·技术 / TECHNOLOGY·

# 针对程序代码语句级能耗优化方法

黄天明，钱德沛，栾钟治

北京航空航天大学，北京 100191

**摘　要：** 本文主要通过插桩技术对程序的指令进行分析，寻找到程序代码中低效冗余的数据存取语句，对这些语句进行改进，在运行结果不变的前提下，使得在运行程序过程中对于计算机相关部件的使用更加合理和高效，降低机器运行该程序的能耗，起到对程序代码能耗优化的作用。本研究通过Intel Pin[1]工具对于程序代码指令的分析发现，常见程序中存在一种低效冗余的数据存取代码，该代码对于数据多次写入后才读。对这种代码改进前后使用根据计算机部件级硬件性能计数器相关读数建立的模型[2]对于程序能耗进行测量，结果显示对程序能耗降低起到了明显作用，可以作为程序代码能耗优化的一种有效方法，特别是对于运行时间长能耗高的科学计算程序。

**关键词：** 程序代码；能耗优化；数据存取；低效冗余；Intel Pin；科学计算

**doi:** 10.11871/j.issn.1674-9480.2018.01.007

# Energy Consumption Optimization Method for Program Code

**Huang Tianming, Qian Depei, Luan Zhongzhi**

*Beihang Unirersity, Beijing 100191, China*

**Abstract:** This paper mainly analyzes the instructions of the program through the instrumentation technology, finds the inefficient redundant data access statements in the program code, and improves these statements. Under the premise of the running result, when the program is executed, the use of computer-related components is more reasonable and efficient, reducing the energy consumption of the machine running the program, and optimizing the energy consumption of the program code. In this paper, we used the Intel Pin [1] tool to analyze the instructions of programs, we found that there is an inefficient redundant data access code in common programs, the data is read after it was written multiple times. For the improvement of this code, the model based on the computer component level hardware performance

**基金项目：**国家重点研发计划项目 (2017YFB0202202)

counter related readings [2] is used to measure the program energy consumption. The result shows that the programs energy consumption is reduced, which can be used as a program code energy optimization. An effective method, especially for scientific computing programs with long running times and high energy consumption.

**Keywords:** program code; energy optimization; data access; inefficient redundancy; Intel Pin; scientific computing

## 引言

随着计算机和互联网的蓬勃发展，给人类提供越来越多便利的同时其消耗的巨大能源也引起关注。美国能源部门指出一个数据中心的能耗超过一个典型商业建筑的 100 倍之多[3]；同时研究机构表明运行一个 300 瓦的微小型服务器一年的电费以达到 338 美元，排出的二氧化碳多达 1300 千克[4]；并且对于有着庞大集群的数据中心，其年均电量的消耗占了全球电量的 1.3%，而这个比例在 2020 可能会达到 8%[5]，数据中心消耗的电能已经超过许多国家的对电能的消耗[6]。面向可持续发展的低成本、低能耗的新型计算系统、模型和应用的研究， 或称为绿色计算，已成为未来信息技术领域面临的重大挑战。

对绿色计算的研究主要分为硬件层面和软件层面。电能由硬件层消耗，很多研究考虑开发出功耗更低的硬件，在保证系统运行效率的同时降低能耗；然而，运行在硬件之上的软件，作为资源的消费者，如果能够进行合理的管理减少资源的使用，或者提高资源的利用率节能效果更加显著，尤其在集群环境下，合理的管理和调度能够减少数据中心大量的能耗使用，如文献 [7] 所说，采用 Facebook 自主研发的调度技术能够使得其数据中心每年减少大约 16% 的能耗消耗。

计算机系统除去在互联网行业的应用，在科学研究及工业设计制造领域也被广泛用来进行复杂的数值计算，也就是科学计算。科学计算是计算机为解决科学研究和工程中的数学问题进行的计算，具有能够进行无损伤模拟实验，全过程全时空诊断实验以及短时间低成本大量重复实验等优点广泛被数学、物理和生物等领域研究人员采用 [8]。这类计算往往特别复杂及持续较长时间，同时由于程序编写人员往往是科研工作者或者非信息技术方向工程师，编写程序只关注了其功能，而容易忽略效率，所以这类软件的能耗有着很大的优化空间。虽然科学计算所消耗的能源在所有计算机系统中所占比例不高，但如果能降低计算时的能耗对于科学研究及工程制造起到降低成本的作用，同时也能够减少碳排放。

本文针对软件能耗优化中代码优化这一方向进行研究，分析代码中能耗低效代码，提出改进方案，最后使用基于多个硬件部件性能计数器建立的模型验证方法的有效。

## 1　软件能耗优化方法

软件能耗优化并非一个新的研究领域，对于嵌入式软件的能耗优化研究非常成熟，使得这些多是运行在电池供电系统上的软件能够在能源受限的情况下有更长的运行时间 [9]。由于通用计算领域芯片指令集复

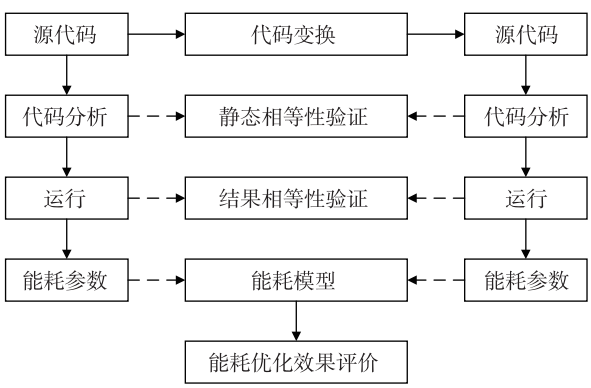


**图1　代码变换步骤图**

**Fig. 1　Code transformation step diagram**

杂，所以对于通用计算平台软件能耗优化的研究还有较大发展空间。

本文针对软件的具体代码，研究其能耗优化的方法，称为面向代码的能耗优化。按照优化的对象，面向代码的能耗优化又可以分为指令级、语句级和模块级三类[10]。指令级优化可采用编译器提供的命令优化代码能耗，粒度小；语句级能耗优化试图采用更高效低能耗的代码结构和数据结构来降低能耗，这一层级贴近程序编写和调试人员，能够对他们进行能耗调试指导，本文研究的层级就是语句级能耗优化；模块级能耗优化是根据上下文选择能耗更低的算法或模块设计方法，来对能耗进行优化。

### 1.1 语句级能耗优化

在程序设计中，程序变换(Program Transformation)是指由现有程序 P 生成新程序 P' 的过程，且 P 和 P' 等价，语句级能耗优化的变换，是在不改变代码功能的前提下，通过改变语句，将能耗较高的代码优化为能耗较低的代码。

对程序代码语句进行能耗优化一般遵循以下三个原则：

(1) 化复杂计算为简单计算，化复杂数据结构为简单数据结构，减少计算能够减少对硬件部件的使用，降低能耗；

(2) 优先使用高性能的存储部件，如缓存优先于内存，内存优先于磁盘，减少数据交换时间，降低能耗

(3) 充分利用 CPU 资源减少等待时间，由于计算机硬件上计算部件和存储部件的速度不匹配，经常出现处理器等待数据的情况，同时由于任务调度问题也会让 CPU 处于空闲状态，造成能源消耗。

本文针对代码语句中数据存储部分一种低效冗余情况进行能耗优化，也就是写入之后没有读的无效写入。

### 1.2 无效写入

实际代码运行过程中，存在一种情况，当对存储器同一个位置连续两次进行成功的写入操作，但中间没有读的时候，或者对某个位置进行一次写入后直到程序终止都没有读取该位置的值，那么第一次写入就没有意义，造成了冗余操作，也就是 DeadSpy[11]。

列表 1 中第 19 行和第二十行产生了一次无效写入，因为第 19 行的 Foo 函数给数组 a 赋值时对内存进行写入，在没有读取它的值的情况下，第 20 行的 Bar 函数再一次给数组a赋值，最后第 21 行的 Foo1 函数才读取了数组 a 的值，所以第 19 行的 Foo 函数里面的对数组 a 的赋值语句便是无效写入。

实际生产中，由于程序编写人员的编写习惯，在缺乏事先对程序整体用到的存取指令完全预测了解的情况下，容易写一些冗余的存取代码。例如在申请某个数组的时候会给其赋予初始值，但是在真正用到时会再一次赋值，这便造成了无效写入。现代编译器在对代码进行优化时针对这方面的冗余存取，显得低效，但这种情况在实际生产代码中并不少见，针对这方面冗余操作进行优化将能够显著提高代码的能耗效率。

## 2 程序代码无效写入定位方法

前文所描述的程序代码语句中无效写入的数据存储情况，使用基于 Intel Pin 的一个对程序运行时上

**列表1 无效写入示例图**

**List1 The example of invalid write**

```
1   #define N (0xfffff)
2   int a[N];
3
4   void Foo() {
5      int i;
6              for(i = 0 ; i < N ; i++) a[i] = 0;
7   }
8   void Bar() {
9              int i;
10             for(i = 0 ; i < N ; i++) a[i] = 0;
11  }
12
13  void Foo1() {
14     int i;
15             for(i = 0 ; i < N ; i++) a[i] = a[i] + 1;
16  }
17
18  int main() {
19             Foo();
20             Bar();
21             Foo1();
22             return 0;
23  }
```

下文及数据属性分析的一个库 CCTlib[12] 开发的客户端工具对程序执行时存取指令进行分析，该库使用 C++ 语言编写，运行在 Linux 平台，用户能够利用这个库开发出自己的程序分析工具。基于该工具给出的 API，结合前文描述的无效写入的低效代码情况，分析出代码中这种语句的存在，进行优化改进，从而实现对程序代码的能耗优化。

### 2.1 Intel Pin

Pin 是一个动态二进制指令插桩工具，它提供了一套丰富的接口给用户使用，用户能够利用这些接口实现对程序不同粒度的分析，如模块级、函数级、调用链和指令级等。其中，调用链 (trace) 是 Pin 使用的一个专用术语，表示一个调用的多个退出方式，例如一个调用链以一个分支处作为起点，可以以函数调用、返回或者跳转作为终点。

使用 Pin，可以对程序执行中每条读写指令进行插桩追踪，为分析程序对内存访问情况提供依据。

### 2.2 CCTLib

CCTLib 是由 Milind Chabbi 等人开发的一款基于 Intel Pin 的库，能够分析出程序执行过程中的调用上下文，并以此建立一棵调用关系树 (calling context tree，CCT)[12]，CCT 的每一个分支代表了一个指令调用序列，而每一个叶子结点代表了一条具体的指令。CCT 能为用户提供对于程序指令插桩分析的 API，如打印指令调用路径、指令执行情况等。利用这些 API，用户可开发客户端工具用来追踪分析程序执行过程中上下文调用情况。以下是 CCTLib 主要几个 API 的介绍。

### 2.3 无效写入判定

使用 Intel Pin 逐条插桩分析程序指令，将内存地址的存取操作记录情况构建一个状态机。状态机状态变换为对于每一个用到的内存地址初始标记为 V (Virgin)，表示没有对其进行存取操作，当进行了存取操作时，根据操作的类型将状态置为 R (Read) 或者 W (Write)。根据对同一地址的存取操作，状态机实现状态转移。出现以下两种情况会判定为无效写入：(1) 当出现连续两次写操作，中间没有读操作时；(2) 在程序结束时，处于W状态的内存地址，即是改地址进行最后一次写操作后，程序直到运行结束也没有对其进行读取。

状态机中 Halt 指令代表该地址知道结束都没有再进行存取操作，Report Dead 行为表示检查出无效写入，可以汇报。由于该状态机从程序开始到结束记录每一个内存存取操作，可以避免出现假阳性或者假阴性的情况，判断结果可靠。

### 2.4 无效写入追踪与记录

文本使用基于 CCTLib 开发客户端工具，该工具使用 Pin 分析追踪每一条程序指令，建立指令调用上下文树 CCT，此时的 CCT 的每一个分支代表了一个指令调用序列，而每一个叶子结点代表了一条“写”指令。程序执行完以后，将每个无效写入以一对 CCT 分支的形式展示给用户。

具体在 CCTLib 上实现为在 Linux 平台上使用影子内存 (shadow memory)[13] 将每个内存位置的状态保存。为了追踪到无效写入，对地址 M 的每条存取指令，都要根据图 2 的状态机更新其状态 STATE (M)，同时保存恢复其上下文调用的指针，在遇到需要报告无效写入时返回句柄。客户端工具维护一个随着程序执行展开的动态增长的调用关系树，这个调用树的每条分支代表一个调用栈。在每个调用 (call) 指令和返回 (return) 指令之前插桩，如果是调用指令，在调用新的节点时则创建新的分支；如果是返回指令，将全局的当前节点作为父节点。

当创建的调用关系树中节点依据图2状态机转移状态时进入到无效写入状态时，将该记录的相关信息以一个三元组的形式记入一个表格 DeadTable，三元组的格式为 <dead context pointer, killing context pointer, frequency>。其中dead context pointer表示给一个地址写入了值，但未读取过的指令的句柄指针，这种指令可能成为无效写入指令的第一次写入；killing context pointer 表示写入了一个 dead context 的地址的指令句柄指针，中间没有读取操作，表示当前

**表1 CCTLib 主要 API 介绍**

**Table 1 Introduction of CCTLib main APIs**

| 函数名 | 参数 | 功能 |
|---|---|---|
| PinCCTLibInit(IsInterestingInsFptr isInterestingIns, FILE* logFile, CCTLibInstrumentInsCallback userCallback, VOID* userCallbackArg, BOOL doDataCentric = false) | IsInterestingInsFptr:给定的预定值，用来限定返回给客户端工具上下文的情况，可选为1. INTERESTING_INS_ALL，客户端工具需要每条指令的调用山下文；2. INTERESTING_INS_MEMORY_ACCESS，客户端工具需要存取指令的调用上下文；3. INTERESTING_INS_NONE，客户端工具只需要函数名称级别的调用关系。<br>logfile:输出文件。<br>userCallback:当指令的isInterestingIns为true时，传递其userCallbackArg值的客户端回调函数。<br>userCallbackArg:提供给userCallback参数回调的指针。 | 调用CCTLib之前初始化。 |
| ContextHandle_t GetContextHandle(THREADID threadId, uint32_t opaqueHandle) | threadId:Pin的线程id。<br>opaqueHandle:句柄通过CCTLib传递给userCallback中的客户端工具。 | 给客户端工具返回上下文句柄 |
| VOID PrintFullCallingContext(ContextHandle_t ctxtHandle) | ctxtHandle:需要打印的句柄 | 打印句柄完整的上下文调用 |

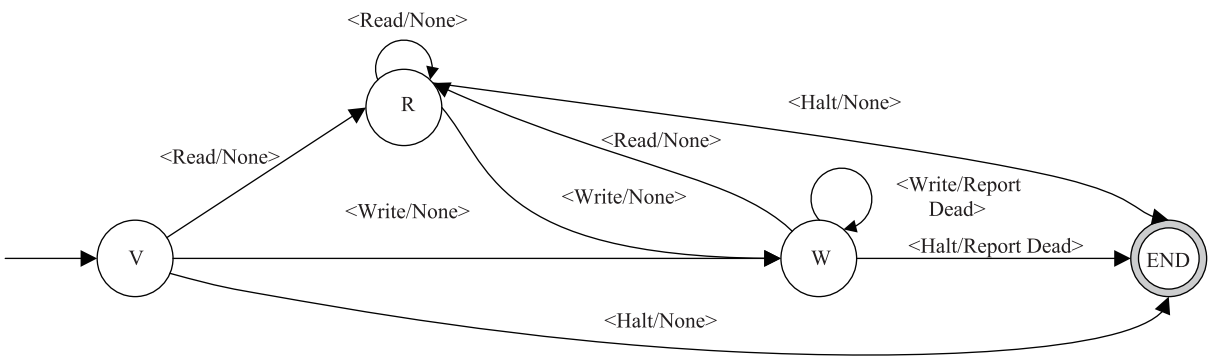


**图2 无效写入状态转移图**

**Fig. 2 Invalid write state transition diagram**

指令已经产生无效写入；frequency 表示无效写入指令对的频率。

当程序运行结束时，将写入 DeadTable 的三元组按照频率逆序排列，同时遍历记录的句柄指针在调用关系树 CCT 中的父节点获得完整的调用链，将该信息呈现给用户，用户根据这些调用链的信息反过来看程序源码，对其进行调试。

## 3 程序代码无效写入优化

在获取到用基于 CCTLib 编写的客户端工具分析出的无效写入指令调用链后，需要这些指令的源代码部分进行调试改进，从而对程序运行能耗起到优化。

### 3.1 优化方法

根据观察分析一些典型的程序，例如 SPEC CPU2006 [14] 中 benchmark 发现，产生无效写入的原因可能不是简单如列表 1 所示那样是程序编写人员的编写习惯不佳，有可能是数据结构选取的不合理以及分支规划的冗余，这些都能在接下来的例子中见到。

像列表 1 所示那中编写冗余，可以选择在不影响程序正常功能的情况下直接删除冗余部分代码；对于编写习惯更好质量更高的工业生产程序代码，产生无效写入的原因更加隐晦，同时由于程序更加复杂，修改起来需要更多的分析。以下是 SPEC CPU2006 中一些 benchmark 的无效写入情况分析和优化方法。

### 3.2 403.gcc 优化举例

对于 403.gcc，经过对各个输入的测试，发现对于输入 c-typeck.i，gcc的无效写入非常多，占到了整个访存量的 73%，针对于输入 c-typeck.i，做以下分

析和优化。

在列表 2 中，展示了在输入为 c-typeck.i 时，gcc 运行时常用到的一个函数 loop.c，函数在列表2部分的功能如下：

(1) 在第 3 行，将给数组 last_set 分配 132KB 的空间，共有 16937 个元素，每个元素占位 8KB。

(2) 5-12 行，遍历传入的参数 loop 中每条指令。

(3) 7-8 行，如果指令匹配一个模式，则调用 count_one_set 函数，该函数的功能是通过最后一条设置虚拟寄存器的指令更新 last_set。

(4) 10-11 行，如果之前的模块完成，那么通过调用 memset 函数来重置整个 last_set 数组，以便在下个循环中使用。

该段代码会产生大量的无效写入，经过分析发现由于程序花费大量时间对 last_set 数组进行重置清零，但在之前用到它的模块中，每次只使用了该数组极少量的元素，但一开始分配就给 last_set 可能用到的最大的数组大小，也就是会有大量的元素在没有被访问的情况下被重复重置清零，也就是重复对其写入 0，所以造成了大量的无效写入。

显然，该程序在该片段对于 last_set 数组的访问很是少量，但是直接给其分配大量的元素单个循环中对其访问的元素极少，在每个循环结束都需要初始化数组的情况下，会造成很多无效的写入。经过抽样发现，对于某个 last_set 数组的元素在连续 34 个 memset 函数调用周期中，被用到的中位数为 2；同时在 99.6% 的情况下每个周期只有 22 个不同的元素会被写入新的值。

于是，一个简单的优化方案为：维护一个大小为 22 个元素的数组，记录 last_set 数组修改过的元素的下标，下次重置清零时只重置数组保存的下标的元素，而不去重置整个 132KB 的数组；如果遇到数组溢出，也就是一个周期中被修改的元素超过 22 个，也就是需要清零的元素超过 22 个，那么周期结束时调用 memset 重置整个数组。

同时，在 403.gcc 的另一个函数 cselib.c 中也存在无效写入的代码，在列表 3 中，该段代码中宏 VARRY_ELT_LIST_INIT 分配数组并初始化为0，之后函数 clear_table 再一次将数组初始化为 0，显然这产生了无效写入。通过阅读源码发现，clear_table 函数还有更轻量级的实现，该实现并不会初始化数组 reg_values，所以此处应是调用了功能更加复杂的接口造成了无效写入。

通过开发的工具对 403.gcc 分析还发现了一些无效写入，不在此一一列举。

### 3.3 Chombo 优化举例

本节将对一个科学计算程序 amrGodunov3d 其中的无效写入进行分析和优化。其中，amrGodunov3d 是 Chombo [15] 中一个标准测试程序，Chombo 是一个使用块结构以及自适应细化网络求解偏微分方程的框架，但由于设计时细节欠考虑，出现了很多低级的无效写入情况。下面将会举 amrGodunov3d 中的无效写入例子进行分析，程序运行时的输入采用程序包提供的 common.input。

**列表2 gcc 无效写入示例 1**

**List2 The invalid write code in 403.gcc(1)**

```
1   void loop_regs_scan (struct loop * loop, …) {
2                   …
3           last set = (rtx *) xcalloc (regs->num, sizeof (rtx));
4           /*扫描loop , 记录寄存器的用途 */
5           for(each instruction in loop) {
6                   …
7                   if(GET_CODE (PATTERN (insn)) == SET || …)
8                           count_one_set (…, last_set, …);
9                   …
10                  if(end of basic block)
11                          memset(last_set, 0, regs->num * sizeof(rtx));
12          }
13          …
14  }
```

**列表3 gcc 无效写入示例 2**

**List3 The invalid write code in 403.gcc(2)**

```
1   void cselib_init (){
2                   …
3           cselib nregs = max reg num () ;
4           /* 将数组reg_values的元素初始化为0 */
5           VARRY_ELT_LIST_INIT (reg_values, cselib_nregs, …);
6           …
7           clear_table (1);
8   }
9
10  void clear_table (int clear_all) {
11          /*将数组reg_values所有的元素初始化为0*/
12          for (int i = 0; i < cselib_nregs; i++)
13                  REG_VALUES (i) = 0;
14                  …
15  }
```

列表 4 所列出的代码片段位于 Chombo-3.2 的示例程序 amrGodunov 下的 PolytropicPhysicsF.ChF 文件中，该段代码位于计算密集型的黎曼解算器内核的三层嵌套循环中，该内核工作在一个存储着 8 字节实数的四维数组上。根据分析无效写入的工具输出显示，第 1-3 行的赋值会被 6-8 行的覆盖，而 6-8 行又会被 12-14 行的覆盖，这显然是对条件分支语句的使用不当造成的。如果对该段代码稍作修改，修改 if-else 的结构，赋值条件不变，顺序从后往前颠倒，即赋值方式为原 12-14 行、原 6-8 和原 1-3 行依次根据条件判断进行赋值操作，通过条件分支跳过不必要的赋值操作，该部分的无效写入即被消除，修改方式如列表 5。

表 4 中给出了在对 amrGodunov3d 进行无效写入优化前后在单节点状态下运行时的能耗的变化。

**列表4　amrGodunov3d 无效写入示例**

**List4　The invalid write code in amrGodunov3d**

```
1     Wgdnv(CHF_IX[i;j;k],WRHO)  = ro  + frac*(rstar - ro)
2     Wgdnv(CHF_IX[i;j;k],inorm) = uno + frac*(ustar - uno)
3     Wgdnv(CHF_IX[i;j;k],WPRES) = po  + frac*(pstar - po)
4
5     if (spout.le.zero) then
6       Wgdnv(CHF_IX[i;j;k],WRHO)  = ro
7       Wgdnv(CHF_IX[i;j;k],inorm) = uno
8       Wgdnv(CHF_IX[i;j;k],WPRES) = po
9     endif
10
11    if (spin.gt.zero) then
12      Wgdnv(CHF_IX[i;j;k],WRHO)  = rstar
13      Wgdnv(CHF_IX[i;j;k],inorm) = ustar
14      Wgdnv(CHF_IX[i;j;k],WPRES) = pstar
15    endif
```

**列表5　amrGodunov3d 无效写入修改方案**

**List5　The modification scheme of the invalid write in List4**

```
1     if (spin.gt.zero) then
2       Wgdnv(CHF_IX[i;j;k],WRHO)  = rstar
3       Wgdnv(CHF_IX[i;j;k],inorm) = ustar
4       Wgdnv(CHF_IX[i;j;k],WPRES) = pstar
5     else if (spout.le.zero) then
6       Wgdnv(CHF_IX[i;j;k],WRHO)  = ro
7       Wgdnv(CHF_IX[i;j;k],inorm) = uno
8       Wgdnv(CHF_IX[i;j;k],WPRES) = po
9     else
10      Wgdnv(CHF_IX[i;j;k],WRHO)  = ro  + frac*(rstar - ro)
11      Wgdnv(CHF_IX[i;j;k],inorm) = uno + frac*(ustar - uno)
12      Wgdnv(CHF_IX[i;j;k],WPRES) = po  + frac*(pstar - po)
13    endif
```

## 4　实验验证

本章实验为了验证前文中提出的无效写入对程序运行能耗的影响，将同一个程序分别在优化前和优化后在相同环境中运行，观察结果，分析无效写入优化对于程序运行能耗的影响。

### 4.1　实验环境

为了控制变量，对比程序将运行在同一主机上，同时主机运行的进程两次测试是保持一致，为了排除气温、主机散热以及电网对于实验准确性的影响，跑完一段程序 10 分钟后跑第二段程序。表 2 和表 3 是对实验主机软硬件的介绍。

### 4.2　实验模型

本文实验部分采用了基于硬件性能计数器采样数值回归建立的模型。硬件性能计数器 [16] 是一组内嵌在处理器上的专用寄存器，计数器统计的与计算机硬件相关的事件，称为性能事件，例如:处理器周期、高速缓存的引用、 高速缓存不命中、分支不命中和总线周期等。

选取硬件性能计数器数值作为建模参数有以下两个原因：一方面，由于程序运行时要与硬件打交道， 性能事件能很好的刻画程序行为特征，例如当程序访存时，通过诸如高速缓存不命中、 TLB 不命中、DMA 事件等性能事件就能了解程序在访存过程

**表2　实验主机硬件**

**Table 2　Hardware of Experimental host**

| 部件 | 描述 |
|---|---|
| CPU | 2.93GHz Intel Core i3 |
| 内存 | 4GB DDR3 1333HZ |
| 磁盘 | Seagate Barracuda 7200.12 |
| 网络 | 1000Mb/s Ethernet |

**表3　实验主机软件**

**Table 3　Software of Experimental host**

| 软件 | 描述 |
|---|---|
| 系统 | Ubuntu 14.04.5 LTS |
| 内核 | Linux 3.13.0-32-generic |
| 编译器 | gcc/g++ 4.8.2 |

中的具体行为；另一方面，性能事件与系统功耗是有直接关系的。性能事件反映了程序的行为特征，因此 通过性能事件就能建立应用程序代码与系统能耗的关系。

通过多元线性回归对选取的工作集 SPEC_CINT2006 数据进行拟合，在性能事件备选集合中添加和删除性能事件，由测量的功耗数据与拟合功耗数据的误差确定最终的模型。在最后确定的模型中，只有 3 个性能事件: 处理器活跃的周期数 (Active Cycles)、引退的指令 (Instruction Retired) 和最后一级缓存不命中 (LLC_Misses)，模型的最终形式

$$P_{system}= 23.834\times Active\ Cycles+2.093\times Instruction\ Retired+72.113\times LLC_{Misses}+47.675$$

其中三个性能事件的含义分别是：

(1) 处理器活跃的周期数(Active Cycles)——这个性能事件的值表示处理器活跃的周期数。当处理器处于空闲状态时，功耗保持一个较低的相对稳定的值，但处于活跃状态时，功耗就会增大很多，所以这个性能事件能够反映处理器功耗的变化。

(2) 引退的指令(Instruction Retired)——离开了“引退单元”的指令(位操作)。引退单元将投机执行的位操作写入用户可见的寄存器，然后不断的检查重排列缓冲区的状态，将已经被执行过并且和任何指令池中的位操作都无关的指令移出。这个性能事件反应了到底有多少指令已经被执行。

(3) 最后一级缓存不命中(LLC_Misses)——记录最后一级高速缓存不命中次数。由于最后一级高速缓存不命中次数和内存的访问次数成比例。因此如果当运行的是 非 I/O 密集型的应用程序时，就可以用这个度量值去估算内存的访问次数。

## 4.3 实验方法

开始实验前，将测试程序备份一份。对一份程序源代码使用无效写入分析工具对其进行分析，使用获取的无效写入信息对源码进行修改，然后分别测试修改前和修改后程序的能耗。

图 3 展示了无效写入分析工具在分析 403.gcc 把 c-typeck.i 作为输入时的分析结果，图中第一行展示了该无效写入的频率以及在整个程序中所占比例，后面两段分别是无效写入的 dead context 和 killing context 即在源码中对应的两处位置的上下文调用链，根据提供的源码位置就可以对其无效写入原因进行分析，继而优化。

在测试程序运行时，使用硬件性能计数器的值作为模型输入，输出计算出的系统功率。获取硬件性能计数器的值的方法是使用性能分析工具 PAPI [17]，将获取到的目标寄存器的值输入到上一节所描述的模型中，计算功率，然后使用贝塞尔 (Bezier) 曲线连接各个采样点，积分求出能耗值。

## 4.4 实验结果

图 4 展示了 SPEC CPU2006-INT 中部分 benchmark 经过无效写入分析工具分析得出的无效写入比例。其中，无效写入比例是无效写入的数据量占程序执行过程中整个写入量的比例。可以看到 403.gcc 这个


**图3 无效写入分析工具输出**

**Fig. 3 The output of invalid write analysis tool**

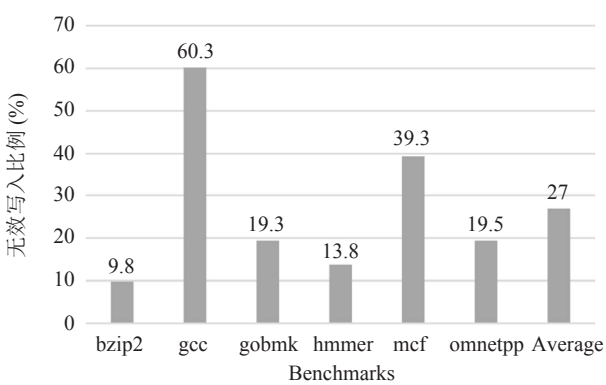


**图4 SPEC CPU2006-INT 部分 benchmark 无效写入分析结果**

**Fig. 4 The result of invalid write in SPEC CPU2006-INT partial benchmarks**

benchmark 的无效写入比例高达 60.3%，401.bzip2 相对低一些，但也有 9.8%，几个 benchmark 的无效写入比例平均值也达到了 27%。下文将展示针对无效写入进行优化前后程序运行能耗结果对比。

主机在不运行测试程序时也是有后台程序在运行，部件也在消耗电能，所以应当将主机未运行测试程序时消耗的电能剔除，才能看到明显的结果对比。表 4 是将主机未运行测试程序时消耗的电能剔除后的结果，剔除方法时将未运行测试程序时主机硬件性能计数器数值读出，使用模型算出长时间的功率平均值，将该平均值作为主机未运行测试程序时的功率，后续根据运行时间将计算电能结果减去该功率下主机的消耗。计算测试程序能耗，可以用以下公式表示：

$$E_{result}=\int_{T_{start}}^{T_{end}} P_1-\overline{P}_2\times T_{end}-T_{start})$$

其中，$E_{result}$ 表示要求的测试程序能耗，$T_{start}$ 和$T_{end}$ 是测试程序运行的起止时间，$P_1$ 表示模型计算出的整体功率，$P_2$ 之前测得主机未运行测试程序时的功率，公式中取用其平均值。

由于同一个 benchmark 在运行不同输入时，调用的函数也不一样，所以针对不同的输入，分别进行优化测试。有些 benchmark，例如 bzip2，由于程序执行时间太短，采样采集硬件计数器数值时间密度相对比较大导致结果误差较大，不作为实验测试对象；而 403.gcc 执行时间较长，比较符合实验条件，针对不同输入对其进行了详细的能耗测量，结果如表 4 所示。

**表4　实验结果**

**Table 4　The result of experiment**

| 程序 | 输入 | 优化前能耗 (J) | 优化后能耗 (J) | 提升 (%) |
|---|---|---|---|---|
| 403.gcc | 166.i | 141.65 | 128.48 | 9.3 |
| | 200.i | 207.34 | 203.2 | 2 |
| | c-typeck.i | 182.37 | 137.69 | 24.5 |
| | cp-decl.i | 133.36 | 115.76 | 13.2 |
| | expr.i | 153.13 | 127.4 | 16.8 |
| | expr2.i | 197.48 | 169.64 | 14.1 |
| | scilab.i | 98.46 | 97.8 | 0.8 |
| | g23.i | 254.07 | 219.26 | 13.7 |
| | s04.i | 227.0 | 166.39 | 26.7 |
| amrGodunov3d | common.input | 154.32 | 147.84 | 4.2 |

其中 gcc 的平均能耗减少达到 13.46%，效果显著，同时，amrGodunov3d 能耗减少达到 4.2%，说明对程序代码中的无效写入进行优化改进能够起到显著的减少程序运行能耗的效果，实现对程序代码能耗的优化。

## 5　结论及未来研究方向

本文介绍了不同粒度下对程序能耗进行优化的方法，重点阐述了程序语句级能耗优化的方法，对程序代码中一种存取冗余情况——无效写入进行了详细的描述和分析得出以下结论：

(1) 程序中无效写入较为常见，出现的原因有多种，例如程序编写人员编写习惯不佳，代码中在循环访问元素个数少、访问元素分布随机的情况下使用大数组，每次重置清零产生大量冗余写入等。

(2) 在使用工具对于目标代码进行分析以后，得到该程序中无效写入的源代码位置，在对于这些无效写入进行优化后，经过测试，发现能够对程序运行所消耗的能耗起到显著的减少。

如前文所说，对于程序能耗优化研究意义重大。本文对于程序代码能耗的优化只是针对于无效写入这一具体存取冗余情景，在未来的研究中应该发现并研究更多程序代码能耗可优化的方向；同时对程序运行时的存取情况分析的复杂性以及对程序代码进行能耗优化时应当保证其正确性，决定了对无效写入这一情况的分析过程的复杂，不利于用户直接使用，后续研究中应当将这一方案向工具化发展。将这一工具作为插件集成到现有的一些集成开发环境中去，使得程序编写人员特别是进行科学计算程序编写的科研人员能够使用简单的操作对编写的程序代码进行能耗优化，起到降低科研及工业成本和减少碳排放的目的。

**参考文献**

[1]　C. K. Luk, R. Cohn, R. Muth, et al. Pin: Building

customized program analysis tools with dynamic instrumentation [J]. Acm Sigplan Notices, 2005, 40(6): 190-200.

[2] S. Yang, Z. Z. Luan, B. Y. Li, et al. Performance Events Based Full System Estimation on Application Power Consumption; proceedings of the IEEE International Conference on High Performance Computing and CommunicationsSydney[C], Australia, December 2016, 2017.

[3] P. M, N. R. O. Energy cost, the key challenge of today's data centers: a power consumption analysis of TPC-C results [J]. Proceedings of the VLDB Endowment, 2008, 1(2): 1299-40.

[4] B. R, R. R. Power and energy management for server systems [J]. Computer, 2004, 37(11): 68-76.

[5] W. Deng, F. Liu, H. Jin, et al. Harnessing renewable energy in cloud datacenters: opportunities and challenges [J]. IEEE Network, 2014, 28(1): 48-55.

[6] C. Ren, D. Wang, B. Urgaonkar, et al. Carbon-aware energy capacity planning for datacenters; proceedings of the Modeling, Analysis & Simulation of Computer and Telecommunication Systems[C], 2012. IEEE Computer Society.

[7] R. Azimi, M. Badiei, X. Zhan, et al. Fast Decentralized Power Capping for Server Clusters; proceedings of the IEEE International Symposium on High PERFORMANCE Computer Architecture[C], 2017. IEEE.

[8] 朱少平. 浅谈科学计算 [J]. 物理, 2009, 8): 545-51.

[9] 罗刚, 郭兵, 沈艳等. 源程序级和算法级嵌入式软件功耗特性的分析与优化方法研究 [J]. 计算机学报, 2009, 09): 1869-75.

[10] 宋杰, 孙宗哲, 李甜甜等. 面向代码的软件能耗优化研究进展 [J]. 计算机学报, 2016, 11): 2270-90.

[11] M. Chabbi, J. Mellorcrummey. DeadSpy: A Tool to Pinpoint Program Inefficiencies [J]. Journal of Software, 2012, 7(4): 1-11.

[12] M. Chabbi. CCTLib,calling context tree library [EB/OL],https://github.com/CCTLib/cctlib

[13] N. Nethercote, J. Seward. How to Shadow Every Byte of Memory Used by a Program; proceedings of the Proceedings of the 3rd International Conference on Virtual Execution Environments[C], New York, NY, USA, 2007. ACM.

[14] J. L. Henning. SPEC CPU2006 Benchmark Descriptions [J]. Acm Sigarch Computer Architecture News, 2006, 34(4): 1-17.

[15] Applied Numerical Algorithms Group, Lawrence Berkeley National Laboratory .Chombo [EB/OL],https://seesar.lbl.gov/anag/chombo

[16] R. Berrendorf, H. Ziegler. PCL—The performance counter library: A common interface to access hardware performance counters on microprocessors [J]. 1998,

[17] P. J. Mucci, S. Browne, C. Deane. PAPI: A Portable Interface to Hardware Performance Counters; proceedings of the Department of Defense Hpcmp Users Group Conference[C], 1999.

收稿日期：2017 年 11 月 1 日

**黄天明：**北京航空航天大学计算机学院中德研究所，硕士研究生，主要研究方向为计算机体系结构。
E-mail: *htming1993@gmail.com*

**钱德沛：**北京航空航天大学计算机学院，教授，主要研究方向为高性能计算机体系结构与实现技术、网格计算、主动网络、计算机网络管理与性能测量等。
E-mail: *depeiq@buaa.edu.cn*

**栾钟治：**北京航空航天大学计算机学院，副教授，主要研究方向为高性能计算机体系结构与实现技术、网格计算、主动网络、计算机网络管理与性能测量等。
E-mail: *rick710055@263.net*