高技术通讯 2013 年 第23 卷 第11 期:1131 ~1138

doi:10.3772/j.issn.1002-0470.2013.11.005

支持算法组件自动替换的编程范式及编译框架①

李恒杰②[*,**] 何文婷[*,**] 陈 莉[*] 刘 雷[*] 吴承勇[*]

([*]中国科学院计算技术研究所 北京 100190)

([**]中国科学院大学 北京 100049)

摘 要 针对算法组件的自动替换蕴含的算法组件的兼容性判定问题,提出了一个算法组件的约束模型,从而将算法组件的兼容性判定转化为约束模型的兼容性判定。为解决转化后的兼容性判定,采用了分治思想,即只需判定原子约束强度,约束模型间的兼容性可由原子约束强度结合约束强度推导规则与放松的组件接口兼容性判定规则自动导出。为使算法自动替换更具实用性,提出了支持算法自动替换的编程范式,使得自动替换前期涉及的手工工作在编程所涉及的不同角色之间得到合理分配。设计了编译框架原型,或支持算法与编译优化选项的自动选择。实验结果显示,该系统在9 个测试用例上获得的平均加速比为2.29。

关键词 算法组件兼容性判定,编程范式,编译框架,迭代编译

0 引 言

算法替换是指把程序中的算法替换成符合上下文语义要求的其他算法,其目的是使程序更符合优化目标(如程序性能)。需要系统能自动完成算法替换的理由是:不同的算法提供了不同的折中方案,程序员可能很难确定哪种算法对其所要解决的问题是最合适的[1];算法实现、编译优化、硬件体系结构之间相互影响甚为复杂,使程序员很难对算法实际性能进行估计[2];由于多核处理器与各种加速器(如 GPU,FPGA)的出现使得硬件平台日益复杂与多样化,需要通过自动算法替换使程序的性能具有可移植性。算法的自动替换蕴含着语义等价性判定问题,而任意两个程序的语义等价性的判定仍未解决[3],因此,支持算法自动替换的方案均需要人工维护或描述语义的等价性。如支持算法自动替换的系统[3-7],其限制算法替换只在算法库中的等价算法之间进行,而这种等价性信息需要人工存储在数据库中或者内建在算法库中。支持算法选择的语言[2,8,9],则把人工维护的工作转嫁到程序员身上。从已有的工作来看,使用语言对算法级语义进行描述要么停留在某一专有领域[8,10],要么使描述方式变得不为普通程序员接受[11]。

为使算法自动替换方案具有通用性,并能利用其调优遗产代码,文献[12]把算法实现封装在软件组件[13]中,使对语句层次上的算法识别和替换,转换到组件层次上来。同时为了扩大可替换算法的范围,从一般地要求算法在语义上等价弱化为在语义上兼容;通过使用说明兼容图将库中的算法组件组织起来,使图中任一结点(组件)的所有子孙结点与该结点兼容。程序员通过组件化程序,并在调用算法处使用图中层次尽可能高的且满足上下文语义的结点,系统能自动将其替换成与其兼容的组件。本文沿用文献[12]中组件化的思想,但在兼容性判定方面做了改进,以降低人工维护语义兼容性的成本。首先就算法组件兼容性判定提出了一个算法组件的约束模型,通过判定约束模型的兼容性来获知组件的兼容性。其次,在组件兼容性判定上采用了分治的思想,即通过引入约束强度、原子约束、约束表达式、约束推导规则及接口兼容性判定规则,把对组件整体的兼容性判定转化为对原子约束强度的判定。

① 863 计划(2012AA010902),973 计划(2011CB302504),国家自然科学基金(60970024)和国家自然科学基金创新研究群体科学基金(60921002)资助项目。

② 男,1983 年生,博士生;研究方向:编译技术;联系人,E-mail:lihengjie@ict.ac.cn

(收稿日期:2012-12-31)

而这种转化,将降低判定组件兼容性时的人力成本与出错概率。为使算法自动替换方案更具实用性,我们提出了一种支持算法自动替换的编程范式,规范了编程过程中各方角色的行为,使得自动替换前期涉及的手工工作得到合理分配。我们的编译框架原型将算法替换和搜索与迭代编译[14,15]相结合。实验结果显示,我们的系统通过为 9 个测试用例自动选择算法与编译优化选项使其获得 2.29 的平均加速比。

1 算法兼容性判定

对给定任务,往往可由多种不同的算法来完成。然而,由于算法在设计与实现时往往会对输入、输出添加各种不同约束,导致解决同一问题的算法在不同应用的上下文中并不一定兼容。如对于静态图的单源最短路径问题,我们有熟知的 Dijkstra 算法与 Bellman-ford 算法。前者仅能处理边权值不为负的情况,后者能处理边权值为负的情况。对边权值为非负的图,两者是兼容的,反之,则不兼容。为了判定算法就具体上下文中的任务而言是否兼容,我们首先给出蕴含算法约束信息的约束模型,然后给出我们的约束强度推导规则与放松的组件接口兼容性判定规则,最后给出约束模型的兼容性判定规则。

1.1 算法组件与组件约束模型

为了简化算法兼容性的判定与算法在程序中的替换,我们把算法实现封装成软件组件,使外部程序只能通过组件接口与组件中的算法实现通信(交换数据)。我们并不对算法组件的接口设计做限制,因此,做数据抽象的接口设计(如基于泛型编程实现的 Boost 算法库)与不做数据抽象的接口设计(如 C 的数值算法库),均纳入我们考虑的范畴。我们对算法组件统一的约束模型描述于图 1 中。

- CPT(IN, OUT, ATTR)
- IN = < I_1(C), I_2(C),.., I_N(C) >
- I_i = <I-NAME$_I$, ALGO$_I$, IMP$_I$>$_i$
- C ∈ {必选参数, 可调参数, 可忽略参数}
- OUT = <O_1, O_2, .., O_M>;
- O_j = <O-NAME$_O$, ALGO$_O$, IMP$_O$>$_j$
- ATTR = <TASK, ALGO$_A$, IMP$_A$>

图 1 算法组件的约束模型

图 1 中,CPT 表示算法组件名,IN 和 OUT 分别用于表示组件输入、输出的约束。每个输入、输出参数上的约束(I_i 和 O_j),即包括算法级的约束 ALGO(如计数排序要求输入为整数,快速排序输出不稳定的排序结果),也包括实现级的约束 IMP(如归并排序的待排序对象使用向量存放还是列表存放)。I-NAME 和 O-NAME 分别用于表示输入、输出参数的名称,对完成同一任务的不同算法组件,若其参数描述的是对应概念,则参数名设为相同。算法的特征约束,用 ATTR 来刻画。它包括算法级特征约束 ALGO$_A$(如安全 hash 函数 SHA1[16]的理论抗攻击强度是 2^{63})和实现级特征约束 IMP$_A$。我们把算法组件接口的参数分成三类:强制性参数,可调参数,可忽略参数。参数的分类涉及到两个层面的意义:一是为了说明程序员在使用算法组件时是否必须为该参数传参;二是为了告知推导系统在判定算法组件兼容时,如何对该参数做处理。我们将在 1.3 节详细解释参数名与参数分类的用途。

1.2 约束表达式与约束强度推导规则

为了刻画算法组件的兼容关系,我们引入约束强度的概念。例如对约束 A = "边权值为非负实数"和约束 B = "边权值为实数",约束 A 的强度要大于 B,则我们记为 A - > B。

输入、输出参数上的约束与算法特征约束通常为复合约束,如某最短路径算法[17]的输入约束为 S = "平面有向图,边权值为实数,环权值非负"。为了简化人工比较复合约束之间强度的工作,先人工把复合约束分解成原子约束,并描述原子约束之间的强度,再通过约束操作符把复合约束表示成约束表达式。我们定义两个约束操作符"∨"和"∧",用来构建约束表达式。若 A 与 B 为原子约束或者约束表达式,则 A∧B 表示既满足约束 A 也满足约束 B;A∨B 表示满足约束 A 或满足约束 B。如对前述的约束 S,我们可将其分解为 S1 = "平面有向图"; S2 = "边权值为实数"; S3 = "换权值非负",则 S = S1∧S2∧S3。约束表达式(复合约束)的强度即可由原子约束强度根据图 2 的规则推导出来。

- 假设S, S', C为任意约束, Φ 为空约束, 则
  S->Φ
- 若S->S' 或者S=S', 则
  S∧C -> S'
  S->S' ∨C
- 对于其他情况, 约束强度不可判

图 2 约束表达式约束强度推导规则

利用原子约束与约束表达式推导可以潜在地减

少人工描述约束强度的工作。从对复合约束强度的比较转化为对原子约束强度的比较,一方面可降低比较工作的复杂度,减少比较时的出错概率,另一方面可潜在地减少比较次数。对 n 个原子约束,我们最多需要人工对原子约束强度两两做比较,一共需要比较 C_n^2 次(由于非同类原子操作通常不可比较约束强度,故实际需要比较的次数远小于 C_n^2 次),从而可以推导出潜在的 $m = C_n^3 + \cdots + C_n^n$ 个约束之间的强度关系(C_m^2 次比较)。然而,虽然将非原子约束定义成原子约束会潜在地影响人工判定约束强度强弱的工作量,但是它并不会影响推导系统的正确性。

1.3 约束模型兼容性判定规则

由于我们对添加入库的算法组件接口并不做限制,这一方面丰富了算法库中的接口,给程序员提供了灵活的选择,另一方面,不一致的接口设计使在算法级兼容的算法组件变得不可替换。接口的不一致主要体现在参数顺序不一致、参数数目不一致、参数的数据类型不一致。若在判定组件是否兼容时采用严格的判定规则,即一旦发现任何一种不一致性则判定组件不兼容,这会使可兼容的组件数大大减少,从而使我们的算法自动替换方案在支持性能自动调优上的优势大打折扣。

经过分析,对参数顺序不一致的情况与参数数目不一致的某些情况,在约束模型中加入一些说明信息之后,可以弱化我们的判定规则。首先需要的是参数名 I-NAME/O-NAME(见 1.1 节)。相同的名字表示参数蕴含的是相同的概念。由于概念的检查通常不能由编译器完成,而又没有简单的方法让程序员向编译器传递这种信息。我们把这种概念性的信息保存在与算法组件对应的约束模型中,不仅使基于约束的判定可行,而且采用基于名字的匹配可以解决参数顺序不一致带来的不兼容问题。

为了解决参数数目不一致带来的不兼容问题,我们把参数分为三类:强制性参数,可调参数,可忽略参数。强制性参数是指程序员在使用该组件时必须为该参数传递实参,在兼容性判定时必须对该参数进行判定。可调参数通常是那些由于考虑到算法的性能,暴露给程序员,让程序员根据自己的任务设定最优值的那些参数。如 BZIP2 压缩算法可以指定分块大小,从而可以调整压缩率与压缩时间。对于可调参数,我们的组件信息数据库为其保存默认值与可选的其他值,程序员在使用该组件时可选择性地为该参数传递实参(系统支持对可调参数的自动调优),在兼容性判定时可以忽略对该参数进行判定。可忽略参数是指,由于算法本身或实现的特殊需求而存在的参数,对其他算法或实现,并不需要这样的参数(如基于桶的 Dijkstra 算法[18]需要额外指定图中边的最大权值,这个权值指明分配桶的数量)。对于可忽略参数,程序员在使用该组件时必须为该参数传递实参,在兼容性判定时若其他算法组件不含对应的该类参数,则可以忽略对该参数的兼容性进行判定,否则还需进行兼容性判定。需要注意的是,将可调参数或可忽略参数指定为强制性参数并不影响兼容性判定的正确性,但会减少组件兼容集中组件的数目。

利用上述放松的接口兼容性判定规则与约束强度的概念,我们可以给出约束模型的兼容性判定规则,如图 3 所示。

- 假设有如下两个算法组件的约束模型:
  $CPT_1 = (IN_1, OUT_1, ATTR_1)$
  $CPT_2 = (IN_2, OUT_2, ATTR_2)$
  我们用$CPT_1 \subset CPT_2$表示CPT_1兼容于CPT_2,则判定$CPT_1 \subset CPT_2$的规则如下,
- 输入约束的兼容条件:
  (1) $\forall I_{1I}(C) \in IN_1$且$C \in$ {强制性参数},$\exists I_{2J}(C) \in IN_2$且$C \in$ {强制性参数},使$\text{I-NAME}_{1I} \equiv \text{I-NAME}_{2J} \wedge \text{ALGO}_{2J} \rightarrow \text{ALGO}_{1I} \wedge \text{IMP}_{2J} \rightarrow \text{IMP}_{1I}$
  (2) $\neg$ ($\exists I_{2J}(C) \in IN_2$且C$\notin$ {可调参数},$\exists \forall I_{1I}(C) \in IN_1$,使$\text{I-NAME}_{1I} \equiv \text{I-NAME}_{2J} \wedge \text{ALGO}_{2J} \rightarrow \text{ALGO}_{1I} \wedge \text{IMP}_{2J} \rightarrow \text{IMP}_{1I}$)
- 输出约束的兼容条件:
  (1) $\forall \text{O-NAME}_{1I} \in O_{1I}$ ($\in \text{OUT}_1$), IFF $\exists \text{O-NAME}_{2J} \in O_{2J}$ ($\in \text{OUT}_2$),使$\text{O-NAME}_{1I} \equiv \text{O-NAME}_{2J} \wedge \text{ALGO}_{1I} \rightarrow \text{ALGO}_{2J} \wedge \text{IMP}_{1I} \rightarrow \text{IMP}_{2J}$
  (2) $\neg$ ($\exists O_{2J} \in \text{OUT}_2$, $\neg \exists \forall O_{1I}(C) \in \text{OUT}_1$,使$\text{O-NAME}_{1I} \equiv \text{O-NAME}_{2J} \wedge \text{ALGO}_{1I} \rightarrow \text{ALGO}_{2J} \wedge \text{IMP}_{1I} \rightarrow \text{IMP}_{2J}$)
- 算法属性约束的兼容条件:
  若$\text{ATTR}_1 = \{\text{TASK}_1, \text{ALGO}_1, \text{IMP}_1\} \wedge \text{ATTR}_2 = \{\text{TASK}_2, \text{ALGO}_2, \text{IMP}_2\}$
  则需满足$\text{TASK}_1 \equiv \text{TASK}_2 \wedge \text{ALGO}_{1I} \rightarrow \text{ALGO}_{2J} \wedge \text{IMP}_{1I} \rightarrow \text{IMP}_{2J}$
- 算法组件的兼容条件:
  $IN_1 \subset IN_2 \wedge \text{OUT}_1 \subset \text{OUT}_2 \wedge \text{ATTR}_1 \subset \text{TTR}_2 \Rightarrow CPT_1 \subset CPT_2$

图 3 约束模型兼容性判定规则

与程序中待替换的组件在约束模型上兼容的组件并不一定就是可替换的。我们还需考虑待替换的组件对平台的要求在当前编译的平台上是否能满足。我们把这种对平台的需求存放在组件信息表中,而不是约束模型中,因此在编译框架自动替换算法组件时,它一方面需要判定约束模型的兼容性,另

一方面还需要检查平台配置清单,以确定组件对平台的需求在程序待编译的平台上是否满足。平台配置清单一般包括表 1 中的内容。

表 1 平台配置清单

配置类型	配置信息
编程环境	TBB,OpenMP,pthread
加速器	GPU
SIMD	SSE2,MMX
处理器核数	4
…	…

2 编程环境与编译框架

以上我们解决了算法自动替换过程中的核心问题,即如何判定待替换的算法是否与原算法兼容。然而为了使我们的工作实用,还需要解决以下几个问题:算法组件从何而来;组件接口应如何设计以满足程序员的不同需求;如何减少算法组件因参数数据类型不一致导致的不兼容;约束模型如何根据算法组件来建立。

2.1 编程环境

随着软件的大型化,编程再也不是个体行为。如这几年开源社区的兴起,使得重要的大型项目聚集了社会上大量的编程人员。这种基于社区编程的背后隐藏着三类角色:项目发起者与维护者,他们负责项目的系统设计、系统的框架实现、代码与文档的维护;项目参与者,他们根据自己的专长或喜好在系统框架内针对某一模块做开发工作;资源使用者,从开源项目中获取代码或者设计方案。

开源社区无疑是算法组件的重要来源。但是,以各具体项目为中心的任务发布使不同项目中的算法实现接口并不兼容,或者算法的实现并未组件化,而是散落项目中的语句。而算法组件化与保证组件间接口兼容所做的代码改写工作不能完全交由算法组件库的维护者来做。借鉴于社区编程这种开发模式,我们提出了支持算法自动替换的编程范式。它重定义了以上三类角色的工作内容,并规范了各自在编程过程中的行为。该编程范式把涉及整个编程过程的人分为三类:项目发起者与维护者,算法级程序员,应用级程序员。项目发起者与维护者以具体算法实现任务为中心发布项目,他们负责对任务进行分类与描述,对完成同一任务的算法,把它们按约束进行分类;把约束分解成原子约束,并将原约束用约束表达式表示,同时指定原子约束之间的强度关系;定义算法层的输入/输出,并发布具体算法的开发任务。算法级程序员精通某些算法,承接他感兴趣的算法开发任务,在为算法的接口做设计时一方面要遵循算法层的输入输出设计,另一方面可以添加一些实现层相关的输入,如影响性能的可调参数等;为可调参数设置默认值,同时给出可调参数的备选值;为满足应用级程序员的不同需求,算法组件库中应接纳具有不同接口的组件,但是在接口设计之初,算法级程序员需要了解已发布的同种算法的接口设计,以减少复合数据结构因成员命名与成员顺序不一致导致的非必要的接口不兼容问题,他们用注释描述待提交组件的实现层的各种约束。组件提交至组件库后,触发一条维护任务,由维护人员视情况在系统中添加新的实现级的原子约束,描述新原子约束与系统中已有原子约束的强弱关系,并用约束表达式表示组件实现级的约束。系统自动将算法组件的实现级约束与按约束分类的该类算法的算法级约束合并,生成编译框架(见 2.2 节)所需的约束模型。应用级程序员是算法组件的“消费者”,他对要完成的任务做模糊查询,根据系统中任务的分类与描述,选择自己需要的一个具体任务,进入该任务查看按照约束分类的算法,进入符合程序约束的算法集中查看当前约束类中算法组件的接口,系统需要能够显示各种接口下兼容的组件数目,应用级程序员根据组件接口与同一接口下兼容组件数来选择算法组件。允许应用级程序员把采用某个接口实现的组件改为采用另一接口实现的组件,以获得更多的他所需的兼容组件。但这些临时修改的组件并不直接提交至组件库,而是提交至当前用户的临时“兼容组件区”,系统默认该区中所有的组件兼容。尔后由维护人员确定是否要将这些临时组件加入组件库。基于组件实现的应用在编译结束后需要反馈给系统一些信息,如使用了哪些算法组件。系统定期对组件信息库做检查,淘汰一些使用率低的算法组件至备份区,以降低维护人员的维护开销。

为了方便以上的各种交互活动,我们提出了图 4 所示的编程环境。其核心是云开发环境,它内建了算法组件库、组件信息数据库,编译框架、版本管理器等。为了方便各类人员对组件信息进行查询以满足前面提到的各种交互任务,我们为开发环境配备一个搜索引擎,各种搜索结果在全局视图中显示。

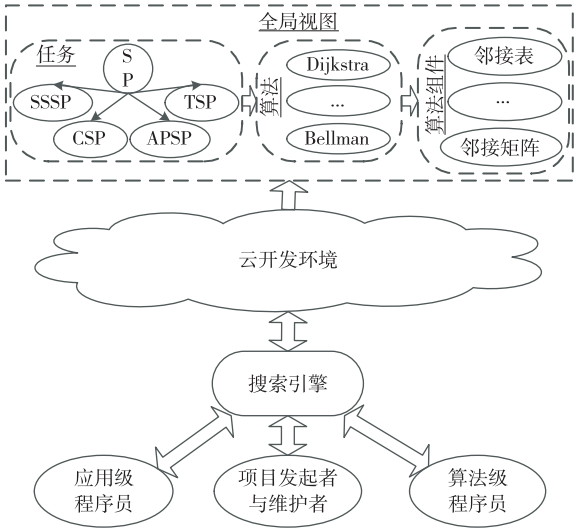

图 4　编程环境

2.2　编译框架

对使用算法组件的程序,可以使用图 5 中的编译框架进行算法的自动替换。下面我们简要介绍其中的各个模块。

我们目前的实现针对 C/C++程序和组件。基于组件的编程并不需要特殊的语法结构支持,因此,最后的程序可以被通用 C/C++编译器处理。我们开发的预处理器(见图 5 虚线框)作为 GCC 的前端负责 4 个方面的工作:(1)分析程序,获取组件引用信息;(2)访问组件信息库,构建兼容组件集;(3)将算法组件的代码添加到用户程序中去构成完整的待编译项目;(4)触发最后的编译。

如果组件兼容集中有多个组件,需要选择符合优化目标的一个,这是图 5 中算法组合空间搜索器

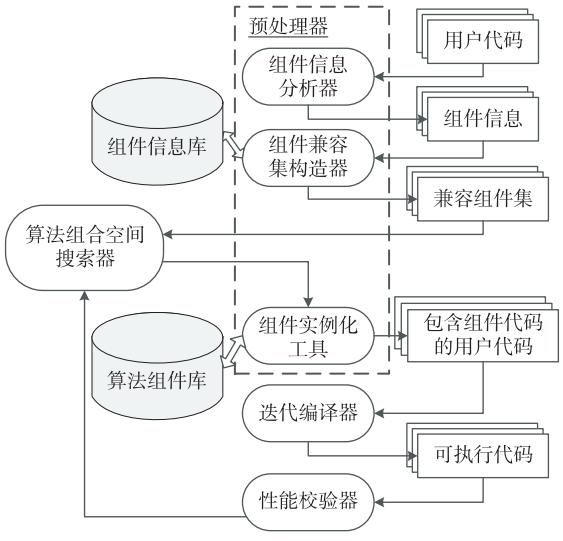

图 5　自动替换算法的编译框架

的工作。搜索的方法类似于迭代编译[14,15]中所使用的方法:基于从候选者中反复随机选择的策略。而在此的候选者不仅包括传统的迭代编译中的优化选项组合,还包括算法和实现。

组件信息库中存储了用于判定组件兼容性与支持组件自动替换的各种表,见图 6。组件信息表中的记录包含组件名和算法名。组件名用于区分同一算法采用不同实现的组件,同时提供算法名是为了方便应用程序员检索;存储位置,指定组件代码在算法组件库中的存储位置;任务名和平台需求信息在算法组件替换时需要。我们分别存储输入、输出和特征约束表达式,因为它们的记录格式并不完全一致,如输入约束表中的记录还需要表示输入参数的分类信息。约束模型表存储用来判断组件兼容性的约束模型。

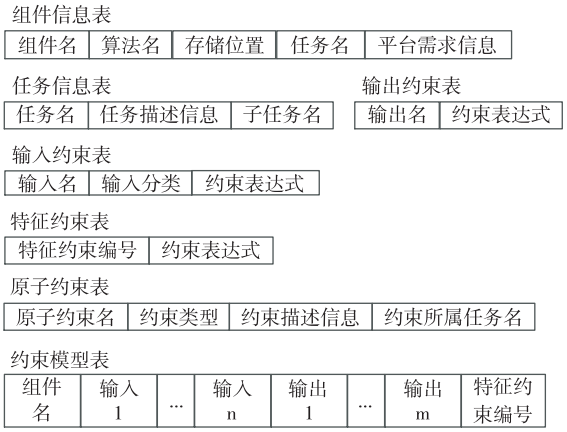

图 6　编译框架的数据库结构

图 5 中的算法组件库与组件信息库一方面与图 4中的搜索引擎一起为用户提供组件信息检索服务,以方便算法级程序员添加组件,应用级程序员使用组件,维护人员维护组件;另一方面作为独立于编译框架的支撑模块,当组件库扩大时,编译框架与组件构造的程序不需做任何修改,即可能使程序通过重编译获得优化。值得一提的是,我们的编译框架能适应于多个算法组件库,而不仅仅是一个。

3　实验及评价

3.1　实验方法

为了使评价更为客观,我们使用的均是真实用例,见表 2。这些用例要么来自于各大基准测试程序,如 SPEC 2006,PARSEC,MiBench;要么来自于为

公众熟知且成熟的应用。

表 2 测试用例

测试程序	代码量	测试输入集	组件化的算法
ASTAR（SPEC 2006）	5842	REF	A * 算法
BSDIFF（Linux 部件）	608	libruby-static. a 的两个版本	前缀排序，BZIP2
BZIP2（SPEC 2006）	8293	REF	Seward's BWT；MTF；RLE；哈夫曼编码
CJPEG（MiBench）	26950	灰度 16 位图像	DCT
DEDUP（PARSEC）	3683	Simlarge	Rabin-Karp；SHA1；Gzip；冗余串识别；
FREQMINE（PARSEC）	2710	Simmedium	Fp-zhu
LIBQUANTUM（SPEC 2006）	4357	REF	Pauli Spin 操作 1；Pauli Spin 操作 2；Not Gate；CnotGate；CCnot Gate
SCOTCH	12040	Ncvxqp5	Heavy-Edge match；FM；Band-FM
VPR（vpr 5.0）	40197	Stdev000	模拟退火算法

我们采用文献[12]中的方法把程序中的算法组件化。由于新的判定组件兼容的方法并不影响接口设计，因此，组件化改动的代码量与组件化导致的性能开销均与文献[12]中的一致，而我们用于替换的算法组件也与文献[12]的相同，详可参看文献[12]，在此不再赘述。

实验在 Intel Xeon E5430 四核处理器上运行，它配备了 12 MB L2 cache，16 GB 内存。编译器是 GCC 4.4.2，操作系统是 RedHatLinux（4.1.2-44）。为每个测试用例，我们搜索了 30 个编译优化组合，见 3.2 节。所有的实验数据均是测试 3 遍后取平均值。

3.2 实验结果及评价

算法组件的自动替换技术把对程序的优化级别提高到算法层次上来，这与传统的编译优化技术相比，具有若干优势。首先，语义上兼容的不同算法在除性能以外的其他指标上也有差异，如内存使用量，算法质量（比如压缩算法的压缩率）等，替换算法可以获得在多指标上的折衷或改进。其次，把串行算法替换成并行算法可以使程序利用上并行硬件资源。最后，程序可以随算法组件库中添入新的算法组件而得到调优，这并不用改写程序或在编译器中添加新的优化技术，而仅需重新编译程序。下面我们用实验来评估该技术的各项优势。

图 7 显示了算法速度与算法质量上的折衷。测试用例中的 SCOTCH 是一个图划分工具，评价图划分的指标有划分速度与划分质量（如划分后切割的边数）。图中用“x”表示的点对应原程序的指标，其他点是通过算法替换后形成的程序对应的指标。由图可见，曲线上的 4 个点是对“x”点的帕累托改进，这意味着通过算法替换构成的新程序有 4 个可以比原程序在算法速度和质量上有改进，而具体选择哪个程序版本，程序员可根据他们的目标而定。

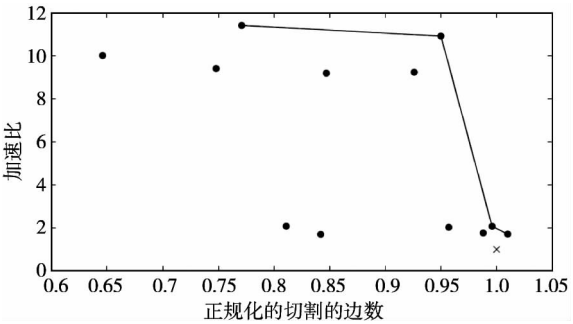

图 7 SCOTCH 中图划分算法在划分速度与划分质量上的折衷

图 8 显示，通过对算法进行替换（图中算法替换），我们的编程环境对遗产代码的性能改善为 1.13 ~ 3.11，平均为 2.07。这远远超过了由迭代编译器（图中迭代编译）自动优化的性能（1.00 ~ 1.53，平均为 1.16）。并且图中所示性能的改进并不以牺牲其他指标上的质量来换取。这个改进要么通过把原来串行的算法替换成并行的算法，如 ASTAR，BZIP2，DEDUP，LIBQUANTUM，VPR；要么替换成采用针对平台的高效实现，如 CJPEG；要么来自

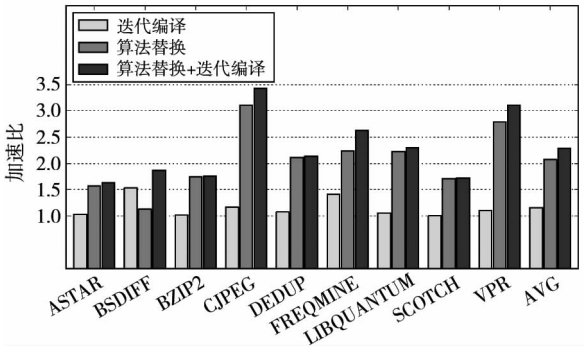

图 8 算法替换与传统迭代编译比较

于对原来算法的改进,如 BSDIFF, FREQMINE, SCOTCH。这说明由编程人员做有限合理的干预(如组件化程序),能极大地改善性能调优结果。另外,由于算法替换与传统的迭代编译具有正交性,两者结合后(图中算法替换 + 迭代编译),性能进一步改善为 1.63 ~ 3.43,平均为 2.29。

算法组件自动替换技术的另一优点是,随着时间的迁移,当有新的算法组件被添加入库时,应用程序仅需重新编译,而不做任何改动即可获得性能提升。我们使用 LIBQUANTUM 这个测试用例演示了性能随着新组件的添加,逐渐改善这一情况(见图 9)。LIBQUANTUM 包含 6 个任务组件,每个组件在我们当前的算法组件库中,有 2 个实现(串行版和并行版),因此一共有 2^6 种算法组件组合。在每个时间步,我们通过随机的往组件库中添加(或不添加)算法组件,来模拟组件库更新的过程。图 9 显示了,每个时间步,由迭代技术搜索所有可获得的组件后获得的程序最优性能。

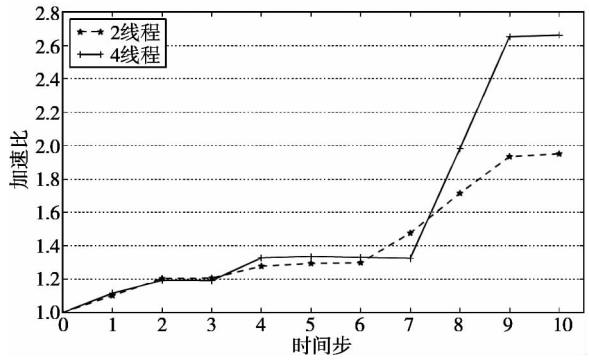

图 9　随着新组件添加入组件库,应用的逐步性能改善图

4　结　论

算法组件的兼容性需要人工来维护,为了降低维护成本与出错概率,我们改进了说明兼容图的方案[12]。为了使通过自动替换算法组件来优化程序这种方案更实用,我们提出了一种规范这一方案中涉及到的不同角色的行为的编程范式,使所涉及的人力工作能够通过各角色的互相合作得到减少和合理的分配。实验结果显示了自动替换算法组件的优化方案相对传统编译优化而言的三大优势。我们的原型编译框架通过结合这两种正交的优化技术,使 9 个测试用例获得平均 2.29 的加速比。

将来的工作包含两方向,一是支持组件接口上采用了兼容(而非相同)的数据结构的组件自动替换。这可以通过把数据结构组件化并描述数据结构组件的兼容性来完成,我们目前的算法组件兼容性模型不需改动。而组件化数据结构所面临的问题会使我们的方案能处理的应用集变小。另一个方向是为算法组件的替换添加运行时系统支持。促使这样做的原因有两个:(1)程序的阶段性行为要求程序在运行的不同阶段通过使用不同的算法使程序性能调优;(2)为了更好地支持应用对平台与输入的自适应性,从而改善程序的可移植性及性能的可移植性,需要我们能够在运行时替换算法组件。

参考文献

[1] Hiral Y, Kurokawa T, Matsuo S, et al. Classification of hash functions suitable for real-life systems. *IEICE Transactions*, 2008, E91-A(1): 64-73

[2] Ansel J, Chan C, Wong Y, et al. PetaBricks: a language and compiler for algorithmic choice. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, 2009. 15-21

[3] Metzger R, Wen Z. Automatic Algorithm Recognition and Replacement. Massachusetts: MIT Press, 2000. 16-18

[4] Thomas N, Tanase G, Tkachyshyn O, et al. A framework for adaptive algorithm selection in STAPL. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, Chicago, IL, USA, 2005. 277-288

[5] Li X, Garzaran M, Padua D. A dynamically tuned sorting library. In: Proceedings of the International Symposium on Code Generation and Optimization, Palo Alto, California, 2004. 111-122

[6] Püschel M, Moura J, Singer B, et al. Spiral: a generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, 2004, 18(1): 21-45

[7] Yu H, Zhang D, Rauchwerger L. An adaptive algorithm selection framework for reduction parallelization. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, 2004. 278-289

[8] Xiong J, Johnson J, Johnson R, et al. SPL: a language and compiler for DSP algorithms. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, Utah, USA, 2001. 298-308

[9] Johnson T, Elgenmann R. Context-sensitive domain-independent algorithm composition and selection. In: Proceed-

ings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Canada, 2006. 181-192

[10] Gilat A. MATLAB: An Introduction with Applications. 4th Edition. Massachusetts: John Wiley & Sons, 2010. 1-384

[11] Cheesman J, Daniels J. UML Components: A Simple Process for Specifying Component-based Software. Boston/San Francisco/New Yok: Addison-Wesley Longman, 2000. 1-166

[12] Li H, He W, Chen Y, et al. SWAP: parallelization through algorithm substitution. *IEEE Micro*, 2012, 32(4): 54-67

[13] Lau K, Wang Z. Software component models. *IEEE Transactions on Software Engineering*, 2007, 33(10): 709-724

[14] Chen Y, Huang Y, Eeckhout L, et al. Evaluating iterative optimization across 1000 datasets. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada 2010. 448-459

[15] Agakov F, Bonilla E, Cavazos J, et al. Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, 2006. 295-305

[16] Schneier B. New Cryptanalytic Results Against SHA-1. http://www. schneier. com/blog/archives/2005/08/new_cryptanalyt. html: Schneier, 2005

[17] Klein P, Mozes S, Weimann O. Shortest paths in directed planar graphs with negative lengths: a linear-space O(n log2 n)-time algorithm. In: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, USA, 2009. 236-245

[18] Dial R. Algorithm 360: shortest path forest with topological ordering. *Com ACM*, 1969, 12(11): 632-633

A programming paradigm and compiler framework for automatic replacement of algorithm components

Li Hengjie[*,**], He Wenting[*,**], Chen Li[*], Liu Lei[*], Wu Chengyong[*]

([*] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**] University of Chinese Academy of Sciences, Beijing 100049)

Abstract

Considering that automatic replacement of algorithm components implies the problem of compatibility checking of the algorithm components, a constraint model for algorithm components is proposed to transfer the compatibility checking of algorithm component into the compatibility checking of constraint models. To solve the compatibility checking, a strategy of "Divide and Conquer", is used. According to it, only the related constraint strength between atomic constraints needs to be checked. The compatibility between constraint models can be deduced from atomic constraints, by making use of the derivate rule of constraints strength and the relaxed rule of compatibility checking for component interfaces. To make it practical, a programming paradigm that normalizes the behaviors of different roles in the development of software is presented. The new paradigm helps to balance the manual work in the first stage of the replacement of the algorithm components between different roles. The proposed prototype compiler framework supports automatic selection of algorithms and compiler optimization options. Using this system, an average speedup of 2.29× is achieved on 9 benchmarks.

Key Words: compatibility checking of algorithm components, programming paradigm, compiler framework, iterative compilation