动态编译技术研究*)

崔慧敏　戴桂兰　王生原　张素琴
(清华大学计算机科学与技术系软件所　北京 100084)

摘　要　本文从动态编译的概念出发，着重讨论了运行时特定化技术和 Just-in-time 编译技术；比较全面地总结并评述了具有代表性的动态编译系统及其采用的 Profiling 技术、重编译技术；并探讨了动态编译技术研究中存在的一些问题及进一步的工作。

关键词　动态编译，运行时特定化，Just-in-time 编译，Profiling，自适应优化

A Study of Dynamic Compilcation

CUI Hui-Min　DAI Gui-Lan　WANG Sheng-Yuan　ZHANG Su-Qin
(Department of Computer Science & Technology, Tsinghua University, Beijing 100084)

Abstract　Based on the concept of dynamic compilation, the paper focuses on discussing the techniques of run-time specialization and just-in-time compilation, reviews the representative dynamic compilers and their used profiling techniques, and techniques for recompilation, and explores further some current issues of dynamic compilers and future work.

Keywords　Dynamic compilation, Run-time specialization, Just-in-time compilation, Profiling, Adaptive optimization

1　引言

传统的编译优化技术是在编译时根据应用程序的静态信息，即编译时就已比较明确的信息进行编译优化。动态编译，即运行时编译，运行时的程序变换。由于下述原因：其一，面向对象语言和技术在当代软件开发中的使用已导致较大程度的延迟绑定，限制了可供静态编译器分析的范围；其二，分别编译和共享库，编译器不一定能够看见所有的代码；其三，有些信息，如运行时常量值信息在编译时是不能确定的，运行环境是不断变化的。而动态编译优化利用在运行时提供的信息，对程序提供更完全的优化。因此，利用动态编译优化技术，可大大扩大优化范围，从而能够产生更有效的代码。

目前对动态编译的研究主要集中在三个方面：1)运行时特定化。根据运行时常量，将程序代码特定化，然后在其中做各种优化工作，如：常量传播、循环展开。2)Just-in-time 编译。主要针对 Java 程序进行的运行时编译，并根据 profiling 收集到的 profile 信息进行自适应的优化。3)动态的二进制代码转换和优化。将针对一种体系结构产生的目标码，直接移植到与之不同的另一类体系结构上运行。

本文着重对运行时特定化和 Just-in-time 自适应编译优化技术分别进行讨论，探讨动态编译的一些关键技术和具有代表性的工作，总结出其存在问题及进一步研究发展方向。

2　运行时特定化

特定化是一种程序变换技术，它以一般的程序及其存在的上下文(特定化上下文)作为输入，产生与上下文有关的特定化程序。源程序可以是一个完整的程序，也可以是一个简单的模块。特定化上下文规定全局变量的值和有关特定化程序的参数入口点。它可能包含别名分析以及描述外部函数的行为。

程序特定化包括编译时特定化和运行时特定化。编译时特定化是指当可能的运行时上下文集合固定并比较小时，程序在编译时能被重写以产生所有可能的特定化。例如：许多机器提供 8-bit 或 24-bit 的颜色，对这种情况，程序能在编译时被特定化。然而，一般来讲，编译时特定化不可能在编译时预见所有可能的特定化上下文，再者，仅一部分可能的特定化上下文能被用到，因此在编译时产生所有可能的特定化上下文将产生大量无用的代码。运行时特定化则能够弥补这些不足，运行时特定化是根据对给定的程序及其上下文描述而在运行时产生特定化代码的过程。一般来讲，编译时特定化可以理解为一种源到源的变换，而运行时特定化直接产生目标代码。

2.1　程序特定化的理论基础

程序特定化的理论基础是部分求值，它根据某些给定的输入而对程序自动进行特定化[1]。设 $p(x,y)$ 表示一个程序，它有两个参数 x 和 y。给定它的部分参数(例如 x)的值(例如 v)，那么对 p 的部分求值会执行所有依赖于 x 的值的计算，并生成一个特定化之后的程序，其中只包括与 x 的值无关的计算，我们可以将这个过程表示为(可将 x 当作一常量处理)：

$$S(p,v)=p_v$$

其中 S 表示部分求值器(partial evaluator)，p_v 表示特定化之后的程序。特定化之后的程序将 x 作为常量，执行剩余的计算并返回结果(其结果可能已经由原来的程序计算返回)。

$$p(v,w)=p_v(w)$$

由于删除了 p 中与 x 有关的计算，因此 $p_v(w)$ 的执行通常比 $p(v,w)$ 要快。

运行时特定化的基本思想是在编译时为一个给定的程序构建本地代码级的生成扩展(generating extension)。假设 $R^{L\to M}$ 代表一个从语言 L 到机器语言 M 的 RTS 系统，p^L 代表

*)基金项目：国家自然科学基金(No. 60083004)资助。

一个需要特定化的程序。那么 R 在编译时构建生成扩展 G_p，它以 p 作为静态参数并在运行时生成以 M 语言表达的 p 的特定化程序。

$$R^{L\to M}(p^L)=G_p \qquad \text{(编译时)}$$

$$G_p(x)=P_x^M \qquad \text{(运行时)}$$

传统的部分求值生成的特定化程序是源语言级的，而在运行时特定化技术中，G_p 生成的特定化程序 P_x^M 是用机器语言 M 表达的[1]。这需要在编译时创建 p 的目标代码片断(称为模板)，而在运行时只需要复制这些模板来生成特定化程序。在 RTS 系统中，第二个过程是非常高效的。

下面以一个简单的函数 *pow* 为例来分析 RTS 系统的工作过程，该函数计算 x 的 n 次方，那么 $R^{L\to M}$的输入如下：

```
(define(pow n x)
  (if (=n 0)
    1
    ( * (pow(-n 1)x))))
```

假设上面的函数针对第一个参数 n 来进行特定化。系统首先确定哪些表达式是可以在部分求值时计算的(称为静态表达式)，以及哪些表达式是需要插入到特定化之后的程序中的(称为动态表达式)。我们可以用下面标记过的程序来表达这些信息：

```
(define(pow n x)
  (if (@=n 0)
    1
    (@ * (@ pow(@-n 1)x))))
```

其中，带下划线的表达式为动态表达式，无下划线的为静态表达式，它们的值可以在特定化时计算。@标记说明该函数在部分求值时执行，如果@标记带有下划线，那么该函数的格式将被插入到特定化之后的程序中；如果@标记带有上划线，那么将对该函数体进行特定化并执行 inline 操作。根据标记信息，系统会生成一个生成扩展和模板(图 1)。生成扩展将每个动态表达式替换为将适当的模板存入内存的格式(例如图中的(load-template tn))。此外，在第一个模板之前和最后一个模板之后附加了 prologue 和 epilogue，从而可以在 inlined 的模板之间传递参数。

生成扩展：

```
(define (pow-gen n)
  (load-template t0)
  (if (=n 0)
    (load-template t1)
    (begin (load-template t2)
           (pow-gen(-n 1))
           ( load - template
           t3))))
```

模板：

label	instructions	comments
t0	push % ebp	//prologue
	movl %esp,%ebp	
t1	movl $1,%eax	//返回 1
	leave	//epilogue
t2	push -8(%ebp)	
t3	addl $4,%esp	
	imull -8(%ebp),%eax	//x * pow(x-1)
	leave	//epilogue

图 1　pow 对应的 generating extension 和模板

当 $n=3$ 时，特定化之后的代码如图 2 所示，相当于(define(pow-3 x)(* x(* x(* x1))))生成的代码。

```
push %ebp //(pow-gen 3):t0
movl %esp,%ebp
push -8(%ebp) //(pow-gen 3):t2
push %ebp //(pow-gen 2):t0
movl %esp,%ebp
push -8(%ebp) //(pow-gen 2):t2
push %ebp //(pow-gen 1):t0
movl %esp,%ebp
push -8(%ebp) //(pow-gen 1):t2
push %ebp //(pow-gen 0):t0
movl %esp,%ebp
movl $1,%eax //(pow-gen 0): t1
leave
addl $4,%esp //(pow-gen 1): t3
imull -8(%ebp),%eax
leave
addl $4,%esp //(pow-gen 2):t3
imull -8(%ebp),%eax
leave
addl $4,%esp //(pow-gen 3):t3
imull -8(%ebp),%eax
leave
```

图 2　$n=3$ 时，特定化之后生成的代码

2.2　代表性工作

运行时特定化可以大大扩大优化范围，因此人们对其做了大量的研究工作，其中具有代表性的运行时特定化系统有法国 Universitaire de Beaulieu 开发的 Tempo 系统和 Washington 大学开发的 DyC 系统。

2.2.1　Tempo　是一个基于模板的运行时特定化环境。它是一个 C 语言的部分求值器，可以执行编译时特定化和运行时特定化。

Tempo 是一个 offline 的部分求值器。部分求值过程可以分为分析和变换两个阶段。分析阶段以源程序和特定化上下文的抽象表示作为输入，根据得到的信息对能在特定化过程中得到简化的程序结构进行标注[2]。变换阶段就是特定化阶段，特定化程序(specializer)根据标记过的程序和实际的特定化上下文进行特定化，生成特定化之后的程序。

Tempo 支持编译时特定化和运行时特定化。编译时特定化生成 C 程序，运行时特定化生成机器代码。编译时特定化阶段只需要对程序进行简单变换并执行标记的动作。运行时特定化过程本身可以分编译时时阶段和运行时阶段。在编译时阶段，对能够实施特定化的基本块进行编译生成代码片断(即模板)，并生成一个专用的行定化程序。在运行时阶段，该特定化程序填充模板中运行时常量的值，从而构建特定化之后的程序。

2.2.2　DyC　是一个基于运行时特定化的 C 语言动态编译系统。DyC 系统包括一个静态编译器和一个动态编译

器[3]。其总体框架图如3所示。为了减少编译时间，静态编译器会生成一个预编译的机器码模板，模板中包含一些 holes 以便存放运行时常量的值。静态编译器还会生成设置(set-up)代码来计算导出运行时常量的值，此外，还生成指导命令(directives)指示动态编译器如何根据模板和设置代码计算出的常量来生成可执行代码。动态编译器(也称为 stitcher)只需要遵循指示代码来复制机器码模板并在 holes 中填充合适的常量值。

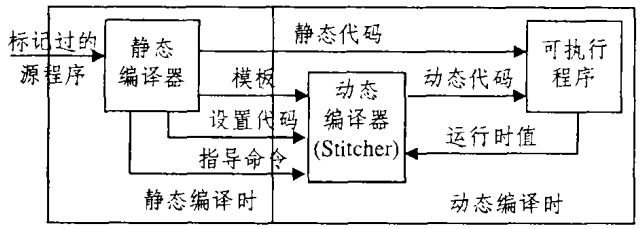

图3 DyC 的总体框架图

在 DyC 系统中，用户可以在源程序中插入指导性的注释来表明程序的哪些部分是需要动态编译的(称为动态域)以及哪些变量是原始运行时常量，通过常数传播和常量展开，系统可以自动标识出动态域中其它导出运行时常量。但是手动插入注释是一项很繁琐也很容易出错的工作，Calpa 系统弥补了这些不足。它结合 profile 信息和程序分析来自动获得注释，从而在 DyC 中驱动动态编译[5]。Calpa 系统由两个主要部分组成：profiler 和程序分析工具。Profiler 在应用程序中插入评测工具(instrumentation)，使之在运行时生成变量值和频率数据等 profile 信息。程序分析工具基于动态编译的开销/获益模型，预见特殊注释对应用程序运行的影响。

与 Tempo 相比，DyC 允许在程序的任何位置标注运行时常量的值，而 Tempo 则只允许在入口点标注。DyC 可以对任意的程序进行运行时特定化，而 Tempo 必须将所有非结构化的代码转换成结构化代码才能进行特定化，在转化过程中会引入额外的条件测试，还可能引入循环[4]。DyC 只能进行运行时特定化，Tempo 还支持编译时特定化。

3 Just-in-time 编译与自适应优化

由于 Java 语言的平台无关性使其得到广泛的应用，同时，由于计算机平台的迅速发展以及动态编译相对于静态编译和解释执行的明显优势，因此近年来 just-in-time 编译已成为编译技术的研究热点。IBM Watson Research Center、Intel Corporation 等对此进行了大量的研究，并研制出相应的基础设施，如：JikesRVM、ORP 等等。这些系统为动态编译技术的研究提供了基础平台。

3.1 关键技术

由于使用 just-in-time 的编译技术主要目标是，一方面提高目标代码的质量，以改进 Java 应用的执行性能；另一方面降低编译、运行时开销，因此如何进行 profiling 以及如何实现自适应的优化是 just-in-time 编译的关系技术之一。Profiling 信息是动态编译的基础，用以指导编译系统进行自适应的编译优化决策，以使系统能够自我调节以适应新的目标环境和新的应用需求。

3.1.1 profiling 技术 是在程序运行时收集特定信息的技术。在动态编译中，减少编译时间开销的一个重要途径就是只为那些对程序总体性能造成影响的部分进行优化，但是要发现这些关键部分，就需要进行 profiling。

profile 制导的编译过程如图4所示，由图中可以看出，在执行 profile 制导的编译优化之前，需要对一个或多个典型输入运行 instrumented 程序来收集 profile 数据[6]，然后就可以利用这些 profile 数据来指导优化编译器重新编译源程序以生成优化代码。

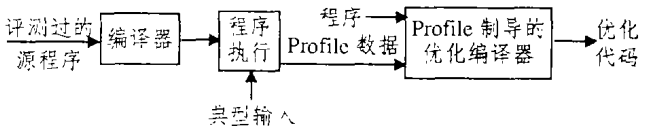

图4 Profile 制导的优化过程

profiling 过程本身需要占用一定的时间空间资源，因此需要在 profiling 带来的效率提高以及额外开销之间进行折衷考虑。在制定 profiling 方案的过程中，要着重考虑 profiling 是否引入运行时开销，profiling 信息的精确性和可用性。profiling 的内容，profiling 什么时候进行，每个应用程序特点或目标环境特征的 profiling 需要多长时间完成是 profiling 技术所要研究的关键问题。

profiling 技术包括在线(On-line)profiling 和离线(Off-line)profiling[11]。在线 profiling 是编译时在生成的本地机器代码中插入评测工具。这些评测工具生成 profile，以在程序运行时记录执行信息，供编译系统使用。由于在线 profiling 收集信息的环境和程序运行环境是相同，因此得到的 profiling 信息具有精确性的特点，但是与此同时也提高了运行时开销[11,14]。离线 profiling 是由用户将 profile 信息记录在 profile 文件中，在程序执行时通过 profile 文件来和动态编译系统进行通讯。离线 profiling 技术中 porfile 只需要生成一次，不需要增加运行时开销，但是因为在收集 Profiling 时所采用的输入集合和程序运行时的输入集合并不相同，所以收集到的 profiling 可能不够精确。

3.1.2 自适应优化决策模型 自适应动态编译优化的控制模型是动态编译的核心部分，它决定动态编译系统的自适应能力和范围。由于不同的运行环境对目标代码的质量要求各不相同，因此优化决策可以看作是一个多重目标的综合决策问题。

自适应优化决策的设计目标在于能够针对某一具体应用，根据其应用程序特点和运行环境的特征的 profiling 信息，动态确定对哪个部分、采用什么样的优化层次进行编译或重编译优化这一编译优化过程提供有效的支持[16]。自适应动态编译优化系统通常包括评测工具、控制器以及重编译系统。其中，评测工具负责对应用程序特点和目标环境的 profiling 信息的获取、分析和维护，并将分析结果传递给控制器。控制器利用有关的 profiling 信息以及编译优化知识库的信息，对优化编译器应该编译哪个方法、优化在哪个层次上进行以及进一步实施 profiling 做出规则。

3.2 代表性工作

3.2.1 ORP ORP(Open Runtime Platform)是 Intel 实验室的 Microprocessor Research Lab(MRL)开发的一个 MRTE(Managed Run-Time Environment)系统。在 ORP 平台上可以设计开发各种 just-in-time 编译器(JIT)，垃圾收集(GC)，多线程以及同步实验。ORP 提供了精确的垃圾收集、线程同步以及多个 just-in-time 编译器。在 ORP 中没有解释器。ORP 支持两种不同的 MRTE 平台：Java 和 CLI(Common

Language Infrastructure)。

ORP 中有三个主要的组件:核心虚拟机(core VM)、just-in-time 编译器(JIT)以及垃圾收集(GC)。核心虚拟机主要负责类的装载。垃圾收集负责管理堆、为对象分配空间,并在堆满时收回垃圾。JIT 编译器负责将 bytecode 编译为本地指令。ORP 允许多个 JIT 共存。每一个 JIT 通过 JIT 接口来与核心虚拟机交互。ORP 中设计了两个 JIT 编译器:一个是快速的代码生成器(称为 O1 JIT),它直接从 JVM bytecode 生成目标代码,而不实施复杂的优化。另一个是优化的编译器(称为 O3 JIT),它将 JVM bytecodes 首先转换为中间表示(IR),然后在中间表示上可以进行很多复杂的优化。

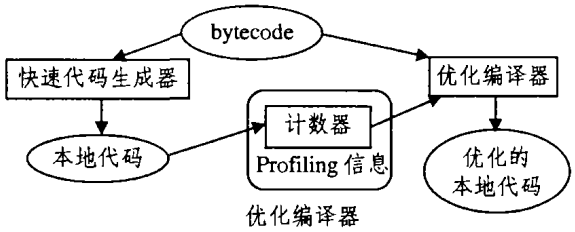

图 5 ORP 的动态编译模型

ORP 的动态编译模型如图 5 所示。ORP 动态编译模型的思想是自适应地选择执行频率较低的方法尽快而简单地生成目标代码,而只对执行频率较高的方法实行开销较大的优化。当一个方法被首次调用时,由 O1 JIT 编译生成本地码,并在生成代码中插入评测代码来收集 Profiling 信息[7,12,13]。O1 JIT 在两种位置插入评测代码:方法入口点和回边。前者表明一个方法是调用相关的,后者表明方法是循环相关的。每次执行这些代码时,评测代码都会更新 profiling 信息。根据收集到的 profiling 信息,源程序中的部分方法将会被标识为执行频率较高并调用 O3 JIT 重新编译,并根据 profiling 信息来指导优化。

在 ORP 中,触发重编译的方法有两种。一种方法是评测(instrumenting)。首先将程序中计数器的值设置为固定的阈值,当程序执行到计数器所在的程序点时,相应计数器的值减一。一旦计数器的值达到零,那么就立即触发重编译过程。这种方法的优点是一旦方法的执行次数达到了阈值就可以立即进行重编译,使得程序可以尽快使用优化之后的代码。但是这种方法的缺点是选择合适的阈值比较困难。另一种触发重编译的方法是使用独立的线程。这种方法可以在多处理器的系统中同时执行编译和执行过程,从而降低编译时间的开销。在这种方法中,系统创建一个独立的线程来定期扫描所有方法的 profiling 信息确定执行频率高的方法,并使用空闲的处理器来执行重编译。这种方法的优点是可以充分利用多处理器的特点来使得编译和执行能够重叠。但是同时也有两个缺点:第一是这个线程需要扫描所有方法的 profiling 信息才能确定哪些方法需要重编译;第二是执行频率较高的方法不能在达到阈值时立即重编译,而需要等到下次扫描时才能被重编译。

3.2.2 Jikes JVM 是由 IBM T. J. Watson Research Center 开发的一个供研究用的 Java 虚拟机。Jikes JVM 采取的是只编译(compile-only)策略[8]。Jikes JVM 所有的子系统都使用 Java 语言编写,但是 Jikes JVM 能够自引导,而不需要在其它的 JVM 上来运行系统,用纯 Java 实现系统的一个优点是 Jikes JVM 可以动态优化自身。

Jikes JVM 中的子系统包括:动态的类加载器、动态链接器、对象分配器、垃圾收集、线程调度、profiler、三个动态编译器以及其他一些运行时支持,例如例外处理和类型检测等。Jikes JVM 中的三个编译器为:1)基本编译器。它通过模拟 Java 的操作数栈来将 bytecodes 直接转换为本地码。它不执行寄存器分配,性能只比解释器稍好。2)快速编译器。与 ORP 类似,快速编译器不执行复杂优化,不生成中间表示。3)优化编译器。它将 bytecodes 转换为中间表示,并在中间表示上进行一系列优化。与 ORP 不同的是优化编译器将所执行的优化分为三个层次,根据系统收集到的 profiling 信息,Jikes JVM 判断达到的阈值,并在调用优化编译器的时候选择不同的优化级别。

Jikes JVM 的结构图如图 6 所示。Jikes JVM 的自适应优化系统包括三个组件,分别是运行时度量子系统、控制器以及重编译子系统[9]。除了这三个主要的组件之外,AOS 数据库能够保存组件的决策并允许组件来查询决策。运行时度量子系统收集正在执行的方法信息,并对这些信息进行总结之后通过管理器时间队列传送给控制器,或者将信息存储到 AOS 数据库中。控制器从运行时度量子系统或者 AOS 数据库中接受信息并根据这些信息来决定如何编译,之后将做出的编译决策传递到重编译子系统来启动合适的编译器。重编译子系统的多个编译线程从控制器生成的编译队列中提取并执行编译计划。Jikes JVM 支持后台重编译(Background recompilation),重编译子系统可以启动编译线程来执行编译策略,编译线程可以和运行线程并行执行。AOS 数据库是自适应优化系统用来记录决策、事件以及静态分析结果的存储空间。

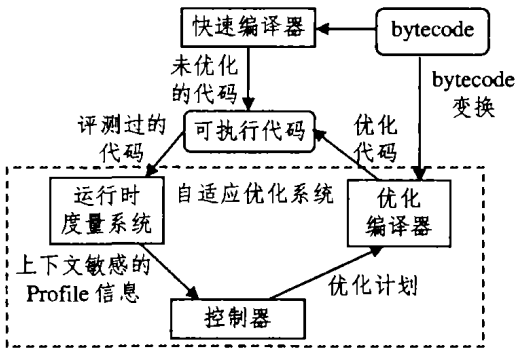

图 6 Jikes JVM 结构图

4 存在问题及进一步工作

动态编译技术的研究已取得了相当大的进展,有大量的先进技术和工具为之提供支持。动态编译技术的发展不仅仅受到传统编译技术因素的影响,许多其它因素也会对其产生不容忽视的影响。在一些重要因素的影响下,编译基础设施的研究与应用存在着一些问题。

(1)运行时特定化应作为一个优化技术嵌入在编译器。由于运行时特定化直接产生机器语言级的特定化程序,因此 RTS 系统在运行时可以执行得足够快。但又由于这种从源程序片断中产生机器指令,但不同的源语言、不同的目标机,都需要专用的编译器。

另外,为提高程序的特定化过程的效率,在特定化时对特定化代码所做的优化较少(如指令调度和寄存器分配),从而使特定化代码的执行效率较低。这主要原因是许多优化算法引入到 RTS 系统比较困难,多数优化算法的开发目标是用于

编译的，快速实现也比较困难。有些优化算法与其执行平台有关。因而也就不能象传统的静态编译器可以在中间表示级做大量的与目标机无关的优化。鉴于运行时特定化的优缺点，将运行时特定化作为一种优化技术嵌入到传统的编译器中。这样一方面可以充分发挥其优势，同时可以有效地弥补其不足。

(2)执行延迟问题。引入运行时编译开销是运行时特定化和 just-in-time 编译都存在的问题。如何在获得较好的性能的同时引入最少的运行时编译开销是动态编译系统需要解决的一个关键问题。减少执行延迟的有效途径主要有两个方面，一是减少运行时编译的代码，即 Lazy compilation；另一个在多处理器系统中利用空闲的处理器来执行运行时编译任务，使得程序的运行和编译可以并行执行，即后台编译[10]。

(3)可重定向问题。开放式平台的显著特点之一是它的平台无关性。动态编译技术的另一个问题是如何解决重定向的问题，真正做到与平台无关，并能根据目标机执行相关的优化。由于动态编译系统运行时编译的特点，因此可以更加充分地利用机器信息进行编译的优化和决策，甚至可以在运行时收集部分机器信息，从而获得更好的可重定向性。但是现有的动态编译系统在这些方面做得都还很少，因此可重定向问题也是动态编译技术研究中的核心问题。

(4)决策模型的自适应问题。由于应用环境千差万别，因此对目标代码的要求也各不相同。而一般的编译优化技术的目标主要在于提高目标代码的执行性能，而对代码占用的存储空间和执行空间以及功耗方面则考虑较少。编译优化是一个多目标优化问题，对一个动态编译优化系统来说，如果能够自适应地实现多目标优化则可以大大提高其自适应性。而现有的系统大多采用由固定优化方法集合组成的双重编译器，不支持不同应用程序特点和不同优化目标的综合考虑，因此，自身的灵活性较差，难以适应目标环境及应用模式的变化。

结束语 动态编译作为一种新兴的编译优化技术，有广阔的发展前景。本文探讨了运行时特定化和 JIT 编译的特点及其代表性系统。动态编译能够扩大优化范围，提高系统性能，但是动态编译优化引入运行时的编译开销，从而可能会产生执行延迟。因此，利用动态编译优化技术，要在动态编译开销和动态编译代码的质量间做出折衷考虑。

参 考 文 献

1 Masuhara H, et al. Dynamic Compilation of a Reflective Language Using Run-Time Specialization. Principles of Software Evolution, 2000. In: Proc. Intl. Symposium on, Nov. 2000. 128～137
2 Noel F, et al. Automatic, Template-Based Run-Time Specialization: Implementation and Experimental Stud-y. Computer Languages, 1998. In: Proc. 1998 Intl. Conf. on, May 1998. 132～142
3 Auslander J, et al. Fast, Effective Dynamic Compilation. In: Proc. of the ACM SIGPLAN'96 conf. on Programming language design and implementation, May 1996
4 Grant B, et al. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Theoretical Computer Science, 2000, 248(1-2)
5 Mock M, et al. Calpa: A Tool for Automating Dynamic Compilation. Microarchitecture, 2000. MICRO-33. In: Proc. 33rd Annual IEEE/ACM Intl. Symposium on, Dec. 2000. 291～302
6 Gupta R, Mehofer E, et al. Profile Guided Compiler Optimizations. The Compiler Design Handbook: Optimizations & Machine Code Generation, Auerbach Publications
7 Michal C, et al. Practicing JUDO: Java Under Dynamic Optimizations. In: Proc. of the ACM SIGPLAN '00 conf. on Programming language design and implementation, Aug. 2000
8 Michael G B, et al. The Jalapeno Dynamic Optimizing Compiler for Java™. ACM. In: Proc. of the ACM 1999 conf. on Java Grande, June 1999
9 Matthew A, et al. Adaptive Optimization in the Jalapeno JVM. In: Proc. of the conf. on Object-oriented programming, systems, languages, and applications, Oct. 2000
10 Chandra K, et al. Reducing the Overhead of Dynamic Compilation. Software: Practice and Experience, 2000, 31(8): 717～738
11 Krintz C. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. Code Generation and Optimization, 2003. CGO 2003. In: Intl. Symposium on, Mar. 2003. 69～78
12 Ali-Reza A, et al. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In: Proc. of the ACM SIGPLAN '98 conf. on Programming language design and implementation, May 1998
13 Michal C, et al. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. In: Proc. of the 2002 joint ACM-ISCOPE conf. on Java Grande, Nov. 2002
14 Matthew A, et al. Online Feedback-Directed Optimization of Java. In: Proc. of the 17th ACM conf. on Object-oriented programming, systems, languages, and appliations, Nov. 2002
15 Bowen A, et al. The Jalapeno virtual machine. IBM Systems Journal, Java Performance Issue, 2000, 39(1)
16 Bruening D, et al. An Infrastructure for Adaptive Dynamic Optimization. Code Generation and Optimization, 2003. CGO 2003. In: Intl. Symposium on, Mar. 2003. 265～275

(上接第 103 页)

参 考 文 献

1 Akyildiz I F, Su W, Sankarasubramaniam Y, Cayirci E. A survey on sensor networks. IEEE Communications Magazine, 2002, 40(8): 102～114
2 Heinzelman W R, Chandrakasan A, Balakrishnan H. Energy-efficient communication protocol for wireless microsensor networks. In: Proc. of the33rd Intl. Conf. on System Sciences (HICSS '00), Jan. 2000. 1～10
3 Intanagonwiwat C, Govindan R, Estrin D. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In: Proc. of the Sixth Annual ACM/IEEE Intl. Conf. on Mobile Computing and Networking (Mobicom'2000), Boston, Massachusetts, August 2000
4 Manjeshwar, Agarwal D P. TEEN: a routing protocol for enhanced efficiency in wireless sensor networks. In: 1st Intl. Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing, April 2001
5 Lindsey S, Raghavendra C S, Sivalingam K. Data Gathering in Sensor Networks using the Energy * Delay Metric. In: Proc. of the IPDPS Workshop on Issues in Wireless Networks and Mobile Computing, 2001
6 Heidemann J, et al. Building efficient wireless sensor networks with low-level naming. In: Proc. of the ACM Symposium on Operating Systems Principles, Banff, Canada, Oct. 2001
7 Intanagonwiwat C, et al. Impact of network density on data aggregation in wireless sensor networks: [Technical Report 01-750]. University of Southern California, Nov. 2001
8 Krishnamachari B, Estrin D, Wicker S. Modelling data-centric routing in wireless sensor networks. In: Proc. of IEEE Infocom, 2002
9 Krishnamachari B, Estrin D, Wicker S. Impact of data aggregation in wireless sensor networks. In: Intl. Workshop on Distributed Event-Based Systems, Vienna, Austria, July 2002
10 Lindsey S, Raghavendra C S. PEGASIS: Power Efficient Gathering in Sensor Information Systems. In: Proc. of IEEE Aerospace Conf. 2002
11 Kalpakis K, Dasgupta K, Namjoshi P. Maximum Lifetime Data Gathering and Aggregation in Wireless Sensor Networks. In: Proc. of IEEE Networks'02 Conf. 2002
12 Yao Y, Gehrke J E. The Cougar Approach to In-Network Query Processing in Sensor Networks. Sigmod Record, 2002, 31(3)
13 Madden S R, et al. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. OSDI, Dec. 2002
14 He T, et al. AIDA: Adaptive Application Independent Data Aggregation in Wireless Sensor Networks. ACM Trans. on Embedded Computing System Special issue on Dynamically Adaptable Embedded Systems, 2003