语义相关的中粒度动态更新技术研究*)

杨沙洲　杨学军

(国防科技大学计算机学院　长沙 410073)

摘　要　软件动态更新技术是保证系统长时间的可靠运转和高可配置能力的关键技术，软件系统愈趋复杂的今天，其重要性得到越来越多的认同。这一技术通常可分为以代码插入技术为代表的细粒度更新和构件级的粗粒度更新两类，两者在灵活性、适用性上都有各自的不足。本文针对现有动态更新技术的不足，将一组具有语义相关性的功能集合作为参与动态更新的对象，提出了一种应用于传统操作系统运行环境的中粒度动态代码更新机制 PRODUP(PROvider-based Dynamic UPdate system)，使得传统操作系统及应用程序也具备一定的动态更新能力，以满足应用过程中越来越多的高可用、高可配置要求。PRODUP 以构件动态重配思想为基础，结合动态链接技术，使动态更新过程符合透明、低耗的要求。并基于 PRODUP 实现了一套可动态更新的内核模块机制，证明了 PRODUP 思想的可行性。

关键词　软件动态更新，中粒度，语义，符号提供者

Research on Medium-granularity Dynamic Update with Semantics Considered

YANG Sha-Zhou　YANG Xue-Jun

(Institute of Computer, National University of Defense Technology, Changsha 410073)

Abstract　Dynamic Update for Software is one of the key technologies to implement a system able to run continuously and steadily for a long time, and to implement high configurability as well, which more and more researchers have recognized along with the situation that the software system is becoming complicated more than ever nowadays. There are two categories in this field. We named them by granularity: Fine granularity with code injection supporting, and Coarse granularity with component supporting. Both of they are of weakness in flexibility and applicability. We make a solution to resolve the problem under the circumstance that dynamic update can be applicable in the traditional operating systems' runtime environment. We group the functional symbols with correlative semantics as a whole, named 'PROVIDER', to take part in dynamic upgrade. That's the granularity in our solution. So we call it 'Medium-granularity'. The aim of the solution is to meet the requirement of high availability and high configurability in modern system with traditional architectures. The conception of the PROvider-based Dynamic UPdate system, so called PRODUP, is originated from the dynamic reconfiguration in component environment and the dynamic linking. It achieved the transparency and low-ost during dynamic updating. In order to make sure the feasibility the PRODUP, we develop a kernel modularity mechanism with dynamic updating ability.

Keywords　Software dynamic update, Medium-granularity, Semantics, Provider

1　引言

所谓"动态更新"是和静态配置相对应的一种系统结构、功能调整方式，即在保持系统正常运行的同时，动态地更新系统的组成部分，从而达到调整过程对系统正常功能影响最小化的目的。

对动态更新能力的需求主要来自自适应系统(自治系统)、高可用系统和软件监测、软件调试、软件升级等技术层应用三个方面[1]，其实现技术可分为两类：一种是以代码插入技术为代表的细粒度更新，例如 Dyninst[2] 和 DLD[3]；另一种则是基于构件环境实施的粗粒度更新和配置技术，CORBA[4] 和 Jini[5] 都对此有深入的研究，IBM 的 k42 操作系统[1]也采用了这一技术。

细粒度更新技术又可以细分为运行库级和代码段级，前者以 DLD 为代表，后者以 Dyninst 为代表。它的特点是适用面广、灵活性高，参与更新的实体没有语义上的要求。对象级的动态更新拥有边界清晰、模块化程度高等优点，但不适用于非面向对象的软件环境。我们知道，尽管构件化操作系统环境已经获得越来越多的应用支持，但非面向对象的操作系统仍然主宰着现实中的大部分服务系统，但在这一领域中，动态更新支持仍然是相当有限的。在研究中我们发现，传统操作系统应用中，很多情况下我们所要更新的模块是与某一种特定功能相关的代码集合，而在现有的没有语义介入的符号组织系统中进行此类动态更新操作具有相当大的难度。例如动态调试技术，通常要求更新应用程序的某一类相关联的符号，而无需更新整个运行环境；这一类符号有可能属于同一个运行库中，也有可能分属不同的运行库(或者应用程序本身)；因此，无论是运行库级的细粒度更新还是代码段级的细粒度更新都不能清晰地描述所要更新的代码集合。但如果将这一类相关联的符号整合在一起作为动态更新的最小单位，就可以满足动态调试活动的需要。在本文中，我们将这一个相关符号集合称为符号功能实现(简称符号实现)的提供者，命名为 Provider，并围绕 Provider 构造了一种新型的动态代码更新机制 PRODUP(PROvider-based Dynamic UPdate system)。

文章分为 4 节，第 1 节(即本节)介绍了应用对软件动态更新的需求和目前动态更新技术的研究现状和不足；第 2 节详细阐述了 PRODUP 系统的设计思想和结构；第 3 节介绍了基于 PRODUP 所实现的一套应用系统原型；最后总结全文，并指出下一步的研究工作。

2　中粒度动态更新环境

传统的非面向对象的应用程序可看作是一系列功能函数的有序组合，各个功能函数在静态的代码空间中被表示为一个个相互关联的符号。ELF 文件格式即是一种应用广泛的

*)课题来源：OS 新技术研究，项目号：2003AA1Z2060。杨沙洲　博士生；杨学军　教授，博士生导师。

代码组织形式[6]，Dyninst 及 DLD 等细粒度动态更新技术都是以 ELF 文件格式为基础的。

正如第 1 节所言，细粒度动态更新技术没有考虑到符号间的语义关联，其根本原因就在于传统的符号组织形式是不包含语义信息的。本文提出的中粒度动态更新环境的核心思想就是将语义信息结合到符号组织系统中。PRODUP 系统的符号组织子系统称为“符号网”，除此之外，PRODUP 还包括面向应用程序的接口层 Guest 和面向可更新代码的接口层 Target，三者之间的关系见图 1。

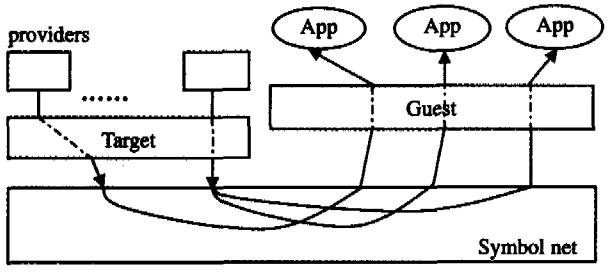

图 1　PRODUP 总体结构

PRODUP 系统、可更新代码的容器 providers，以及应用代码 App 构成了整个具备中粒度动态更新能力的应用程序。providers 通过 Target 接口层和符号网发生联系，将自身作为一组符号的实现提交到符号网中，而 App 则使用 Guest 所提供的接口访问符号网中的符号，进而访问到实际提供符号实现的 providers 中的代码。

2.1　内置语义关联的符号网

传统的符号组织系统以符号名为索引进行组织，PRODUP 的符号网保留了这一组织方式，并在此之上构造同属于一个 provider 的符号链，因此每一个符号都同时位于符号的语法网络和语义网络之中，构成 PRODUP 符号网的一个节点。对符号的访问和传统运行库的访问方式相同，仍然通过符号名进行，而动态更新操作则从 provider 着手，一次更新即完成一个 provider 中所有符号的更新动作。例如，某应用程序的符号网中存在 sym1、sym2、sym3、sym4 四个符号，其中 sym2、sym3 属于 providerA，我们分别用 sym2(A)和 sym3(A)来表示；假设 sym2、sym3 符号的另一个实现版本位于 providerB 中，则用 providerB 更新 proiverA 后，符号网中将包括 sym1、sym2(B)、sym3(B)和 sym4 四个符号。

由此可见，结合 provider 信息构造的符号网实际上是重新组织非语义化的运行库的一种方法。在 PRODUP 中，我们没有对 provider 之间的关系进行约束，因此，不同的 provider 可以代表不同的语义关联，包含不同的符号集合，它的内容由组织动态更新的用户决定。

2.2　动态更新过程

符号网是可更新符号及其实现代码的容器，动态更新操作实际上就是对符号网内容的更新。PRODUP 的 Target 模块是 provider 进入或离开符号网的接口，实现了对 provider 的加载(load)和卸载(unload)两种操作，加载操作可实现符号网的版本升级，卸载操作用于清理不被使用的“空闲”provider。

放松的更新时机约束　一般来说，进行动态更新时，如果被更新的模块处于“空闲”状态，即没有对该模块功能的访问需要，更新操作最简单。这是一种最严格的更新时机约束，可以认为静态更新机制就是这一约束的极端情况。Linux 等操作系统的“可加载模块”机制所采用的约束方式就是这种方式(见第 3 节)。

PRODUP 的设计和大多数粗粒度更新机制一样，允许更新当前有访问需要、但并不处于“访问中”的模块。例如即使应用程序需要使用 providerA 的 func1，但只要启动更新操作时 func1 不处于运行状态，就允许 providerA 被更新。

符号实现的多版本并存　和运行库级的细粒度更新机制不同，provider 是根据应用需要来决定所包含的内容的，因此在参与一个应用程序的多个 provider 中，所提供的符号实现并不要求一一对应。在动态更新过程中，这种非对称的 provider 的组织方式很容易出现同一 provider 的一部分符号被更新，而另一部分符号仍需提供服务的情况。例如 providerA 提供了 sym1、sym2 的实现，而 providerB 只提供 sym2，providerB 被加载时，sym2 被更新了，而 sym1 仍将使用 providerA 的实现。显然，如果 sym1 不处于“空闲”状态，providerA 就不能卸载，sym2 就同时有来自 providerA 和 providerB 的两个版本并存。为了保证一致性，仅由最新版本的符号实现对外提供服务。

为了在符号网中实现多版本并存，我们扩展了符号网中符号节点的功能属性，使其不是仅包含一个实现，而是将对同一符号提供不同版本服务的符号实现组织成链表，表头为最新版本，我们称之为活跃实现(如图 2 中的 sym2(B))。如果某个 provider 所提供的所有实现均处于不活跃状态，则允许卸载该 provider。

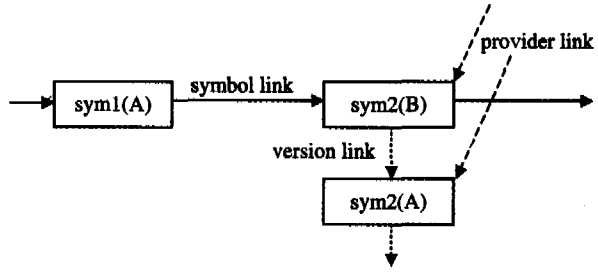

图 2　多版本符号实现的组织

非阻塞更新　允许符号实现的多版本并存的同时也方便了卸载操作的延后。一般情况下，对处于“访问中”状态的模块进行动态更新或者返回失败，或者更新动作挂起，等被更新模块离开“忙”状态后再继续。PRODUP 选择与 k42、DLD 类似的非阻塞方式，即更新操作总是立即返回，忙状态的符号实现继续服务于已启动的访问请求，而更新完成后的新访问则使用新的符号实现。仅当原有符号实现所属 provider 的所有符号都处于空闲状态时，该 provider 才被卸载。

符号之间的依赖关系　粗粒度的更新机制将符号内置于强关系的对象(构件)类型中[1,4,5]，符号对外不单独提供服务。细粒度更新技术大多不考虑符号之间的依赖关系[2,3,7,8]，而将这一工作交给程序员来完成。

在 PRODUP 中，provider 里包含了对外部环境依赖关系的描述，只有当外部环境能够满足所依赖的符号时，加载才能成功；同时，被依赖的符号和“访问中”的符号拥有相同的状态，仅当 provider 所提供的符号实现中没有任何一项被其他 provider 依赖，该 provider 才被认为是“空闲”的。

动态更新应用的很多情况中，新版本并不完全替代老版本，而是依赖于老版本，从而为老版本扩充功能，在 PRODUP 中表现为同名符号间的跨版本访问。我们的符号依赖机制可以清楚地描述这种需求。

2.3　对动态符号的访问

对应用程序来说，最理想的访问方式是对所请求的功能是否动态化完全不可知，像访问静态符号那样访问动态符号。

在粗粒度更新系统中就是采用的这种方式。因为动态符号的实现者是可变的，所以应用程序不能假定其实现者的地址，而只能通过某一种“媒介”来间接访问。显然，这种方式会带来一定的开销，例如 K42[1,9] 系统，对动态对象的访问都经由中间媒介对象(Mediator)进行，Mediator 中维护着一张地址表，查表的开销是不可避免的。

与间接方式相对应的是直接方式。如果说间接方式偏重于灵活性的话，直接方式则更注重访问的高效。DLD、Dyninst 等细粒度更新系统即采用直接方式，在进行动态更新时直接修改应用程序，使其总是能访问到当前最新的代码。此方法最大的问题在于如果没有类似 ELF 文件的重定位表那样的数据结构支持，符号的调用者是很难搜集齐的。

为了满足灵活性要求，PRODUP 采用了一套高效的间接访问机制：符号网中维护一张所有在编符号的“活跃实现地址指针表”，表中各项的地址记录在符号节点中，应用程序中对某符号(例如 syml)的引用表示为一个二级指针，其中存储着地址指针表中该符号所对应的位置(如图 3)。当应用程序访问 syml 时，它通过 syml 查到指针表的对应项，再由此对应项中保存的地址找到实际的 syml 符号实现。

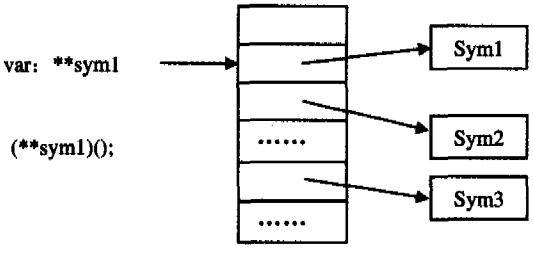

图 3 最小开销间接访问方式

在进行动态更新时，Target 模块负责更新地址表内容，而 Guest 模块则提供如上对地址表的访问例程。只要保证 Target 和 Guest 对地址表操作的互斥，应用程序对动态符号的访问就可以维持在一个高效、透明的水平。

3 基于 PRODUP 思想的内核调试器

Linux/FreeBSD 等类 Unix 操作系统都将驱动程序等内核功能部件组织成可加载的模块，需要时才从存储设备上动态加载到内核空间中。当已被加载的模块不再被使用(即处于空闲状态)时，模块才允许卸载。内核可加载模块机制中限制时机的限制给模块的开发和使用带来了很多不便。同时，内核中一些关键功能是系统运行的基础，例如调度器、内存管理等，只要系统在运行，就不可能存在这些功能不被使用的情况，这也限制了内核的扩展能力。我们基于 PRODUP 思想对 FreeBSD[10] 的 KLD 机制进行了动态化改进，使之具备了动态更新能力。

KLD 的动态化包括三个方面的工作：

• PRODUP 核心功能的实现：此部分主要包括符号网的维护、Target 接口和 Guest 接口，设计为一个模块；

• 内核动态化：将内核符号访问动态化，即采用 Guest 接口来访问(在实验系统中，我们没有更改内核代码，而是自行设计了一套使用 Guest 接口的应用模块)；

• 各个 provider 的实现：每个 provider 实现为一个模块。

在实验中，kldload/kldunload 被用作 provider 的加载/卸载器。

PRODUP 给应用程序运行带来的开销主要体现在三个方面：初始化开销、加载开销和访问开销(含锁开销)。我们采用 Intel P4 2.1G、512M 内存的 PC 机，在 FreeBSD5.2.1 环境下，以仅包含一个符号的 provider 为目标，针对与该符号相关的初始化耗时、加载耗时，以及在内核中访问该符号的耗时进行了测试，获得以下数据。

测试项目	非动态化的耗时	动态化的耗时	开销
初始化	0ms	12ms	12ms
加载	13ms	17ms	4ms
访问	21ms	22ms	1ms

实验证明，PRODUP 可以实现过程式环境中的动态更新。

总结与研究展望 PRODUP 是一种基于传统操作系统和应用程序的动态更新技术，它的特点是以语义为纽带组织符号集合作为动态更新的单位，粒度适中，灵活性高，适用性强。PRODUP 可以在一定程度上满足软件动态性的需要，也为高可用系统提供了技术准备。

但一套稳定可靠的动态更新机制还有很多问题需要解决，例如符号间的依赖关系，很多情况下并不是显然的，靠程序员进行罗列会极大降低易用性。而且很多系统各个符号间的相关性可能比较强，因此难以分离出比较独立的符号集合构成一个 provider，这时就必须引入相关性分析、状态保存、迁移等技术来解决强相关的符号的更新。

另一个有待研究的问题是符号的引用跟踪。PRODUP 系统中，地址指针表能够保证所有显式的符号引用都能访问到有效的符号实现，但我们无法跟踪隐含的访问，例如应用程序通过函数指针保存了失效的符号实现地址。这一问题在大多数动态更新系统中都存在[2,3,7,9]，通常或者要求编译器提供支持跟踪间接引用，或者要求应用程序编写时遵循一定的规范(例如类似 Java 的编程风格)。

随着软件系统功能和结构越来越复杂，对软件的动态更新能力的需求也越来越突出。很多研究者都致力于实现一个灵活、高效且应用范围广的动态更新环境，PRODUP 就是这样一种尝试。

参 考 文 献

1 Soules C A N, Appavoo J, Hui K, et al. System Support for Online Reconfiguration. In: Proc. of the Usenix Technical Conf. 2003
2 Buck B, Hollingsworth J K. An API for runtime code patching. The International Journal of High Performance Computing Applications, 2000, 14: 317～329
3 Ho W W, Olsson R A. An approach to genuine dynamic linking. Software|Practice and Experience, 1991, 21(4): 375～390
4 Group O M. The common object request broker: Architecture and specification. version 3.0.3. http://www.omg.org/technology/documents/formal/corba_iiop.htm, Aug. 2004
5 Sun Microsystems, Jini Architectural Overview. http://wwws.sun.com/software/jini/whitepapers/architecture.html. January 1999
6 Lu Hongjiu. ELF: From the Programmer's Perspective. ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz.
7 Hollingsworth J K, Miller B P, Cargille J. Dynamic Program Instrumentation for Scalable Performance Tools. Scalable High-Performance Computing Conf., Knoxville, Tenn, 1994. 841～850
8 Beyer S, Mayes K, Warboys B. Application-compliant networking on embedded systems. In: Proc. of the 5th IEEE Intl. Workshop on Networked Appliances, Oct. 2002
9 Hui K, Appavoo J, Wisniewski R, et al. Supporting Hot Swappable Components for System Software. In: Eighth Workshop on Hot Topics in Operating Systems, Elmau, Germany, May 2001
10 The FreeBSD Project. FreeBSD homepage. http://www.freebsd.org, 2004