计算机科学
COMPUTER SCIENCE

基于申威平台寄存器溢出策略的预选先验优化

蔡淳豪, 梁淑萍, 姜军, 邵宁远

引用本文

蔡淳豪, 梁淑萍, 姜军, 邵宁远. 基于申威平台寄存器溢出策略的预选先验优化[J]. 计算机科学, 2025, 52(6): 82-87.

CAI Chunhao, LIANG Shuping, JIANG Jun, SHAO Ningyuan. Pre-selection Optimization for Spill Heuristic on Shenwei Platform [J]. Computer Science, 2025, 52(6): 82-87.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240800128

基于申威平台寄存器溢出策略的预选先验优化

蔡淳豪　梁淑萍　姜　军　邵宁远
无锡先进技术研究院　江苏 无锡 214122
(MYYcch@163.com)

摘　要　在国产多核处理器申威平台上,申威JDK的C2即时编译器通过图着色寄存器分配算法完成寄存器分配工作。即时编译器在分配寄存器时并没有考虑国产处理器的指令特征,导致编译器生成了过多的访存代码,从而无法更全面地发挥国产处理器的性能。为了更充分地发挥申威国产处理器的性能,提出了一种减少图着色寄存器分配时溢出代码的编译优化策略。优化策略基于图着色寄存器分配算法,根据在申威平台上特殊信息的寄存器规律,构造先验模型,并按照先验模型调整溢出策略,从而减少访存代码的生成。该策略在申威JDK上实现了优化,并基于基准测试集SPECjbb2015和SPECjvm2008验证了优化的效果。实验结果表明,优化后SPECjbb2015的max-jOPS和critical-jOPS分数分别提升了4.20%和5.98%,SPECjvm2008的总分数提升了2.02%。

关键词:图着色寄存器分配;访存寻址;溢出代码;编译优化
中图分类号　TP314

Pre-selection Optimization for Spill Heuristic on Shenwei Platform

CAI Chunhao,LIANG Shuping,JIANG Jun and SHAO Ningyuan
Wuxi Institute of Advanced Technology,Wuxi,Jiangsu 214122,China

Abstract　The C2 just-in-time compiler implemented on the SWJDK allocates registers according to the graph coloring register allocation algorithm. The just-in-time compiler ignores the characteristics of the SW processor when allocating registers,which results in excessive memory access codes. In order to get the most out of SW processor,this paper proposes a compilation optimization algorithm. The optimization algorithm is based on the graph coloring register allocation algorithm. And the spill strategy is adjusted based on a priori assumptions about the characteristics of registers representing special information on the SW server. The proposed algorithm has been implemented in SWJDK. The optimization of the algorithm also has been verified based on the benchmark SPECjbb2015 and SPECjvm2008. After optimization,the max-jOPS of SPECjbb2015 increases by 4.20% and critical-jOPS of SPECjbb2015 increases by 5.98%. The SPECjvm2008 increases by 2.02%。

Keywords　Register allocation via graph coloring,Memory addressing,Spill code,Compiler optimization

1　引言

自摩尔定律提出至今,中央处理单元(Central Processing Unit,CPU)的核心变得更快也更多,与此同时,基于存储器的访存速度远远慢于CPU的运算速度。因此对于大多数计算机应用程序来说,其现在以及未来一段时间内的性能的主要限制因素都将是内存访问[1]。在如今追求效率的软件开发时代,考虑到开发效率、可读性与可移植性,大型复杂的软件都需要依赖编译器生成基于特定平台的汇编代码。而在现代编译器结构中,寄存器分配器负责完成平台相关寄存器的分配和指派工作。寄存器分配器会构建一个映射,从符号寄存器映射到物理寄存器和内存位置的某种组合,其通过向代码中插入访存操作来体现这种映射[2]。访存操作的插入会极大地影响计算机应用程序的性能,所以寄存器分配器需要最小化其添加到代码中的被称为溢出代码(Spill Code)的访存指令,从而尽可能发挥硬件CPU的性能[3]。但是,针对申威处理器平台所提供的寄存器规模与访存指令寻址模式,寄存器分配器需要有针对地给出减少溢出代码的逻辑。相较于广泛应用于主力服务器的x86架构以及广泛应用于移动端的arm架构,申威架构的访存寻址模式暂时只支持偏移寻址中的寄存器加16位偏移这一寻址模式,而不支持带移位的寻址、基址变址寻址、多寄存器寻址和堆栈寻址。指令层面的差异,会导致编译器生成代码在平台间存在差异,从而在寄存器分配时产生了不同的场景。

基于以上原因,本文针对国产多核申威处理器平台,提出了一种减少溢出代码的编译优化算法,并在配套的编译器申

到稿日期:2024-08-23　返修日期:2024-10-16
基金项目:科技部重点支持项目(GG20210701)
This work was supported by the Key Project of the Ministry of Science and Technology(GG20210701).
通信作者:姜军(goodsun_jj@163.com)

威 JDK 上实现了在原有分配逻辑上的分配改进，以充分发挥国产多核申威处理器的性能。基于 Java 虚拟机（JVM）基准测试 SPECjbb2015 和 SPECjvm2008，所提算法在两类测试的分数上都得到了一定的提升，验证了其有效性。

2 传统图着色寄存器分配算法的溢出策略

编译器的寄存器分配阶段介于优化阶段和最终汇编代码生成阶段之间。优化阶段产出的中间表示（Intermediate Representation，IR）是在基于目标机器具有无限数量的高速通用 CPU 寄存器的假设情况下编写的[4]。寄存器分配器就需要将 IR 中无限数量的符号寄存器映射到平台相关 CPU 中实际存在的 N 个通用寄存器。为了解耦符号寄存器中的干涉条件，可能需要向程序中添加额外的访存代码，将部分符号寄存器内的信息从寄存器溢出到存储器，然后在需要的地方重新加载它们。这部分访存代码也就是所谓的溢出代码[5]。

寄存器分配主要需要完成前后两部分的工作：第一部分需要根据平台相关中间表示图来构建（Build）待分配寄存器的干涉图；第二部分则需要对干涉图进行着色（Select）甚至重构来完成寄存器分配。算法流程大体如图 1 所示。

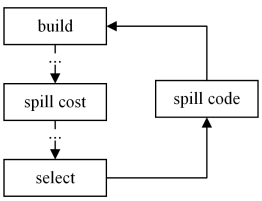


图 1 寄存器分配原有算法流程

Fig. 1 Original algorithm flow for register allocation

先忽略第一部分，在建立干涉图之后，由 Kempe 定理[6]可知，如果图 G 中存在某个小度节点 p，把节点 p 及其关联的边都从图 G 中移除后，可以得到图 G'。当且仅当图 G' 是可着色的，原图才是 N 可着色的。很自然地，可以通过算法减少干涉图的小度节点来简化干涉图，直到整个图被丢弃。然后，空图的节点按删除的相反顺序重新添加，在恢复节点的同时，为节点染上一种颜色，完成寄存器的分配[5]。但是这种着色算法会在图 G' 的所有节点都是大度节点时失效。当着色算法因为所有节点都是大度节点而失效后，必须添加溢出代码并删除节点来修改干涉图，直到可以进行着色完成寄存器分配为止[7]。

基于上述算法逻辑，产生溢出代码的原因是平台相关的中间表示构造的干涉图不能被 N 个寄存器着色。在着色算法固定的情况下，应合理设计溢出代码算法来最小化溢出代码量。最小化溢出代码量的问题与着色问题相同，都是 NP 完全问题。Chaitin 的原始算法在每条定义后添加存储操作，在每条使用前添加读取操作[8]。另有一些更复杂的优化算法还提到了几乎处处溢出[9]和实质化[10]的思想。Kurlander 等[11]随后提出了一种利用空延迟槽暂存溢出代码的算法来减少溢出代码生成。Koseki 等[12]基于此提出了溢出代码移动的部分溢出算法。Bergner 等[13]提出的干扰区溢出算法是一种基于 Chaitin 着色的分割干扰区的溢出算法。这些或基于生命周期的溢出算法，或基于访存周期的溢出算法，都集中在图着色算法的结束部分，缺乏对平台相关特性的考虑。

本文基于图着色寄存器分配算法的逆向分析，提出了基于国产化 CPU 平台的编译优化策略。所提算法通过优化特殊节点和平台相关中间表示的方式减少溢出代码，从而在原有基础上减少应用程序的访存指令，提升了性能。

3 特殊作用寄存器的分配优化

3.1 溢出代价相同时的溢出策略

在 Chaitin 提出的图着色寄存器分配算法中，溢出阶段如果有多个变量都有可能作为溢出的候选，那么寄存器分配器会依据溢出策略选择被溢出的变量。申威 JDK 作为一款即时编译器，具有在程序运行过程中动态收集程序运行概要信息的功能，溢出策略依赖于这些信息较为精准地选择溢出代价最小的变量进行溢出。这里的溢出策略可以粗略地描述为以下两点[14]：

1）虚拟寄存器生命周期在频繁执行路径中作为定义/使用，将提高当前点虚拟寄存器的溢出代价；

2）虚拟寄存器生命周期在一个大区域中，将降低当前点虚拟寄存器的溢出代价。

申威 JDK 即时编译生成的汇编代码中会频繁使用到线程、堆基地址等信息，存放这些特殊信息的寄存器被称为特殊作用寄存器。这些信息通常分别具有各自的信息特性，例如堆基地址在启动 Java 虚拟机后不会被修改，线程信息在线程内被定义后就不会被更改，即在同一线程内的不同方法中该线程信息都是固定的。依据干涉图构建算法，线程信息和堆基地址具有很长的生命周期，这种方法结束都依旧存活的信息，却可能因为不当的溢出策略而在方法内被重复定义。

现有图着色寄存器分配算法在定义虚拟寄存器生命周期时并不会超出当前方法的生命周期，方法间寄存器的使用需要依照方法调用规定来保留存活的寄存器。然而在现实应用程序中，这类跨方法生命周期的信息会因为只在方法内部被定义了生命周期而被低估了溢出代价，从而导致不当的溢出策略带来冗余的溢出和冗余的访存操作。

```
(a) A1 = thread
(b) A2 = disp
(c) A3 = A1 + A2
(d) A4 = OP A3
(e) A5 = OP A4
(f) A6 = OP A1, A5
(g) A7 = OP A6
(h) return OP A4, A7
```

图 2 申威处理器节点匹配完成后的示例代码

Fig. 2 Code after matching of SW processor

通过图 2 所示的案例来说明在申威平台上的这种情况。案例中 thread 表示一个寄存器，寄存器存储线程本地分配缓冲区的地址指针，在 Java 虚拟机中发挥特殊作用；disp 为大于访存指令格式中规定的偏移最大值的常数。通过第（c）步加法运算指令获得一个内存地址，后续根据这个内存地址进行一系列的 OP 操作。针对此类特殊案例，计算出每个虚拟寄存器的生命周期，并根据干涉图构造算法画出如图 3 所示的干涉图。

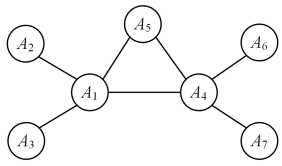


图 3 寄存器干涉图

Fig. 3 Register interference graph of the code

为了简化算法的演示，假设只使用两种颜色对图 3 进行着色。根据图着色算法，首先将度小于 2 的节点 A_2，A_3，A_6，A_7依次移除，只留下度全为 2 的节点 A_1，A_4，A_5构成三阶完全图。此时可以验证无法仅仅使用两种颜色完成对剩余子图的着色，因此需要依据溢出策略选择被溢出的节点。

根据申威平台 JDK 中实现的图着色寄存器分配算法的溢出策略，将选择 A_1作为溢出节点，并将 A_1溢出到栈帧 S_1的偏移处。重写后的代码如图 4 所示。

```
(a) A11 = thread
(b) [S1] = A11
(c) A2 = disp
(d) A3 = A11 + A2
(e) A4 = OP A3
(f) A5 = OP A4
(g) A12 = [S1]
(h) A6 = OP A12, A5
(i) A7 = OP A6
(j) return OP A4, A7
```

图 4 程序重写后的代码

Fig. 4 Code after rewriting

再次构造干涉图(如图 5 所示)，经过移除小度节点的化简后，依旧存在由 A_{12}，A_4，A_5构成的三阶完全图。由于依据申威平台 JDK 中实现的算法，不会再次溢出 A_{11}和 A_{12}，因此再次依据溢出策略选择 A_4作为溢出节点。

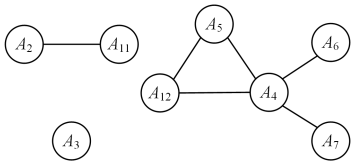


图 5 重写后代码的寄存器干涉图

Fig. 5 Register interference graph of code after rewriting

将 A_4溢出到栈帧 S_2的偏移处后，这段代码能够用两个寄存器 R_1和 R_2完成寄存器分配。简化代码如图 6 所示。

```
(a) R1 = thread
(b) [S1] = R1
(c) R2 = disp
(d) R1 = R1 + R2
(e) R2 = OP R1
(f) [S2] = R2
(g) R2 = OP R2
(h) R1 = [S1]
(i) R1 = OP R1, R2
(j) R2 = OP R1
(k) R1 = [S2]
(l) return OP R1, R2
```

图 6 寄存器分配后的代码

Fig. 6 Code after register allocation

3.2 基于先验假设的代价函数优化

不难发现，对于示例代码，存在如图 7 所示的最优寄存器分配溢出策略。

```
(a) R1 = thread
(b) R2 = disp
(c) R2 = R1 + R2
(d) R2 = OP R2
(e) [S2] = R2
(f) R2 = OP R2
(g) R1 = OP R1, R2
(h) R2 = OP R1
(i) R1 = [S2]
(j) return OP R1, R2
```

图 7 寄存器分配后的代码

Fig. 7 Code after register allocation

对比图 6 与图 7 的代码可以发现：1)对 A_1虚拟寄存器的溢出操作是多余的，这将带来额外的两次访存操作，从而影响应用程序的性能；2)溢出存储线程信息的 A_1虚拟寄存器还可能会给类似垃圾收集读写屏障的地方带来额外的访存操作。

出现这种额外访存的原因在于，寄存器分配算法的溢出策略没有准确地选择出最优的寄存器溢出方案。现有的溢出策略在给定程序的单层循环内计算变量 x 的代价函数近似地表示为：

$$cost(x)=\frac{def_n(x)+use_n(x)}{degree(x)} \tag{1}$$

代入图 2 的示例代码中，因被溢出的寄存器需要在 A_1，A_4，A_5之中，故只需要计算它们的溢出代价，如表 1 所列。

表 1 变量的溢出代价

Table 1 Spill cost of the variables

变量	权重	*degree*	*cost*
A_1	3	4	0.75
A_4	3	4	0.75
A_5	2	2	1

从表 1 中所列的代价函数可以很清晰地看出，变量 A_1和 A_4的溢出代价相同且都为最小。

针对这种已知特殊信息的情况，从寄存器分配的溢出策略角度可以选择提前增加这类信息的溢出代价，使其优先于其他虚拟寄存器的溢出。在代价函数计算部分添加对变量信息的判断，在实施时具有很大难度。由于寄存器分配阶段在指令选择之后，获取线程信息的操作被定义为特殊的 get 节点，因此在实现时只需要匹配到这些特殊的 get 节点就可以定位到这些被用来存放特殊信息的虚拟寄存器。这里给出先验假设：当需要被溢出的虚拟寄存器$\{x_1,x_2,\cdots,x_n\}$的溢出代价相同时，假设可以根据接合后的图标识出存储特殊信息的虚拟寄存器 $x_s\in\{x_1,x_2,\cdots,x_n\}$，那么 x_s 处基于先验的规定使用要求，提前预估出溢出代价值并将其加入溢出代价函数的计算中，以避免类似案例中的无效溢出。在先验假设条件下，变量 x 的损失函数表达式变为：

$$cost(x)=\frac{def_n(x)+use_n(x)+\|x\|_s}{degree(x)} \tag{2}$$

其中：

$$\|x\|_s=\begin{cases}num(G), & x=x_s\\ 0, & x\neq x_s\end{cases} \tag{3}$$

其中，$num(G)$值表示溢出点之后程序块内变量 x 预估被使用的次数，基于输入的程序构造图 G 计算得出。从溢出模型角度看，$num(G)$值表示指令块中溢出点预估的溢出 x_s 的溢出代价。考虑到循环内变量的代价按指数增长，循环内的代价函数应该变为：

$$cost(x)=\frac{\sum_{n}(def_n(x)+use_n(x)+\|x\|_s)\times\lambda^n}{degree(x)} \tag{4}$$

权重系数 $\lambda>1$ 表示溢出内存循环中寄存器相较于溢出外层循环寄存器成倍更高的溢出代价。优化后的算法流程如图 8 所示。

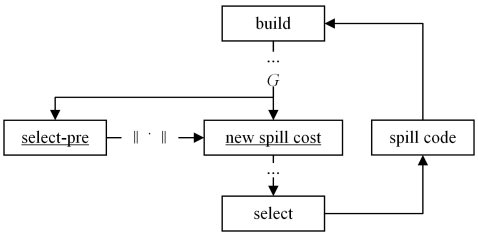


图 8 预选先验优化后的寄存器分配算法流程

Fig. 8 Flowchart of register allocation algorithm optimized by pre-selection

对比原算法（见图 1）和优化后的算法（见图 8）可以看出，经过优化后的算法首先需要预选出存储特殊信息的虚拟寄存器，其次依据输入的图计算预选虚拟寄存器在虚拟点的预估溢出代价，然后用预估的溢出代价重构溢出代价函数，最后用重构的溢出代价函数计算溢出代价。以图 2 初始案例为例，预选出存储线程信息的 A_1 虚拟寄存器，在溢出点之后，A_1 在指令片段中被使用的次数为 2，所以 $num=2$，再根据优化后的代价函数计算新的变量溢出代价，如表 2 所列。

表 2 变量的新溢出代价

Table 2 New spill cost of the variables

变量	权重	*degree*	$\|\cdot\|_s$	*cost*
A_1	3	4	2	1.25
A_4	3	4	0	0.75
A_5	2	2	0	1

由于增加了虚拟变量 A_1 的溢出代价，此时溢出代价最小的为 A_4。溢出 A_4 的寄存器分配方案如图 7 所示。

4 基于申威平台的寄存器分配优化

4.1 国产申威处理器特征

国产申威处理器是一款 RISC 架构的高性能多核处理器，拥有自主知识产权的指令集架构。其在寻址模式设计和指令设计方面都独特性，这些独特性首先体现在申威处理器配套的基础编译工具申威 GCC 上，再进一步传递到 Java 等高级语言编译器上。本文以申威 JDK 作为实验对象，验证编译优化的效果。

根据国产申威处理器的指令手册，访存指令作为存储器指令，格式如图 9 所示。

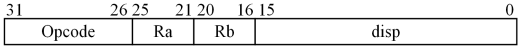


图 9 申威处理器的简单运算指令格式

Fig. 9 Memory instruction format of SW processor

申威访存指令的寻址模式属于偏移寻址类型，操作数的有效地址由指令中的寄存器（Rb）内容与指令的低 16 位偏移量求和而形成。相较于 arm 架构支持的带移位的寻址、基址变址寻址、多寄存器寻址和堆栈寻址，申威需要为类似的访存寻址额外生成对应的运算指令。例如，当需要通过基址变址访存时，需要提前通过加法指令计算出目的地址，再通过访存指令读取数据；当需要通过移位寻址访存时，也需要先通过移位或者移位加指令来计算目的地址再访存。这也是图 2 案例中语句（a）－（c）出现的原因。

针对国产申威处理器的存储器访问指令只能采用偏移寻址方式的设计，指令手册中给出了基于带扩展因子运算指令的加速访存优化，并在申威 GCC 中给出了实现思路和效果验证。申威 JDK 在 C2 即时编译器中通过领域特定语言（Domain-specific Language，DSL）描述了一种符合扩展因子运算指令的新节点，依托自底向上重写的匹配规则实现在即时编译出来的代码中通过扩展因子运算指令加速访存地址计算。

4.2 基于申威平台的寄存器分配先验优化

针对申威平台上通过简单运算计算地址的程序特性，3.2 节中给出的优化算法能够有效规避存储特殊信息的虚拟寄存器被溢出。原因在于，申威平台上用于计算地址的简单运算指令会大量定义并使用虚拟寄存器，这些定义与使用拉高了部分虚拟寄存器的溢出代价，导致基准算法中 A_1 被标记为溢出点。首先，根据 3.2 节中给出的优化算法，在申威 JDK 中构造了表示特殊信息的图节点，用于在接合图中标识存储特殊信息的虚拟寄存器；然后，根据数据流图计算溢出点的额外预估代价并重构代价函数；最后，根据重构的代价函数计算溢出代价，依旧选择溢出代价最低的点进行溢出。由于在优化算法中动态拉高了存储特殊信息虚拟寄存器的溢出代价，这可以在一定程度上规避存储特殊信息的虚拟寄存器被溢出。

优化算法能够初步解决实际测试集中遇到的问题，但是依旧无法评估类似垃圾收集读写屏障这类 JDK 对虚拟寄存器使用带来的额外访存开销。考虑到国产申威处理器具有 32 个整数寄存器的硬件特点，进一步在 3.2 节优化算法基础上人为地将 *num* 值取极限，将这类特殊信息分配到约定为调用者保存的指定物理寄存器内，并将对应物理寄存器排除出寄存器分配池。从寄存器分配角度看，这类特殊信息的生命周期被拉长到了整个申威 JDK 的存活周期，由申威 JDK 主动修改，不再依赖即时编译模块。考虑到申威处理器相对丰富的寄存器个数，这种优化可以有效减少这类信息的寄存器溢出情况。

将图 2 所示代码中的第（d）条语句具化为调用语句，A_4 为调用结束的返回结果，第（f）－（g）两句为计算地址的偏移操作和读取内存操作，如图 10 所示。

```
(a) A1 = thread
(b) A2 = disp
(c) A3 = A1 + A2
(d) A4 = CALL A3
(e) A5 = OP A4
(f) A6 = A1 + A5
(g) A7 = LOAD A6
(h) return A4 + A7
```

图 10 存在调用的示例代码

Fig. 10 Code with call instruction

当涉及调用语句时，寄存器分配器需要将所有生命周期跨越调用语句的虚拟变量溢出到栈上。在这里，只有存储有线程信息的 A_1 寄存器的生命周期跨越了调用语句，于是需要在调用前进行保留，在调用后进行寄存器信息的恢复。此处结合申威平台寄存器现状，允许分配的寄存器个数为 20，分配情况如图 11 所示。

```
(a) R1 = thread
(b) R2 = disp
(c) R2 = R1 + R2
(d) [S1] = R1
(e) R2 = CALL R2
(f) R1 = [S1]
(g) R3 = OP R2
(h) R3 = R1 + R3
(i) R3 = LOAD R3
(j) Return 2 + R3
```

图 11 申威处理器节点匹配完成后的优化代码

Fig. 11 Optimized code after matching of SW processor

很显然，此时无法通过调整损失函数的方式来避免这类情况的溢出。考虑到此类信息并不会被轻易修改，于是将这类特殊信息分配到约定为调用者保存的指定物理寄存器内，同时将这类寄存器定义为只读类型。具体表现为，在 Java 算法调用过程中不需要分析它们的生命周期，但是在使用(Use)对应信息的操作时可以直接从这些寄存器中读取，或作为偏移寻址直接使用这些信息，即可以将图 11 转换为如图 12 所示的形式。

```
(a) R1 = disp
(b) R1 = Rthread + R1
(c) R1 = CALL R1
(d) R2 = OP R1
(e) R2 = Rthread + R2
(f) R2 = LOAD R2
(g) return R1 + R2
```

图 12 优化寄存器分配后的代码

Fig. 12 Code after optimized register allocation

对比图 11 和图 12 可以看出，优化后的代码节约了一步定义操作和一步存一步取操作，能够提供一定的性能优化。另一方面，算法中将特殊信息分配到调用者保存的物理寄存器内，保证了 Java 调用 C 程序时寄存器的正确性。

5 实验结果与分析

5.1 实验环境及测试集

本文针对申威 JDK 编译器的 11.0.15 版本进行了第 4 章中寄存器分配算法的改进，实验测试平台采用国产处理器申威，配备 Linux 操作系统，内核版本设置为 4.19。性能对比测试集采用 Java 虚拟机基准测试 SPECjbb2015 和 SPECjvm2008 测试集。

SPECjbb2015 测试集通过模拟企业级 Java 应用程序，测量在高负载情况下多线程、多核处理器环境中的性能表现。SPECjbb2015 模拟了多用户并发访问，并测量系统在处理大量并发事务时的响应时间、吞吐量和处理能力，以此评估服务器硬件和 Java 虚拟机的性能。SPECjbb2015 测试集在测试结束后将给出两个性能分数 max-jOPS 和 critical-jOPS，以每秒完成的操作数量(Operations Per Second，OPS/s)作为分数单位。max 分数用于衡量软硬件系统在高负载情况下的性能极限。critical 分数用于衡量软硬件系统在响应时间限制情况下的性能极限[14]。

SPECjvm2008 测试集包含 10 个测试模块，如表 3 所列。

表 3 SPECjvm2008 测试用例说明

Table 3 Description of tests in SPECjvm2008

测试用例名称	应用领域
compress	文件压缩
crypto	加解密
derby	数据库逻辑
mpegaudio	音频解码
scimark.large	大数据集运算
scimark.small	小数据集运算
serial	序列化
startup	启动
sunflow	图片渲染
xml	XML 文件验证转换

运算类测试用例与加解密类测试用例有更详细的算法划分，划分后的测试用例在测试后将得分几何平均为模块得分。最后再将模块的分数求几何平均，得出 SPECjvm2008 的总体得分。

5.2 实验结果与分析

针对本文实现的编译优化，对 SPECjvm2008 和 SPECjbb2015 测试集的 Composite 模式进行优化评测，测试选用的垃圾收集器为 JDK11 默认的 G1GC 垃圾收集器。测试结果如图 13 所示。

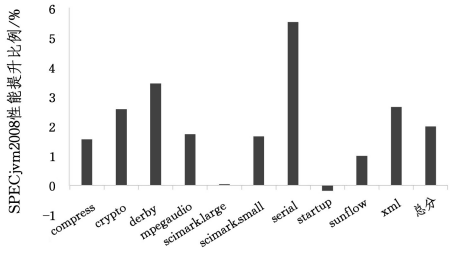


图 13 SPECjvm2008 子测试用例的性能提升比例

Fig. 13 Improvement of sub tests in SPECjvm2008

由图 13 可知，经过算法优化后，swjdk11.Optimized 运行 SEPCjvm2008 测试集的 serial 测试用例有相对明显的提升，提升了 5.61%，所有测试用例的总分提升 2.02%。SPECjbb2015 测试集的 max 分数提升了 4.20%，critical 提升了 5.98%。当选用并行 GC 作为垃圾收集器时，两个测试集的性能变化不明显。

针对 G1GC 性能提升最大的部分，在 C2 编译出来的用于实现 G1GC 写屏障[15]的部分。在多线程运行程序的情况下进行并发标记，需要通过 SATB 写屏障技术记录标记过程中引用修改前的情况。G1GC 通过线程本地队列的方式来减少并发标记时资源的竞争，伪代码如图 14 所示。

```
def satb_write_barrier(field,newobj):
  if $gc_phase==GC_CONCURRENT_MARK:
    oldobj= *field
    if oldobj !=Null:
      enqueue($current_thread.stab_local_queue,oldobj) *field=newobj
```

图 14 SATB 写屏障伪代码

Fig. 14 Pseudo code for SATB writing barrier

SATB写屏障算法细节可以参考文献[15]给出的解释。从伪代码中可以发现，写屏障生效时会频繁使用线程信息来访问线程本地分配缓冲区，因此优化线程信息的寄存器分配能够有效提升应用程序性能。但是对于不常使用的信息，额外占用寄存器资源则会造成性能下降。

结束语 寄存器分配器作为即时编译器中不可或缺的一部分，与微处理器间有着极强的关联性。在将基础寄存器分配算法应用于特定平台的特定编译器时，需要仔细考查平台特性。本文从减少溢出访存的角度，提出对图着色寄存器分配算法的局部优化实现，并在一定程度上提高了申威JDK的性能。然而国产CPU申威处理器与市场通用处理器的微结构差异较大，为了发挥申威处理器的硬件性能，需要更多地从硬件角度思考算法的局限性，并基于此进行算法层面的优化。

参 考 文 献

[1] ULRICH D. What every programmer should know about memory[EB/OL]. https://people. freebsd. org/~lstewart/articles/cpumemory. pdf.

[2] COOPER K D, TORCZON L. Engineering a compiler(2nd ed.)[M]. Beijing: Posts & Telecom Press, 2020: 499-527.

[3] ALFRED V A, MONICA S L, JEFFREY D U. Compilers Principles, Techniques & Tools[M]. Singapore: Pearson Education, 2007: 237-265.

[4] BRIGGS P, COOPER K, TORCZON L. Improvements to Graph Coloring Register Allocation[J]. ACM Transactions on Programming Languages and Systems, 1994, 16(3): 428-455.

[5] CHAITIN G J. Register allocation & spilling via graph coloring[J]. ACM SIGPLAN Notices, 1982, 17(6): 98-101.

[6] KEMPE A B. On the geographical problem of the four colours[J]. American Journal of Mathematics, 1879, 2(3): 193-200.

[7] BRIGGS P, COOPER K D, KENNEDY K, et al. Coloring heuristics for register allocation [C]// Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. New York: ACM, 1989: 275-284.

[8] SPECjbb2015 Benchmark User Guide [EB/OL]. (2017-05-25)[2024-04-14]. https://www. spec. org/jbb2015/docs/userguide. pdf.

[9] BERNSTEIN D, GOLUMBIC M, MANSOUR Y, et al. Spill code minimization techniques for optimizing compliers[J]. ACM SIGPLAN Notices, 1989, 24(7): 258-263.

[10] BRIGGS P, COOPER K D, TORCZON L. Rematerialization [C]// Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. 1992: 311-321.

[11] KURLANDER S M, FISCHER C N. Zero-cost range splitting [C]// Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. 1994: 257-265.

[12] KOSEKI A, KOMATSU H, NAKATANI T. Spill code minimization by spill code motion[C]// 2003 12th International Conference on Parallel Architectures and Compilation Techniques. IEEE, 2003: 125-134.

[13] BERGNER P, DAHL P, ENGEBRETSEN D, et al. Spill code minimization via interference region spilling[J]. ACM SIGPLAN Notices, 1997, 32(5): 287-295.

[14] SPECjvm2008 User's Guide [EB/OL]. (2008-04-16)[2024-04-14]. https://www. spec. org/jvm2008/docs/UserGuide. html.

[15] NARIHIRO N. Tetteikaibou g1gc algorithm hen[M]. Beijing: Posts & Telecom Press, 2016: 155-173.

[16] CHAITIN G. Register allocation and spilling via graph coloring[J]. ACM SIGPLAN Notices, 2004, 39(4): 66-74.

[17] BRIAN G, JOSHUA B, DOUG L. Java concurrency in practice[M]. Beijing: China Machine Press, 2012: 128-135.

[18] CYTRON R, FERRANTE J, ROSEN B K, et al. Efficiently computing static single assignment form and the control dependence graph[J]. ACM Transactions on Programming Languages and Systems, 1991, 13(4): 451-490.

[19] CLICK C. From quads to graphs: An intermediate representation's journey: CRPC-TR93366-S [R]. Technical Report Center for Resesearch on Parallel Computation, Rice University, 1993.

[20] GEORGE L, APPEL A W. Iterated register coalescing [J]. ACM Transactions on Programming Languages and Systems, 1996, 18(3): 300-324.

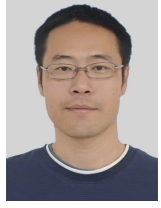
[21] JOHANSSON E, SAGONAS K. Linear scan register allocation in a high-performance erlang compiler[C]// International Symposium on Practical Aspects of Declarative Languages. Berlin: Springer, 2001: 101-119.

[22] PELEGRI-LLOPART E, GRAHAM S L. Optimal code generation for expression trees: an application burs theory[C]// Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1988: 294-308.

[23] PALECZNY M, VICK C, CLICK C. The Java HotSpot™ server compiler[C]// Java™ Virtual Machine Research and Technology Symposium(JVM 01). 2001.

[24] GALINIER P, HAO J K. Hybrid evolutionary algorithms for graph coloring[J]. Journal of Combinatorial Optimization, 1999, 3: 379-397.

CAI Chunhao, born in 1997, master. His main research interests include compilation principle, advanced language virtual machine and JIT compilation.



JIANG Jun, born in 1980, master. His main research interests include compilation design and optimization, architecture oriented performance analysis and optimization, and so on.

（责任编辑：柯颖）