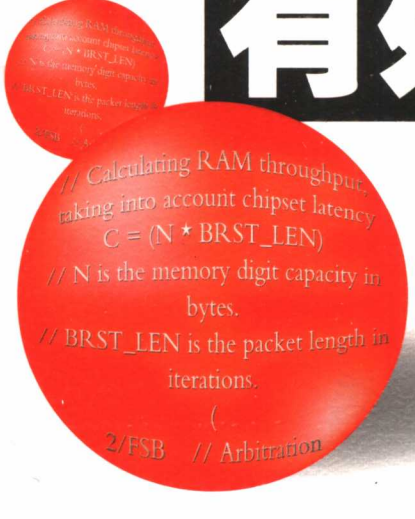
Broadview
www.broadview.com.cn

安全技术大系

透视优化技术　细叙代码实例　量析代码性能　对比优化方式

Code Optimization: Effective Memory Usage

代码优化：有效使用内存



[美] Kris Kaspersky 著
谭明金 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

含光盘1张

安全技术大系

代码优化：有效使用内存

Code Optimization: Effective Memory Usage

[美] Kris Kaspersky 著

谭明金 译

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书系统深入地介绍了各种代码优化编程技术。全书分为4章。第1章集中介绍如何确定程序中消耗CPU时钟最多的热点代码，即有关所谓程序剖分技术，以及典型剖分工具的实用知识。第2、3章分别全面介绍RAM子系统与高速缓存子系统的代码优化知识。第4章主要介绍机器代码优化技术。各章在讨论基本原理的同时，详细给出了典型的代码实例，并对优化性能进行了定量的分析。

该书特别适合于作为应用程序员及系统程序员的学习与开发之用。同时，本书对在硬件方面的专业人员与技术工作者有一定的参考价值。

版权贸易合同登记号　图字：01-2003-8826

图书在版编目（CIP）数据

代码优化：有效使用内存 /（美）凯斯宾单（Kaspersky, K.）著；谭明金译. —北京：电子工业出版社，2004.10
（安全技术大系）
书名原文：Code Optimization: Effective Memory Usage
ISBN 7-121-00351-1

I. 代… II. ①凯… ②谭… III. 代码－程序设计 IV. TP311.11

中国版本图书馆CIP数据核字（2004）第093685号

责任编辑：孙学瑛
印　　刷：北京智力达印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编100036
经　　销：各地新华书店
开　　本：787×980　1/16　印张：21.75　字数：462千字
印　　次：2004年10月第1次印刷
印　　数：4000册　定价：48.00元（含光盘1张）

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

译 者 序

时间就是金钱，效率就是生命。这句名言对于程序，尤其是性能至关重要的程序来说，同样一点都不显过分。如何获取良好的程序运行性能，就是代码优化的研究内容。

代码优化对于许多程序员而言显得有点神秘，这是由于它涉及软件算法与硬件体系结构等方面的深层知识。对于希望掌握代码优化技术的计算机从业人员来说，要揭开代码优化技术神秘面纱的一条捷径是：阅读一本介绍代码优化技术的好书。Kris Kaspersky 撰写的《代码优化：有效使用内存》一书就是这样一本好书。我无意在此为该书做免费的广告（尽管向读者推荐高质量的书籍也是作者或者译者应尽的责任和义务），说它好是由书的内容决定的（当然，是否真好最终要由读者来评判，但我相信读者在仔细阅读并理解了书中的内容之后，定会认为吾言不虚）。

本书系统地介绍了各种代码优化编程知识，其内容涵盖关于如何确定程序中消耗 CPU 时钟最多的热点代码的所谓程序剖分技术与典型剖分工具的实用知识，RAM 子系统与高速缓存子系统的工作原理与代码优化知识，以及机器代码优化技术等。该书在透彻介绍各种优化技术原理时，详细给出了典型的代码实例，对性能进行了定量分析。特别是，全书针对各种优化方式的有利与不利之处进行了细致入微的对比分析，使读者可以轻松地选用满足自己应用需要的优化机制。实用、深入、系统与有效是本书在写作上的一大特色。

当然，本书的质量是由作者在该领域很高的学术造诣、丰富的开发经验及长期的写作经历等多方面的因素来保证的。Kris Kaspersky 是一位技术撰稿人，其文章涉及黑客、反汇编与代码优化技术的各个方面，同时他还是《黑客反汇编揭秘》一书的作者。长期以来，他一直致力于研究与安全和系统程序开发有关的许多课题，这包括编译器开发、优化技术、安全机制研究、实时操作系统内核的创建与反病毒程序的生成等。

本书的翻译得到王璐、沈鑫剡、胡勇强、伍红兵、龙瑞、魏涛、汪东、张斌、李炎新等同志的大力协作与帮助，在此一并致谢。

由于代码优化涉及的内容十分丰富，相关知识的层次比较深，加之译者水平有限，译文中必定存在疏漏和不当之处，请读者不吝赐教。

译 者

前言　优化概述

0.1　优化的好处

现在还值得在优化问题上花时间吗？集中精力学习 MFC 或者.NET 技术不是更好吗？现代计算机的性能不是如此强大甚至连最新操作系统，如 Windows XP 都不能使它慢下来吗？

新一代程序员一直对优化心存疑虑。不过，从一个极端走到另一个极端，并非明智之举。即使最先进的处理器也没有强大到足以应付所有的任务。如果将它们难以完成的任务罗列出来，得到的仍然是一个长长的列表：现实世界中各种各样的物理过程建模、多媒体信息处理、文字识别，等等。哪怕是开发一个高效的数据压缩程序也会面临很多挑战。

处理器的性能的确在快速地提升，然而，PC 用户对处理器性能的需求增长，也是并驾齐驱的。与早些时候人们情愿启动程序后等待它的运行这种数据处理情形不同的是，现在，大多数用户希望所有的操作能够立即完成或者在几分钟内完成，并且数据处理量也大大增加了。目前，文件的大小动不动就超过了几百兆字节。曾记否，整个硬盘容量尚不及现在硬盘容量的十分之一的那个时代？

有什么样的目标就有什么样的手段去达到目标。这种理念决定了本书叙述的方向。因此，读者应将如下的一些建议用于后面的所有优化算法当中去：

1．优化应该尽可能保持与硬件无关，并且对于其他操作系统，是可移植的，而不需要另外的付出或者导致效率的大幅度降低。换句话说，在程序中使用内联汇编语言函数基本上是不可以接受的。最好避免使用那些只有特定编译器才具备的非标准或者扩展语言能力。

2．优化不应该使开发过程（包括测试在内）的劳动强度增加 15%以上。比较理想的情况是，所有关键算法都应该以独立库的形式来实现，因为独立库是不需要予以另外处理的。

3．优化算法应该使程序性能的提高不少于 20%。本书很少去考虑那些使程序性能提高不足 20%的优化方法。全书关注的焦点集中在不需要程序员额外付出就可以使程序性能至少加倍的算法上。尽管似乎显得有点不可信，但这样的算法确实存在！

4．优化应该使代码修改起来毫不费力。许多优化技巧可能会“杀死”程序，因为即使一个细微的修改都可能使优化丧失效能。所有变量都可以在寄存器之间进行仔细的分配，

微代码可以实现并行化，任何能够发挥作用的特性都可以使用。但这些措施并不能弥补代码在灵活性方面出现的损失。这类程序的执行速度甚至连一个时钟周期都不能提高，代码的尺寸也不能哪怕减少一个字节。因此，本书将仅仅讨论那些对更改程序结构所引入的变化并不敏感的优化方法。

这里要试图证实的关键概念是：普遍认定的购买更新与性能更强的处理器比在优化方面花费时间与精力要好的看法是不对的。此外，优化并不像大多数人想像的那样费劲。例如，本书提供的许多通用方法就不需要对每个任务进行单独调整。书中力图描述一些在系统层次具有可移植性的优化技术，从而避免在没有特殊需要的情况下使用汇编语言。当然，绝对不使用汇编语言也是不可能做什么事情的，特别是在讨论剖分（profiling）技术及机器优化算法的章节中更是这样。不过，作者希望贯穿代码的全部注释会使有汇编语言的地方更容易理解，即使对于没有汇编语言编程经验的读者来说也是如此。

0.2　读者对象

本书描述了计算机的各组件之间的交互组织、体系与机制。本书着重从机器代码与数据结构的层次上，讲述了如何进行高效编程，以及各种代码优化技术。

该书是为在 C/C++语言编程方面具有一定经验的应用程序开发者写的，同时也是为具有一些汇编语言知识的系统程序开发人员写的。不过，书中描述的优化方法并不局限于任何一种高级编程语言。因此，具备 C 语言知识仅仅对于理解书中给出的源代码是必要的。

当然，作者也希望本书能够为计算机硬件开发的专业人员与技术工作者带来一些帮助。本书尽可能详细地描述硬件的操作原理，并且讨论在最常用的计算机组件中存在的瓶颈。

书中提供的绝大多数素材都是以作者的经验为基础的。所有信息都经过了仔细的测试与验证。尽管如此，仍然不能保证书中没有错误。（毕竟，“漏洞总是竞相出现的”。）

本书的素材主要适用于 AMD Athlon 以及 Intel Pentium II、Pentium III 与 Pentium 4 微处理器。较早的处理器只是在必要的时候才提到并加以考虑。

0.3　优化基础

程序员（即使是训练有素的程序员）都免不了经常不假思索地去与有问题的汇编语言函数较劲。这不是正确的因应之道！正如第 4 章将要展示的那样，在大多数情况下，机器优化与手工优化之间的差异是可以忽略不计的。除了修饰部分以外，程序通常并没有什么可以优化与提高的。在通常情况下，编译器会给出一个较为理想的结果，因而只做出一点努力并不能使程序的性能提高几个百分点。如果仅仅是在改写一个或者多个汇编语言函数

之后，程序的性能才稍有改善的话，那将是非常不幸的事情。时间与精力都浪费掉了，而得到的只是一个遗憾。

因此，在进行手工优化之前，有必要确定编译器生成的代码是否真的没有经过优化，以及能够进行何种程度的优化。同样，对现有的潜在性能进行评估也很有好处。不过，走另一个极端而认定编译器总是生成优化代码也是不明智的。所有一切取决于实现的处理算法与高级编程语言上下文环境之间的对应关系如何。有些要用一系列 C 或者 Pascal 语言操作符才能实现的任务可以通过单条机器指令来完成。认为编译器会用适当的机器命令取代这组指令，未免显得天真。（与此不同，编译器把每条指令编译成一条或者多条机器指令。）

可见，对代码进行优化处理时，需要考虑一些基本的优化规则。

规则一

在进行代码优化之前，先要有一个同一代码的可靠的、非优化的版本。这意味着，在开始优化代码之前，一定要确保程序能够正确地运行。

在软件开发过程中是不可能生成优化代码的，这样做会实实在在地影响指令的设计。即使仅仅在算法中引入一点点变化，也几乎总会导致对代码进行彻底的修改。因此，在确信用高级编程语言实现的算法能够正确运行之前，切不可启动代码优化过程。并且，如果发现软件的某个地方存在漏洞，那么第一个要怀疑的部分就是经过优化的代码片断。（经过优化的代码基本上不具备可读性，并且不可理解，而调试这样的代码会是一个繁复冗长的过程。）一个没有优化但经过了仔细调试的代码版本会很有帮助：如果用非优化代码取代优化代码而没有产生新的错误，那么意味着软件 bug 存在于优化代码之中。假如这样做之后，错误依然存在，则需要在其他地方去寻找 bug。

规则二

应利用算法优化措施，而不是通过提升系统特性来获取最大限度的性能提升。没有什么优化措施可以极大地提高诸如冒泡排序或者线性搜索之类的算法的效率。正确的指令规划及其他的程序设计技巧最多能将程序的性能提高几个百分点。与此不同，实现一个快速排序和折半查找算法会至少将程序的速度提高 10 倍，而不管源代码写得粗糙与否。因此，如果程序运行速度显得太慢，应该尽量使用更加高效的算法，而不是去优化一个效率低下的算法。

规则三

不要将代码优化与汇编语言实现混为一谈。如果在剖分阶段检测到了程序中的热点（hotspot），并不要急于将它们用汇编语言进行重写。首先要做的是尽量在高级程序设计语言中采取各种可能的步骤：删除消耗资源的算术运算（特别是除法与余数计算）；最大限度

地减少分支的数目；用数目更小的迭代操作（译者注：循环语言结构中重复执行的循环体部分称为迭代操作，这部分代码执行一次成为一次迭代）替换循环，等等。在极端情况下，试着换用别的编译器。（不同编译器的代码编译质量往往存在很大的差异。）

规则四

在试图用汇编语言重写程序之前，先查看一下编译器生成的汇编代码并估算它的效率。也许，程序性能差不是编译器的责任，可能是由处理器或者内存子系统引起的。对于要进行大量计算的科技应用程序和需要海量内存的图形软件来说，情况尤其如此。简单地将程序转换为汇编语言，并不能提高内存子系统的吞吐量或者使处理器的运算执行得更快。首先，通过编译器生成代码的汇编语言列表（例如，对于 Microsoft Visual C++编译器来说，使用/FA 命令行选项就可以达到目的）。然后，扫描列表，找出其中诸如 `MOV EAX,[EBX]\MOV [EBX], EAX` 之类的严重错误。原则上，改善由编译器生成的代码的性能将比从零开始用汇编语言实现要容易得多。因为前者需要的时间要少得多，而得到的结果却可能相差无几。

规则五

如果编译器生成的汇编语言列表虽然显得很不错，但是程序运行起来仍然很慢，就可以考虑将它加载到一个反汇编工具之中去。在这里可能发现很多东西，比如，编译器对跳转指令的存放位置的定位是不准确的，从而可能没有合适地定位跳转指令。通过其倍数为 16 的地址来对齐跳转指令可以得到最好的性能。将整个循环体放在一个高速缓存行（cache line）（也就是 32 个字节）中，甚至可以得到更好的结果。不过，这说得太远了点。机器代码优化手段是另外一个不同的主题。我能够给出的最好建议是，不妨去仔细研究一下由顶尖处理器制造商（Intel 与 AMD）发布的文档资料。

规则六

如果可用的处理器指令能够实现比较高效的算法，那么就用不着去管编译器而可以着手去实现汇编语言代码。不过，这种情况是很少见的。再说，我们不是生活在荒岛之上。经过了仔细调试并且性能很好的可用高度优化代码库文件，不下几吨重。如果能够买到一个库文件，那么自己为什么还要去重新开发呢？

规则七

在开发汇编语言代码时，不管存在什么样的干扰，都应该给出一个精巧而高效的方案。

是的，确实存在一些没有公布的性能、不寻常的样式及其他的诀窍。然而，这些方法的任何一种都不是可移植的、与处理问题无关的或者每个人都清楚的。（老实说，谁能够轻而易举地弄明白自己十年前编写的源代码呢？）我曾经不止一次因为使用自己以往用过的诀窍而搬起石头砸自己的脚。其中，最令人痛苦的事情是，用到的这些诀窍通常并不是必需的；使用它们纯粹是因为要“露一手”。可见，不要忽视对程序进行注释，并且应该将所有的汇编函数存放在一个单独的模块之中。在程序代码中要避免使用内联汇编函数：这些函数很少是可移植的，并且在移植到其他平台或者编译器以后经常会带来负面效果。

汇编语言技巧仅仅用在软件版权保护方面才是合理的。不过，这又是另外一个显得非常不同的讨论话题。

0.4 常见的谬误

关于优化方面的错误说法为数不少。虽然专业人员大多会对此一笑了之，但是这些说法会对新手造成不利的影响。本节将揭示其中的几个讹传。

讹传 1

编译器会替用户完成所有的优化任务。

认为编译器具有超凡脱俗的能力是毫无根据的。即使是一个具有强大的优化能力的编译器，也仅仅能够对设计良好的代码进行高效的转化。如果源代码残缺不全，没有什么编译器可以更正它的错误。可见，不应该将所有与性能和效率有关的任务都交给编译器去做！一个比较好的做法是协助编译器去完成这些任务。达到此种目的的具体方法需要单独加以讨论，这方面的内容可以在第 4 章中找到。

讹传 2

最大限度的效率只有在单纯使用汇编语言编程时才可以得到，用高级语言编程是不可能得到这样的结果的。

将程序代码改为汇编指令，很少可以提高程序的性能。用手工优化方式转化高质量的源代码所得到的结果，在性能上只比由优化编译器给出的结果好 10%~20%。这个差异虽然是很大的，但是它并没有大到足以消除单纯使用汇编语言进行程序设计所带来的困难与工作强度。

第 4 章提供了这个方面的详细内容，其中包括机器优化与手工优化质量的比较。

讹传 3

与优化编译器不同，人不可能利用处理器体系方面的所有性能。

仅仅只有 Intel 开发的编译器才能够生成处理器微体系结构认为具有优化性质的代码。第 1 章（见“VTune 实用剖分知识”一节）将给出这方面的实际例子。

尽管如此，最新的处理器其实能够优化被传递过来要它进行处理的代码。进行手工优化的企图容易遭到失败：没有什么代码对于现有的处理器体系结构来说是优化的，诸如 PII、P4、AMD-K6 与 Athlon 之类的处理器所具有的特性相互之间存在迥然的不同。

这方面的例外情形体现在对性能具有苛刻要求的小范围内的特定任务（诸如破译口令字）上。在这些例外当中，手工优化被证明是行之有效的，由此得到的结果比任何编译器都好。

讹传 4

x86 处理器不值得使用，要理解什么是真正的性能就必须使用 PowerPC。

每种处理器体系结构都有自己的优点与不足。我虽然没有优化过要在 PowerPC 上运行的任何代码，但我有几个熟人在从事用于 PowerPC 的优化编译器的开发，他们对其中的一些“特性”感到很失望。

x86 处理器系列同样具有许多限制与问题。不过，这不能成为程序员编写粗糙的代码而不去改进它的借口。

到目前为止，该处理器系列拥有最为复杂的指令系统之一，它为系统程序员提供了几乎不受限制的编程能力。应用程序员甚至想像不出编译器从他们那里剽窃到了哪些经验！

目　录

CHAPTER 1

第 1 章　程序剖分

本书通篇使用剖分（profiling）一词来描述对整个程序及其特定片断的性能进行分析以检测出热点（hotspots）的过程。这里的热点指的是程序中需要最长执行时间的代码部分。

根据著名的“90-90 规则”[1]，程序 10%的代码量通常要消耗多达 90%的系统资源。如果将执行每条指令所需要的时间用相对于指令线性地址递增图表的形式表现出来，那么得到的图表将给出几个明显的尖峰，这些尖峰是从包含大量低“山”的平原上凸现出来的（参见图 1.6）。这些尖峰就是所谓的热点。

为什么不同程序段的图形线结构会如此不一样呢？这是因为大多数数值“辗磨”算法都是以循环形式（也就是多次重复执行同一代码片段）进行组织的。通常，这些循环并不是按顺序进行处理的，而是形成一种存在多级嵌套的层状结构。这样一来，具有最深嵌套层次的循环将占用程序执行时间中狮王级的份额。因此，对这些循环进行优化可以使性能得到最大程度的提升。

需要注意的是，对代码量大、执行慢、但很少调用的函数进行优化实际上是没有意义的，由此得到的性能提升，完全可以忽略不计（除非这些函数的代码可能写得太粗糙）。

如果程序实现了一个简单算法并且源代码不超过几百行，那么通过肉眼扫视代码列表就可以容易地找出热点部分。不过，随着源代码尺寸的增大，寻找热点的任务将变得更加复杂。正在进行分析的程序很可能包含几千个错综复杂的函数，这些函数相互作用的方式

1　开头 90%的代码用去开发时间的第一个 90%，剩下 10%的代码消耗开发时间的另外一个 90%。这条诙谐的格言是由贝尔实验室的 Tom Cargill 提出来的。1985 年秋，Jon Bentley 在《ACM 通信》的“计算机科学的减震器”专栏中提及它，因而流传开来。在该栏目中，这条规则被称为“可信规则”。这个称谓似乎没有什么吸引力

也十分复杂。（其中，一些函数很可能由外部的库函数或者操作系统的 API 进行调用。）在分析这种程序时，很难确定究竟是其中哪个函数致使程序的性能很差。要是这样，就只好去使用专用软件工具来解套了。

剖分软件（profiler）是用于进行程序优化的主要工具。“盲目的”优化很少能够产生好的结果。有一条谚语是这样说的：“飞行编队的步伐由最慢的飞机决定”。软件代码的行为与此类似：应用程序的性能取决于程序的瓶颈。这样的情形时常出现在单条机器指令上（例如，在具有深层嵌套的循环中反复执行的除法指令）。程序员可能在改善其他代码片段方面花费了大量的精力，但是蓦然发现他或者她所有的付出仅仅换来应用程序的性能提升 10% 或者 15%。这会使程序员大吃一惊。

这样就可以得出第一条优化规则：**消除并不“最热”的热点实际上不能使应用程序的性能得到实质性的提升**。重返前面给出的那个比喻，提高倒数第二慢的那架飞机的速度基本上不能从总体上提升整个编队的飞行速度。这架飞机的步伐是否会影响最慢的飞机的速度是另外一个需要讨论的话题。由于需要全面的剖分技术知识，本书不再对该主题进行讨论。

1.1 剖分的目标与目的

剖分的主要目标是**从总体上考察标定应用程序的运行时性能**。根据详细程度的不同，程序“**点**”（spot）一词可以用来指称单条机器指令或者用高级程序设计语言写的整个代码片段。（例如，它可以是一个函数、一个循环或者单行源代码。）

大多数先进的剖分软件支持如下的基本操作集合：

- 确定每个程序点的整个执行时间（程序点总计时）
- 确定每个程序点的执行时间类型（程序点计时）
- 确定隐藏在后面的原因，以及冲突或者损失源（损失信息）
- 确定每个程序点的调用数目（计数）
- 章确定程序覆盖范围（范围）

1.1.1 总执行时间

收集应用程序执行每个点所需时间的过程，可以检测到该应用程序中“最热”的部分。注意一点很重要，即直接的测量会显示整个程序执行时间的 99.99%被 `main` 函数消耗掉了。然而，很显然 `main` 函数并不是热点。真正的热点是 `main` 函数调用的函数！为了避免在程序员中造成混淆，剖分软件通常从程序里每个函数的总执行时间中减去执行子函数所需要的时间。

作为例子，现在来看作为 Microsoft Visual C++编译器的一部分而提供的 profile.exe 剖

分软件所生成的结果（如表 1.1 所列）。

表 1.1　Visual C++提供的 profile.exe 工具所生成的剖分结果

Func time	%	Func+Child time	%	Hit count	Function
350 192	95.9	360 982	98.9	10000	_do_pswd (pswd_x.obj)
5 700	1.6	5 700	1.6	10000	_CalculatedCRC (pswd_x.obj)
5 090	1.4	10 790	3.0	10000	_CheckCRC (pswd_x.obj)
2 841	0.8	363 824	99.6	1	_gen_pswd (pswd_x.obj)
1 226	0.3	365 148	100.0	1	_main (pswd_x.obj)
98	0.0	0 098	0.0	1	_print_dot (pswd_x.obj)

第二列（Func+Child Time）列举了每个函数的总执行时间。占用时间最多的是 main 函数（像预料的那样）。占据第二位的是 gen_pswd 函数（99.6%），紧随其后的分别是 do_pswd 函数（98.9%）与 CheckCRC 函数（3.0%）。仅仅占总时间微不足道的 1.6%的 CalculateCRC 函数，初看起来似乎显得不怎么重要。因此，只要快速地浏览一下收集起来的数据就可以看到这样的三个热点：main 函数、gen_pswd 函数与 do_pswd 函数（如图 1.1 所示）。

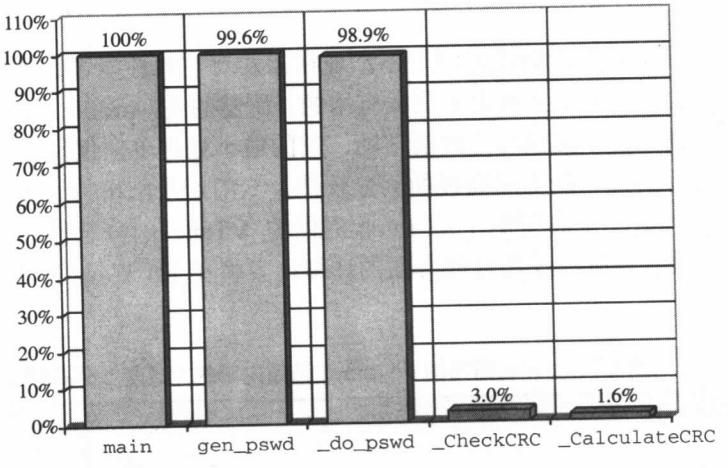


图 1.1　体现每个函数的执行时间的图。初看起来，似乎有三个热点，但实际上不是这样

实际上，应该立即将 main 函数从热点中排除掉。很显然，该函数不能为性能降低承担责任。gen_pswd 函数与 do_pswd 函数则是两个需要加以考虑的函数。如果这是两个独立的函数，那么它们就表示两个热点。不过，这里的情况并不如此。假如从父函数（gen_pswd）的总执行时间中减去子函数（do_pswd）的总执行时间，那么得到的父函数时间值是 0.8%——比总执行时间的 1%还小。