HOPE

HOPE COMPUTER COMPANY LTD.

计算机系统程序员问题详解

李 伟 编译

北京希望电脑公司

计算机系统程序员问题详解

李 伟 编译

北京希望电脑公司

一九九一年五月

前　言

近年来，计算机科学的发展已形成了数学化证明程序正确性的技术．然而，这些技术是相当耗费时间的，且对大多数应用而言非常难于掌握．即使如此，它仍被用来为某些安全操作系统开发安全性核心（中央机构）．

在这些技术尚未成熟之前，大多数程序员仍需以相同的方式调试程序．幸运的是，我们现在可以比以前工作得快了．

在谈及操作系统程序设计时，我们始终认为，其工作颇具艺术性．因为我们总是在设法提高其性能，而不是简单地去完成一个程序设计任务．本书的主要对象就是那些系统程序员以及那些分析系统程序的应用程序员．书中对调试、操作过程与工具、问题的预防、编程技巧和规范以及测试与管理等系统程序员经常遇到的问题进行了详细地讨论并给出了可行的解法方法与实例．

本书的组织如下：

第一章"调试"讨论了解决当前系统问题的方法与技术．第二章"过程和工具"介绍了可使调试更容易的产品和标准的实践过程．第三章"预防"和第四章"编程训练"详细说明了在程序开发工作中尽量减少错误的技术．第五章"保持符号化"将有助于程序员避免编写过分专业化的汇编语言代码．第六章"测试"给出了检测程序错误的方法．第七章"管理"讨论了一些技术方面的问题．最后，在附录中给出了系统程序员要经常用到的一些技术信息．

本书所讨论的内容颇为实用，但也涉及到许多编程领域内的高深知识．因此，对编译者来说，一定会在书中出现不少不妥当之处．还有赖于诸多同行的热情批评与指正．

最后，衷心感谢北京希望电脑公司对本书问世所给予的大力支持和热心帮助．感谢所有为本书做过工作的同志！

目　录

第一章　调试

解决一个系统问题的目的就是从发生的故障或不希望得到的结果中找到隐藏的而往往是隐藏很深的原因。大多数故障可以根据结果马上找到。一个很好的示例是当运行一个新汇编的程序时发生了一个指令中断。汇编程序在汇编它不认识的源指令时将产生一串零，而零是非法的操作码。请注意其结果，一个结果不同的指令中断使用了不正确的汇编语言语法。

虽然原因与结果之间的不同将多次出现，这个例子并不算太复杂，因为其结果可以使我们马上找到原因。在更困难的问题中，原因和结果不论在时间还是内存位置上都会离得很远。

如果说任何可能发生的问题都可以改正，这只能是一种轻率的说法。在陷入困境之前，最好说这种改正需要对涉及到的全部函数都进行重新设计。但是如果是已知的一系列条件带来的故障，那么它就可以被诊断。如果我们知道是特定的一系列行动产生的故障，那么就可以采用直接技术进行诊断。

最坏的情况是一些不可重现的故障。这些故障可以预知但它们只发生一些很难出现的环境下，例如它们需要很高或特殊的系统调入。如果不能重现这种故障，那么就必须收集相似的情况来获取信息。因为不能在我们控制的环境重现故障，那么诊断的花费就会很高，解决问题也就需要很长的时间。

本章讨论的方法将提高用户收集程序故障信息的能力，减轻解除故障的难度。

§1.1　有错的数据

覆盖(overlay)(正确的数据中混有错误的数据)是一种引起故障的常见原因。程序错误的使用了一个指针，把一些特定的数据放入错误的控制块的某一位置。(幸运的话，插入的数据是一种可识别的模式，可能可以表明其来源，不幸的是这种幸运是不可靠的:覆盖可能就是一串零)。

在对一个调用控制块的子程序进行编码时，一定要使用一个基寄存器(在一台 370 上，基寄存器中零指是内存中的前 4096 个字节，这里覆盖的后果非常严重)，而且一定要使用正确。

§1.1.1　错误控制块

当用户认为指向一个长控制块，但却错误调用了一个短控制块时，就会覆盖下一个(不是当前的)控制块。成组的控制块在内存中是连续存放的，所以覆盖不会被立即发现。这一段延迟使这一错误变得很严重。

§1.1.2 太长的数据段

指向正确的块时长度问题也可能带来麻烦。一个缓冲区通常有一个固定格式的头。这里含有控制信息。缓冲区的正文段跟在后面。如果在缓冲区中放入了太多的数据，下一缓冲区的头包括其中的控制信息将被破坏掉。这种覆盖通常直到下一次使用这些信息时才被发现。

§1.1.3 有错的代码

有时覆盖破坏的是代码而不是数据。当程序操作与正常逻辑不符时，把内存中的代码与程序中的比较一下。在这里覆盖的发生时间与故障的发生时间隔得也很长。不要忘记检查一下字符或汇编了的数据是否遭到了破坏。

§1.1.4 一种预防技术

有一种简单的技术可以帮助进行复杂的内存覆盖的查找任务。这一技术包含在控制块中加上一个 EBCDIC 标记(eyecatcher)段。之所以如此命名是因为其最初的用途是在转储中做为此块的标记。其另外的作用是增大识别出引起覆盖的程序的可能性。当一个子程序把一个控制块传递给另一子程序时，接收者将把标记段的值与其希望的值作比较，通常是控制块的缩略语。如果其值不相同，子程序将停止运行，防止发生可能的覆盖。创建的转储通常含有足够的信息来找到引起故障的程序。没有这些检错码的话，通常引起覆盖的程序和后来试图使用已有错的数据的程序离得很远。这种原因与结果间的很大的差距使发现问题所在非常困难。

因为标记块的使用已变得很普遍，用户可以对其进行检查而不用它来是操作系统或个人编码。标记段通常位于控制块开始，但这种技术并不依赖于它的位置。因为只对其值的相等性进行比较，所以标记段在不同块中的位置可以不同。控制块通常用四字节为单位存储三字节长的地址，高位字节为 0，这种地址不匹配合法的 EBCDIC 字符串。有人可能认为每 4 字节含有 31 位的地址的结构是一种合法的地址，可以减少这种方法的副作用。确实如此，但这还不够。在 31 位地址的环境中，一个地址可以匹配一个 EBCDIC 控制块的名字，可以这一测试在应该失败时成功。但这一机率大约为 1:232，这是一个可以接受的事实。

如果自己分配控制块，应该在释放控制块前改变标记块的名字。这将防止不自觉的再次使用已释放的内存。

发生数据覆盖时，程序故障经常在很久后才性。直接的方法一般不能从结果找到原因，而下一节将要讨论的"陷阱"(traps)则可以帮助用户找到这种问题的原因。

§1.2 陷阱(traps)

前面已经说过，调试的目的是从故障找到原因．有时只能更靠近(指在时间上)原因。一个比较早被查出的故障经常可以被改正．陷阱允许用户倒回去。

典型情况下，服务程序执行对其输入的合法性检查，它们捕获非法输入避免进行不正确的操作．范围检查是很常见的:如果在一个带分支的表中选择函数，在提供的数值中搜寻之前请试一个支持范围之外的函数名值．如果知道一个控制必然存在一个内存范围中，检查一下提供的指针是否指向相应的范围．子程序使用的越频繁，这种合法性检查的价值也就越大．

要避免设置一个不能靠近问题的陷阱．例如，一个使循环变为一个立即的程序故障的陷阱无助于调试过程的进行，因为这只是节省了检测循环的时间．当使用陷阱时，它应该教给我们一些关于故障的新东西．只是提供更快的恢复通常无助于接近问题的解决。

按下列五个步骤进行，可以设置一个陷阱:

1. 全面地研究故障(结果)
2. 按程序逻辑倒回去，使用所有可用的信息．
3. 选择一个或多个认为可以进行问题检测的早期检测的点．
4. 决定陷阱要采取何种行动．理想的情况是陷阱可以捕获足够多的数据使问题可以马上解决．为了保证这一点，必须做到:自动转储我们感兴趣的存储区内容，这样就可以不依赖于一个操作符来进行相应的动作．不幸的是随着检测的进行要检测的存储区是在不断改变的，这需要进一步的转储．停止系统运行可以允许我们在控制台检测数据，但在大型系统上这种行动不宜经常做，系统停止时我们不在场的话，这样做也于事无补．简单地向目录发送一条信息通常得不到足够的数据，而且会使处理器混淆或出错．没有人会喜欢收到这样的信息"Y 子程序中的 X 模块出了点问题"．
5. 代码和测试陷阱适合于你选的点．

测试一下陷阱．要确定建立的陷阱能测试出异常条件，同时还可以检测正常的系统操作．例如，如果编辑器有时要不存储当前要更新的文件而结束编辑，可以设置一个陷阱，这一陷阱负责检查用户的不存储文件的终端请求，　止拦截这一正常条件．陷阱在没有问题时关闭是不好的．如果陷阱影响一个频繁使用的系统子程序，在陷阱安装前后对系统的运行情况作一下比较可以得到一些发生冲突时的具体数据．

这五个步骤对在任何条件编写一个好的陷阱都有所帮助．陷阱产生的数据至少会对进一步编写更好的陷阱有所帮助，最终达到问题的解决．

§1.3 存储转储信息

有时(越少越好)可能会出现一些以上所讲方法无法解决的问题．那么请试试本节的最

后一招。

在不正常使用的情况下，这种方法解决不了任何问题，但如果使用正确，它可以在相同的外部症状下解决两到三个不同的故障。一个问题只出现一次是不可能解决的，出现两三次才能识别出其中的类似的东西，然后才能提出假设并设置陷阱，更接近于故障。假设问题的解决不需存储二十到三十次转储信息。可能会用到磁带或缩微胶片，虽然没有用纸那么方便，但其所占体积要小得多。

如果问题每月只出现一次，那么存储转储信息将是要解决它所必须的技术。一个高可靠性的系统只能来自对罕见的、难办的故障的排除。

§1.4 最近有何变动?

程序的改动常常会带来意想不到的麻烦。由于对改动的检查做的不够，一些问题的发生也就不那么明显。如果一组卡片被用过，那么其中一张就可能会丢失。联接编辑器和调入器可能不检查卡片组中是否有卡片丢失。

我有一次改动了一个程序，它就完全不能运行了。经过几小时的烦人的检查发现调入模块中的系统状态信息不连贯或缺少。我的联接编辑器少了一经 SETSSI 控制卡片。

在改动大型的程序时，使用新的基存储器 USING(对编码和控制块)，删除以前的 DROP 可能会使模块中跟随其后的许多行改变。这与你的改动无关，因为用的错误的基存储器，按照规则，两个 USING 指向同一区域时，汇编程序选择序号较高的一个。因而一个删掉的 DROP 不影响后面代码的机此为 50:50。

一般来说都要问程序最近有何变动，由于这些变动经常引起故障。做研究工作的科学家都对其试验列出分类清楚的目录;系统程序员也应对其所做的事列出类似的目录。简单的记下想做的事将会省去机器的一些无用的运转。

§1.5 程序要做什么事!

要搞清楚一个问题的详细情况否则就会迷失目标。一个好的习惯是通览所有看到的数据(指令记数器、存储器、存储区、程序存储和程序输出)来判断程序哪有问题。如果数据不对了，就可能是指令记数器的问题。

问题是怎么做到这一点?下面是找出故障应遵循的五个步骤:

1. 由于BAL和BALR指令很常用，我们需要检查寄存器。寄存器中存的返回地址，特别是寄存器 14 中的可以指出最后执行的子程序。这样就可以查出是否已从子程序中返回。
2. 一旦找出最后调用的子程序，主可以继续查找其它寄存器，查找子程序内和邻近的代码，判断哪些指令已经执行过。
3. 最后，可以检查相应的数据区(被检查的程序段使用的数据区)来更准确的判断哪些指令已经执行过。

这一序列可以找出出错的指令，或许是一个指令使用了错误的地址或一个寄存器中的数据有错．跟踪存储区链可能也会找到线索．如果机器有一个堆栈(370 机器没有)它将对跟踪程序路径有所帮助。

不要过度信赖指令计数器．如果出现错误传输，这种信赖会有害于调试的进行。从全局上看待故障有利于更快的解决它。

§ 1.6 专门的调试工具

没有合适的工具什么事都不容易做．一些常用的专门程序可以帮助用户进行调试工作．有三种程序提供的帮助最多:转储数据打印程序，调试器(debugger)和跟踪程序(tracing programs)。

§ 1.6.1 转储数据打印程序

最先想到的调试工具是转储数据打印程序如果一个控制块格式整齐的话转储很容易，按其内存中的排列顺序打印就行了。OS／V 转储数据打印程序支持 exit 的时间要比查看转储信息的时间少的多．在一些若干如运行控制块链的时间花费越少，问题解决的也就越快．

§ 1.6.2 调试器

一台动态的调试工具可以节省大量的时间．一个好的工具可以设置断点停止运行来对程序和数据进行测试和改动．这种程序具有一种特殊的文法，不过一旦掌握，使用它们将非常自然和有效．一些系统只提供一个调试器而没有转储程序，调试程序支持字符表，所以可以在调试中使用程序标号．

一个调试器经常用来取代转储程序．如果调试器支持符号表，存储区包括符号信息就可打印出来，因而可读性也就更强．一个调试器可以使用户直接找到问题只需要少量的输出，改善用户的调试速度．需要输出的减少意味着只需要一台很小的窗口，通过它检测故障，这可能使用户看不到邻近的数据区．有时候可能希望打印出一个问题的转储信息而不是使用调试器，因为这样可以看到更多的信息。

调试器的最大用途是用作监视程序运行．执行一个可执行程序并观察其内部操作对用户大有帮助．这一技术与登山者攀登峭壁时所用的技术类似:断点就是他用的钢锥，用它你一点点地通过程序，执行一小段然后检测寄存储区来保证其正确无误。

§ 1.6.3 跟踪器

动态调试器的一个相关工具就是跟踪程序，它记录程序操作过程中发生的事件．虽然跟踪的灵活性很小，但它很有用．但是如果对所有指令一视同仁的进行跟踪，将产生大量的无用输出．按当今的计算速度，跟踪一秒钟内机器运行的每一条指令将会产生一百万行

的输出!

§1.6.4 程序事件记录

IBM 的 system / 370 机器包括一个特殊的硬件功能，程序事件记录(PER，Program —Event Records)来帮助进行调试工作。这一硬件包括四种类型记录的标志位，一位标志寄存器参数和地址区的开始与结束，这都在机器的控制寄存器中。四种类型的记录是:(1)成功的分支(2)指令到来(instructio fetch)(3)存储改变和(4)寄存器改变。

操作系统软件使 PER 非常适合于系统程序员，在 MVS 上，SLIP 操作命令用来产生 PER 陷阱(注意 MVS 不支持寄存器改变陷阱)，因为 VM 实现是交互式的而且易于使用，它能满足一个需要全功能调试器的需要。而 MVS 实现更适合于捕获传计数据中的异常事件和错误。在我看来，存储区改变是 PER 的最强的功能，因为不存在好的替代它的方法。

§1.6.5 PER 陷阱和表现

使用 PER 会戏剧性地增大系统资源的消耗硬件为所有与存储有关的函数提供一个单一的地址范围。如果仅有一个存储范围被跟踪，硬件可在无需软件的帮助下提供所有需要的过滤如果要跟踪多个存储范围。硬件需要软件帮助过滤中断。软件过滤要比硬件慢，所以应尽量避免使用。在 MVS 上，把 PER 陷阱限制在一个特定的地址空间(使 ASID 操作符)，可以减少此陷阱对整个系统的影响。当然，如果不知道哪个地址空间引起覆盖或使用你感兴趣的子程序，就不有用这种方法限制陷阱。

所有这些工具都会使问题更简单，只要能掌握它。但是有时必须跨越一点成见。现实生活中的与一个系统程序员对话:

"为什么不安装 TESTRAN?"
"它不好用"
"你怎么知道"?
"Joe 五年前试过了。")

用户不可能准确地了解什么样的工具适用于要做的工作。请花一些时间学习使用调试程序。

§1.7 装备(instrucmetnatation)

大型系统通常依赖一些重要的内部资源:缓冲区和一些其用途不好做外部描述的控制块，不用费力地去查找存储区传储来得到这些信息，给系统加一个资源监视器。监视器可以做人工使用传储数据所做的事，报告结果，可以自动的也可响应请求。安装这类设备所需编程工作经常意外的少，这是与其效益相比较而言的。

虽然我已使用 JES2 和 HASP 多年，但我也是慢慢地认识到我们需要监视缓冲区和可用控制块的设备．我常常在事后发现，有些故障来源于资源的短缺．

最后由我设计由我的同事写了一个程序．一个新的 JES2 命令触发资源计数，结果将报告给总处理器并记录在记录文件上．用命令实现允许我们 很快查明是否是由于资源短缺带来的故障，我们也可更灵活地对付它．这一命令每天都自动发出，一个事后处理器将就所得数据给出报表．编码的工作大约需三个多月，但其效益是巨大的．八页纸的编码带来的好处远大于其投资，我真不知道没有它时我们是怎么过来的．

软件设备不需大量的或复杂的开发工作:一个简单的工具常常就可满足所有的要求，它当然没有工具好得多．

§1.8 重新设计

系统代码从安装起就有其使用年限．几年后它的初始环境已经改变或消失，但代码却依旧使用．当一个程序只能由一个人才会维护和改动，它就需要重新设计．(参看"预防"一节来学会怎样避免产生新的问题)，这样的工作要花费一些时间，但其效益是可观的．

我们就有过这样的经历．在把 BCRE(我们自己使用的处理系统)转换为 IBM 的 ISO 时，一个同事开发了一个命令用于工作输出提取．命令试图用 HASP 来模拟 BCKE 的接口，但失败了．TSO 对交换的使用，对 SMF 统计的使用，以及我们对那些古老的、复杂的，非文本化的接口的可怜的知识结合在一起就产生了阻塞和低存储区覆盖．(这一接口后来经过了上面提及的重新设计，结果是没有人能完全弄懂那些代码．)一夜之间它变成了我们可靠性方面的主要问题．标准的调试技术无法提供帮助，所以我们不得不丢掉它重新开始．

我开发了一个全新的接口来取代 TSO 和 HASP 的代码．新接口是按使现有故障不可能再发生设计的．所有的数据移动(有一个例外)都在 TSO 存储区保护键下进行，以此来防止无意中的覆盖．民有的 WAIT 都处在监视时钟的保护下、允许进行丢失的 POST 条件进行检测．

仔细准备的流程图和全部进行预先实验免去了设计中的许多问题．

代码的替换取得令人意想不到的成功，我们却花了很长时间才认识到应该这样做(在它的整个使用过程中只发生了一次系统故障，而旧的接口则是大约一天两次)．这使我们感到这几千行的代码带来的价值远大于其替换的费用，因而我们也就不再进行非常需要进行的进一步替换了．

§1.9 没有变动

编程中最令人沮丧的可能是对程序进行了大量改动后，安装好进行测试时发现新程序与旧程序运行时完全一样，没有新的错误，也没有新的改进，什么都没有．

一个同事曾经回忆说:"一个科学家改正了一个对海拔高度的计算，但我们的朋友们却一直怀疑这一结果．科学家说明是无稽之谈，她花了两年才找到这一新算法．最后她拿到程序检查了一下，正确的子程序是在那，可没有使用．系统中没有人调用正确的子程序，

它只是放在那而没被装上"。

创造性的避免测试新代码的方法包括:"使用 NODECK 一起进行汇编来防止产生新的目标模块，没有经过检查的联接编辑(将引起故障)或逻辑错误。有时新代码把一个卡片插入其应在位置的后面一反经过一个无条件分支，它就会不可到达! 这一变动装上后就会浪费很可观的时间。

技术的发展增加了这一问题发生的可能性大量的存储系统确实代表着主存外虚拟存储实现直接存取的概念，但由于设计上的原因，它们增加了丢失改动数据的可能性。如果把当前数据拷贝到永久存储区与进行更新不同步进行的话，就可能产生 当前数据的错误。缓冲区直接存取控制也可能有或没有这一方向信息，这依赖于各个特定的设计。以前的设计(如 IBM 的 3880-23)给出一个"写向(write through)"方法，在 I/O 操作信号结束之前把数据通过缓冲区写到永久存储区，一次成功的写操作意味着没有数据丢失，一些新的设计(如 IBM 的 3990-3 和 3480 磁带控制单位)增长了数据丢失的可能性，写操作信号可能在数据记录到磁记录区以前已经结束了。

当一次改动不能执行其功能，哪怕是多么不可能，也存在着新代码没有装在的可能性。

§ 1.10 独立块(Absolute Patches)

由于我们许多人用汇编编程，用独立修正技术(absolute modification techniques)开发的块是以十六进制，八进制或二进制串的形式写的，我们对这种语言都很陌生，一个块错误，人工从助记符变为操作码，则可以避免。如果块很大，就有在着两个危险。块中可能存在着与我们进行的工作无半的错误，或者当把块从数据转换为汇编语言时可能发生抄写错误(这些错误当然也存在于小的模块，但随着模块的增大，错误出现的机率也就越大)。

当把以独立格式改动过的程序进行重编译时，必须首先把独立改动变为源语言否则块将在传输中丢失。这里的症状是很久以前被改正的错误又出现了。在重编译之间的独立块越多，丢失一个机会也就越大。管理控制，尤其是自动维护目录技术，可以减少，但不能根除这一问题。

我在大的改正中不使用一个块除非这能节约大量的时间，例如如果临时没有程序源代码。(最好想办法把源代码搞到手)关于块的问题完全是无用的因为它的解决一点也不能提高程序的质量。

§ 1.11 运行情况监视器(performance monitor)

一个精密的运行情况监视可对改善软件运行情况大有帮助。这一技术类似于工程计划中的关键路径方法(critical path method)，但它在系统进行操作时执行。

§ 1.11.1 指令地址图

计算机的指令地址段带有一个硬件的监视器。软件监视器执行优先级要比监视程序，使用计时器中断，周期性执行指令地址段的监视程序的 PSW 的优先级高得多。

```
地址          百分比  0-----10-----20-----30-----
OOC5000-FF    0%
00C5100-FF    10%   * * * * * * *
00C5200-FF    2%    *
00C5300-FF    1%    *
00C5400-FF    0%
00C5500-FF    5%    * * * *
00C5600-FF    30%   * * * * * * * * * * * * * * * * * * * * * * *
00C5700-FF    3%    * *
```

指令地址直方图示例

当地址处在预料的范围中时，监视器就会打印出一个高分辨率的地址分配图。这一直方图将与软件做比较。我们对有问题的地区进行检测来提高 CPU 的使用效率。

看一下上图，可以清楚地看到程序检测地址段(5600—(56FF 费时最多，占总数的 30%。当我们检查完这一段程序后，我们可以跳到 C5100 到 C51FF，这一段占 10%。

§ 1.11.2 使用监视器的输出

要把如内循环等瓶颈进行隔离有两个办法。我们可以在"热点"处做一些改动来提高代码运行速度，或改变程序设计来减少进入此程序的次数。一般情况下后一种方法会带来更大的改善，但需要更多的转换。我喜欢先进行局部变动，衡量一下性能的改善，决定是否需要做进一步的工作。

这一技术改变了早期的编码方式。以前，每条指令都小心编写来节省尽可能多的时间。今天我们则努力寻找值得进行效率改进的地方。(我现在仍在写 LA RX，3(，RX)而不是 LA RX，3(RX)因为在一台 360／40 上第一种格式是一个较快的机器循环。现在我使用的机器两个都很快)。

§ 1.11.3 硬件监视器的困难

硬件监视技术存在着问题。监视是很精巧的工作。监视器必须足够的快来捕捉数据，足够的功能来开发直方图，足够的敏感来避免计算机中的电压骤减(这将引起硬件故障)计算机厂商的工程师不会太热心，因为他们的计算机面临着危险。(工程师们会不自觉的害怕使用设备数目的减少。最常使用硬件监视器的场合，系统行为分析一般还要使用更多的

设备。例如，一个大的中央处理器，更多的通道，更多的控制单位，更多的或磁鼓！)最后，有时还要消除由不正确的监视器设置带来的人为错误。在监视指令地址数据时，我最喜欢的是在数据缓冲区里报告程序动作。在 MVS 系统中有多个地址空间，任务就定为艰巨，需要用保护键过滤选择想要的地址空间。

§1.11.4 软件监视器

一个软件监视器可带来相似的结果，但对于操作系统软件来说此监视器必须自己衡量其运行效果。但是，有些比较精巧的系统交互操作可以被检测，因为软件监视器可以把指令计数器和主存联系起来，告诉我们更多的系统状态。另外，软件监视器可以收集与缺省页相连的指令地址并把这些地址绘成一个单独的直方图;因而它们可以帮助识别那些带来不必要的翻页的程序码。

§1.11.5 效果

结果具有很高的价值。硬件监视器可以改正 HASP 系统节约它总共消耗的三分之一的 CPU 时间，在已经算效率较高的软件包中这不能不说一大进步。硬件监视指出很大一部分时间消耗在一个待定的 HASP 子程序上。研究一下这一子程序，可以看到其寻找控制块链的功能还能做得更快。我们认识到在链在相同类型的控制块是连续的，那么就可以改变子程序不在链头而是在相应类型的第一个控制块开始查找。一旦控制块的类型变了就停止查找。注意只要改动一个子程序就可获得这一戏剧性的执行性能的提高。

在找到瓶颈以前不要开始进行优化。Pareto 的原则告诉我们:80%的时间花在了 20%的程序段上。对这 20%的改善就可对程序产生很大的影响。不要把时间花费在 80%的程序上。

§1.12 “浪费”一点机器时间

在解决问题中，不要舍不得用一些机器时间来降低问题的难度。如果在一个模块中查找一个文字串，让机器做这一工作。如果用合适的断点，进行一个交互测试还是要用很多时间才能找到故障发生的条件，就别去做它。

计算技术经济学表明人力花费在飞速增长，同时相对来说计算机时间越来越便宜(几年前，IBM 宣布它的软件开发经费第一次超过了其硬件开发经费。这无可争辨地反映这一工业的发展趋势)。如果一处错误可以由人做一小时完成，请先调查一下是否可由计算机几分钟解决问题。

§1.13 厂家支持

厂商支持大部分现在使用中的软件，包括维护更新，故障列表，和支持人选(support personnel)