本为 5.0rh-15;Dyner 版本为 3.0.

我们在 send_bits 函数的入口设置断点.该函数带有两个参数,其中第 1 个参数 value 是待输出代码的值,我们的断点条件就是 value 取某个特定值.如果 send_bits 函数在 value 取某个特定值的情况下出错,用这样的条件断点可以发现程序的错误.计时从调用 spec_compress 函数开始.

我们首先测量了条件断点的响应时间(response time),也就是从程序开始运行到条件断点触发所需的墙钟时间(wall clock time).表 1 列出了对 20 次测试结果取算术平均以后的数据.其中第 1 列是测试中条件断点的执行次数(# of operations),第 2~4 列分别是 GDB,Dyner,XBUSTER 的响应时间(time).为了便于比较,表中第 6,7 列分别列出了 GDB 与 XBUSTER,Dyner 与 XBUSTER 响应时间的比值.

Table 1 Response time of conditional breakpoint

表 1 条件断点响应时间

# of operations	Time (s)			GBD / XBUSTER	Dyner / XBUSTER
	GDB	Dyner	XBUSTER		
191	0.045 799	0.009 339	0.009 356	4.895 1	0.998 2
1876	0.341 232	0.009 648	0.009 648	35.368	1.000 0
36 538	6.652 141	0.027 244	0.027 119	245.29	1.004 6

从表 1 的数据可以看出,条件断点执行次数越多,XBUSTER 比 GDB 快得就越多.即使在条件断点执行次数很少(191 次)的情况下,XBUSTER 仍比 GDB 快近 5 倍.XBUSTER 的执行时间和 Dyner 相当,这是因为我们的 XBUSTER 和 Dyner 都使用了 Dyninst.

我们还分别测量了不设置条件断点和条件断点不触发情况下的程序运行时间(算术平均值),根据得到的结果计算出了条件断点执行一次所需要的时间,见表 2.

© 中国科学院软件研究所 http://www.jos.org.cn

Table 2 Average time of conditional breakpoint

表 2 条件断点平均执行时间

	Program time (s)		Time of breakpoint $T3=T2-T1$ (s)	#of breakpoint N	Average time of breakpoint $T=T3/N$ (ms)
	No conditional breakpoint (s)	Including conditional breakpoint (s)			
GDB	0.857 305	340.453 246	339.595 941	1 828 573	186
XBUSTER	0.857 305	1.022 749	0.165 444	1 828 573	0.1

从计算结果可以看出,传统条件断点(GDB)平均每次执行(average time of breakpoint)时间为 186ms,而快速条件断点(包括 XBUSTER 与 Dyner)只有 0.1ms,多出的部分就是传统条件断点的进程切换与系统调用引入的额外开销.比较和测试结果表明,XBUSTER 的快速条件断点与传统条件断点相比,具有很高的效率,当条件断点执行次数为 36 538 时,XBUSTER 比 GDB 快 245 倍;与 Dyner 等基于动态插装的系统相比效率相当,但是 XBUSTER 功能更强、可移植性更好.

5 结 语

快速条件断点是高性能调试器必备的特性之一.本文介绍了快速条件断点的基本原理,讨论了快速条件断点设计时需要考虑的代码生成、代码插装和源断点等问题.我们使用经过改进的动态补丁码工具 Dyninst,实现了具有快速条件断点的并行调试器 XBUSTER.性能测试结果表明,XBUSTER 的快速条件断点与传统条件断点相比,具有很高的效率.与现有基于动态插装的系统相比,XBUSTER 具有功能性强、可移植性好等鲜明特点.进一步的工作包括对断点代码的优化、在性能测量中使用快速条件断点等等.

References:

[1] Brown J, Zosel M, Zwakenberg R, Seager M, Williams A. ASCI debugging requirements. Los Alamos National Laboratory, 1998. http://www.lanl.gov/ projects/asci/PSE/ASCIdebug.html.

[2] Liu J, Yu HL, Shen MM, Zheng WM. Implementation techniques of a distributed debugger in LINUX cluster systems. Computer Engineering, 2002,28(4):7~9 (in Chinese with English abstract).

[3] Kessler PB. Fast breakpoints: Design and implementation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. White Plains: ACM Press, 1990. 78~84.

[4] Paradyn Developer's Guide. Paradyn Project. 2001. ftp://grilled.cs.wisc.edu/paradyn_manuals/developerGuide.pdf.

[5] Wu LC, Mirani R, Patil H, Olsen B, Hwu WMW. A new framework for debugging globally optimized code. In: Proceedings of the ACM SIGPLAN'99 Conference on Programming Languages Design and Implementation. ACM Press, 1999. 181~191.

[6] Buck B, Hollingsworth JK. An API for runtime code patching. International Journal of High Performance Computing Applications, 2000,14(4):317~329.

[7] Hollingsworth JK, Miller BP, Cargille J. Dynamic program instrumentation for scalable performance tools. In: Scalable High-Performance Computing Conference Knoxville. 1994. 841~850.

[8] Hollingsworth JK, Miller BP, Goncalves MJR, Naim O, Xu ZC, Zheng L. MDL: A language and compiler for dynamic program instrumentation. In: International Conference on Parallel Architectures and Compilation Techniques (PACT). New York: ACM Press, 1997. 201~212.

[9] Brown JS. The application of code instrumentation technology in the Los Alamos debugger. Los Alamos National Laboratory, 1992.

[10] Hollingsworth JK, Altinel M. Dyner User's Guide. Dyninst Project, Release 3.0. Paradyn Project. 2002. http://www.dyninst.org/docs/dynerGuide.v30.pdf.

[11] DyninstAPI Programmer's Guide. Release 3.0. Dyninst Project. 2002. http://www.dyninst.org/docs/dyninstProgGuide.v30.pdf.

[12] Liu J, Shen MM, Zheng WM. Research on perturbation imposed on parallel programs by debuggers. Chinese Journal of Computers, 2002,25(2):122~129 (in Chinese with English abstract).

附中文参考文献:

[2] 刘建,余宏亮,沈美明,郑纬民.LINUX 机群系统并行程序调试器的设计与实现.计算机工程, 2002,28(4):7~9.

[12] 刘建,沈美明,郑纬民.调试器对并行程序干扰特性的研究.计算机学报,2002,25(2):122~129.

© 中国科学院软件研究所 http://www.jos.org.cn