

广东中兴新支点

KProfiler 用户手册

嵌入式操作平台

声 明

本资料著作权属广东中兴新支点技术有限公司所有。未经著作权人书面许可，任何单位或个人不得以任何方式摘录、复制或翻译。

侵权必究。



是广东中兴新支点技术有限公司的注册商标。广东中兴新支点技术有限公司产品的名称和标志是广东中兴新支点技术有限公司的专有标志或注册商标。在本手册中提及的其他产品或公司的名称可能是其各自所有者的商标或商名。在未经广东中兴新支点技术有限公司或第三方商标或商名所有者事先书面同意的情况下，本手册不以任何方式授予阅读者任何使用本手册上出现的任何标记的许可或权利。

由于产品和技术的不断更新、完善，本资料中的内容可能与实际产品不完全相符，敬请谅解。如需查询产品的更新情况，请联系广东中兴新支点技术有限公司。

若需了解最新的资料信息，请访问网站 <http://www.gd-linux.com/>。

手册说明

本手册是 KProfiler 产品的全面使用说明书，从使用者角度介绍如何运用 KProfiler 收集采样数据并进行解析。

内容介绍

章 名	概 要
第 1 章 概述	介绍何为 KProfiler，其功能特色及快速入门
第 2 章 部署	分别介绍内核态、用户态及图形界面 GUI 工具三种部署方式
第 3 章 命令行基本操作	介绍命令行方式如何收集采样数据并解析
第 4 章 图形界面基本操作	介绍图形界面下如何收集采样数据并解析
第 5 章 高级功能	对高级功能进行列举介绍
第 6 章 典型案例	通过介绍几个典型案例介绍 KProfiler 应用场景
第 7 章 精确计时统计	介绍精确计时统计功能
第 8 章 FAQ	常见故障分析方法
第 9 章 开源 LICENSE 说明	简单介绍 GPL
附录 A 典型事件列举	按照 ARCH 类型列举典型事件

版本更新说明

文档版本	发布日期	备 注
KProfiler_V3.02.20_P1B3	2013-3-20	2013 年第一次版本发布。

本书约定

介绍符号的约定、键盘操作约定、鼠标操作约定以及四类标志。

- 1) 符号约定

- 样式 **按钮** 表示按钮名；带方括号 “**【属性栏】**” 表示人机界面、菜单条、数据表和字段名等；
- 多级菜单用 “->” 隔开，如 **【文件】->【新建】->【工程】** 表示 **【文件】** 菜单下的 **【新建】** 子菜单下的 **【工程】** 菜单项；
- 尖括号 <路径> 表示当前目录中 include 目录下的 .h 头文件, 如 <asm-m68k/io.h> 表示 /include/asm-m68k/io.h 文件。

2) 标志

本书采用二个醒目标志来表示在操作过程中应该特别注意的地方：



提示：介绍提高效率的一些方法；



警告：提醒操作中的一些注意事项。

联系方式

电 话：020-87048510

电子信箱：cgel@gd-linux.com

公司地址：广州市天河区科技园高唐软件园基地高普路 1021 号 6 楼

邮 编：510663

目 录

1. 概述.....	1
1.1. 功能说明.....	1
1.2. 对待测系统影响.....	3
1.3. 版本相关.....	3
1.4. 快速入门.....	3
1.4.1. 命令行方式	3
1.4.2. 图形界面方式	7
2. 部署.....	9
2.1. 内核部署.....	9
2.1.1. 获取	9
2.1.2. 编译支持 KProfiler 内核	10
2.2. 用户态工具部署.....	25
2.2.1. 获取	25
2.2.2. 目录结构	25
2.3. 图形界面 GUI 工具部署	27
2.3.1. 获取	27
3. 命令行基本操作.....	27
3.1. 收集采样数据.....	28
3.1.1. 加载内核 KProfiler	28
3.1.2. 用户态控制工具 kprofiler	28
3.2. 解析采样数据.....	36
3.2.1. 解析采样数据工具	36
3.2.2. 解析场景	42
3.2.3. 符号表路径设置规则	43
3.2.4. 解析结果的统计信息分析	44

4. 图形界面基本操作	45
4.1. 收集采样数据.....	45
4.1.1. 加载内核 KProfiler	45
4.1.2. 建立目标机连接	45
4.1.3. 配置启动采样	47
4.1.4. 控制启动采样	54
4.2. 解析采样数据.....	55
4.3. 日志管理.....	56
5. 高级功能.....	57
5.1. 阻塞采样.....	57
5.1.1. 应用场景	57
5.1.2. 使用介绍	57
5.1.3. 相关命令	58
5.1.4. 典型应用	59
5.2. 函数调用关系.....	64
5.2.1. 应用场景	64
5.2.2. 使用介绍	64
5.2.3. 相关命令	65
5.2.4. 典型应用	66
6. 解析数据解读	67
6.1. 热点函数数据解析.....	67
6.2. 函数调用关系数据解析.....	69
6.3. 源码级数据解析.....	71
7. 典型案例.....	73
7.1. 性能调优.....	73
7.1.1. CPU 占用率高但系统性能低下.....	73
7.1.2. CPU 占用率高但业务性能上不去.....	78
7.1.3. CPU 占用率低性能无法提高.....	81

7.2. 特殊事件应用	84
7.2.1. 数据缓存 miss 统计	84
7.2.2. 系统/函数指令执行效率统计	90
8. 精确计时功能	94
8.1. 原理简介	94
8.2. 使用说明	94
8.2.1. 生成性能日志	94
8.2.2. 解析性能日志	95
8.2.3. 配置信号	96
8.2.4. 精确计时接口	97
8.2.5. 使用精确计时功能辅助阻塞点定位	97
8.3. 解析结果说明	97
8.4. 用于开启时钟寄存器访问权限的内核模块代码	100
9. FAQ	101
9.1. 如何判断当前环境的采样类型	101
9.2. 设置调用图深度不成功	102
9.3. 多个事件采样时报错	102
9.4. 评测结束无结果显示	103
9.5. 开始采样执行清除采样数据操作报错	103
9.6. 结束采样后打包数据报错	103
9.7. 计数器采样，不评测内核依然可采集到内核数据	104
9.8. 设置堆栈深度后无堆栈回溯信息显示	105
9.9. 查看解析结果 overflow 值较大	105
9.10. 查看解析结果 event_overflow 值较大	106
9.11. 两种采样方式结果差异很大	106

9.12. 计数器采样，结果大多落在软中断	107
9.13. 在 zemu mips64 仿真环境下配置 oprofile 后内核启动异常死机.....	108
9.14. 图形界面 GUI 控制目标端采样遭遇干扰导致采样失败.....	109
9.15. 图形界面 GUI 进行数据采样无法获取采样结果	110
9.16. 当 Linux 环境配置了 nmi_watchdog 时如何采样.....	111
10. 开源 LICENSE 说明	112
10.1. GPL 简介.....	112
10.2. 开发指导.....	112
10.3. 源码获取方式.....	113

1. 概述

1.1. 功能说明

基于 OProfile 研发的 KProfiler 是一款低开销的系统全局性能监视工具，用于对 Linux 系统进行评测（profiling）和性能监控。通过评测表或图形，形象的显示出为特定的处理器事件收集的采样百分数或数量以用于总结或分析。KProfiler 可以工作在不同的体系结构上，包括 IA32, IA64 和 AMD Athlon 系列，以及 arm（包括 nommu）、ppc（powerpc）、mips 等。由于其开销小，KProfiler 内核部分的代码已经被包含在 CGEL 内核中，包括 CGEL3.0、CGEL3.0-KTH 和 CGEL4.0 部分体系。

KProfiler 可以帮助用户获取诸如系统热点函数及其调用链、缓存/TLB 命中率、函数执行效率等信息，能够帮助用户排查 CPU 冲高，程序性能无法提升等问题。它收集有关处理器事件的信息，其中包括 TLB 的故障、停机、存储器访问、缓存命中率。通过收集到的评测数据，用户可以很容易地找出性能问题。

KProfiler 是一种基于统计的分析工具，可以在系统运行过程中为内核、库及用户程序收集采样。采样工作依照以下两种采样机制进行：

■ 基于性能计数器采样机制

此种采样机制需要硬件支持。在 KProfiler 启动时设置用户关心的事件及计数器的溢出值。在系统运行过程中，如果 KProfiler 监测到发生本事件一次，则计数器加 1。当计数器的值超过设定的阈值时，会触发一个不可屏蔽（NMI）中断。此刻，中断处理函数会保存中断触发时的处理器状态，如程序寄存器、程序指针 PC、堆栈信息等。使用虚拟机启动的内核不支持计数器采样。i386/piii 默认事件为 CPU_CLK_UNHALTED，Mips732 默认事件为 CYCLES，若了解更多的事件信息，请参考附件 A。

■ 基于定时器方式采样机制

支持没有性能计数器的硬件或者不能使用性能计数器的场景。定时器方式是借助 OS 时钟中断的机制。每个时钟中断都会激发 KProfiler 进行一次采样。中断处理函数保存中断触发时刻的处理器状态供用户分析处理。不同于基于性能计数器的采样，一旦代码禁用了中断，则不能进行数据采样。基于定时器方式采样，无法设置采样时间间隔，而是使用系统默认值（一个 TICK）。

KProfiler 使用了一个内核模块和一个用户空间守护进程，内核模块主要功能为 KProfiler 初始化、oprofilefs 注册、样本采集等。用户态工具运行于后台，负责从 oprofilefs 中导出样本数据并进行解析。基于定时器方式采样不能配置事件，基于性能计数器的采样如果不配置时，则使用默认事件，通常是处理器周期，即该事件将对处理器循环进行计数。事件的样本计数将决定事件每发生多少次计数器才增加一次。KProfiler 被设计成可以在低开销下运行，从而使后台运行的守护进程不会扰乱系统性能。

基于 OProfile 研发的 KProfiler，修复了大量 BUG 的同时，还根据自身的需要制定了特有的功能，其功能对比如下：

表 1-1 KProfiler 与 Oprofile 功能对比

功能	开源 OProfile	KProfiler
采样过程控制方式	脚本控制	程序控制
Mips 架构的调用关系回溯	不支持	支持
阻塞采样	不支持	支持
配置 NO_HZ 后定时器方式采样	不能准确反映	采样结果正确
处理器空闲时处理器时钟周期事件	不能准确反映	采样结果正确
交叉解析	多步骤完成	一条命令实现
定时采样	不支持	支持
支持架构扩展	不支持	支持
动态切换采样方式	不支持	支持
采样进程优先级设定	不支持	支持
打包功能	不支持	支持手动和自动打包
采样对内核符号表和 objdump 依赖	依赖	脱离
绑定采样	不支持	支持
信号触发采样	不支持	支持

KProfiler 不仅仅提供了命令行方式的采样解析，同时，还提供了友好的图形化界面，用户只需在用户界面上根据界面的提示配置自己所需要的功能，而不用去了解命令行的组织形式和参数意思。清晰明了的图表形式输出数据的解析结果，便于用户分析、理解。强大的可视化控制界面方便了初学者使用的同时，还大大提高了 KProfiler 的使用效率。

1.2. 对待测系统影响

使用 KProfiler 对操作系统和应用程序及库进行性能评测的过程会对操作系统带来额外的开销，但经过试验所得，配置 KProfiler 功能，但不运行则对系统几乎没有任何影响。若使用 KProfiler 进行采样，则依据采样方式的不同，对系统产生的影响也不尽相同：

- 基于定时器方式采样，因其每秒中断次数很少（几百的数量级），除网络测试外，其余影响均在 5% 以下；
- 基于性能计数器的采样，默认事件阈值情况下（1 万中断/秒），影响基本都在 10% 以下，也能满足要求；
- 至于网络测试结果，可能会存在较大影响，但这与您自身设置或硬件配置有关：
 - ✧ 设置的阈值过小，导致每秒中断数太多，对应处理器周期事件其默认阈值为 100000，对应 1 万次中断/每秒，说明 1 万次/秒时的采样效果应该是最好的，8 万次/秒的处理压力算是比较大；
 - ✧ 由于硬件条件的限制，导致测试数据差别较大；
 - ✧ NMI 中断过多时，对网络性能影响较大。

总的来讲，KProfiler 是一个低系统消耗的系统评测工具，其对系统性能的影响在可接受范围之内。

1.3. 版本相关

用户可以通过查看放置 KProfiler 二进制文件目录下的 `kprofiler_version` 文件来获取 KProfiler 的版本号和构建时间。

1.4. 快速入门

KProfiler 工具的使用总体上分为三大步骤：内核配置编译并启动内核，数据采样和数据解析。下面分别按照命令行方式和图形界面方式简单地介绍一下主要操作顺序和基本步骤。

1.4.1. 命令行方式

1.4.1.1. 内核配置编译并启动内核

- 1) 为内核配置 KProfiler 支持，并对配置项均采用编译进内核的配置方式：

- MIPS 架构：配置方式如下：

```
Profiling support --->
    Performance-monitoring counters support --->
        [*]Profiling support<EXPERIMENTAL>
        [*]OProfile system profiling<EXPERIMENTAL>
```

配置完毕后，检查 lsp/defconfig 文件，是否成功配置 KProfiler 支持，如下所示：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
```

➤ X86/x86_64 架构：配置方式如下：

```
Instrumentation Support--->
    [*]Profiling support<EXPERIMENTAL>
    <*>OProfile system profiling<EXPERIMENTAL>
    [ ]Kprobes<EXPERIMENTAL>
    [ ]Activate markers
Kernel hacking--->
    [*]Compile the kernel with frame pointers
```

配置完毕检查 lsp/deconfig 文件，若是成功配置，则显示如下：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
CONFIG_FRAME_POINTER=y
```

➤ ARM 架构：配置方式如下：

```
Profiling support --->
    [*]Profiling support<EXPERIMENTAL>
    <*>OProfile system profiling<EXPERIMENTAL>
```

配置完毕检查 lsp/deconfig 文件，若是成功配置，则显示如下：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
```

➤ PPC85xx 架构：配置方式如下：

```
Code maturity level options--->
    [*]Prompt for development and /or incomplete code/drivers

Instrumentation Support--->
    [*]Profiling support<EXPERIMENTAL>
```

```
<*>OProfile system profiling<EXPERIMENTAL>
[ ]Kprobes<EXPERIMENTAL>
[ ]Activate markers
```

配置完毕检查 lsp/deconfig 文件，若是成功配置，则显示如下：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
```

- 2) 构建内核；
- 3) 启动内核。

1.4.1.2. 运行、采样并解析

下载 KProfiler 用户态工具到目标端，使用如下命令完成采样和数据解析。

```
./kprofiler          #表示默认启动计数器进行采样，并默认使用-p all 选项    （所有采样数据分别按照
cpu、库、线程号和线程组号、内核区分保存），默认使用-c 5 选择，设置调用图深度值为 5 层，默认使用
-r 1 清除之前采样

./cg.out              #目标端启动用户程序

./kprofiler --stop    #停止采样

./opreport            #统计用户态程序和内核的采样情况
```

1.4.1.3. 解析数据分析

```
./opreport
```

命令运行后，对于用户态程序和内核采样得到的解析结果具体含义，在此处以一个实例进行介绍：

```
Sample Statistical Information :
      CPU0      CPU1
received      10741      10589  #CPU0 上采样次数为 10741 次，CPU1 上采样次数为 10589
overflow       0         0  #采样过程中的内核缓存溢出次数为 0
%(overflow)    0         0  #采样过程中的内核缓存溢出次数占此 CPU 上采样个数的百分比为 0
backtrace(aborted)0         0  #堆栈回溯过程中的失败次数为 0
%(backtrace)   0         0  #堆栈回溯过程中的失败次数占此 CPU 上采样个数的百分比为 0
received(total) no-mapping(backtrace) %  event-overflow %  no-mapping(sample) %
error          %              #总的采样统计
21330          0         0         0         0         0         0         0         0
```

```

Start Sample Time : 06:47:25      #开始采样时间
Stop Sample Time : 06:47:38      #结束采样时间
Sample time : 13.786123s         #整个采样持续时间

Running sample number : 21330     #运行时采样次数

image=/home/lcz/cg.out
image=/vmlinux
CPU: AMD64 processors, speed 2310.28 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit mask of 0x00 (No unit
mask) count 2310000              #采用计数器方式采样
Running Samples:                 #统计总的采样情况，采样落在各进程及函数的次数，以及所占比例
    event| samples|    %|                (usr:% sys:%)|    app
-----
CPU_CLK_UNHALTED      21328      #整个系统中，进程采样数为 4564
                        4546  21.3%      (4546:21.3% 0:0%)      /home/zy/test/User.out
# User.out 用户态采样数为 4546，占总的采样比 21.3%，User.out 内核态采样数为 0，占采样比也为 0
                        (3385:15.9%)      test_add_4
#User.out 进程中采样热点函数 test_add_4，其采样数为 3385，占总的采样比为 15.9%
                        (1003:4.7%)      test_add_5
                        16782  78.7%      (0:0% 16782:78.7%)      /vmlinux
#vmlinux 用户态采样数为 0，内核态采样数为 16782，占总的采样比为 78.7%
                        (5670:26.6%)      poll_idle
#采样落在 poll_idle 这个函数中的次数为 5670:次，占总共的采样次数比例为 26.6%
                        (5545:26.0%)      test_ti_thread_flag
                        (5490:25.7%)      need_resched
.....

```

总的采样情况：

received(total)	no-mapping(backtrace)	%	event-overflow	%	no-mapping(sample)	%	error	%
21330	0	0	0	0	0	0	0	0

其各参数的含义分别是：

- received(total): 总采样次数为 21330;
- no-mapping(backtrace) %: 用户态程序堆栈回溯时找不到进程空间的样本个数为 0，占总的采样次数的百分比为 0;


- event-overflow %: 采样过程中的文件缓冲溢出次数为 0, 占总的采样次数的百分比为 0;
- no-mapping(sample) %: 统计采样的用户态程序 pc 值找不到对应的虚拟区 (VMA) 的样本个数为 0, 占总的采样次数的百分比为 0;
- error%: 统计采样的用户态程序 pc 找不到进程空间的样本个数为 0, 占总的采样次数的百分比为 0。

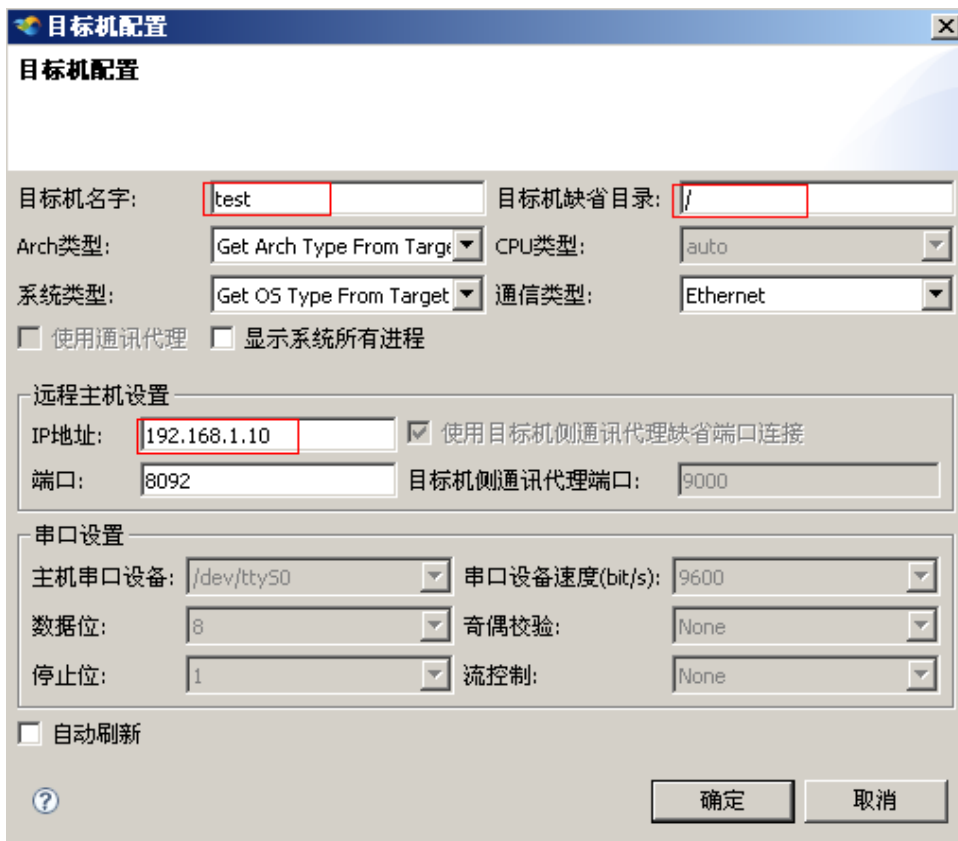
1.4.2. 图形界面方式

1.4.2.1. 内核配置编译并启动内核

参见 1.4.1.1, 与命令行方式相同。

1.4.2.2. 运行、采样并解析


- 1) 开启目标板, 在确认启动和初始化成功后, 在 KIDE 的目标机视图里点增加目标机配置项, 在弹出窗口中, 为目标机命名, 输入以/ (斜杠) 结尾的目标板 (机) 的缺省目录, 输入目标机端的 IP 地址, 最后点**确定**完成配置;



The image shows a 'Target Machine Configuration' dialog box with the following fields and options:

- 目标机名字:** test
- 目标机缺省目录:** /
- Arch类型:** Get Arch Type From Target
- CPU类型:** auto
- 系统类型:** Get OS Type From Target
- 通信类型:** Ethernet
- ☐ 使用通讯代理
- ☐ 显示系统所有进程
- 远程主机设置**
 - IP地址:** 192.168.1.10
 - ☒ 使用目标机侧通讯代理缺省端口连接
 - 端口:** 8092
 - 目标机侧通讯代理端口:** 9000
- 串口设置**
 - 主机串口设备:** /dev/ttyS0
 - 串口设备速度(bit/s):** 9600
 - 数据位:** 8
 - 奇偶校验:** None
 - 停止位:** 1
 - 流控制:** None
- ☐ 自动刷新
- Buttons:** 确定 (OK), 取消 (Cancel)

图 1-1 新建目标机

- 2) 对新建目标机，点击右键，选择“打开 KProfiler”，开启 KProfiler 视图；
- 3) 点击  按钮，新增一个启动采样配置；
- 4) 在 KProfiler 采样配置窗口，【通用】选项页，为此 KProfiler 配置添加描述，选择待连接目标机，其余选项配置保持默认缺省即可；

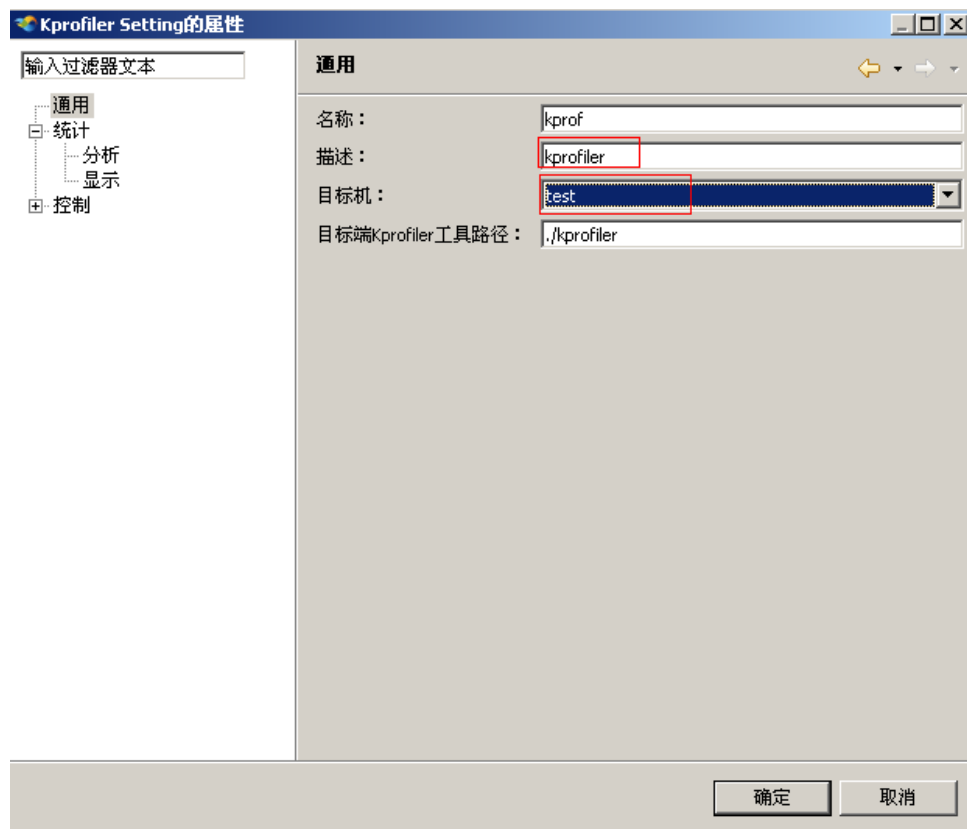


图 1-2 通用选项配置

- 5) 点击 **确定**，完成 KProfiler 设置；
- 6) 对新建 KProfiler 配置点击右键，选择“控制”->“挂载”，完成此配置与目标机的绑定；

- 7) 挂载成功，对 KProfiler 配置点击右键，选择“控制”->“启动”，开启数据采样，在目标端启动用户态程序；
- 8) 完成采样后，对 KProfiler 配置点击右键，选择“控制”->“停止”，结束采样；
- 9) KProfiler 会自动打印出解析结果。
- 10) 数据采样解析完毕后，对 KProfiler 配置点击右键，选择“控制”->“卸载”，取消与目标机的连接，方便其他人进行使用。这是由于每台目标机在一个时间段内仅能用于一套 KProfiler 的采样、解析。

1.4.2.3. 解析数据分析

停止采样后，KProfiler 会自动打印出解析结果，同时显示出对内核以及用户态程序的解析情况，其具体显示信息及含义同 1.4.1.3。

2. 部署

2.1. 内核部署

由于 KProfiler 内核部分的代码已经被包含在 CGEL 内核中，包括 CGEL3.0、CGEL3.0-KTH 和 CGEL4.0 部分体系，因而通过构建 CGEL 内核的 LSP 项目，并进行 KProfiler 内核配置后，即可获取 KProfiler 内核功能模块。

2.1.1. 获取

关于 KProfiler 的内核模块部分可对 CGEL 内核进行内核配置获得，CGEL 内核随同 KIDE 一并发布，可直接登录 FTP（<ftp://10.75.8.168/ZX-TSP/2012年/KIDE/>，用户名和密码都为 zteuser）获取 KIDE 安装文件，下载安装即可；也可以登陆南京服务器（[ftp://10.44.27.159/chengyan version/ZX-TSP/2012年/KIDE/](ftp://10.44.27.159/chengyan_version/ZX-TSP/2012年/KIDE/)，用户名和密码都为 zteuser）。

2.1.2. 编译支持 KProfiler 内核

2.1.2.1. 命令行下编译

■ 内核配置

进入内核构建目录，执行如下命令即可进行内核配置。由于 CGEL 内核源码获取方式不同，其工程构建目录也不同。若以安装 KIDE 方式，其路径为<\target\build\cgelX.X\product\build>；若从服务器直接获取源码，则路径为<\target-cgl\build\cgelX.X\product\build>。

```
make ARCH=XXX
CROSS_COMPILE=../../bin/ARCH-linux-gnu-      LSPSRC=../../bsp/cgelX.X/../../ZTE-XXX-XXX
menuconfig
```

如 ppc 架构：

```
make ARCH=ppc
CROSS_COMPILE=/ppc860_gcc4.1.2_glibc2.5.0/bin/powerpc-860-linux-gnu-
LSPSRC=../../bsp/cgel3.0/ppc/8xx/ZTE-DTB-mpc850 menuconfig
```

打开内核配置界面后，按照架构不同，分别选择相应的配置选项完成 KProfiler 功能的配置。

➤ MIPS 架构：配置方式如下：

```
Profiling support --->
    Performance-monitoring counters support --->
        [*]Profiling support<EXPERIMENTAL>
        [*]OProfile system profiling<EXPERIMENTAL>
```

配置完毕后，检查 lsp/defconfig 文件，是否成功配置 KProfiler 支持，如下所示：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
```

➤ X86/x86_64 架构：配置方式如下：

```
Instrumentation Support--->
    [*]Profiling support<EXPERIMENTAL>
    <*>OProfile system profiling<EXPERIMENTAL>
    [ ]Kprobes<EXPERIMENTAL>
    [ ]Activate markers
Kernel hacking--->
```

☒ Compile the kernel with frame pointers

配置完毕检查 lsp/deconfig 文件，若是成功配置，则显示如下：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
CONFIG_FRAME_POINTER=y
```

➤ ARM 架构：配置方式如下：

```
Profiling support --->
☒ Profiling support<EXPERIMENTAL>
<*>OProfile system profiling<EXPERIMENTAL>
```

配置完毕检查 lsp/deconfig 文件，若是成功配置，则显示如下：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
```

➤ PPC85xx 架构：配置方式如下：

```
Code maturity level options--->
☒ Prompt for development and /or incomplete code/drivers

Instrumentation Support--->
☒ Profiling support<EXPERIMENTAL>
<*>OProfile system profiling<EXPERIMENTAL>
[ ] Kprobes<EXPERIMENTAL>
[ ] Activate markers
```

配置完毕检查 lsp/deconfig 文件，若是成功配置，则显示如下：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
```

PPC85XX 架构还支持 CGEL4.0 内核，其内核配置 KProfiler 支持的方式如下：

```
General setup --->
☒ OProfile support<EXPERIMENTAL>
☒ OProfile system profiling<EXPERIMENTAL>
```

配置完毕检查 lsp/deconfig 文件，若是成功配置，则显示如下：

```
CONFIG_PROFILING=y
CONFIG_OPROFILE=y
CONFIG_HAVE_OPROFILE=y
```



提示：由于环境存在差异，内核配置文件的放置位置可能有所不同，详情请咨询 bsp 包提供者。

■ 构建内核

按照如下步骤进行：

第 1 步：运行 Cygwin，内核若以安装 KIDE 方式获取则进入目录<\target\build\cgeI3.X\product\build >，若单独获取内核源码包，则进入目录<\target-cgl\build\cgeI3.X\product\build >；

首次使用，可以通过以下命令获取帮助信息；

```
make ARCH=XXX
CROSS_COMPILE=/.../bin/XXX-linux-gnu- LSPSRC=../../bsp/cgeI3.0/../../ZTE-XXX-XXX help
```

如：

```
make ARCH=ppc
CROSS_COMPILE=/ppc860_gcc4.1.2_glibc2.5.0/bin/powerpc-860-linux-gnu-
LSPSRC=../../bsp/cgeI3.0/ppc/8xx/ZTE-DTB-mpc850 help
```

其中，参数 ARCH、CROSS_COMPILE、LSPSRC 三个参数用户可以根据自己的具体情况设置。

第 2 步：编译前执行 clean 命令，清除之前编译的中间文件和可执行文件；

```
make ARCH=XXX
CROSS_COMPILE=/.../bin/XXX-linux-gnu- LSPSRC=../../bsp/cgeI3.0/../../ZTE-XXX-XXX clean
```

如：

```
make ARCH=ppc
CROSS_COMPILE=/ppc860_gcc4.1.2_glibc2.5.0/bin/powerpc-860-linux-gnu-
LSPSRC=../../bsp/cgeI3.0/ppc/8xx/ZTE-DTB-mpc850 clean
```

第 3 步：执行编译命令。

```
make ARCH=XXX
```

```
CROSS_COMPILE=../../bin/XXX-linux-gnu- LSPSRC=../../bsp/cgel3.0/../../ZTE-XXX-XXX all
```

如：

```
make ARCH=ppc  
CROSS_COMPILE=/ppc860_gcc4.1.2_glibc2.5.0/bin/powerpc-860-linux-gnu-  
LSPSRC=../../bsp/cgel3.0/ppc/8xx/ZTE-DTB-mpc850 all
```

第 4 步：编译成功完成后，在目录<target\build\cgel3.0-kth\product\object >下会动态生成一个目录 ZTE-XXX-XXX，如 ZTE-DTB-mpc850，目录 images 下存放生成的内核映像，如 uImage。

关于内核的构建、加载，详细信息可参考《广东中兴新支点 CGEL 产品开发指南》和《广东中兴新支点 CGEL 用户手册》。

2.1.2.2. 图形界面 KIDE 下编译

■ 新建 LSP 项目

第 1 步：在【CGEL 项目】分支选择【LSP 项目】，然后点下一步。

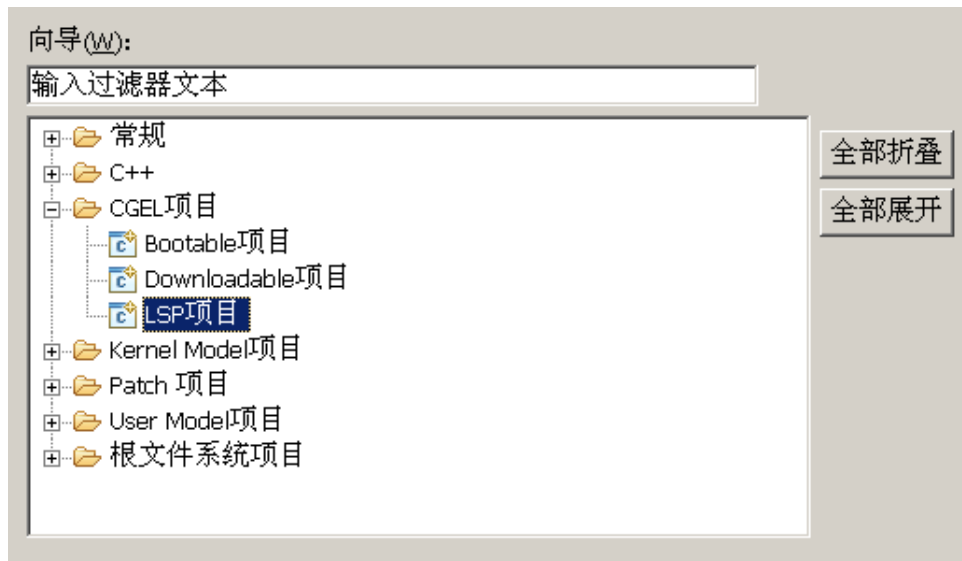
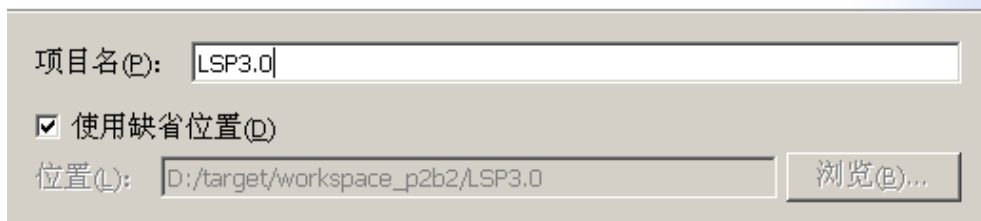


图 2-1 选择 LSP 项目

第 2 步：为 LSP 项目命名，然后点下一步。如果不使用默认的工作空间目录，将【使用缺省位置】的勾去掉。

c 项目向导

c 项目向导

A screenshot of the 'Project Wizard' dialog box. It has a title bar with a folder icon and the letter 'C'. The dialog contains a text field for 'Project Name (P):' with the value 'LSP3.0'. Below it is a checked checkbox for 'Use default location (D):'. At the bottom, there is a text field for 'Location (L):' with the value 'D:/target/workspace_p2b2/LSP3.0' and a 'Browse (B)...' button to its right.

项目名(P): LSP3.0

☒ 使用缺省位置(D)

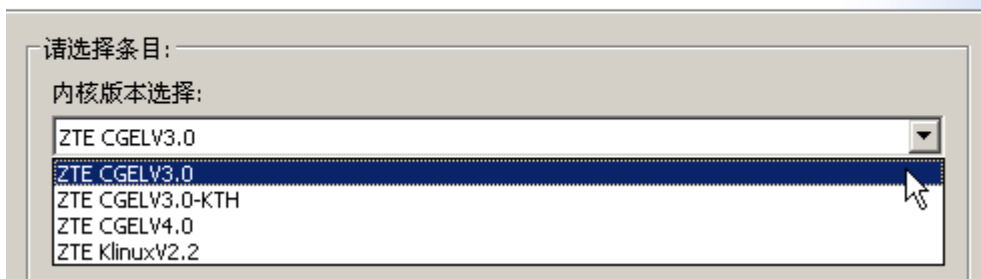
位置(L): D:/target/workspace_p2b2/LSP3.0 浏览(B)...

图 2-2 LSP 项目取名

第 3 步，选择内核版本，KProfiler 对 CGEL3.0、CGEL3.0-KTH、CGEL4.0 都支持，此处我们选择以 CGEL3.0 内核为例。

选择项目的一个内核版本

选择需要的内核版本

A screenshot of the 'Select Kernel Version' dialog box. It has a title bar with a folder icon and the letter 'C'. The dialog contains a list box titled 'Please select item:' with the label 'Kernel version selection:'. The list box contains the following items: 'ZTE CGELV3.0', 'ZTE CGELV3.0-KTH', 'ZTE CGELV4.0', and 'ZTE KlinuxV2.2'. The item 'ZTE CGELV3.0' is currently selected and highlighted in blue. A mouse cursor is visible over the list box.

请选择条目:

内核版本选择:

- ZTE CGELV3.0
- ZTE CGELV3.0-KTH
- ZTE CGELV4.0
- ZTE KlinuxV2.2

图 2-3 选择 CGEL 内核版本

第 4 步，为目标板选择 CPU 型号，编译工具链按默认，之后点**完成**。

图 2-4 中，【处理器体系选择】是目标板的 CPU 体系选择。【处理器选择】是具体的 CPU 型号，本例是一个 i386 CPU 单板；【LSP 模板选择】是广东中兴新支点嵌入式平台提供的一些已有的 LSP 模板，选择一块与自己的开发板最接近的模板；【工具链选择】一般选默认工具链，如果有分支请根据需要选择。

选择项目的一种类型

选择需要的处理器体系和处理器。



请选择条目：

处理器体系选择：
i386

处理器选择：
i386

LSP模板选择：
ZTE-VERMACHINE-i386

工具链选择：
CrossChain(GCC 4.1.2, GLIBC 2.5.0)

图 2-4 选择 CPU 体系

第 5 步，完成 LSP 项目建立，图 2-5 是 1 个新建的 LSP 项目。

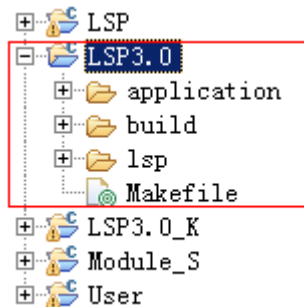


图 2-5 LSP 项目创建成功

提示：若对 LSP 项目重命名，则需要重新编译，否则将导致符号表文件和内核映像文件的路径有误。

■ 内核配置

要使用 KProfiler 工具，首先在内核配置时必须配置 KProfiler 支持，不同的体系架构其配置菜单有所不同，分别描述如下：

➤ ARM

打开内核配置界面后，选择【Profiling support】这一项，如图 2-6:

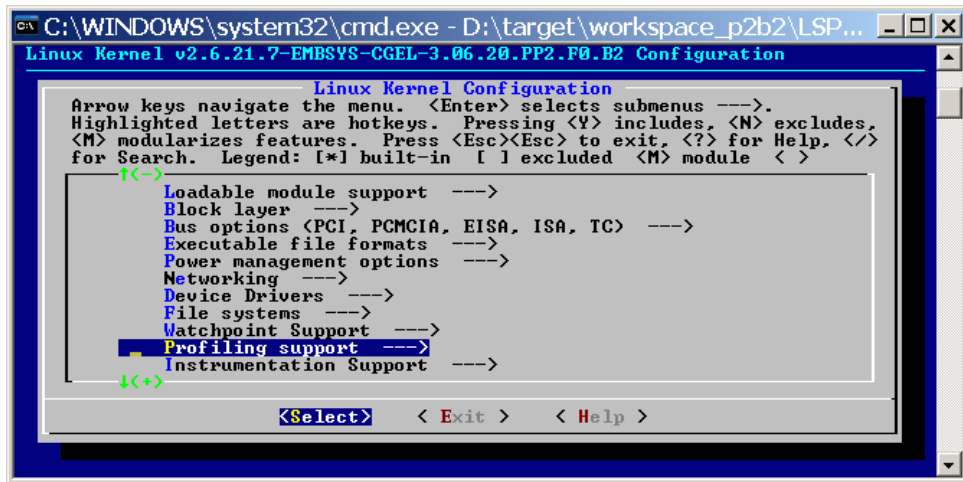


图 2-6 选中【Profiling support】

选中【Profiling support】和【OProfile system profiling】配置选项，如图 2-7:

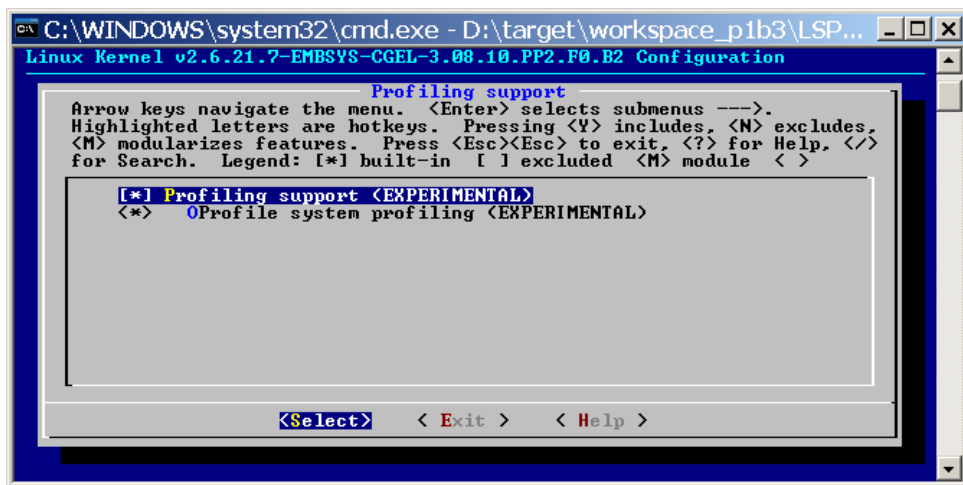


图 2-7 配置 KProfiler 支持

此处【OProfile system profiling】选项配置为编译进内核模式。编入内核的情况下，内核启动后直接加载 OProfile 模块。

➤ X86/x86_64

打开内核配置界面后，选择【Instrumentation Support】这一项，如图 2-8:

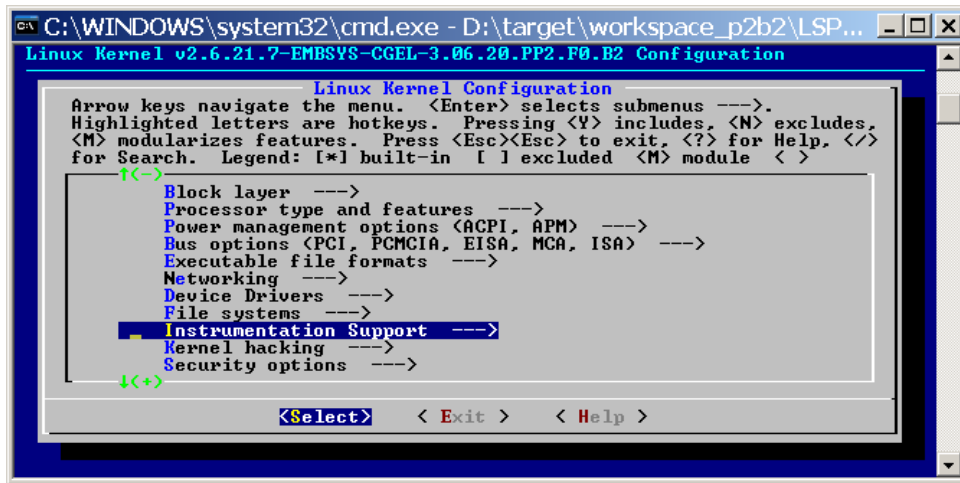


图 2-8 选中【Instrumentation Support】

选中【Profiling support】和【OProfile system profiling】配置选项，如图 2-9：

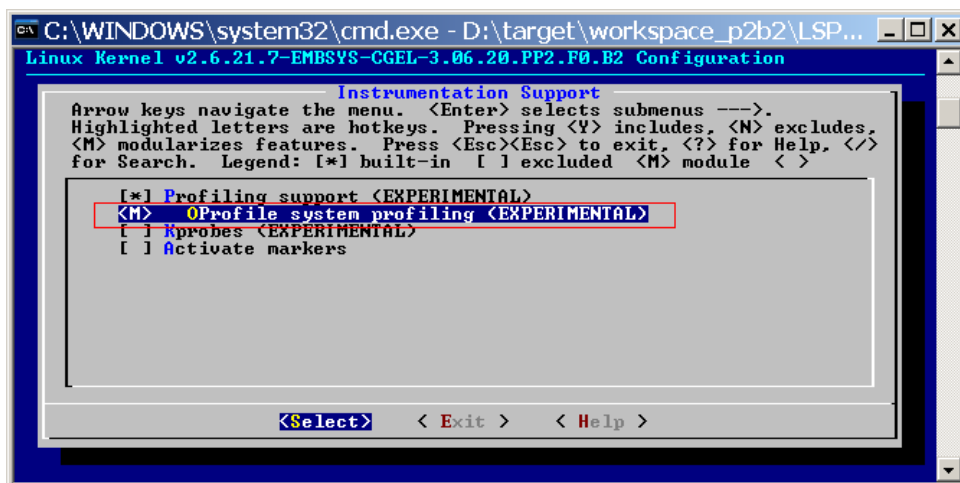


图 2-9 配置 KProfiler 支持 1

【OProfile system profiling】选项可配置为编译进内核或编为模块两种方式。编入内核的情况下，内核启动后直接加载 OProfile 模块；编为模块时，则需在内核启动后，手工插入模块才能启动内核态的 KProfiler 支持。图 2-9 为编成模块的情况。编进内核的配置如图 2-10 所示：

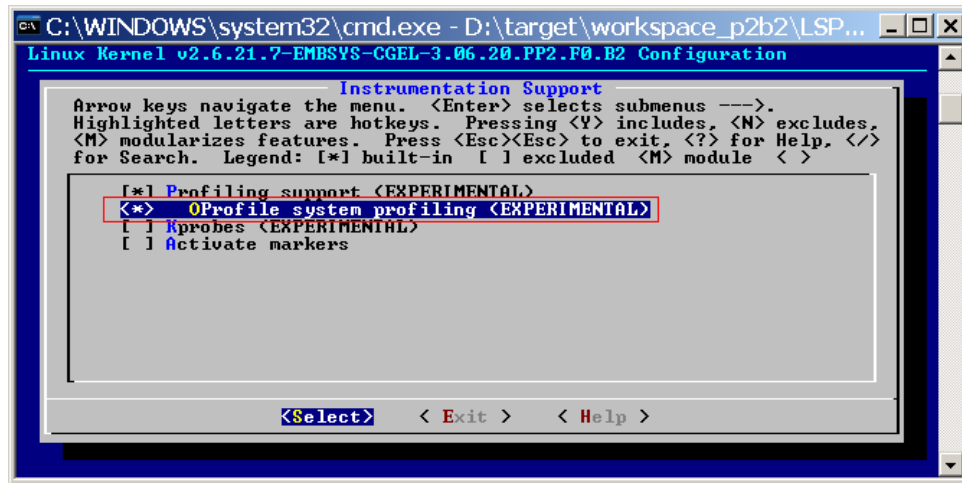


图 2-10 配置 KProfiler 支持 2

➡ 提示：建议将 OProfile 编译成模块，这样操作便于不用重新编译内核，大大节省了编译时间。

返回内核配置初始界面，选中【Kernel hacking】，如图 2-11：

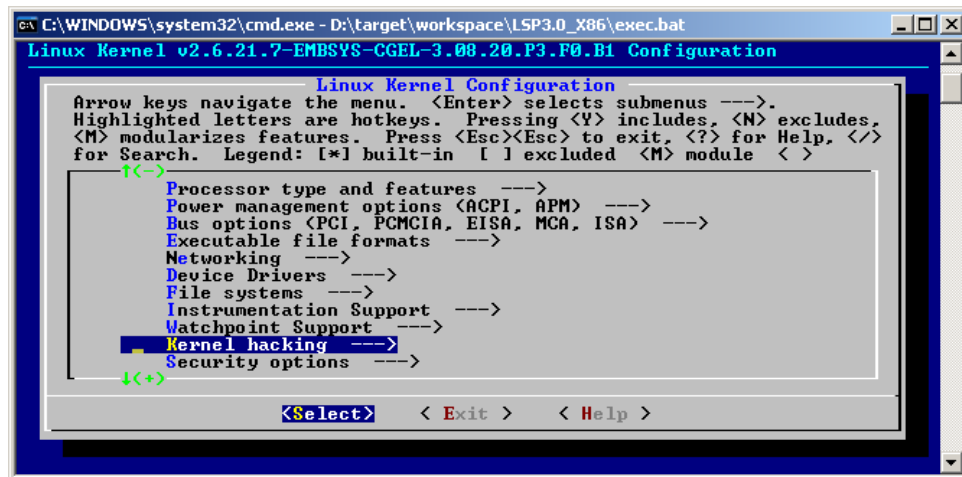


图 2-11 选中【Kernel hacking】

进入后，选中【Compile the kernel with frame pointers】，如图 2-12：

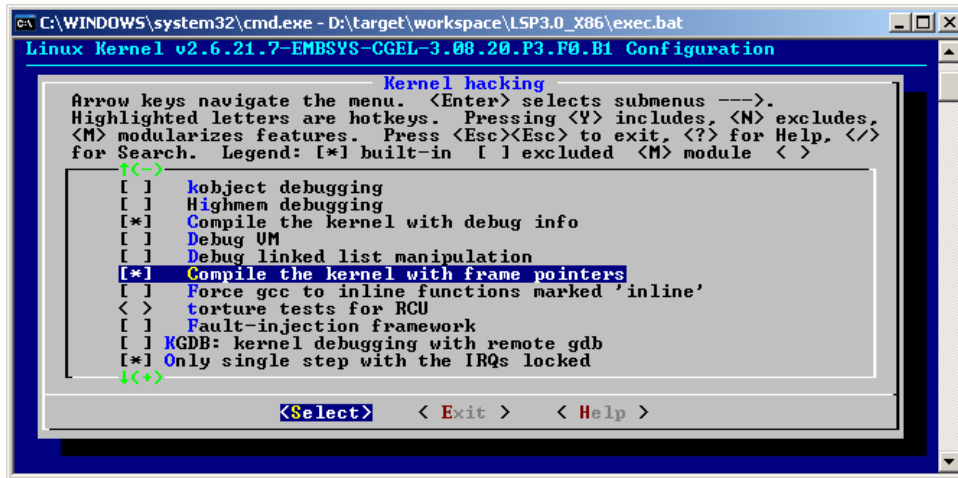


图 2-12 配置 frame-pointer 支持

提示：KProfiler 进行调用关系样本采集时都需要通过读取栈寄存器中的值进行堆栈向上回溯，因此，在构建内核时应配置 frame-pointer 支持，在构建用户态程序时添加 `-fno-omit-frame-pointer` 选项以保存堆栈信息。

➤ MIPS

打开内核配置界面后，选择【Profiling support】这一项，如图 2-6:

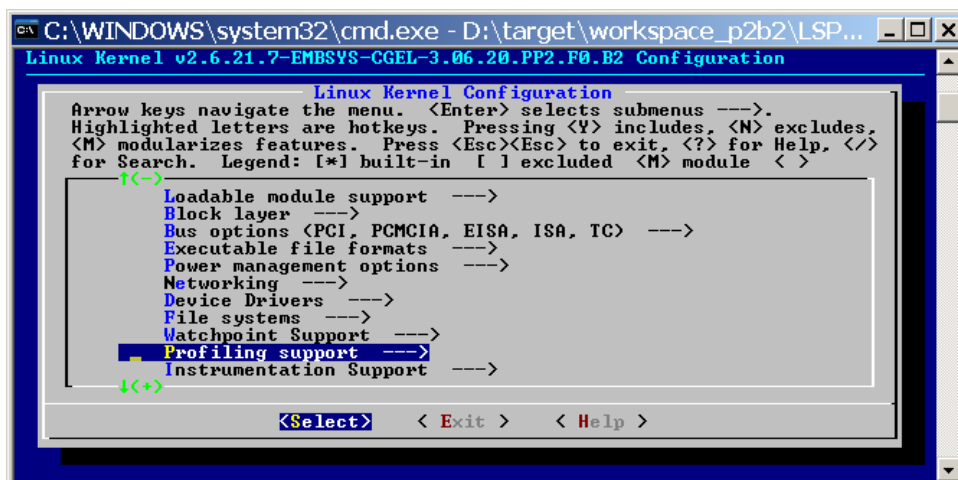


图 2-13 选中【Profiling support】

选中【Profiling support】和【OProfile system profiling】配置选项，如图 2-7：

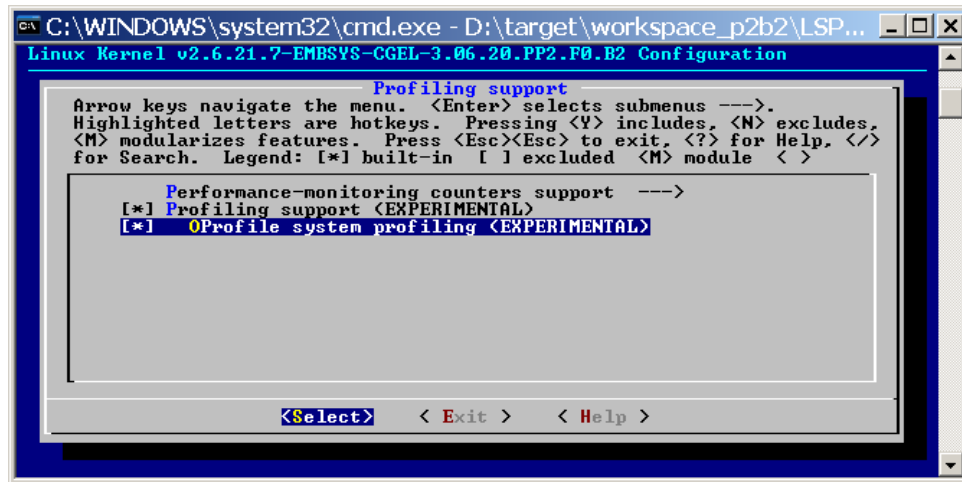


图 2-14 配置 KProfiler 支持

此处【OProfile system profiling】选项配置为编译进内核模式。编入内核的情况下，内核启动后直接加载 OProfile 模块。

➡ 提示：若是用于 mips 体系的 zemu 虚拟机，则由于尚未提供性能计数器支持，所以不能对内核配置 KProfiler 支持。

➤ PPC85xx

ARM、MIPS、X86 和 x86_64 在内核配置的默认菜单中即可找到配置 OProfile 的选项，但在很多其他的体系架构下直接进行内核配置时是找不到 OProfile 的配置入口，这是因为 OProfile 在当前的内核配置中还作为实验选项，要想开启该选项，必须首先将代码成熟度级别配置正确。如图 2-15：

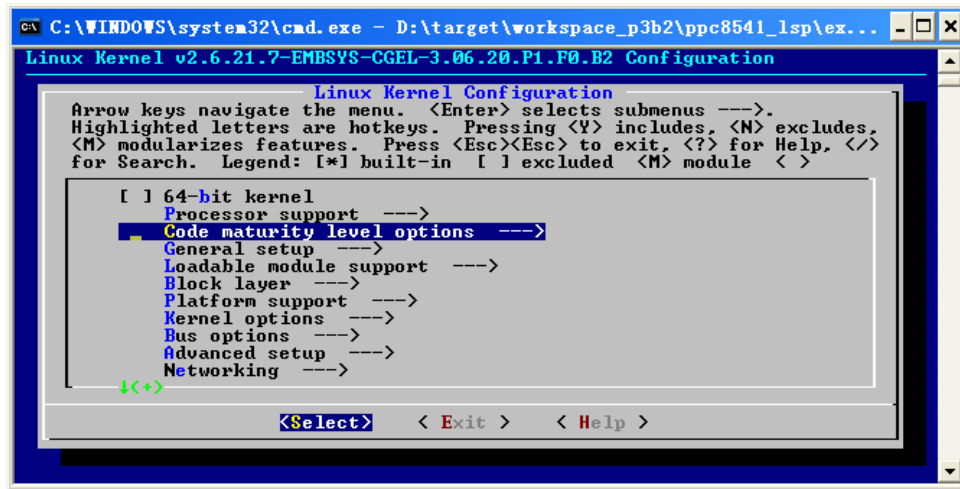


图 2-15 配置代码成熟度级别

进入该配置项，选中【Prompt for development and /or incomplete code/drivers】，如图 2-16：

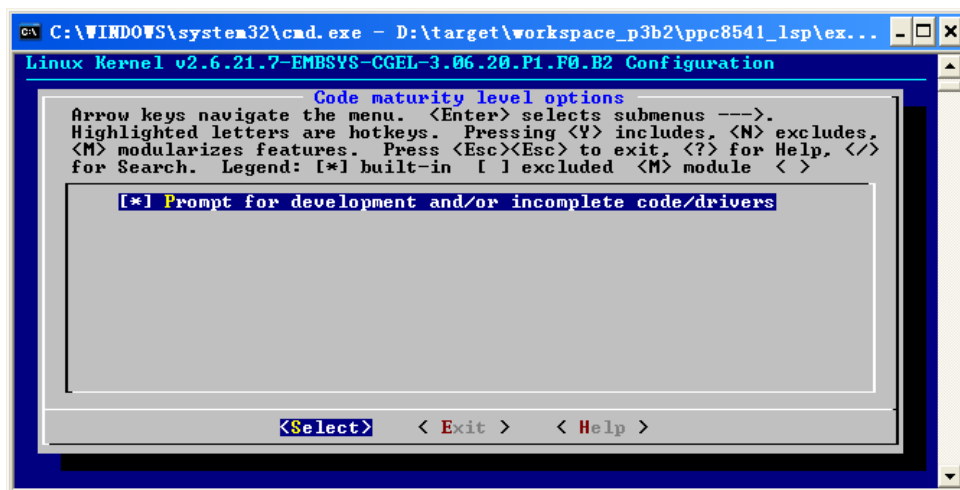


图 2-16 修改代码成熟度级别

然后返回上一层配置菜单，即可查看到【Instrumentation Support】配置项，后续的配置步骤同 x86/x86_64。

 提示：并非只有 PPC85xx 架构需手动配置 OProfile 入口，在其他架构下若是遇到找不到 OProfile 配置入口的情况，请参照如上方式修改代码成熟度级别配置。

PPC85XX 的架构，提供了对 CGEL4.0 内核支持，但其内核配置 KProfiler 支持的方式不同于之前介绍的 CGEL3.0 内核配置方式。

打开内核配置界面后，选择【General setup】这一项，如图 2-17：



图 2-17 配置【General setup】

进入该配置项，选中【OProfile support<EXPERIMENTAL>】，随之打开【OProfile system profiling<EXPERIMENTAL>】选项，请一并对其进行配置选中，如图 2-18，此处【OProfile system profiling<EXPERIMENTAL>】选项配置为编译进内核模式。编入内核的情况下，内核启动后直接加载 OProfile 模块。



图 2-18 配置 KProfiler 支持

■ 构建 LSP 项目

内核配置完成之后，点击 LSP 项目，在右键菜单中选中【构建项目】，完成之后将生成支持 KProfiler 的内核映像文件 bzImage（X86 架构），如图 2-19：

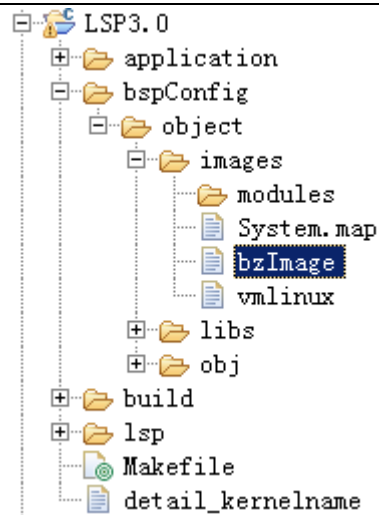


图 2-19 完成 LSP 项目构建

若内核配置时将 oprofile 编译为模块，则构建完 LSP 项目后，会生成 oprofile.ko，如图 2-20:

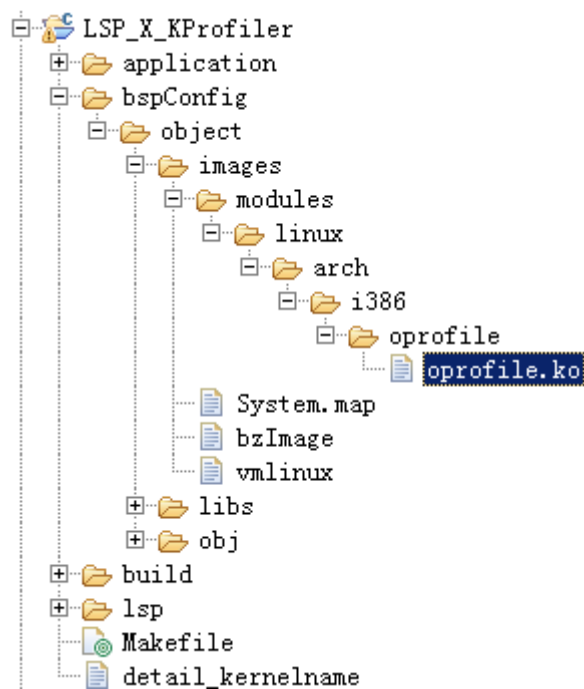


图 2-20 生成 oprofile.ko 模块

2.2. 用户态工具部署

2.2.1. 获取

关于用户态 KProfiler 工具，有两种获取方式：

- 登录 FTP（<ftp://10.75.8.168/ZX-TSP/2012年/KIDE/>，用户名和密码都为 zteuser）获取 KIDE 安装文件，下载安装即可，因为 KProfiler 已经集成到 KIDE 中。
- 登陆南京服务器（<ftp://10.44.27.159/chengyan version/ZX-TSP/2012年/KIDE/>，用户名和密码都为 zteuser）获取 KIDE 安装文件；
- 登录 FTP（<ftp://10.75.8.168/ZX-TSP/2012年/KProfiler/>，用户名和密码都为 zteuser）获取二进制的 KProfiler 用户态工具。

2.2.2. 目录结构

由于用户态 KProfiler 工具有两种获取方式，因而其目录结构也分为 KIDE 和 FTP 两种情况予以介绍。

■ KIDE 下目录结构

KIDE 下 KProfiler 目录结构按照各个架构的构建结果单独放置，提供各架构单板的采样和 cygwin 下的交叉解析功能。目录结构如下所示：

```
kide
|-- kprofiler.x86                                #用于 x86 架构下数据采样和本地解析
|   |-- bin                                    #放置采样数据所需文件
|   |   |-- kprofiler
|   |   |-- ophelp
|   |   |-- oprofiled
|   |
|   |-- share                                  #存放对应架构的各个事件和掩码定义
|   |   |-- oprofile
|   |   |   |-- i386
|   |   ....
|-- kprofiler.x8664                             #除 Win32 有所不同外，其余架构放置文件夹内容均同 x86
|-- kprofiler.armeb
|-- kprofiler.armel
|-- kprofiler.mips64eb
```

```

|-- kprofiler.mips64el
|-- kprofiler.mipseb
|-- kprofiler.mipsel
|-- kprofiler.ppc_generic      #用于通用 ppc 架构数据采样和本地解析
|-- kprofiler.ppc_e500        #针对 ppce500 架构
|--kprofiler.ppc_e500mc       #针对 ppce500mc 架构
|--kprofiler.ppc_e500v2       #针对 ppce500v2 架构
|-- kprofiler.win32           #用于 Windows 环境下交叉解析
|   |-- bin                  #放置解析数据所需文件
|   |   |-- opannotate.exe
|   |   |-- opgprof.exe
|   |   |-- opreport.exe
|   |
|   |-- share                #存放对应架构的各个事件和掩码定义
|   |   |-- oprofile
|   |   |   |-- i386
|   |   |   |-- x86-64
|   |   |   |-- arm
|   |   |   |-- mips
|   |   |   |-- ppc
|   |
|   ....

```

FTP 下目录结构

FTP 下按照各个架构的构建结果单独放置，不同于 KIDE 下的 KProfiler 文件夹，FTP 下 KProfiler 包含了所有功能，可以提供单板的数据采样、本地解析和交叉解析功能，且为便于在单板上使用，文件已被压缩打包。目录结构如下所示：

```

|-- kprofiler.x8664
|   |-- kprofiler.x8664.tar.bz2
|-- kprofiler.x86
|   |-- kprofiler.x86.tar.bz2
|-- kprofiler.armeb
|   |-- kprofiler.armeb.tar.bz2
|-- kprofiler.ppc_generic      #放置通用 ppc 架构数据采样与解析工具
|   |-- kprofiler.ppc_generic.tar.bz2
|-- kprofiler.mips64el

```

```
| |-- kprofiler.mips64el.tar.bz2
|-- kprofiler.mips64eb
| |-- kprofiler.mips64eb.tar.bz2
|-- kprofiler.mipseb
| |-- kprofiler.mipseb.tar.bz2
|-- kprofiler.mipsel
| |-- kprofiler.mipsel.tar.bz2
|-- kprofiler.armel
| |-- kprofiler.armel.tar.bz2
```

2.3. 图形界面 GUI 工具部署

2.3.1. 获取

关于 KProfiler 图形界面 GUI 工具，随同 KIDE 一并发布，可直接登录 FTP（<ftp://10.75.8.168/ZX-TSP/2010年后/KIDE/>，用户名和密码都为 zteuser）获取 KIDE 安装文件，下载安装即可；也可登陆网站 <http://www.gd-linux.com/> 获取；以及登陆南京服务器（<ftp://10.44.27.159/chengyan version/ZX-TSP/2010年后/KIDE/>，用户名和密码都为 zteuser）。

KProfiler 图形界面 GUI 工具包含了 KProfiler 用户态工具，待主机端 KIDE 与目标端完成 KProfiler 配置，并建立链接后，会自行下载 KProfiler 用户态工具到目标端。

3. 命令行基本操作

由于 KProfiler 使用了一个内核模块和一个用户空间守护进程共同完成样本数据的采集到分析，以实现系统性能评测和性能监控。这两者的使用需遵循一定的顺序，以保证整个性能分析过程的顺利实施。具体流程如下：

- 1) 加载 KProfiler 内核；
- 2) 使用用户态工具设置相关采样配置；
- 3) 启动用户空间守护进程；
- 4) 开始收集样本数据；
- 5) 使用用户态工具 opreport 和 opannotate 分析采样数据。

3.1. 收集采样数据

3.1.1. 加载内核 KProfiler


KProfiler 的采样方式有基于性能计数器和定时器两种方式，目前已提供对两种采样方式的动态切换功能。因而，我们只需按其编译方式的不同，分别介绍如何加载内核 KProfiler，对于采样方式，可通过启动用户态工具采样时参数设置或之后的动态切换完成。

■ KProfiler 链接入内核

此方式最为简单，直接启动内核即可。

■ KProfiler 编译为模块

在成功启动内核后，通过使用 **insmod** 或 **modprobe** 命令手动插入模块即可。

 **提示：**若使用 **modprobe** 加载模块，模块文件必须放置在 `/lib/modules/[uname -r]` 目录下，且目录下应有模块依赖文件 `modules.dep`（可通过 **depmod** 命令生成）。

对于已经启动的内核，可以使用如下命令查看内核是否已经配置 KProfiler 支持：

```
cat /proc/filesystems | grep oprofilefs
```

如果输出 **nodev oprofilefs**，则表示当前内核支持 **kprofiler**，否则，请加载 **oprofile.ko** 模块或为内核配置 KProfiler 支持。

3.1.2. 用户态控制工具 kprofiler

KProfiler 提供了一个用户态工具 **kprofiler**，通过启动采样、停止采样、查询采样、打包采样四大命令完成对 KProfiler 整个采样过程的控制。

3.1.2.1. 命令行选项

■ 采样启动命令

```
kprofiler [--start[-e <event> | -t] [-c <backtrace depth>]
```

```
[ -S <blocked_tid or blocked_pid[,blocked_tid or blocked_pid]>]
[ -T <sample time>] [ -s <sample path>] [ -B <cpu_num[,cpu_num]>]
[ -p <separate-opt [,separate-opt]>] [ -r {0|1}] [ -W]
[ -u <"prog [args]">] [ -U <"prog [args]">] [ -P <priority>] [ -k] [ -V] [ -h]
[ -m <[cpu:]threshold>]]
```

常用命令列举:

```
./kprofiler #默认以性能计数器方式启动采样，默认使用-p all 选项，指定采样数据分别按照内核、线程组号、线程号、cpu 号和库进行区分存放，默认使用-c 5 选项，设置调用图深度值为 5，默认使用-r 1 清除之前采样

./kprofiler --start -t #开始采样，-t 表示基于时钟采样，当使用其他参数启动采样一定要添加--start 的参数
```

表 3-1 采样启动命令参数

参数定义	功能与含义	默认值
-e {事件设置}	<p>（该参数需要检查 arch 是否支持硬件计数器方式，该参数与定时器工作方式的-t 参数互斥）</p> <p>该参数设置数据采样的硬件计数器对象，其中事件设置参数的内容格式为：name : count : unitmask : kernel : user</p> <p>Name: 事件名称，如 CPU_CLK_UNHALTED、CYCLES。各 arch 下支持的命令可以通过查询命令获得事件列表；</p> <p>Count: 计数器重置值，即设置的事件发生了指定次数后进行一次采样，也就是采样频率，并清 0；</p> <p>Unitmask: 硬件位屏蔽码，该值与事件名称匹配，同样从查询的事件列表中获取；</p> <p>Kernel: 是否采样内核态，1= 是，0= 否</p> <p>User: 是否采样用户态，1= 是，0= 否</p>	<p>-e default</p> <p>（表示采用默认采样事件，此刻 CPU 将自动调整为非低能耗状态，保证默认事件的产生，采样结束后恢复采样前状态）</p>
-t	<p>（该参数与硬件计数器工作方式的参数-e 互斥）</p> <p>按照系统定时器的方式进行数据采样。该模式所有 arch 都支</p>	<p>-t</p> <p>（当 start 命令缺少-e 与-t 参数时，</p>

	持。	默认采用-e 方式)
-k{0,1}	<p>(该参数仅在内核采用定时器采样方式时才有意义)</p> <p>当选择采用定时器采样时，该参数用于控制内核是否关闭 no_hz 功能。</p> <p>(1，采样内核关闭 no_hz 功能；0，不关闭 no_hz 功能)</p>	-k 1 (默认关闭 no_hz 功能)
-p {type}	<p>该参数设置 kprofile 数据划分规则。其中 type 包含以下划分方式：</p> <ul style="list-style-type: none"> ➤ none: 数据不划分； ➤ library: 按库进行划分； ➤ kernel: 按内核和内核模块进行划分； ➤ thread: 按线程进行划分； ➤ cpu: 按 CPU 进行划分； ➤ all: 以上综合 	-p all (默认为 all，即将数据按照内核，库，线程组号，线程号和 cpu 进行划分)
-c {depth}	该参数设置采样时记录的函数调用栈回溯深度。虽然该回溯深度的设置不受约束，但若堆栈回溯层数越多，采样对系统性能影响就越大。	-c 5 (默认记录 5 层)
-B {CPU 号}	指定 CPU 集绑定，即将守护进程绑定到指定 CPU 号上的 CPU 运行，同时指定多个 CPU 时，用逗号隔开	Null (默认不绑定 CPU 集，且该功能仅在采样环境为多核的情况下使用才有意义)
-T {采样时间}	指定采样时长，采样时长单位为秒，可设置小数点后三位，既支持微妙级的采样	Null (默认不指定采样时长)
-m <[cpu:]threshold> [-d delay]	以系统 CPU 占有率为触发条件开启采样，从而实现对某一个 CPU 或者系统整体的占有率 (默认为整个系统) 进行监控，当占有率超过 threshold 时，触发采样，threshold 取值范围是 [10,90]；delay 参数指明监控占有率时的 CPU 占有率的计算周期，默认是 1 秒，有效值区间[0.2,5]	无

-s {会话保存路径}	指定当前会话相关文件（含数据文件）保存的路径。	-s /var/lib/oprofile （默认保存到该路径）
-S {tid}	指定需要进行阻塞采样的进程/线程。若输入参数中存在多个进程/线程，则使用逗号将参数分隔开来。若输入的是进程号，kprofiler 将获取该进程下的所有线程并进行阻塞采样	Null（默认不采集阻塞任务），具体详见 5.1
-P	指定守护进程优先级，取值范围为 [1-99]	Null（默认为普通优先级）
-W	触发启动采样功能，即KProfiler运行起来之后就一直处于睡眠状态，直到收到SIGUSR1信号才真正启动采样	Null（默认不进行阻塞）
-r {0,1}	是否清除上次采样的数据。 （选择-r 0，则其它子参数-i、-s 等无效）	-r 1 （默认清除上次采样数据）
-u{“目标程序参数”}	运行并采样目标程序。当目标程序运行结束，或者执行 ctrl+c 操作，或者对目标程序/kprofiler 进程执行 kill 操作时，kprofiler 会自动执行停止采样流程，同时目标程序退出。	目标程序需带绝对路径，目标程序及参数需放入双引号中，例如 test.out 程序位于/home 目录下，执行需要带参数 -r 1 --path=/home，则其使用形式如下： -u “/home/test.out -r 1 --path=/home” 默认情况下不使用该参数选项。
-U{“目标程序参数”}	运行并 阻塞 采样，当目标程序运行结束，或者执行 ctrl+c 操作，或者对目标程序/kprofiler 进程执行 kill 操作时，kprofiler 会自动执行停止采样流程，同时目标程序退出。	同-u 参数
-V	设置打印信息级别	Null（默认不设置打印级别）
-h	显示启动模式下的帮助信息	Null（默认不显示帮助信息）

提示：当在内核配置时配置了“NO_HZ”选项时，KProfiler 会自动屏蔽了无时钟滴答功能，以保证计时采样的正常进行。

提示：在 x86 架构上，使用默认事件既 CPU_CLK_UNHALTED 事件进行采样，cpu 进入 idle 流程后，会自动调整为非低能耗状态，保证 CPU_CLK_UNHALTED 事件的产生，采样结束后恢复采样前状态。

提示：-T 参数与-u（或-U）参数相互独立，互不影响。在同时使用了-T 参数与-u（或-U）参数的情况下，如果定时采样的时间到达后但目标程序仍在运行，那么 kprofiler 会结束目标程序的运行并执行停止采样流程；如果定时采样的时间到达前目标程序运行结束，或者执行 ctrl+c 操作，或者对目标程序/kprofiler 进程执行 kill 操作时，kprofiler 会自动执行停止采样流程，同时目标程序退出。

■ 采样停止命令

```
kprofiler --stop [-k][-m][-u] [-z][-h]
```

常用命令列举：

```
./kprofiler --stop -k 1 -u 0      #结束采样，-k 1 杀死守护进程，-u 0 默认保持 oprofilefs 挂载
./kprofiler --stop -z kprofiler  #停止采样，并将采样数据打包为 kprofiler.zip，且放置在和
                                kprofiler 同一级的目录下
```

表 3-2 采样停止命令参数

参数定义	功能与含义	默认值
------	-------	-----

-k {0,1}	是否杀掉采样守护进程。	-k 1 (默认停止采样时即杀掉守护进程)
-m {数据备份文件名}	该参数设置将当前会话路径 (/var/lib/oprofile) 下的当前采样数据文件 (samples/current) 重命名为指定的数据文件名, 并保存在相同目录 /var/lib/oprofile/samples/ 下。 可以用来进行数据下载, 以及避免多次启动采样后的数据覆盖/累加 (由 --start -r 决定) 当前会话默认路径为 /var/lib/oprofile 下, 该默认路径可以由 --start -s 参数设置。	Null (默认不进行数据备份)
-u {0,1}	是否 umount oprofilefs。选择 umount 操作, 则必须杀掉守护进程。	-u 0 (默认停止采样后不取消文件系统挂载)
-z{数据打包文件名}	指定采样数据打包后的压缩包的名字和放置路径 (相对路径和绝对路径都可)	Null (默认不进行打包)
-h	显示结束模式的帮助信息	Null (默认不显示帮助信息)

提示：由于用户执行 **kprofiler --stop** 结束采样后，默认并不会取消 oprofilefs 文件系统的挂载，因而若用户加载 oprofile.ko 内核模块，在结束采样后希望卸载 oprofile.ko 内核模块，则必须先卸载 oprofilefs 文件系统，可使用命令 **umount /dev/oprofile** 取消文件系统挂载。

提示：由于默认不进行采样数据打包，若是希望进行打包，则一定要指定“-z {数据打包文件名}”。例如 -z liu，采样数据压缩后生成 liu.zip 压缩包，且放置在和 kprofiler 同一级的目录下。

■ 采样查询命令

```
kprofiler --show [-E|-S|-v][-h]
```

常用命令列举：

```
./kprofiler --show -S
```

#查询当前采样参数设置情况

表 3-3 采样查询命令参数

参数定义	功能与含义	默认值
-E	查询当前 arch 中支持的硬件计数器事件列表，含事件名称、屏蔽位码等信息。	Null
-S	查询当前采样的参数设置情况。	查询命令无任何子参数时，则默认为查询当前状态。
-v	查询当前工具包的版本信息。	Null
-h	显示查看模式的帮助信息	Null（默认不显示帮助信息）

➡ 提示：采样查询命令的三个参数不能同时使用。

■ 采样打包命令

```
kprofiler --pack [-d][[-n]][-h]
```

常用命令列举：

```
./kprofiler --pack -d /samples -n test
```

#将文件夹/samples 下的文件及子目录打包为 test.zip，
且放置在与 kprofiler 程序同一级目录下

表 3-4 采样打包命令参数

参数定义	功能与含义	默认值
------	-------	-----



-d	指定需要被打包的文件或者文件夹。	默认打包的文件夹/var/lib/oprofile
-n	指定生成的打包文件名字和路径。	默认压缩包名为 kprofiler.zip
-h	显示打包模式的帮助信息	Null（默认不显示帮助信息）

➔ 提示：停止采样后，用户输入打包命令，KProfiler 会将用户指定的文件夹打包为指定名字的.zip 压缩包。当没有停止采样时，用户需要对当前的采样文件进行打包，KProfiler 也会自动停止采样，杀死守护进程后进行打包操作。

➔ 提示：如果用户设置的打包路径下有同名的压缩包，KProfiler 会提示用户是否进行覆盖，用户直接选择覆盖 (overwrite)，不覆盖或者叠加 (append) 即可。

➔ 提示：成功打包后的压缩包后缀为.zip，在 Windows 上可使用 7zip 或 winrar 等工具进行解压。

➔ 提示：用户对采样文件进行打包，不会对 opd_pid 文件进行打包。

3.1.2.2. 使用性能计数器注意事项

■ 事件配置规则

在进行控制采样过程中，对于事件的配置需遵循特定的原则，以保证采样的正常进行。基于性能计数器的采样遵循两个原则：

- 1) 设置的事件个数不能超过性能计数器总数，性能计数器的个数可通过查看/dev/oprofile 路径下的名字为数字的文件夹来进行确认；
- 2) 使用同一个性能计数器的多个事件互斥原则。例如：若事件 1 和事件 2 都只使用编号为 3 的性能计数器记数，那么在一次采样过程中不能同时设置事件 1 和事件 2。

■ 适用场景

目前所有虚拟机均不支持性能计数器的采样方式，所以在虚拟机上使用 KProfiler 时，只能选用定时器的方式进行采样。

■ 调整采样频率

使用默认事件时，KProfiler 启动采样之前，会读取文件 `/proc/cpuinfo` 获取 CPU 的主频，根据主频大小将采样频率调整为每秒钟采样 1000 次。如果 KProfiler 不能通过文件 `/proc/cpuinfo` 来获取 CPU 主频，例如在某些 mips 架构上不能获取 CPU 主频，则采用默认采样频率（发生 600000 次 CYCLES 事件采样一次）。用户也可手动对特定事件的采样频率进行调整，通过使用 `-e` 参数即可实现，如：

```
./kprofiler--start -e CYCLES:10000:0:1:1      #此处时间设置中的 10000 表示当 CYCLES 事件发生  
10000 次采样一次，因而此处采样频率调整为每秒钟采样“CPU 主频/10000”次
```

3.1.2.3. 注意事项

在使用 KProfiler 进行采样时，存在一些容易犯错的问题，导致采样过程无法正常进行，在此处列举出来以示提醒。

3.1.2.4. KProfiler 关键元素存放路径说明

使用 KProfiler 时，请勿将采样的用户态程序符号表放置在很深的目录下。这是由于 KProfile 在生成采样数据文件时就会将这些路径加上自己特定的前缀，缺省时放置在 `/var/lib/oprofile/samples/current` 目录下，`/var/lib/oprofile/` 目录可通过 `kprofiler` 的 `-s` 参数进行修改，指定采样文件存放路径。若采样的用户态程序符号表放置路径太深，则采样出来的数据文件其绝对路径名超过了解析环境限定的最大路径长度（比如在 `cygwin` 上进行交叉解析的时候，`cygwin` 支持的最大路径长度为 260），则统计数据不能正常显示。

3.2. 解析采样数据

3.2.1. 解析采样数据工具

对于搜集到的采样数据，KProfiler 提供了多种分析、统计方式，主要包括概要统计、统计指定程序、综合统计、调用关系统计、源码级统计、指令级统计等。目前，KProfiler 提供了两款用户态工具用于数据的解析，分别为 `opreport` 和 `opannotate`。通过输入不同的参数，KProfiler 将以不同的方式对采样数据进行分析、统计，最终得到用户期望的统计结果。现在，就对这两类解析工具使用的命令参数做详细介绍：

■ `opreport` 解析工具

常用命令列举：

```
./opreport      #仅显示占有率超过 1%的热点程序，对每个热点程序，只显示占有率超过 1%的热点函数

./opreport -c
#仅显示占有率超过 1%的热点程序，对每个热点程序，只显示占有率超过 1%的热点函数，对每个热点函数的父函数，仅显示调用百分比超过 5%的函数

./opreport -l /user.out      #统计应用程序采样情况

./opreport -s /path/test/var/lib/oprofile -c /cg.out -p /test
#统计用户态程序采样情况，-s 指定采样文件本地存放路径，-c /cg.out 针对采样时进程/cg.out 的调用图信息进行统计，/test 为应用程序 cg.out 存放路径

./opreport -s /path/kprofile_test/var/lib/oprofile -l /vmlinux -p /test_vm
#统计内核采样情况，/test_vm 为内核符号表文件 vmlinux 本地存放路径

./opreport cpu:0      #查看 cpu0 上所有进程的样本统计

./opreport tgid:10757      #显示一个进程下面所有线程的样本统计（采样时使用了-p all 选项）

./opreport tid:13809      #显示一个线程的样本统计（采样时使用了-p all 选项）
```

表 3-5 opreport 命令参数列表

参数定义	功能与含义
缺省 default	仅显示占有率超过 1%的热点程序，对每个热点程序，只显示占有率超过 1%的热点函数
-s,--session-dir <path>	指定目标端采样文件或者主机端采样文件本地存放的路径，不指定该项默认路径为：/var/lib/oprofile
-l <symbols[, symbols]>	统计指定程序，如 opreport -l /vmlinux 统计内核采样情况、opreport -l /user.out 统计应用程序采样情况，对内核采样信息统计的同时，会判断 vmlinux 的版本号和构建时间与采样内核是否一致
-c, --callgraph	调用图统计，统计函数调用关系，默认仅显示占有率超过 1%的热点程序，对每个热点程序，只显示占有率超过 1%的热点函数；对每个热点函数的父函数，仅显示调用百分比超过 5%的函数
-g, --debug-info	查看每个符号所属的文件和行数
-d, --details	显示所有选定的符号指令细节，即各符号采样的指令和其采样数
-e, --exclude-symbols<symbols[, symbols]>	在所给的符号表中排除指定的符号

-w, --show-address	显示每个符号的 VMA 地址
-p, --image-path <path>	额外寻找符号表的路径
-b, --blocked-symbols=<blockedsymbols[, blocked symbols]>	统计阻塞任务采样的符号信息，详见 5.1
-O, --consistency	屏蔽 vmlinux 一致性检查功能
-T, --top <top number>	指定显示各个采样符号表的热点函数个数
-t, --top-cg <top number>	指定显示各个采样符号表的热点函数堆栈回溯层数
cpu:<cpu num[, cpu num]>	过滤显示指定 cpu 号的信息
tgid:<processs id[, process id]>	过滤显示指定进程号的信息
tid:<task id[, task id]>	过滤显示指定线程号的信息
event:<event name[,event name]>	过滤显示指定事件的信息
-V, --verbose {all {debug,bfd,level1,sfile,stats}}	给予详细的调试输出
-v / --version	显示版本
-, --help	显示帮助信息
-h, --help-detail	显示详细帮助信息
--usage	显示用法

■ opannotate 解析工具


常用命令列举：

```
./opannotate -S /test/test.out -b /root/liu/ -d /kprofiler/      #指对采样路径为/test/的 test.out 进行
                                                                源码级分析，其编译路径为/root/liu，现在存放源代码的路径为/kprofiler/
./opannotate -a /test/test.out      #指对采样路径为为/test/路径下的 test.out 进行指令级分析
```

表 3-6 opannotate 命令参数列表

参数定义	功能与含义
-S, --source	源码级统计，统计采样数据中源码命中的情况，需要用-d 或者-s 参数指定源码路径，应用程序编译时需加-g 选项
-p, --image-path <path>	额外寻找符号表的路径
-i [symbols] / --include-symbols	解析结果中只包括指定符号的数据
--include-file <filenames>	解析结果中只包含指定文件名的数据
-e, --exclude-symbols <symbols>	排除在所给的符号表中指定的符号
--exclude-file <filenames>	排除在所给的文件名中指定的文件数据
-b, --base-dirs <path>	指定调试信息中带的编译路径，此项需要和--search-dirs 一起使用
-d, --search-dir <path>	本地寻找源文件的路径，如符号表 test 编译的时候的路径为 /cygdrive/e/app/test.c，本地存放的路径为/oprofile/test/test.c，符号表采样的时候的路径为/oprofile/test，则使用 opannotate 源码分析的时候的命令为： ./opannotate -s /oprofile/test -search-dirs=/oprofile/test/ --base-dirs=/cygdrive/e/app/
-A, --blocked-assembly	指令级统计，统计阻塞任务采样数据中汇编语句命中情况
-B, --blocked-source	源码级统计，统计阻塞任务采样数据中源码命中情况
-a, --assembly	指令级统计，统计采样数据中指令（汇编语句）命中情况，应用程序编译时需加-g 选项
-s, --session-dir <path>	指定采样数据存放路径，默认为/var/lib/oprofile

cpu:<cpu num[, cpu num]>	过滤显示指定 cpu 号的信息
tgid:<processs id[, process id]>	过滤显示指定进程号的信息
{archive:<archive-path>}	解析 oparchive 打包的数据
-v, --version	显示版本
-V, --verbose {all {debug,bfd, level1,sfile,stats}}	给予详细的调试输出
-, --help	显示帮助信息
-h, --help-detail	显示详细帮助信息
-n, --consistency	屏蔽 vmlinux 一致性检查功能
--usage	显示用法
--objdump-params <parameters>	指定 objdump 命令所添加的额外参数
-O, --objdump-file	指定 objdump 文件用于指令解析

 提示：通常在解析前使用--session-dir 选项自定义数据文件的存放路径，再次解析时也一定要使用

--session-dir 选项手动指定数据文件存放路径才能保证解析结果的正确，否则解析的为默认路径下的

数据文件。

➔ 提示：对于用 strip 工具剪裁后的符号表，不能使用 KProfiler 分析工具进行符号信息的分析。

➔ 提示：使用指定线程号进行排序是针对某一线程组而言的，所以使用前需指定线程组，可通过在解析命令后添加“tgid:线程组”获取指定线程组数据。

■ 特殊功能

➤ 内核符号表 vmlinux 一致性检查

KProfiler 提供了对内核符号表 vmlinux 一致性检查的功能，以确保是针对同一内核进行采样和解析。即是在对采集到的内核信息进行解析之前，KProfiler 将首先检查所解析的内核符号表文件是否与之前采样的内核保持一致，若不一致，解析将无法继续进行。由于 vmlinux 默认放置于根路径下，若是使用 opreport -l /vmlinux 命令解析内核采样信息时，若其 vmlinux 未放置于根目录下，则需要使用 -p 手动指明 vmlinux 存放路径才能解析。

✧ 屏蔽 vmlinux 一致性检查

可以手动屏蔽 vmlinux 一致性检查功能。当屏蔽此功能后，若出现指定的 vmlinux 和采样内核的版本不一致的情况，只会提示用户，而不会报错退出。

➤ 过滤功能

对于采样文件的解析结果，可以过滤出期望的解析结果，以便对指定数据的查看。

过滤 CPU：在指定 CPU 解析命令后添加 CPU:字符串，用于获取指定 CPU 数据，如：

```
./opreport CPU:4,1  
#只显示 CPU4 和 CPU1 的解析数据
```

过滤线程组：在指定线程组解析命令后添加 tgid:字符串，用于获取指定线程组数据，如：

```
./opreport tgid:336  
#显示线程组号为 336 的所有线程的解析结果
```

过滤线程：在指定线程解析命令后添加 tid:字符串，用于获取指定线程数据，如：

```
./opreport tid:25,57
```

#只显示线程号为 25 和 57 的两个线程的解析结果

3.2.2. 解析场景

针对目标环境的约束，KProfiler 对于所采集到的数据提供了既可选择直接在目标机上进行数据解析，即本地采样数据解析；也可选择下载到主机端后进行分析，即交叉应用数据解析。根据不同的分析场合，选择相应的用户态工具。

使用工具 **opreport** 进行交叉应用数据解析时，有两点需要注意：

- 1) 本地必须拥有**目标端**上采样数据，默认存放在/var/lib/oprofile 路径下(用户采样时可通过 kprofiler 选项-s 来指定)；放置位置可自定义，但一定要保证 abi 文件和 samples 目录在同一级目录下，如 abi 文件放置目录为/tmp/abi，则 samples 目录放置路径一定为/tmp/samples；通常在解析前使用--session-dir 选项自定义数据文件的存放路径，再次解析时也一定要使用--session-dir 选项指明数据文件存放路径才能保证解析结果的正确，否则解析的为默认路径下的数据文件；

如：

```
./opreport --session-dir=/path/test/var/lib/oprofile -c /cg.out -p /test
```

#交叉解析，统计用户态程序采样情况，--session-dir 指定采样文件本地存放路径，-c 指定显示采样进程 cg.out 的调用图统计信息，/test 为应用程序 cg.out 本地存放路径

- 2) 对目标端程序的交叉分析，需保证目标程序的本地放置路径一定要和其目标端放置路径一致，或通过使用--image-path 命令参数来搜索程序。

使用工具 **opannotate** 进行交叉应用数据解析时，有三点需要注意：

- 1) opannotate 工具指定数据文件的方式同 opreport，即使用--search-dir=path 选项来指定源代码放置路径，--base-dirs=path 选项来指定调试信息里带的编译路径；

如：

```
./opannotate -s /oprofile/test --search-dirs=/oprofile/test/ --base-dirs=/cygdrive/e/app/
```

- 2) opannotate 工具无论对程序是源码级还是汇编级分析，拷贝过来的程序路径必须和目标端上程序的放置路径一致，否则需要使用--image-path 命令参数来指定程序的存放路径；
- 3) 使用 opannotate 进行汇编级分析时，需使用本地默认路径为/usr/bin 下的 objdump 工具进行反汇编分析，若是对目标端程序进行分析，则无法使用本地 objdump 对其进行反汇编，所以在此时添加一个--objdump-file 选项用以指定目标端 objdump 文件的全路径来分析目标端程序，此 objdump 文件可在对应架构工具链下的 bin 目录获取。

3.2.3. 符号表路径设置规则

对指定符号表进行解析时，一定要遵循相应的符号表路径设置规则，否则解析结果易出错。由于采样数据的解析分为本地解析和交叉解析两种场景，所以符号表设置规则也按照场景不同进行分类。

在按照解析场景不同对符号表路径设置规则进行了解之前，必须首先区分这两种解析场景，其方法是查看解析命令是否使用了 `--session-dir/-s` 选项，若使用了该选项，则表示是交叉解析，且必须使用 `--image-path/-p` 选项来指定符号表存放路径，否则则为本地解析。

■ 本地解析符号表指定规则

（以下以 `-l` 选项为例，`-c` 和 `-b` 选项指定情况一致）

- 指定带有路径符号表

```
./opreport -l /temp/user.out [-p /path]
```

查看指定的符号表路径（`/temp/`）是否存在待解析的符号表，找到即解析符号采样信息，否则显示告警信息。

- 指定不带路径的符号表

```
./opreport -l user.out [-p /path]
```

到采样路径下去寻找符号表，找到即解析符号信息，否则到 `-p` 指定的路径（不遍历子目录）下查看符号表是否存在，符号表存在且同名可读则解析符号信息。

本地解析时，在采样路径下一般可找到用户态符号表文件（包括库文件），`-p` 一般用来指定 `vmlinux` 以及内核模块的放置路径。

■ 交叉解析符号表指定规则

（以下以 `-l` 选项为例 `-c` 和 `-b` 选项指定情况一致）

交叉解析必须使用 `-p` 指定采样环境的符号表放置路径。

- 指定带路径的符号表


```
./opreport -l /temp/user.out -p /path
```

查看指定的符号表路径（`/temp/`）是否存在待解析的符号表，找到即解析符号采样信息，否则显示告警信息。

- 指定不带有路径的符号表

```
./opreport -l user.out -p /path
```

首先将-p 指定的路径/path 作为根路径和采样路径进行拼接,之后到拼接的路径下去寻找同名字号表,找到即解析符号信息,否则再到-p 指定的路径 (/path) 下查看同名字号表是否存在,存在即解析符号信息。

 提示：当使用-l 指定解析符号表时，解析结果需要把此符号表调用的库，以及陷入内核部分的符号信息都一并解析出来。

3.2.4. 解析结果的统计信息分析

使用 KProfiler 分析工具对采样文件的数据统计之前会显示采样时的统计信息，主要分为以下两方面：

➤ 各 CPU 核上采样的统计信息：

- ✧ received: 单个 cpu 上采样的次数，记录本 cpu 上总共采样的个数，该数据在进入中断处理函数之后就会加一，因而最准确的反映原始采样状态的数据；
- ✧ backtrace(aborted): 单个 cpu 上回溯样本异常终止的个数，如果在回溯的过程当中发现回溯样本 pc 值等于-1，则加一；
- ✧ %(backtrace): 单个 cpu 上回溯样本异常终止的个数占单个 cpu 上采样次数的百分比；
- ✧ overflow: 单个 cpu 上从 cpu_buffer 往 event_buffer 同步时，因同步速度小于采样速度造成的样本丢弃个数；
- ✧ %(overflow): 单个 cpu 上 cpu_buffer 往 event_buffer 同步样本丢失的个数占单个 cpu 上采样次数的百分比；

➤ 总的采样统计：

- ✧ received(total): 统计所有 cpu 上采样的总数，其数据为各个 cpu 上采样总数的累加值；
- ✧ no-mapping(backtrace): 用户态程序堆栈回溯时找不到进程空间的样本个数；
- ✧ %(no-mapping(backtrace)): 统计用户态程序堆栈回溯时找不到进程空间的样本个数占总的采样次数的百分比；
- ✧ event-overflow: 统计守护进程 oprofiled 读/dev/oprofile/buffer 内容时，因读取的速度小于 cpu_buffer 将数据同步到 event_buffer 的速度而丢弃的记录个数；
- ✧ %(event-overflow): 采样过程中的文件缓存溢出次数占总的采样次数的百分比；

- ✧ no-mapping(sample): 统计采样的用户态程序 pc 值找不到对应的虚拟区 (VMA) 的样本个数;
- ✧ %(no-mapping(sample)): 统计用户态程序 pc 值找不到对应的虚拟区 (VMA) 的样本个数占总的采样次数的百分比;
- ✧ error: 统计采样的用户态程序 pc 找不到进程空间的样本个数;
- ✧ %(error): 统计采样中用户态程序 pc 找不到进程空间的样本个数占总的采样次数的百分比。

4. 图形界面基本操作

4.1. 收集采样数据

4.1.1. 加载内核 KProfiler

与命令行方式相同, 参见 3.1.1。

4.1.2. 建立目标机连接

4.1.2.1. 目标机配置

开启目标板, 在确认启动和初始化成功后, 在 KIDE 的目标机视图里点增加目标机配置项。

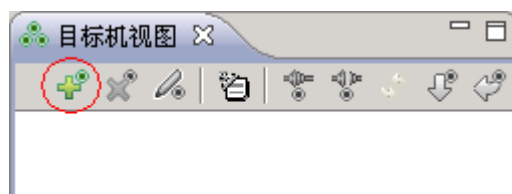


图 4-1 增加调试目标

在图 4-2 弹出窗口中, 为目标机命名, 输入以/ (斜杠) 结尾的目标板 (机) 的缺省目录, 输入目标机端的 IP 地址, 最后点**确定**完成配置。

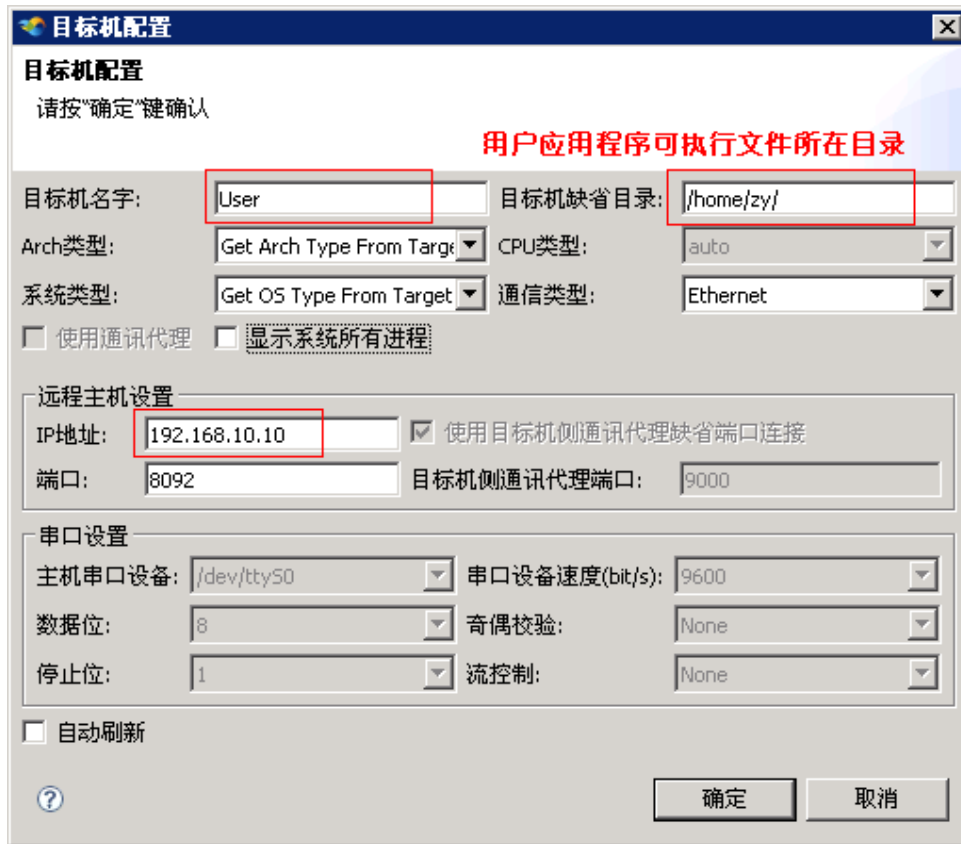


图 4-2 设置目标机配置

4.1.2.2. 目标机连接


目标机连接之前，需要确保目标代理 ztam 已经启动。查看 ztam 版本的方法是运行“ztam -v”命令。

启动 ztam 命令为：

```
./ztam [-v] | [-h] | [agent] | [port] | [-TCP] | [-UDP]
```

参数说明：

- -v: 查看 ztam 版本号；
- -h: 查看 ztam 帮助信息；
- agent: 使用通信代理进行调试；
- -TCP: 采用 TCP 协议；
- -UDP: 采用 UDP 协议；
- port: 指定目标代理 ztam 连接的端口号；

点击按钮来与目标板连接。在 Windows 命令行下用 Ping 命令 ping 目标机 IP，可以查看宿主机和

目标机之间的连通情况。



图 4-3 连接目标板


目标板连接成功后，可以将需要采样的用户态程序下载到目标板，下载成功之后，可以在目标机视图所选的连接下查看显示已下载的对象。

4.1.3. 配置启动采样

对已连接的目标机，点击右键，选择“打开 KProfiler”，开启 Kprofiler 视图。



图 4-4 打开 KProfiler

也可以点击 KIDE 工作台右上角  图标，选 **其他 (O)**，打开所有透视图列表，点击 Kprofiler 透视图图标即可；或者在主菜单-【窗口】->【打开透视图】，选 **其他 (O)** 来开启 Kprofiler 视图。

在开启的 Kprofiler 视图中，点击  按钮，新增一套 Kprofiler 配置项。

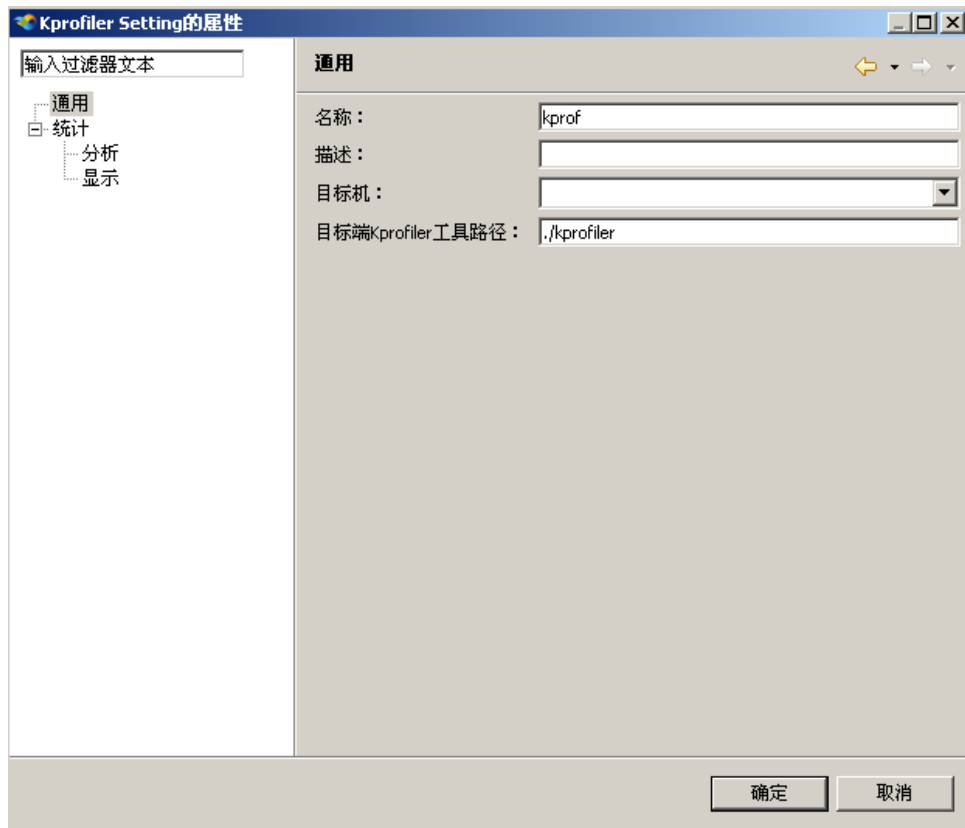


图 4-5 新增 kprofiler 配置

4.1.3.1. 通用

通用选项页，为此 KProfiler 配置添加描述，选择待连接目标机，设置目标端 KProfiler 工具路径，此处设置为针对目标机 ztam 存放位置的相对路径，用户态 kprofiler 工具将会自动下载到此目录下。

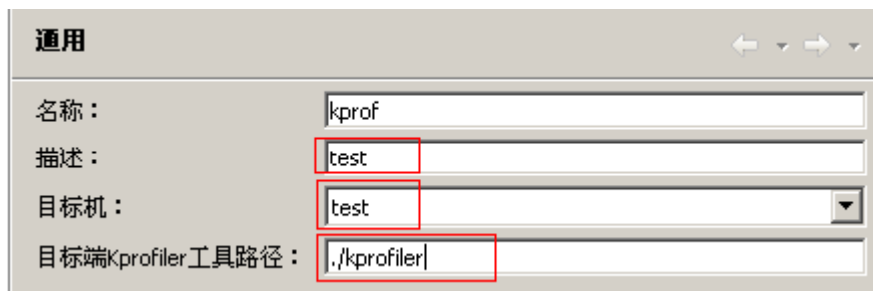


图 4-6 通用选项配置

4.1.3.2. 统计

对于搜集到的采样数据，KProfiler 提供了多种分析、统计方式，主要包括概要统计、统计指定程序、综合统计、调用关系统计、源码级统计、指令级统计等。在统计选项页中，通过设置不同的参数，KProfiler 将以不同的方式对采样数据进行分析、统计，最终得到用户期望的统计结果。

统计选项页，主要对应于命令行的 `opreport` 和 `opannotate` 命令参数设置。

■ 分析

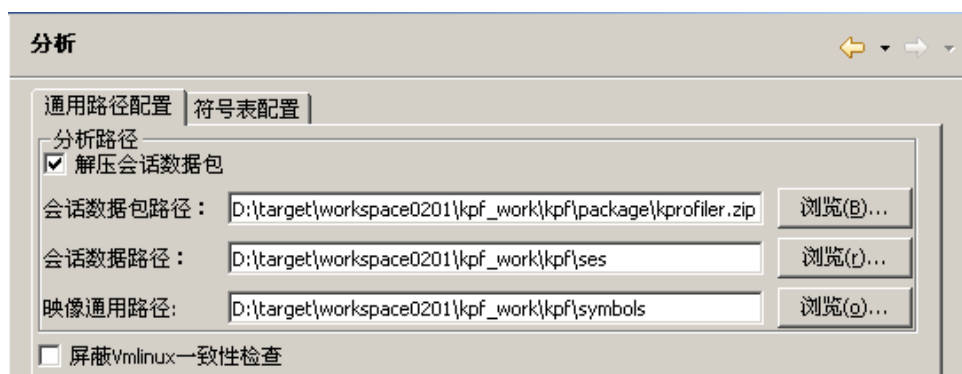


图 4-7 分析选项页参数设置

通用路径设置：

➤ 解压会话数据包

在本地分析数据前，将指定的采样数据包的文件解压到指定的路径中；

➤ 会话数据包路径

此路径下放置目标机采样数据的压缩包文件；

➤ 会话数据路径

此路径下放置对采样数据包解压后的文件；

➤ 映像通用路径

此路径下放置需要解析的的 `elf` 映像和内核映像文件，若设置错误，则无法正确解析映像文件内部函数调用情况；

➤ 屏蔽 vmlinux 一致性检查

KProfiler 提供了对内核符号表 `vmlinux` 一致性检查的功能，以确保是针对同一内核进行采样和解析，但可以手动屏蔽 `vmlinux` 一致性检查功能。当屏蔽此功能后，若出现指定的 `vmlinux` 和采样内核的版本不

一致的情况，只会提示用户，而不会报错退出；



图 4-8 符号表配置项

符号表配置：对指定符号表进行解析时，需要添加额外的符号表路径，在图 4-8 点击**添加**，创建一个新的符号表，图 4-9：需添加符号表路径，设置符号表类型“running/blocked/exclude”，以及为符号表指定模式。

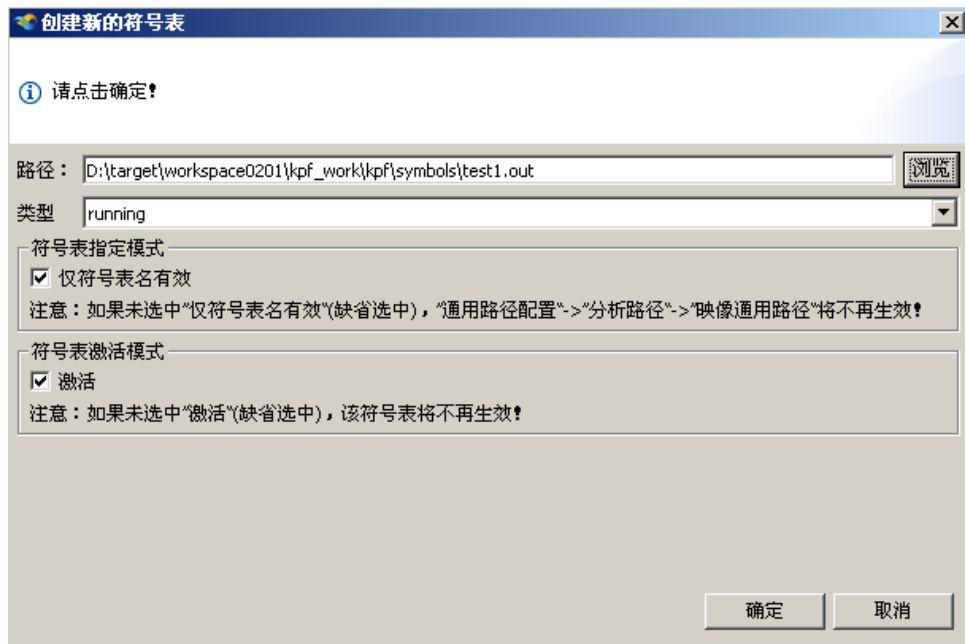


图 4-9 创建新的符号表

■ 显示

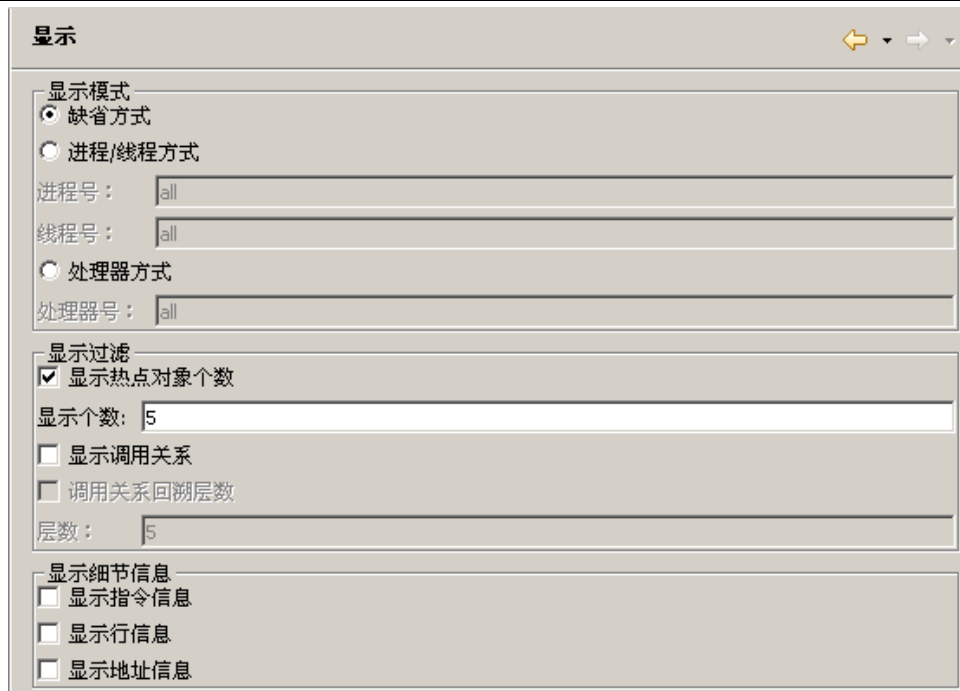


图 4-10 显示选项页参数设置

➤ 显示模式

- ✧ 缺省方式：概要统计；
- ✧ 进程/线程方式：对指定进程/线程进行统计，勾选后，填写具体的进程号/线程号 ID，也可以显示全部进程的数据或者某一进程下所有线程的数据；
- ✧ 处理器方式：指定对某个特定 CPU 进行统计或者显示所有 CPU 数据；

➤ 显示过滤

- ✧ 显示热点对象个数：指定显示各个采样符号表的热点函数个数；
- ✧ 显示调用关系：调用图统计，统计函数调用关系；
- ✧ 调用关系回溯层数：指定显示各个采样符号表的热点函数堆栈回溯层数；

➤ 显示细节

- ✧ 显示指令信息：显示所有选定的符号指令细节，即各符号采样的指令和其采样数；
- ✧ 显示行信息：显示每个符号所属的文件和行数；
- ✧ 显示地址信息：显示每个符号的 VMA 地址。

4.1.3.3. 控制

■ 停止

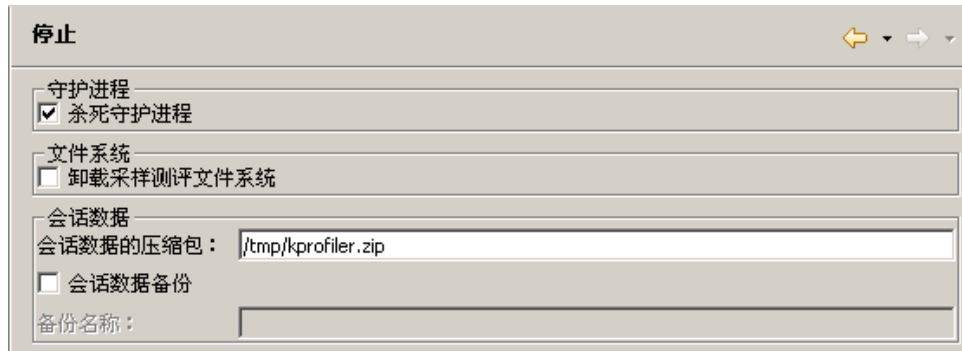


图 4-11 展示了 KProfiler 的“停止”采样参数设置窗口。该窗口包含以下配置项：

- 守护进程**
 - ☒ 杀死守护进程
- 文件系统**
 - ☐ 卸载采样测评文件系统
- 会话数据**
 - 会话数据的压缩包：/tmp/kprofiler.zip
 - ☐ 会话数据备份
 - 备份名称：

图 4-11 停止采样参数设置

- 守护进程：采样结束后是否杀死守护进程；
- 文件系统：采样结束后是否卸载文件系统；
- 会话数据：设置采样结束时将所采样文件压缩打包，并决定是否对采样会话文件（包括采样数据）进行备份；

■ 启动

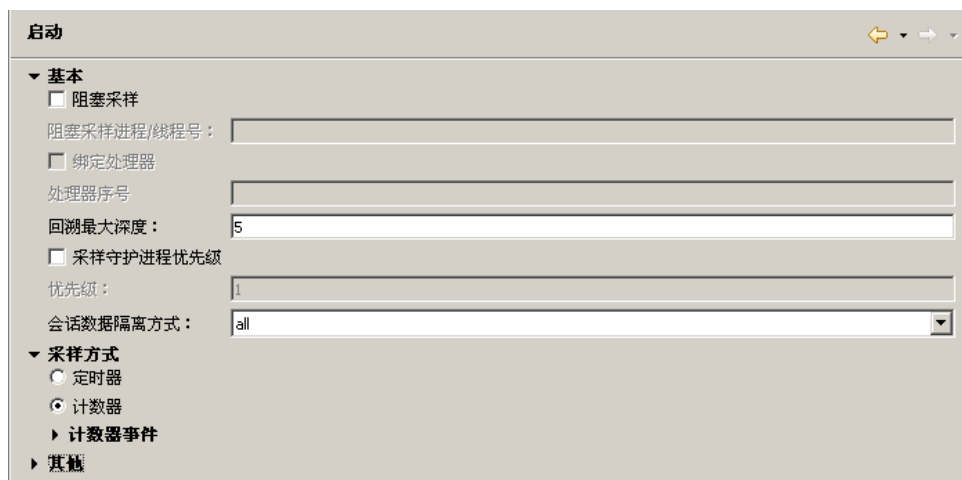


图 4-12 展示了 KProfiler 的“启动”采样参数设置窗口。该窗口包含以下配置项：

- 基本**
 - ☐ 阻塞采样
 - 阻塞采样进程/线程号：
 - ☐ 绑定处理器
 - 处理器序号：
 - 回溯最大深度：5
 - ☐ 采样守护进程优先级
 - 优先级：1
 - 会话数据隔离方式：all
- 采样方式**
 - ☐ 定时器
 - ☒ 计数器
 - 计数器事件
- 其他**

图 4-12 启动采样参数设置

- 基本

- ✧ 阻塞采样：启动阻塞采样，并设置阻塞进程/线程号；
 - ✧ 绑定处理器：指定 CPU 集绑定，即将守护进程绑定到指定 CPU 号上的 CPU 运行，同时指定多个 CPU 时，用逗号隔开；
 - ✧ 回溯最大深度：设置调用图深度，默认为 5 层，与【统计】->【显示】选项页中“显示调用关系”搭配使用；
 - ✧ 采样守护进程优先级：指定守护进程优先级，取值范围为 [1-99]；
 - ✧ 会话数据隔离方式：所有采样数据分别按照 cpu、库、线程号、线程组号和内核区分保存。
- 采样方式
- ✧ 提供定时器采样与计数器采样两种方式，若目标机处理器不支持性能计数器的环境，则此处计数器选项将呈现灰色，无法选中。
 - ✧ 选择计数器的采样方式，则可对采样事件进行选择，在“计数器事件”的列表中进行选择。
- 其他

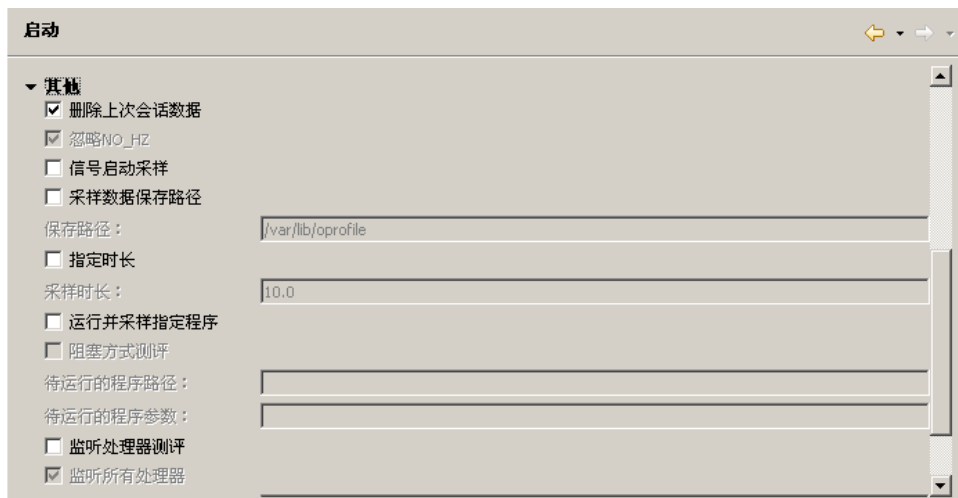


图 4-13 启动选项其他参数配置

- ✧ 删除上次会话数据：清除之前的采样数据；
- ✧ 忽略 NO_HZ：当选择采用定时器采样时，用于控制内核是否关闭 NO_HZ 功能；
- ✧ 信号启动采样：由信号触发启动采样功能，即 KProfiler 运行起来之后就一直处于睡眠状态，直到收到 SIGUSR1 信号才真正启动采样；
- ✧ 采样数据保存路径：指定当前会话相关文件(含数据文件)保存的路径，默认为/var/lib/oprofile；

- ✧ 指定时长：指定采样时长，采样时长单位为秒，可设置小数点后三位，既支持微妙级的采样；
- ✧ 运行并采样指定程序：运行并采样目标程序。当目标程序运行结束，或者执行 `ctrl+c` 操作，或者对目标程序/kprofiler 进程执行 `kill` 操作时，kprofiler 会自动执行停止采样流程，同时目标程序退出；阻塞方式测评：是针对开启运行并采样指定程序后，开启阻塞采样功能；
- ✧ 监听处理器测评：以系统 CPU 占有率为触发条件开启采样，从而实现对某一个 CPU 或者系统整体的占有率（默认为整个系统）进行监控，当占有率超过“处理器占有率临界值”时，触发采样，临界值取值范围是 `[10,90]`；“处理器占有率更新时间间隔”指明监控占有率时的 CPU 占有率的计算周期，默认是 1 秒，有效值区间`[0.2,5]`。

点击**确定**完成一套 kprofiler 配置，该配置可以进行增加、删除、修改、拷贝的操作。

➡ 提示：目前虚拟机都不支持计数器采样方式，但由于 KProfiler 无法区分是运行于真实硬件，还是在虚拟机上运行，所以某些架构的虚拟机在界面上显示可以选择计数器采样方式，望用户避免，以防采样失败。

4.1.4. 控制启动采样

KProfiler 针对每一套配置，都提供了一整套“控制”操作，包括挂载、卸载、启动、停止、下载等 5 项操作，便于用户对整个采样过程的控制，如图 4-14。



图 4-14 控制操作

4.1.4.1. 挂载/卸载

在 KProfiler 视图中，可以在任何一个配置的**右键**菜单里找到各套配置的【控制】->【挂载】和【控制】->【卸载】选项。

挂载操作，主要用于将主机端与目标端进行匹配，从而确保一个主机端在同一时间段内对于同一个目标机有且只有一个采样对象，保证了整个采样的独立完整。

采样结束后，提供**卸载**操作，以便于其他主机端可对目标机的采样对象进行操作。

4.1.4.2. 启动/停止

在 KProfiler 视图中选定某个处于挂载状态的配置，在**右键**菜单中找到【控制】->【启动】操作，启动采样，目标机进入对采样对象的数据收集阶段。

完成数据采样收集之后，选定该配置，在**右键**菜单中选择【控制】->【停止】操作，即可中断采样。

4.1.4.3. 下载

在 KProfiler 视图中，选定仍处于挂载状态配置，在**右键**菜单中选择【控制】->【下载】操作，即可将目标端的采样数据文件下载到本地目标端，方便数据的保留及分析。

4.2. 解析采样数据

停止采样后，KProfiler 会自动打印出数据解析结果供用户查看，如图 4-15：其各统计信息含义请参见 3.2.4。

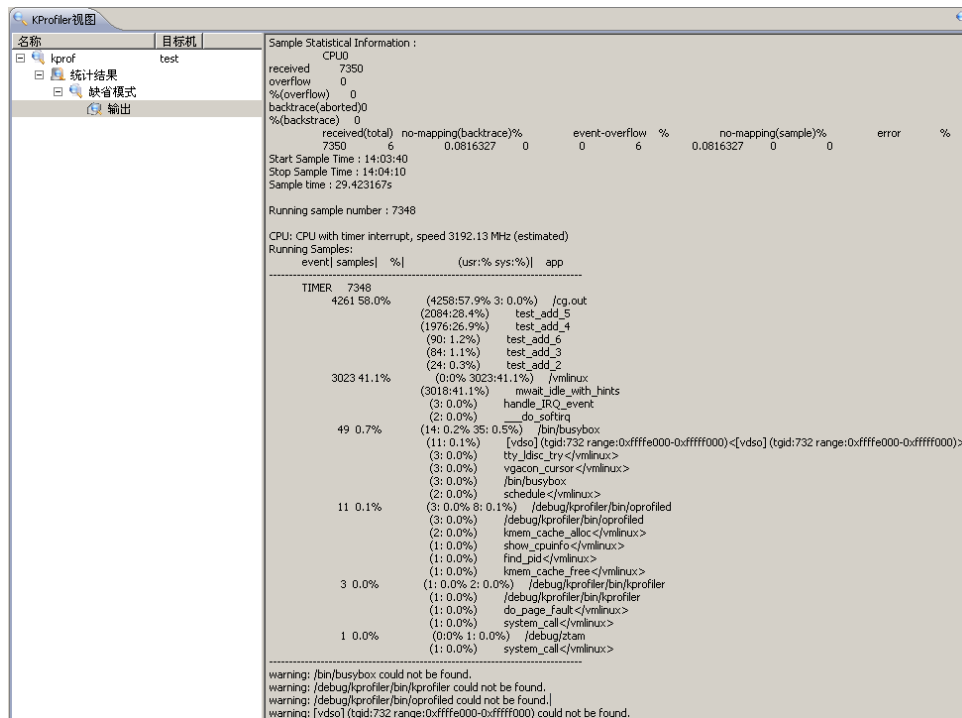


图 4-15 输出解析结果

提示：若是在 KProfiler 配置中，【统计】->【显示】选项页修改了解析结果的显示方式，则请对此配置点击右键，选择【结果】->【显示】，则会更新为最新的显示方式。

KProfiler GUI 图形界面除完成一整套的采样、解析功能外，还可以仅用于解析文件，此时就无需配置目标机，也无需挂载、采样等操作，只需新建 KProfiler 配置，在【统计】->【分析】选项页，为其设置对应的分析路径。最后，在此配置右键菜单选择【结果】->【统计】，则会显示出解析结果。

4.3. 日志管理

KProfiler GUI 图形界面，还提供了日志管理功能，能够将用户在使用 KProfiler 工具进行性能评测时与目标机交互的记录信息保存到日志文件中。日志文件的名称缺省为 kprofilerlog.txt，还可取名为 kprofilerlog_x.txt，其中 x 取值为 1-9。日志文件保存路径为 KIDE 当前工作空间目录下的 log 目录。关于日志文件保存方式，用户可以在 KIDE 首选项中进行设置，即点击 KIDE 主菜单“窗口”->“首选项”->“KProfiler”->“日志”，如图 4-16:

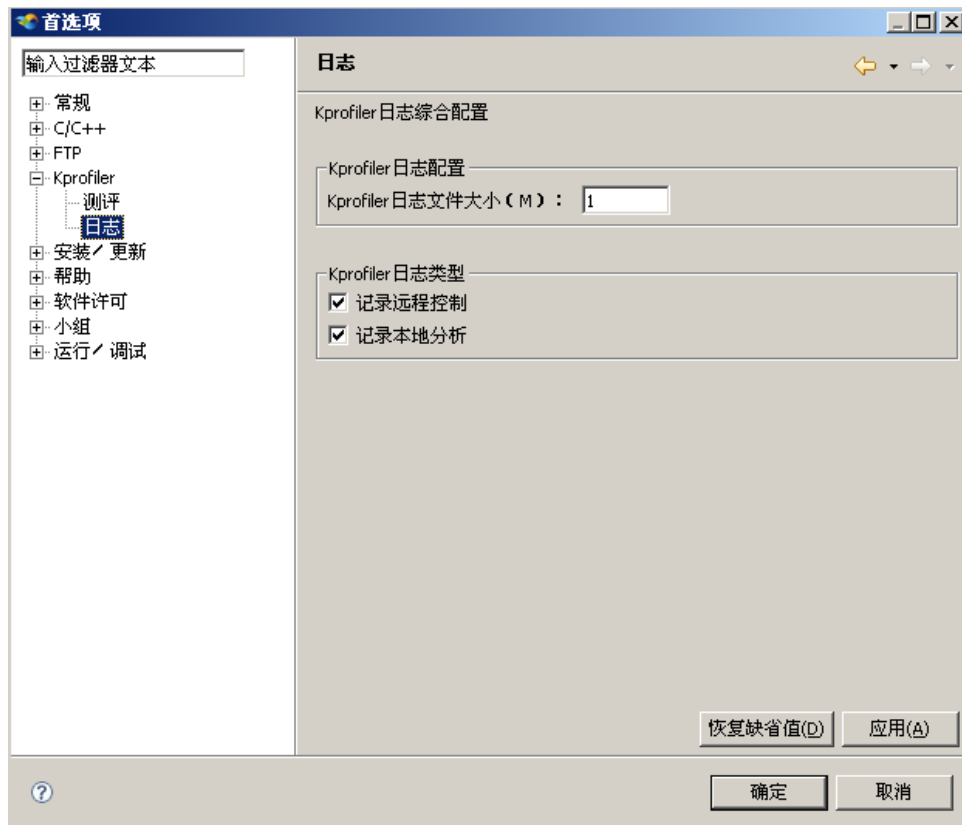


图 4-16 KProfiler 日志设置

5. 高级功能

5.1. 阻塞采样

5.1.1. 应用场景

当 CPU 占有率不高但业务进程处理能力又无法提高的情况下使用阻塞采样。这是由于出现此状况的原因可能是因为资源等待而导致的 CPU 空转，通过使用阻塞采样可以得到采样时段进程阻塞及运行时间的情况和进程阻塞的位置。

5.1.2. 使用介绍

通过使用 `kprofiler -start` 的子参数 `-S` 来指定需要进行阻塞采样的线程号/进程号，若存在多个线程/进程需要指定阻塞采样，则使用逗号将 tid 号分隔开来即可。

如：

```
./kprofiler --start -S 21,69, 74
```

若是使用图形界面，只需在启动配置窗口，【控制】->【启动】选项页，基本设置中，勾选“阻塞采样”，输入需要进行阻塞采样的线程号/进程号，如图 5-1：

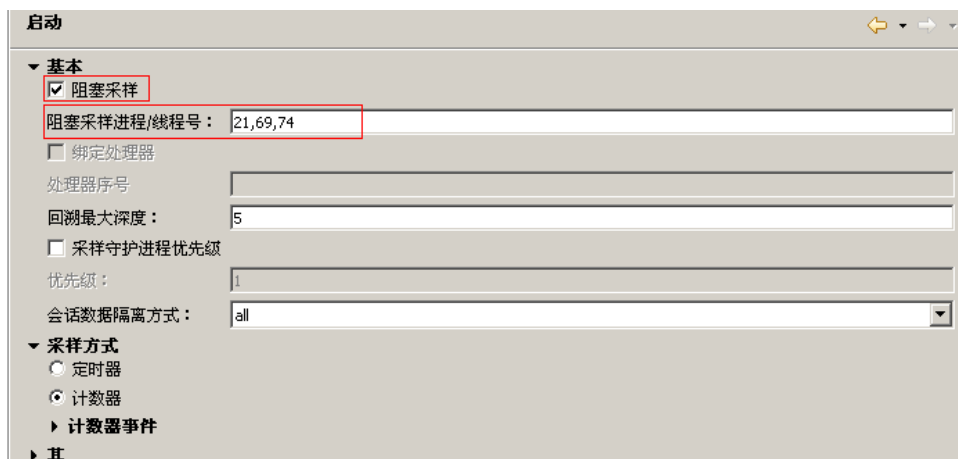


图 5-1 设置阻塞采样配置

如果指定了进程号的话，自动把此进程的子线程号添加到指定的线程组中。

若是存在以下几种情况，将不会进行阻塞采样：

- 1) 若指定的采样线程/进程未阻塞，则无法生成阻塞采样数据；
- 2) 若用户输入的线程号/进程号在当前环境中不存在，将给予“线程/进程不存在”打印提示，内核也将不会对不存在的线程号/进程号进行阻塞采样；
- 3) 若在采样过程中未对阻塞任务进行调度，则无法进行阻塞采样。

5.1.3. 相关命令

■ 启动采样

```
./kprofiler --start -S 21, 69, 74
```

使用 kprofiler 采样工具：

表 5-1 阻塞采样启动命令参数

参数定义	功能与含义
------	-------

-S tid	指定需要进行阻塞采样的进程/线程。若输入参数中存在多个进程/线程，则使用逗号将参数分隔开来。若指定了进程号，则自动把此进程的子线程号添加到指定的线程组中。若输入的是进程号，kprofiler 将获取该进程下的所有线程并进行阻塞采样
--------	---

■ 解析采样数据

主要通过如表 5-2 所示相关解析命令，实现 KProfiler 阻塞采样数据的解析。

表 5-2 阻塞采样相关命令参数

参数定义	功能与含义
opreport -b <block task name>	统计指定阻塞采样任务的符号信息
opannotate -A <block task name>	统计指定阻塞采样任务的指令级信息
opannotate -B <block task name>	统计指定阻塞采样任务的源码级信息

采用图形界面，则只需在完成 kprofiler 配置后，在 KProfiler 视图对此配置依次点击【控制】->【加载】，【控制】->【启动】，结束采样后，点击【控制】->【停止】，即刻自动打印出解析结果。

5.1.4. 典型应用

下面通过一个使用 KProfiler 查找进程阻塞点的典型用例来演示如何进行阻塞采样，方便大家了解 KProfiler 的阻塞采样功能。

■ 阻塞程序源码

```
#include <stdio.h>
#include <unistd.h>
void sleep_time() {
    int s = 0;
    int i;
    for (i = 0; i<5; i++) {
```

```
        s=i+3;
        sleep(3);
    }
}

void sleep_time_test3() {
    int s = 0;
    int i;
    for (i = 0; i<10; i++) {
        s=i+3;
    }
    sleep_time();
}

void sleep_time_test2() {
    int s = 0;
    int i;
    for (i = 0; i<10000; i++) {
        s=i+3;
        sleep_time_test3();
    }
}

void sleep_time_test1() {
    int s = 0;
    int i;
    for (i = 0; i<1000; i++) {
        s=i+3;
    }
    sleep_time_test2();
}

int main() {
    sleep(20);
    sleep_time_test1();
    return 0;
}
```

运行以上代码程序后，虽然是个死循环程序，但是 **cpu** 占用率很低，基本上都是在 **idle** 当中的，该任务的 **cpu** 利用率很低，需要查找当前的资源等待而导致的 **cpu** 空转的原因，通过使用阻塞采样来得到采样时段任务阻塞及运行时间的情况和阻塞的位置。

■ 程序采样、解析

按照如下步骤查找进程阻塞点：

- 1) 运行程序，通过 ps 命令得到此程序的线程号 754：

```
731 root      1104 S    /bin/sw
751 root      1028 S    /bin/sh
753 root      1024 S    /bin/sh
754 root        92 S    ./blocked_liu
755 root     1016 R    ps
```

- 2) 输入 KProfiler 阻塞采样命令，对指定阻塞任务进行采样

```
# ./kprofiler --start -t -c 9 -S 754 -p none      #使用定时器方式进行采样，堆栈回溯深度为 9，针对
                                                754 号线程进行回溯采样，不设置任何的测评区分

Using 2.6+ OProfile kernel interface.
Daemon started.
Start sample time : 14:02:20
Profiler running.
```

- 3) 采样一段时间后结束采样

```
# ./kprofiler --stop

Using 2.6+ OProfile kernel interface.
Stopping profiling.
Stop sample time : 14:04:20
Sample time : 119.910790s
Killing daemon.
```

- 4) 使用解析工具对阻塞采样数据进行解析

首先，直接运行 oprofile 查看其概要信息：

```
./oprofile
```

结果如下所示：

```
Sample Statistical Information :

                                CPU0
received                        30009
overflow                        0
```

```

%(overflow)      0
backtrace(aborted)0
%(backtrace)     0
received(total)  no-mapping(backtrace) %  event-overflow %  no-mapping(sample) %  error %
30009           0           0  0           0  0           0  0  0
Start Sample Time : 14:02:20
Stop Sample Time : 14:04:20
Sample time : 119.910790s

Running sample number :29125           #运行时采样次数为 29125 次
Block sample number : 28684           #阻塞时采样次数为 28684 次
Schedule number :2019                 #内核调度次数为 2019 次

-----

CPU: CPU with timer interrupt, speed 2310.72 MHz (estimated)
Profiling through timer interrupt
Running Samples:                       #运行采样数据
Event|  samples|      %|      (usr:% sys:%)|  app
-----
TIMER    29125
          28535      97.7      (0: 0 28535: 100) /vmlinux
                               (28535: 100)      /vmlinux
          588        2.0      (588:2.0 0:0) /oprofiled
                               (588: 2.0)      /oprofiled
.....
Blocked Samples:
Event|  samples|      %|      (usr:% sys:%)|  app
-----
TIMER    28684
          28684      100      (28684:100 0:0) /blocked_liu
                               (28684:100)      nanosleep

```

从解析结果，我们可以发现 KProfiler 采样到的阻塞信息全部发生在 `nanosleep` 这个系统调用函数中，采样次数为 28684 次。但是仅从以上得到的阻塞信息无法获取到底是程序的那一点上发生了对 `nanosleep` 函数的调用，导致阻塞。因而我们需要查看其回溯信息来得到程序中的阻塞点。

输入如下命令：

```
./opreport -c /output/allbin/blocked_liu
```

能够得到调用 nanosleep 函数发生阻塞的具体函数：

Blocked Callgraph Sampling:

Event	samples	%	(usr:% sys:%)	app

TIMER	28684			
	28684	100	(28684:100 0:0)	/blocked_liu
			(28684:100)	nanosleep
			__(28684:100.0%)	sleep

从解析结果，我们可以看出，nanosleep 这个系统调用全部是通过 sleep 函数，而 sleep 函数绝大部分时间被 sleep_time 这个函数调用：

```
(0:0)    sleep
         |__(25517: 88.9%)  sleep_time
         |__(3167: 11.0%)  main
```

可以看到采样过程中，sleep_time 调用 sleep 的次数占 sleep 所有被调用次数的 88.9%，main 调用 sleep_time 的次数占 sleep_time 所有被调用次数的 11%，所以阻塞主要是发生在 sleep_time 这个函数当中。

且从回溯结果中还可以看到 sleep_time 的调用关系，这使得定位阻塞热点更加容易。

■ 解析结果分析总结

使用 KProfiler 对程序进行阻塞采样后，从解析的结果我们可以得到：

- 1) 此程序有阻塞发生，且全部阻塞发生在系统调用函数 nanosleep 上；
- 2) 系统调用 nanosleep 全部是通过 sleep 函数进行调用的；
- 3) 阻塞点 sleep 被 sleep_time 调用次数占 sleep 所有被调用次数的 88.9%，而 sleep_time 被 main 函数调用次数占 sleep_time 所有被调用次数的 11.0%，所以程序真正的阻塞热点发生在 sleep_time 这个函数当中。

5.2. 函数调用关系

5.2.1. 应用场景

函数调用关系通常应用于寻找热点路径。

情况一、当找到热点函数后，发现该函数的处理非常简单，则导致其成为热点函数的原因就可能是该函数被频繁调用。这时可通过调用关系找到该函数被调用最多的路径，采取有针对性的优化措施。

情况二、记录到的热点函数的确消耗大量 CPU，但该函数在很多流程中均会使用，无法直接定位是哪个流程处理低效。利用函数调用关系，则可以找到最热点的流程分支，进行重点分析。通常为内核系统调用函数、库函数等等，如 `memset`、`spin_lock` 等等。

5.2.2. 使用介绍

在启动采样添加 `-c` 启动参数，用于设置采样时记录的函数调用栈回溯深度；并在解析采样时也添加 `-c` 参数，用于统计函数调用关系。

若是使用图形界面，需在启动配置窗口，【控制】->【启动】选项页，基本设置中，输入“回溯最大深度”值，如图 5-1：

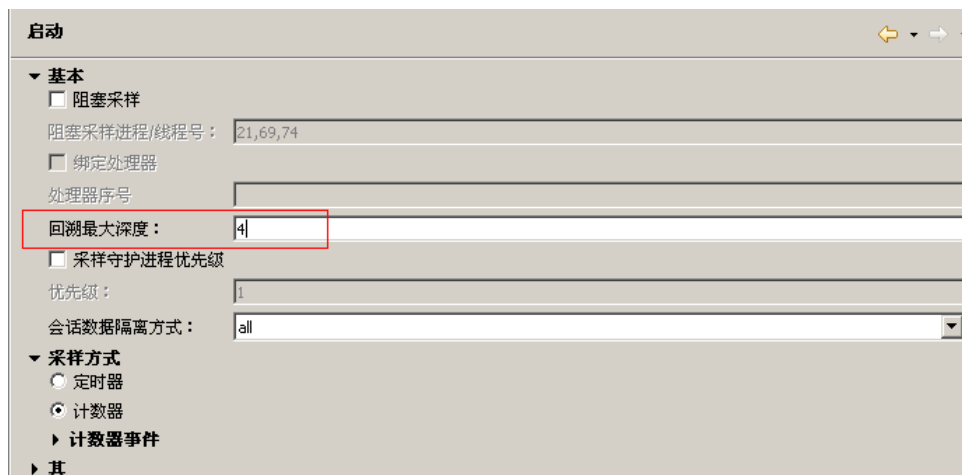


图 5-2 设置采样回溯深度

在【统计】->【显示】选项页，依次勾选“显示调用关系”、“调用关系回溯层数”，输入回溯层数，如图 5-1：

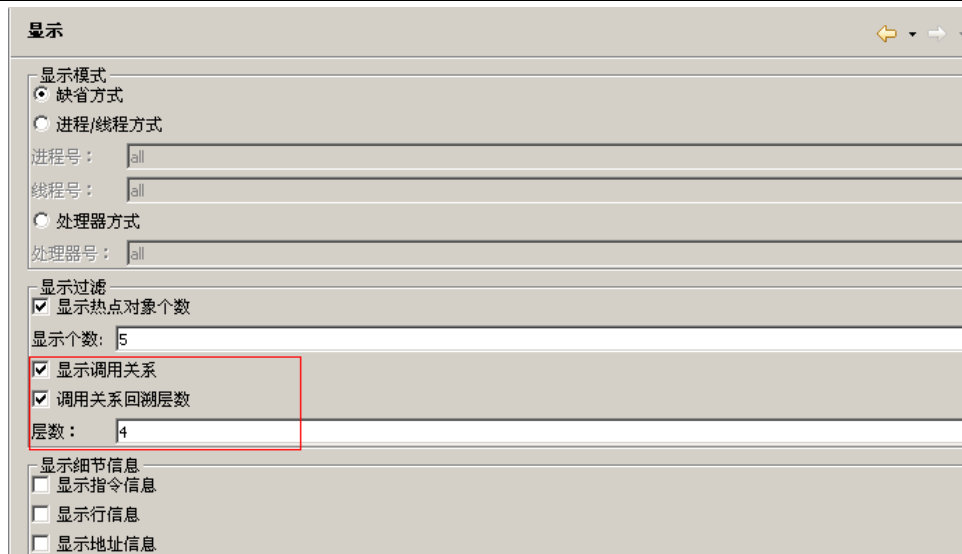


图 5-3 设置显示统计函数调用关系

5.2.3. 相关命令

■ 启动采样

```
./kprofiler --start -c 4
```

使用 kprofiler 采样工具。

表 5-3 函数调用启动命令参数

参数定义	功能与含义
-c/--callgraph	调用图统计，统计函数调用关系，默认设置为-c 5

■ 解析采样数据

使用 opreport 解析工具。

表 5-4 函数调用解析命令相关参数

参数定义	功能与含义
------	-------

-c / --callgraph

调用图统计，统计函数调用关系

采用图形界面，则只需在完成 kprofiler 配置后，在 KProfiler 视图中对此配置依次点击【控制】->【加载】，【控制】->【启动】，结束采样后，点击【控制】->【停止】，即刻自动打印出解析结果。

5.2.4. 典型应用

应用示例：

```

#./kprofiler                                #启动计数器采样， 默认使用-c 5 设置调用图深度值，默认使用-p all 选项
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.

#./kprofiler --stop                          #结束采样
Stopping profiling.
Killing daemon.

#./opreport -c /cg.out                      #开始解析， -c 统计函数调用关系
    
```

由于我们采用堆栈回溯的方式对此次采样数据进行解析，现通过对一个区段内的解析结果进行分析，帮助大家理解采样的情况，以及在整个回溯过程中，函数的调用关系，其他的以此类推即可。

```

-----
(2135:49.9%)      test_add_5
                  |__(2160:99.0%)    test_add_4
                  | |__(4120:99.9%)    test_add_3
                  | | |__(4209:99.8%)    test_add_2
                  | | | |__(4251:100%)    test_add_1
                  #从以上解析显示的数据中可以看到起调用序列为 test_add_1 -> test_add_2 ->test_add_3
                  ->test_add_4 -> test_add_5
    -----
    
```

6. 解析数据解读

KProfiler 提供的解析采样数据工具（具体使用请参见 3.2.1）可以对采样数据进行热点函数分析、函数调用关系分析、采样函数源码级分析等，以方便用户从不同角度分析问题。

为了加深用户对采样数据解析结果的理解，下面以实例的形式对采样数据的不同解析结果进行详细解读。

6.1. 热点函数数据解析

热点函数数据解析是对程序运行过程中各函数占用 CPU 情况的统计，它反映了函数消耗 CPU 时间的分布状态。占用 CPU 越多，说明函数的优化潜力越大，对它进行优化带来的性能改进会越大。热点函数数据解析默认只显示占有率超过 1% 的函数。

```
-----
#./opreport  #解析命令，使用 opreport 不带任何参数，表示进行热点函数解析
The command line of sampling is: ./kprofiler --start -t -S 770 -T 30  #显示采样时使用的命令
#以下为样本解析结果的表头
Sample Statistical Information :  #样本统计信息

                CPU0      #各 cpu 统计结果，如果有多个 cpu，则此处有多列
received        7503      #cpu0 共采集到的样本个数
overflow        0         #cpu0 上因溢出而丢弃的样本个数，消除这个溢出值的方法见 9.9
%(overflow)     0         #cpu0 上因溢出而丢弃的样本个数占本 cpu 总样本数的百分比
backtrace(aborted) 0      #堆栈回溯过程中的失败次数
%(backtrace)    0         #堆栈回溯过程中的失败次数占此 CPU 上采样个数的百分比
received(total)          no-mapping(backtrace)  %
#总样本数，各 cpu 样本之和      #堆栈回溯时找不到对应地址映射的样本个数及百分比
event-overflow  %              no-mapping(sample)  %
#event-overflow 的样本数及百分比，见 9.10      #样本中（非堆栈回溯数据）找不到对应地址映射的样
本个数及百分比
error  %
#出错的样本个数及百分比
7503                0  0
0  0                0  0
0  0
Start Sample Time : 09:46:33  #开始采样时间
```

```

Stop Sample Time : 09:47:03    #结束采样时间
Sample time : 30.107982s      #整个采样持续时间

Running sample number : 7503    #采集到的运行时样本个数，与阻塞样本个数对应
Block sample number : 7162     #阻塞样本个数
Schedule number : 8           #阻塞采样时被阻塞进程的调度次数

opreport command line is: ./opreport    #解析样本数据时使用的命令
CPU: CPU with timer interrupt, speed 3093.08 MHz (estimated)
#使用的采样方式，本例基于 timer 方式，及 cpu 主频信息

#以下为样本解析的详细结果：
Running Samples:                #运行时样本的详细信息
event| samples| %|              (usr:% sys:%)| app
#事件名，本例是 timer 方式所以显示为 TIMER    #用户态样本个数及百分比 内核态样本个数及百分比
进程名称
-----
TIMER      7503    #总的运行时样本数据
           3866  51.5%      (0:0% 3866:51.5%)    /vmlinux
#纯内核态样本个数，即不是用户程序陷入内核态的样本，有 3866 个，占总数的 51.5%
           (3866:51.5%)      mwait_idle_with_hints
#纯内核态的样本中，mwait_idle_with_hints 函数的样本个数 3866，占样本总数的 51.5%
           1256  16.7%      (1256:16.7% 0:0)    /app1
#app1 进程的样本为 1256 个，占总数的 16.7%，其中用户态 1256 个，占总数 16.7%
           (780:10.4%)      memcpy</lib/libc-2.5.so>
#采集到 memcpy 中的样本有 780 个，占样本总数的 10.4%，memcpy 位于/lib/libc-2.5.so 这个文件中
           (328:4.4%)      func1
-----

Blocked Samples:                #阻塞时样本的详细信息，以下各字段、数据含义与上述类似。注：若采样时没
有配置阻塞采样参数，则不会显示阻塞采样结果
event| samples| %|              (usr:% sys:%)| app
-----
TIMER      7162
           7162  100%      (7162:100% 0:0%)    /block_test_32.out
           (7162:100%)      __nanosleep_nocancel

```

6.2. 函数调用关系数据解析

函数调用关系数据解析就是分析热点程序中热点函数被其它函数调用的调用链关系，以及父函数调用子函数的次数与百分比。调用关系的显示是在热点函数的基础上进行的，对每个显示出来的热点函数，以树形结构在其下方展示整个调用链，其中：

1、热点函数（即树的根）那行的数据与热点函数结果中的含义相同

2、父函数在子函数的下方

3、以竖线连接的函数标识为同一级的调用关系

4、树节点（不包括根）中，括号中的数字，前一个代表父函数调用子函数的次数，百分比表示在同级别的调用中，该调用关系占用的百分比，百分比越高表示调用的越频繁

```
-----
# ./opreport -c #解析命令，使用 opreport -c，表示进行函数调用关系解析
The command line of sampling is: ./kprofiler --start -t -S 770 -T 30 #显示采样时使用的采样命令
#以下为样本解析结果的表头
Sample Statistical Information : #样本统计信息
                                CPU0      #各 cpu 统计结果，如果有多个 cpu，则此处有多列
received      7503      #cpu0 共采集到的样本数
overflow      0          #cpu0 上因溢出而丢弃的样本数，消除这个溢出值的方法见 9.9
%(overflow)   0          #cpu0 上因溢出而丢弃的样本数占本 cpu 总样本数的百分比
backtrace(aborted) 0      #堆栈回溯过程中的失败次数
%(backtrace)  0          #堆栈回溯过程中的失败次数占此 CPU 上采样数的百分比
received(total)              no-mapping(backtrace) %
#总样本数，各 cpu 样本之和      #堆栈回溯时找不到对应地址映射的样本个数及百分比
event-overflow %              no-mapping(sample) %
#event-overflow 的样本数及百分比，见 9.10 #样本中（非堆栈回溯数据）找不到对应地址映射的样本个数及百分比
error %
#出错的样本个数及百分比
7503              0 0
0 0              0 0
0 0
Start Sample Time : 09:46:33 #开始采样时间
Stop Sample Time : 09:47:03 #结束采样时间
Sample time : 30.107982s     #整个采样持续时间
```

Running sample number : 7503 #采集到的运行时样本个数，与阻塞样本个数对应

Block sample number : 7162 #阻塞样本个数

Schedule number : 8 #阻塞采样时被阻塞进程的调度次数

opreport command line is: ./opreport -c #解析样本数据时使用的命令

CPU: CPU with timer interrupt, speed 3093.08 MHz (estimated)

#使用的采样方式，本例基于 timer 方式，及 cpu 主频信息

#以下为样本解析的详细结果：

Running Callgraph Sampling: #运行时样本的函数调用关系信息

event| samples| %| (usr:% sys:%)| app

#事件名，本例是 timer 方式所以显示为 TIMER #用户态样本个数及百分比 内核态样本个数及百分比
进程名称

TIMER 7503 #总的运行时样本数据

3866 51.5% (0:0% 3866:51.5%) /vmlinux

#纯内核态样本数有 3866 个，占总数的 51.5%

① (3866:51.5%) mwait_idle_with_hints

② |__ (3866:100%) mwait_idle

③ |__ (3866:100%) cpu_idle

① 纯内核态的样本中， mwait_idle_with_hints 函数的样本个数 3866，占样本总数的 51.5%

② mwait_idle 调用 3866 次 mwait_idle_with_hints，占 mwait_idle_with_hints 总的被调用次数的 100%

③ cpu_idle 调用 3866 次 mwait_idle，占 mwait_idle 总的被调用次数的 100%

1256 16.7% (1256:16.7% 0:0) /app1

#用户态样本数有 1256 个，占总数的 16.7%

(780:10.4%) memcpy</lib/libc-2.5.so> #以下是 memcpy 的调用关系

④ |__ (588:76.3%) test_func1

⑤ | |__ (472:80.3%) test_func 2

⑥ | | |__ (472:100%) test_func 3

⑦ | |__ (116:19.7%) test_func 5

⑧ |__ (184:23.6%) cpy_test 1

⑨ |__ (184:100%) cpy_test 2

④ test_func1 调用 588 次 memcpy，占 memcpy 总的被调用次数的 76.3%

⑤ test_func2 调用 472 次 test_func1，占 test_func1 总的被调用次数的 80.3%

⑥ test_func3 调用 472 次 test_func2，占 test_func2 总的被调用次数的 100%

⑦ test_func5 调用 116 次 test_func1，占 test_func1 总的被调用次数的 19.7%

⑧ cpy_test1 调用 184 次 memcpy，占 memcpy 总的被调用次数的 23.6%

⑨ cpy_test2 调用 184 次 cpy_test1，占 cpy_test1 总的被调用次数的 100%

(328:4.4%) func1 #以下是 func1 的调用关系

|__(261:79.6%) func2

| |

Blocked Callgraph Sampling: #阻塞时样本的函数调用关系信息，以下各字段、调用关系含义与上述类似

event| samples| %| (usr:% sys:%)| app

TIMER 7162

7162 100% (7162:100% 0:0%) /block_test_32.out

(7162:100%) __nanosleep_nocancel

|__(5257:73.4%) sleep_time

| |__(5257:100%) sleep_time_test3

| |__(5257:100%) sleep_time_test2

| |__(5257:100%) sleep_time_test1

| |__(5257:100%) main

|__(1905:26.6%) main

|__(1905:100%) __libc_start_main

|__(1905:100%) _start

6.3. 源码级数据解析

源码级数据解析是统计采样数据中源码命中的情况，能显示函数代码行样本个数和百分比，以便用户对性能问题进行精确定位。

./opannotate -S -b /cygdrive/d/test-multi.c -d /test-multi.c #源码级解析命令，-b 后接源码编译路径，-d 后接源码本地路径（也可指到源码所在的目录一级，如./opannotate -S -b /cygdrive/d/test-multi.c -d ./）

The command line of sampling is: ./kprofiler --start -t #显示采样时使用的命令

#以下为样本解析结果的表头

Sample Statistical Information : #样本统计信息

CPU0 #各 cpu 统计结果，如果有多个 cpu，则此处有多列

received 5159 #cpu0 共采集到的样本个数

```

overflow          0          #cpu0 上因溢出而丢弃的样本个数，消除这个溢出值的方法见 9.9
%(overflow)       0          #cpu0 上因溢出而丢弃的样本个数占本 cpu 总样本数的百分比
backtrace(aborted) 0          #堆栈回溯过程中的失败次数
%(backtrace)      0          #堆栈回溯过程中的失败次数占此 CPU 上采样个数的百分比
received(total)   no-mapping(backtrace) %
#总样本数，各 cpu 样本之和 #堆栈回溯时找不到对应地址映射的样本个数及百分比
event-overflow   %          no-mapping(sample) %
#event-overflow 的样本数及百分比，见 9.10 #样本中（非堆栈回溯数据）找不到对应地址映射的样
本个数及百分比
error            %
#出错的样本个数及百分比
5159             0          0
0 0              0          0
0 0
Start Sample Time : 16:18:38 #开始采样时间
Stop Sample Time : 16:18:59 #结束采样时间
Sample time : 20.694823s     #整个采样持续时间

Running sample number : 5159 #采集到的运行时样本个数
Running Sampling:
/*
 * Total samples for file : "/cygdrive/d/test-multi.c"
 *      124  2.4036          #在/cygdrive/d/test-multi.c 文件中采样到 124 次，占总采样数的
2.4036%
 */

#include <stdlib.h>
:
: int fast_multiply(x, y)
: {
:     return x * y;
: }
: int slow_multiply(x, y)
/* slow_multiply total: 124 2.4036 # slow_multiply 函数采样到 124 次，占总采样数的 2.4036%
TIMER:0 |
samples %| samples % */
: int i, z;
62 1.2018 :for (i = 0, z = 0; i < x; i++) #采样落在此代码语句上的次数为 62 次，占总采样数的 1.2018%

```


• • • • •

程序运行过程中数据进行采样，从而找出性能较差的函数。

7.1.1.2. 目标环境准备

使用 x86 架构的目标环境，KProfiler 编译成模块，采用基于性能计数器方式进行采样。具体步骤如下：

- 1) 在 KIDE 中构建 LSP 项目，在内核配置选项中将 KProfiler 编译成模块，获取内核映像文件 bzImage 和 KProfiler 功能模块 oprofile.ko，具体参见 2.1；
- 2) 启动内核，加载 KProfiler。

```
# insmod oprofile.ko //插入 oprofile 模块，基于性能计数器采样
```



提示：若用户在执行信息采样之前，希望对文件系统下内容进行查看，则首先请检查是否存在

dev/oprofile/文件夹，若不存在，则使用 `mkdir dev/oprofile` 命令创建 oprofilefs 文件系统挂载目录，

再执行 `mount -t oprofilefs nodev dev/oprofile/` 命令手动进行文件系统挂载。

7.1.1.3. 用户程序准备

此处准备一个示例用于演示如何编译和分析一个编写不当的代码，以找出哪个函数性能不佳。这个小示例只包含两个函数 `slow_multiply()` 和 `fast_multiply()`。这两个函数都是用于求两个数的乘积，`fast_multiply` 是直接使用乘法，`slow_multiply` 是以累加代替乘法。代码如下：

```
int fast_multiply(x,y)
{
    return x * y;
}
int slow_multiply(x,y)
{
    int i, j, z;
    for (i = 0, z = 0; i < x; i++)
        z = z + y;
    return z;
}
```

```
int main()
{
    int i,j;
    int x,y;
    for (i = 0; i < 200; i ++)
    {
        for (j = 0; j < 30 ; j++)
        {
            x = fast_multiply(i,j);
            y = slow_multiply(i,j);
        }
    }
    return 0;
}
```

单从分析代码，即可感受到使用累加的 `slow_multiply` 应该比直接使用乘法的 `fast_multiply` 性能差很多，下面我们使用 KProfiler 对其进行分析。

可以使用 KIDE 构建用户程序，编译时添加调试信息（-g），构建成功生成用户程序 `user.out`，之后下载到目标端以备测试，具体方法参见《广东中兴新支点 KIDE V3 用户手册》4.2.3。

7.1.1.4. 数据采样

下载 KProfiler 用户态工具到目标端，解压后运行起来

```
#!/kprofiler                                //开始使用计数器的默认事件进行采样，默认清除之前采样，
                                              默认使用-p all 选项和-c 5 选项

Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.

#!/home/lcz/exec.out                        //运行采样程序

#!/kprofiler --stop                        //结束采样，默认杀死守护进程，默认保持 oprofilefs 文件系统挂载
Stopping profiling.
Killing daemon.
```

7.1.1.5. 数据解析

结束采样，使用 opannotate 工具进行源码级统计，结果如下：

```
#./opannotate -S /home/lcz/exec.out -b /cygdrive/e/target/workspace/exec/debug/../ -d
/home/lcz/orig_code/

          CPU0          CPU1
received    10499      10517
overflow      0          0
%(overflow)   0          0
backtrace(aborted) 0          0
%(backtrace)  0          0
received(total) no-mapping(backtrace) % event-overflow % no-mapping(sample) % error %
21016          0          0      0      0      0      0      0      0
Start Sample Time : 01:01:58
Stop Sample Time : 01:02:09
Sample time : 10.475329s
Running sample number :20004          #运行采样次数
image=/home/lcz/exec.out
image=/no-vmlinux
/*
 * Command line: ./opannotate -s /home/lcz/exec.out -b /cygdrive/e/target/workspace/exec/debug/../ -d
/home/lcz/orig_code/
 *
 * Interpretation of command line:
 * Output annotated source file with samples
 * Output all files
 *
 * CPU: AMD64 processors, speed 1000 MHz (estimated)
 * Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit mask of 0x00 (No
unit mask) count 1000000
 */
Running Sampling:
/*
 * Total samples for file : "/cygdrive/e/target/workspace/exec/debug/../set_cpu.c"
 *
 *
 *      6 85.7143
```

```

*/

#include <stdlib.h>

:
: int fast_multiply(x, y)
: {
:     return x * y;
: }
: int slow_multiply(x, y)
/* slow_multiply total:      6 85.7143
CPU_CLK_UNHALTED:1000000 |
samples %|  samples % */
:     int i, z;
3 42.8571 :      for (i = 0, z = 0; i < x; i++)
3 42.8571 :          z = z + y;
:     return z;
: }
: int main()
: {
:     int i,j;
:     int x,y;
:     for (i = 0; i < 200; i ++ ) {
:         for (j = 0; j < 30 ; j++) {
:             x = fast_multiply(i, j);
:             y = slow_multiply(i, j);
:         }
:     }
:     return 0;
: }

```

其中:

➤ * CPU: AMD64 processors, speed 1000 MHz (estimated)

* Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit mask of 0x00 (No unit mask) count 1000000

表明本次采样基于计数器方式进行采样，使用的事件是 CPU_CLK_UNHALTED，其设置为发生 1000000 次 CPU_CLK_UNHALTED 事件采样一次。

- /* slow_multiply total: 6 85.7143 表明 slow_multiply 函数采样到 6 次，占 exec.out 的所有采样数据的 85.713%
- 3 42.8571: z = z + y; 此采样数据表明采样落在此代码语句上的次数为 3 次，占 exec.out 所有采样数据的 42.8571%

从上述结果可以看到，在每个 TICK 进行采样的过程中，总共采样到 6 次，并且运行在 slow_multiply 函数中，而相同功能的 fast_multiply() 函数并未进行采样，说明 slow_multiply 函数耗时较多。

7.1.1.6. 图形界面操作

若是使用图形界面，首先参照 4.1.1 和 4.1.2 加载 KProfiler 内核，建立目标机连接，在新建 KProfiler 配置时，只需完成通用选项配置，其他配置保持默认即可。

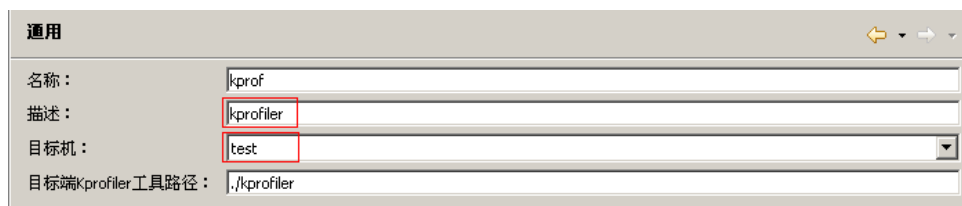


图 7-1 通用选项配置

完成 KProfiler 配置后，在 KProfiler 视图中，对此配置依次执行【控制】->【挂载】，【控制】->【启动】，待启动采样后，在目标端运行用户态程序，运行一段时间后，执行【控制】->【停止】，中断采样，之后 KProfiler 会自动打印出解析结果。其输出结果同 7.1.1.5 介绍。

7.1.2. CPU 占用率高但业务性能上不去

7.1.2.1. 现象描述

■ 运行环境

PSS 项目 SBCW (x86_64) 单板，CPU 主频 2133.41M 8CPU

控制面绑定在 0-1 号 CPU，媒体面绑定在 2-7 号 CPU

■ 故障现象

PSS 项目在进行大话务量测试时，发现媒体面性能上不去，使用 **top** 命令查看，结果如下：

```
Mem: 8652660K used, 7516652K free, 0K shrd, 2216K buff, 902120K cached
CPU0: 17.1% usr 25.1% sys  0.0% nic 54.4% idle  0.0% io  0.0% irq  3.2% sirq
CPU1: 40.6% usr 38.8% sys  0.0% nic 17.4% idle  0.0% io  0.3% irq  2.5% sirq
CPU2: 41.4% usr 35.1% sys  0.0% nic 22.0% idle  0.0% io  0.1% irq  1.1% sirq
CPU3: 37.9% usr 34.7% sys  0.0% nic 25.6% idle  0.0% io  0.2% irq  1.4% sirq
CPU4: 35.1% usr 34.5% sys  0.0% nic 27.4% idle  0.0% io  0.6% irq  2.2% sirq
CPU5: 35.0% usr 37.4% sys  0.0% nic 25.4% idle  0.0% io  0.0% irq  1.9% sirq
CPU6: 37.3% usr 33.9% sys  0.0% nic 25.8% idle  0.0% io  0.4% irq  2.4% sirq
CPU7: 37.2% usr 34.4% sys  0.0% nic 25.8% idle  0.0% io  0.0% irq  2.4% sirq
Load average: 0.00 0.00 0.00 9/728 1836
```

可见，内核态占用了 35% 左右的 CPU。这样的情况不正常，因为媒体面主要进行用户态程序的处理，内核态占用 CPU 应该是越小越好，所以我们需要找到故障时内核的热点函数，针对其进行优化，减小内核态 CPU 占用率，提高性能。

使用 KProfiler 对大话务量测试的环境进行采样，经过整理后，样本数较高的前几位函数如下：

表 7-1 样本数较高函数列表

cpu2		cpu3		cpu4		cpu5		cpu6		cpu7		symbol_name
sample%		sample%		sample%		sample%		sample%		sample%		
3005	11.0515	3289	12.2322	3273	11.9583	3291	12.0753	3270	12.0847	3198	11.7799	futex_wake
1163	4.2772	1284	4.7754	1182	4.3186	1222	4.4837	1153	4.2611	1171	4.3134	find_busiest_group
863	3.1738	956	3.5555	893	3.2627	968	3.5518	864	3.193	1017	3.7461	schedule
446	1.6402	425	1.5806	422	1.5418	421	1.5447	380	1.4043	423	1.5581	mwait_idle
370	1.3607	373	1.3872	370	1.3518	372	1.3649	377	1.3933	411	1.5139	_atomic_dec_and_lock
400	1.4711	405	1.5062	396	1.4468	373	1.3686	392	1.4487	380	1.3997	system_call
295	1.0849	378	1.4058	336	1.2276	365	1.3393	325	1.2011	354	1.304	csum_partial_copy_generic

对样本数据进行分析，发现出现性能问题时，CPU 占有率较高的函数都在内核态，分别是 `futex_wake`、`find_busiest_group`、`schedule`、`mwait_idle` 等，`futex_wake` 与用户态程序的锁处理相关，`find_busiest_group`、`schedule` 和 `mwait_idle` 与内核调度相关，鉴于 `futex_wake` 的 CPU 占有率相当大，因而有充分的理由怀疑是内核处理用户态程序的加/解锁操作过于频繁，导致调度开销过大。

最终结论：用户态程序的锁处理不当，导致内核开销太大，应从应用程序的锁处理相关部分入手进行性能优化。

7.1.2.2. 解决方案

PSS 项目组对应用程序代码进行了排查，发现在数据库相关处理中大量使用了加/解锁的操作，因而对这部分进行了改进，在保证数据正确性的前提下，去除所有不必要的锁操作，重新制作版本进行测试。

7.1.2.3. 最终效果

大话务量测试，性能得到极大提高，达到了性能测试的指标。

7.1.2.4. 优化后，使用 KProfiler 采样的数据

用户程序进行锁优化处理后，再对大话务量进行测试，此时内核态的 CPU 占有率普遍在 3-4% 左右，基本为优化前的 10%，验证了我们根据 KProfiler 采样结果的分析。

优化后的数据整理如下：

top 结果：

```
Mem: 10573016K used, 5596300K free, 0K shrd, 2652K buff, 1261164K cached
CPU0: 22.8% usr 15.5% sys 0.0% nic 56.2% idle 0.0% io 0.0% irq 5.3% sirq
CPU1: 35.3% usr 3.8% sys 0.0% nic 59.3% idle 0.0% io 0.0% irq 1.4% sirq
CPU2: 32.0% usr 4.5% sys 0.0% nic 62.2% idle 0.0% io 0.3% irq 0.7% sirq
CPU3: 33.2% usr 3.1% sys 0.0% nic 61.6% idle 0.0% io 0.3% irq 1.6% sirq
CPU4: 34.6% usr 2.7% sys 0.0% nic 60.5% idle 0.7% io 0.1% irq 1.0% sirq
CPU5: 34.4% usr 3.0% sys 0.0% nic 60.9% idle 0.0% io 0.1% irq 1.4% sirq
CPU6: 34.4% usr 5.0% sys 0.0% nic 59.2% idle 0.0% io 0.0% irq 1.2% sirq
CPU7: 34.6% usr 5.0% sys 0.0% nic 58.7% idle 0.0% io 0.3% irq 1.2% sirq
```

Load average: 3.25 1.71 0.82 10/731 2512

热点函数结果：

表 7-2 热点函数列表

cpu2		cpu3		cpu4		cpu5		cpu6		cpu7		symbol
samples	%	samples	%	samples	%	samples	%	samples	%	samples	%	
24	0.215	13	0.114	25	0.205	19	0.164	17	0.154	26	0.232	e1000_irq_enable
8	0.072	12	0.106	15	0.123	24	0.208	18	0.163	25	0.224	kmem_cache_free
5	0.045	15	0.132	35	0.286	17	0.147	13	0.117	23	0.206	pskb_trim
16	0.144	14	0.123	22	0.18	18	0.156	30	0.271	23	0.206	kfree
11	0.099	13	0.114	9	0.074	8	0.069	4	0.036	23	0.206	drop_futex_key_refs
14	0.126	16	0.141	53	0.434	43	0.372	21	0.19	20	0.179	try_to_wake_up
18	0.162	13	0.114	7	0.057	13	0.112	15	0.136	20	0.179	hrtimer_interrupt
14	0.126	12	0.106	29	0.237	28	0.242	8	0.072	19	0.17	memset_c
10	0.09	14	0.123	8	0.066	14	0.121	7	0.063	19	0.17	pick_next_task_rt
6	0.054	19	0.167	13	0.106	18	0.156	20	0.181	18	0.161	ppro_setup_ctrs
17	0.153	17	0.15	12	0.098	15	0.13	7	0.063	18	0.161	futex_wait
17	0.153	18	0.158	16	0.131	18	0.156	12	0.108	16	0.143	skb_clone
21	0.188	26	0.229	28	0.229	12	0.104	7	0.063	16	0.143	apic_timer_interrupt

从上述表格中看到，内核态各函数的占有率普遍较小，无明显的热点函数（瓶颈）。

7.1.3. CPU 占用率低性能无法提高

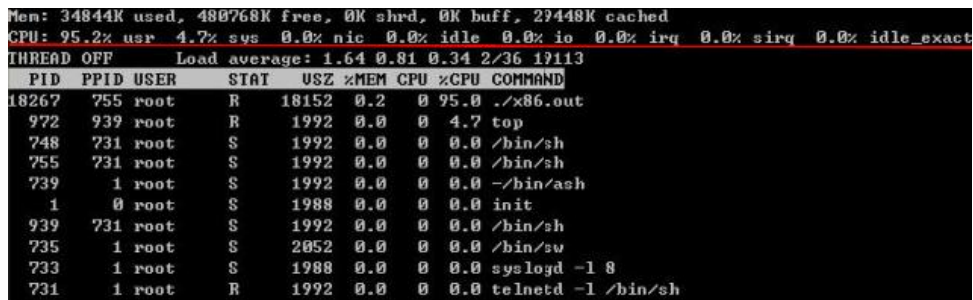
7.1.3.1. 现象描述

■ 运行环境

x86_64, bsp: LENLEVO-PC-X86_64

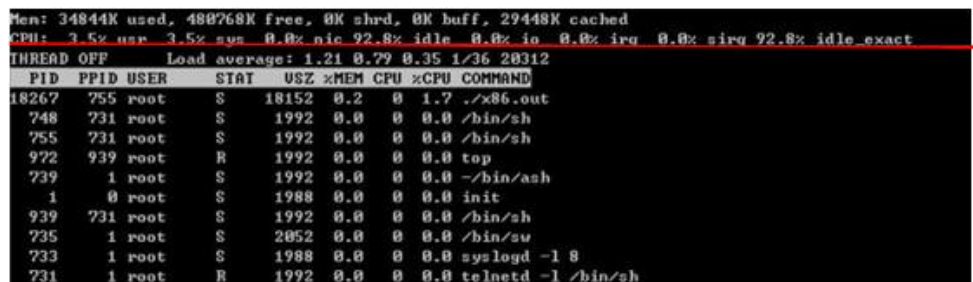
■ 故障现象

运行用户态程序，发现 cpu 占用率增高至 80%后突然下降，且一直处于 cpu 占用率很低的一种情况，使用 top 命令查看，用户态程序 cpu 占用在 2%左右。



```
Mem: 34844K used, 480768K free, 0K shrd, 0K buff, 29448K cached
CPU: 95.2% usr 4.7% sys 0.0% nic 0.0% idle 0.0% io 0.0% irq 0.0% irq 0.0% idle_exact
THREAD OFF Load average: 1.64 0.81 0.34 2/36 19113
PID PPID USER STAT USZ %MEM CPU %CPU COMMAND
18267 755 root R 18152 0.2 0 95.0 ./x86.out
972 939 root R 1992 0.0 0 4.7 top
748 731 root S 1992 0.0 0 0.0 /bin/sh
755 731 root S 1992 0.0 0 0.0 /bin/sh
739 1 root S 1992 0.0 0 0.0 ~/bin/ash
1 0 root S 1988 0.0 0 0.0 init
939 731 root S 1992 0.0 0 0.0 /bin/sh
735 1 root S 2052 0.0 0 0.0 /bin/sv
733 1 root S 1988 0.0 0 0.0 syslogd -l 8
731 1 root R 1992 0.0 0 0.0 telnetd -l /bin/sh
```

图 7-2 CPU 占用率增高情况



```
Mem: 34844K used, 480768K free, 0K shrd, 0K buff, 29448K cached
CPU: 1.5% usr 1.5% sys 0.0% nic 92.8% idle 0.0% io 0.0% irq 0.0% irq 92.8% idle_exact
THREAD OFF Load average: 1.21 0.79 0.35 1/36 20312
PID PPID USER STAT USZ %MEM CPU %CPU COMMAND
18267 755 root S 18152 0.2 0 1.7 ./x86.out
748 731 root S 1992 0.0 0 0.0 /bin/sh
755 731 root S 1992 0.0 0 0.0 /bin/sh
972 939 root R 1992 0.0 0 0.0 top
739 1 root S 1992 0.0 0 0.0 ~/bin/ash
1 0 root S 1988 0.0 0 0.0 init
939 731 root S 1992 0.0 0 0.0 /bin/sh
735 1 root S 2052 0.0 0 0.0 /bin/sv
733 1 root S 1988 0.0 0 0.0 syslogd -l 8
731 1 root R 1992 0.0 0 0.0 telnetd -l /bin/sh
```

图 7-3 CPU 突降保持值很低状态

7.1.3.2. 解决方案

当 CPU 占有率不高但业务进程处理能力又无法提高的情况下使用阻塞采样。这是由于出现此状况的原因可能是因为资源等待而导致的 CPU 空转，通过使用阻塞采样可以得到采样时段进程阻塞及运行时间的情况和进程阻塞的位置。

7.1.3.3. 实施方法

■ 数据采样

```
./kprofiler --start -S 755 -T 20 -t
```

■ 数据解析

```
./opreport
```

■ 数据分析

```
Blocked Samples:
event| samples| %| (usr:% sys:%)| app
-----
TIMER:0 2518 100% (2518:100% 0:0%) /x86.out
2518 2518 100% (2481:98.5%) ./[vdso] (tid: 755 range:0xffffe000-0xffffffff000<./[vdso] (tid: 755 ra
nge:0xffffe000-0xffffffff000>
(68: 1.4%) clone
```

图 7-4 解析结果查看

由此可见，绝大部分阻塞集中在 vdso，使用 opreport 的回溯解析功能，查看到 98.5%的阻塞均在函数 stoplax_ctx 里，如图 7-5:

```
Blocked Callgraph Sampling:
event| samples| %| (usr:% sys:%)| app
-----
TIMER:0 2518 100% (2518:100% 0:0%) /x86.out
2518 100% (2481:98.5%) ./[vdso] (tid: 755 range:0xffffe000-0xffffffff000<./[vdso] (tid:
7 range:0xffffe000-0xffffffff000>
!_(2481:100%) stoplax_ctx
!_(2481:100%) main
!_(2518:100%) __libc_start_main
!_(2518:100%) _start
(37: 1.5%) __libc_disable_asynccancel
!_(37:100%) pthread_create
!_(37:100%) main
!_(2518:100%) __libc_start_main
!_(2518:100%) _start
```

图 7-5 查看阻塞函数

■ 走查代码

源码:

```
static void stoplax_ctx()
{
    int ret;
    int ret2;
```

```

ret=sem_wait(&sem4);
if(-1==ret)
{
    printf("semBTake faile\n");
}
if(count++ >1000)                //发现当切换次数大于 1000 后会进入 sleep 状态
{
    sleep(1);
}
ret2=sem_post(&sem4);
if(-1==ret2)
{
    printf("semBTake faile\n");
}

```

■ 结论

取消 sleep，则不会再进入阻塞状态，CPU 也会上去。

7.1.3.4. Top 命令查看优化结果

Mem: 35012K used, 408600K free, 0K shrd, 0K buff, 29336K cached									
CPU: 87.6% usr 11.9% sys 0.0% nic 0.0% idle 0.0% io 0.1% irq 0.1% sirq 0.0% idle_exact									
THREED OFF Load average: 1.34 0.92 0.52 2/36 22224									
PID	PPID	USER	STAT	VSZ	KMEM	CPU	%CPU	COMMAND	
30909	755	root	R	18152	0.2	0	96.0	./x86.out	
9027	939	root	R	1992	0.0	0	0.4	top	
748	731	root	S	1992	0.0	0	0.0	/bin/sh	
755	731	root	S	1992	0.0	0	0.0	/bin/sh	
939	731	root	S	1992	0.0	0	0.0	/bin/sh	
739	1	root	S	1992	0.0	0	0.0	~/bin/ash	
1	0	root	S	1988	0.0	0	0.0	init	
735	1	root	S	2052	0.0	0	0.0	/bin/sv	
733	1	root	S	1988	0.0	0	0.0	syslogd -l 8	
731	1	root	S	1992	0.0	0	0.0	telnetd -l /bin/sh	

图 7-6 优化后结果

7.2. 特殊事件应用

7.2.1. 数据缓存 miss 统计

7.2.1.1. 场景分析

当 CPU 要读取一个数据时，会经历如下流程：

- 1) 首先从缓存（Cache）中查找，如果找到就立即读取并送给 CPU 处理；否则，进入 2)；
- 2) 从二级缓存（若存在）中读取，如果二级缓存中也没有，则进入 3)；
- 3) 用相对慢的速度从内存中读取并送给 CPU 处理，同时把该数据所在的数据块调入缓存，使得之后对整块数据的读取都从缓存中进行，不必再调用内存。

这个从一级缓存中未读到数据的情况称为一次**数据缓存 miss**。

通过优化的读取机制，可以大大提高 CPU 读取缓存的命中率（大多数 CPU 可达 90% 左右）。即是说 CPU 下一次要读取的数据 90% 都在缓存中，只有大约 10% 需要从内存中读取，从而大大缩短了 CPU 直接读取内存的时间。

本例针对在采样过程中发生 D-Cache miss 的情况，对数据缓存的 miss 统计。

7.2.1.2. 目标环境准备

使用 CDMA powerpc 8541 架构的目标环境，KProfiler 编入内核，采用性能计数器方式进行采样。具体步骤如下：

- 1) 在 KIDE 中构建 LSP 项目，在内核配置选项中将 KProfiler 编译进内核，获取内核映像文件 uImage，具体参见 [2.1](#)；
- 2) 启动内核，自动以性能计数器方式加载 KProfiler。

7.2.1.3. 用户程序准备

用户代码：

```
int SumA(int array[][1024],int n)
{
    int sum = 0,x,y;
    for (y = 0; y < n; y++)
```

```
{
    for (x = 0; x < n; x++)
    {
        sum += array[x][y];
    }
}

return sum;
}

int SumB(int array[][1024],int n)
{
    int sum = 0,x,y;
    for (x = 0; x < n; x++)
    {
        for (y = 0; y < n; y++)
        {
            sum += array[x][y];
        }
    }

    return sum;
}

int main(int argc,char** argv)
{
    int n = 1 << 10;
    int array[n][n];
    int x,y;

    for (x = 0; x < n; x++)
    {
        for (y = 0; y < n; y++)
        {
            array[x][ y] = x;
        }
    }

    printf("The sum result if %d\n",SumA(array,n));
```

```
return 0;
}
```

使用 KIDE 构建用户程序，编译时添加调试信息（-g），构建成功生成用户程序 8541_user.out，之后下载到目标端以备测试，具体方法参见《广东中兴新支点 KIDE 2.8 用户手册》4.2.3。

7.2.1.4. 数据采样

下载 KProfiler 用户态工具到目标端，解压后运行起来

```
#./kprofiler --start -e DL1_RELOADS:7500:0:1:1 -p none //开始采样，-e DL1_RELOADS 表示基于
DL1_RELOADS 事件（CACHE 命中率事件）进行采样，采样结果不设定测评区分
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
#./8541_user.out //运行用户程序
#./kprofiler --stop //结束采样
Stopping profiling.
Killing daemon.
```

7.2.1.5. 数据解析

结束采样，使用 oprofile 工具统计应用程序的采样情况，结果如下：

```
#./oprofile //统计采样情况
.....
CPU: e500, speed 660 MHz (estimated) //采用性能计数器方式采样
Counted DL1_RELOADS events (This is historically used to determine dcache miss rate (along with
loads/stores completed). This counts dL1 reloads for any reason. ) with a unit mask of 0x00 (No unit
mask) count 7500
event| samples| %| (usr:% sys:%)| app
-----
DL1_RELOADS 178
157 78.6% (157: 88.2% 0:0%) /8541_user.out
(140:78.6%) SumA
//函数 SumA 采样 140 次
```

			(17:9.5%)	main
	//main 函数采样 17 次			
19	10.6%	(0:0%	19:10.6%)	/vmlinux
			(17:9.5%)	clear_pages
	//内核函数 clear_pages 采样 17 次			
2	1.1%	(2:1.1%	0:0%)	/lib/ld-2.5.so

可以看到在 8541_user.out 里共有 157 个采样数据。

使用 opannotate -S 查看源码级的统计结果。

```
#./opannotate -S                                     //源码级统计，统计采样数据中源码命中的情况
.....
/*
 * Command line: ./opannotate -s
 *
 * Interpretation of command line:
 * Output annotated source file with samples
 * Output all files
 * CPU: e500, speed 660 MHz (estimated)
 * Counted DL1_RELOADS events (This is historically used to determine dcache miss rate (along with
loads/stores completed). This counts dL1 reloads for any reason. ) with a unit mask of 0x00 (No unit
mask) count 7500
 */
/*
 * Total samples for file : "/cygdrive/d/target/kexec/8541_user/debug/./user.c"
 *
 *      157 30.9665
 */

#include <stdlib.h>
#include <stdio.h>
int SumA(int array[][1024],int n)
:{
/* SumA total:      140 27.6134 */
:   int sum = 0,x,y;
:   for (y = 0; y < n; y++)
:   {
69 13.6095 :           for (x = 0; x < n; x++)
```

```

:      {
71 14.0039 :          sum += array[x][y];
:      }
:  }
:  return sum;
:}
:int SumB(int array[][1024],int n)
:{
:  int sum = 0,x,y;
:  for (x = 0; x < n; x++)
:  {
:      for (y = 0; y < n; y++)
:      {
:          sum += array[x][y];
:      }
:  }
:  return sum;
:}
:int main(int argc,char** argv)
:{
/* main total:    17   3.3531 */
:  int n = 1 << 10;
:  int array[n][n];
:  int x,y;
:  for (x = 0; x < n; x++)
:  {
17  3.3531 :      for (y = 0; y < n; y++)
:      {
:          array[x][ y] = x;
:      }
:  }
:  printf("The sum result if %d\n",SumA(array,n));
:  return 0;
:}

```

从统计结果可以看出，函数 SumA 共采样 140 次，其中在 for (x = 0; x < n; x++)语句处采样 69 次，在 sum += array[x][y]语句处采样 71 次，说明此函数的这两条语句存在大量数据缓存 miss，需要进行优化。

7.2.1.6. 图形界面操作

若是使用图形界面，首先参照 4.1.1 和 4.1.2 加载 KProfiler 内核，建立目标机连接。

新建 KProfiler 配置时，首先完成通用选项配置，如图 7-7。

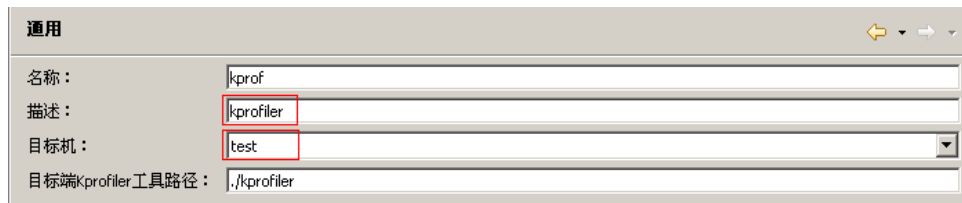


图 7-7 通用选项配置

在【控制】-【启动】选项页，设置“会话数据隔离方式”为 none，表示采样结果不设定测评区分，同时选中计数器采样方式，去掉“使用缺省事件配置”的勾选，勾选特定采样 CPU，并勾选特定的采样事件“DL1_RELOADS”，表示基于 DL1_RELOADS 事件（CACHE 命中率事件）进行采样，如图 7-8：

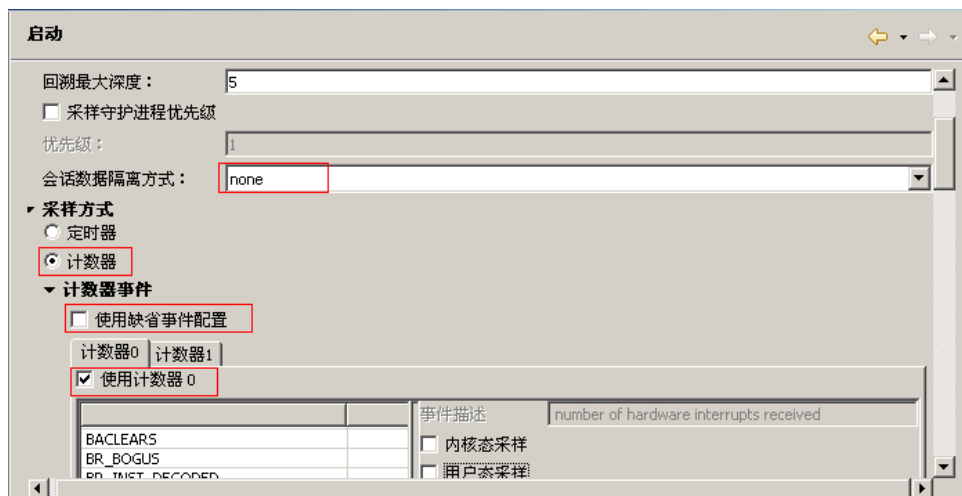


图 7-8 启动选项配置

其他选项页保持默认缺省配置即可。完成 KProfiler 配置后，在 KProfiler 视图中对此配置依次执行【控制】->【挂载】，【控制】->【启动】，待启动采样后，在目标端运行用户态程序，运行一段时间后，执行【控制】->【停止】，中断采样，之后 KProfiler 会自动打印出解析结果。其输出结果同 7.2.1.5 介绍。

7.2.2. 系统/函数指令执行效率统计

7.2.2.1. 场景分析

指令执行效率是衡量系统性能的一个重要指标，它包括 CPI（cycles per instruction，每条指令消耗的平均处理器周期数）和 IPC（instructions per cycle，每处理器周期执行的指令数）。因而，对指令执行效率进行把控至关重要，KProfiler 即可帮助我们实现这一愿望。

7.2.2.2. 目标环境准备

使用 X86 64 架构的目标环境，采用基于性能计数器方式进行采样，KProfiler 编译为模块方式，具体步骤如下：

- 1) 在 KIDE 中构建 LSP 项目，在内核配置选项中将 KProfiler 编译成模块，获取内核映像文件 bzImage，具体参见 2.1；
- 2) 成功启动内核后，通过使用 insmod 或 modprobe 命令手动插入模块，即可以性能计数器方式加载 KProfiler；

```
# insmod oprofile.ko
```

```
//插入 oprofile 模块，基于性能计数器方式采样
```

7.2.2.3. 用户程序准备

用户代码：

```
int fast_multiply(x, y)
{
    return x * y;
}
int slow_multiply(x, y)
{
    int i, j, z;
    for (i = 0, z = 0; i < x; i++)
        z = z + y;
    return z;
}
int main()
{
    int i, j;
```

```

int x,y;
for (i = 0; i < 200; i++) {
    for (j = 0; j < 30 ; j++) {
        x = fast_multiply(i, j);
        y = slow_multiply(i, j);
    }
}
return 0;
}

```

7.2.2.4. 数据采样

下载 KProfiler 用户态工具到目标端，解压后运行起来

```

#./kprofiler --start -e CPU_CLK_UNHALTED:200000:0:1:1 -e RETIRED_INSTRUCTIONS:5000:0:1:1
//开始采样，-e CPU_CLK_UNHALTED:200000:0:1:1 和-e RETIRED_INSTRUCTIONS:5000:0:1:1 表示
基于计数器方式对这两个事件进行采样（采样事件由用户自行设置），默认使用-c 5 选项设置调用图深度
值，默认使用-p all 选项，及默认使用-r 1 清除之前采样

Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.

#./a.out //运行用户程序
#./kprofiler --stop //结束采样
Stopping profiling.
Killing daemon.

```

7.2.2.5. 数据解析

结束采样，使用 oprofile 工具统计应用程序的采样情况，结果如下：

```

# ./oprofile //统计应用程序采样情况
CPU: AMD64 processors, speed 1000 MHz (estimated) //采用性能计数器方式采样
Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit mask of 0x00 (No unit
mask) count 200000

```

Counted RETIRED_INSTRUCTIONS events (Retired instructions (includes exceptions, interrupts, re-syncs)) with a unit mask of 0x00 (No unit mask) count 5000

event	samples	%	(usr:% sys:%)	app

CPU_CLK_UNHALTED	22341			
	12250	54.8%	(12250:54.8% 0:0%)	/home/dyz/a.out
			(4469:20.0%)	slow_multiply
			(3934:17.6%)	fast_multiply
			(3847:17.2%)	mian
	1400	6.3%	(0:0% 1400:6.3%)	home/xulan/p_k/vmlinux
			(760:3.4%)	__handle_mm_fault
			(512:2.2%)	acpi_pm_read
.....				

RETIRED_INSTRUCTIONS	655640			
	557655	85.0%	(557655:85.0% 0:0%)	/home/dyz/a.out
			(187528: 28.6%)	mian
			(185566:28.3%)	slow_multiply
			(184561:28.1%)	fast_multiply
.....				

对 CPU_CLK_UNHALTED 和 RETIRED_INSTRUCTIONS 两个事件分别进行采样,因而使用 opreport 工具解析数据时分别显示了两个事件的解析情况。针对 CPU_CLK_UNHALTED 事件, /home/dyz/a.out 中的 **slow_multiply** 函数采样到 **4469** 次,实际发生此事件的次数为“**4469*200000**”,占总采样的 **20.0278%**;同理,针对 RETIRED_INSTRUCTIONS 事件, /home/dyz/a.out 中的 **slow_multiply** 函数采样到 **185566** 次,占总采样的 **28.3030%**,其实际发生此事件的次数为“**185566*5000**”次。

■ 系统指令执行效率计算:

➤ CPI(cycles per instruction)计算: 每条指令花费的周期数,即总的的周期数除以总的指令数
事件一为 CPU_CLK_UNHALTED, 即是 cycle, 周期数;

事件二为 RETIRED_INSTRUCTIONS, 即是 instruction, 指令数;

$$CPI=((4469/0.200278)*200000)/((185566/0.283030)*5000)$$

➤ IPC (instructions per cycle) 计算: 每个周期执行的指令数,即总的指令除以总的周期

$$IPC=((185566/0.283030)*5000)/((4469/0.200278)*200000)$$


■ 函数 slow_multiply 的指令执行效率计算:

➤ CPI(cycles per instruction)计算:

$$(4469 * 200000) / (185566 * 5000)$$

➤ IPC (instructions per cycle) 计算:

$$(185566 * 5000) / (4469 * 200000)$$

 提示：根据架构不同，代表 cycle 周期数与 instruction 指令数的事件名称有所不同，因而，若要使用此功能，用户需自行分析事件的含义。

7.2.2.6. 图形界面操作

若是使用图形界面，首先参照 4.1.1 和 4.1.2 加载 KProfiler 内核，建立目标机连接。

新建 KProfiler 配置时，首先完成通用选项配置，如图 7-9。

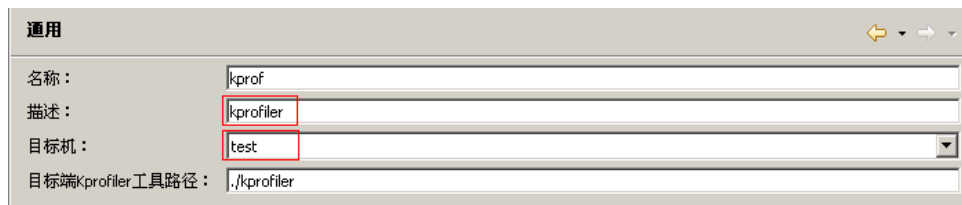


图 7-9 通用选项配置

在【控制】-【启动】选项页，设置“会话数据隔离方式”为 all，表示所有采样数据分别按照 cpu、库、线程号、线程组号和内核区分保存，同时选中计数器采样方式，去掉“使用缺省事件配置”的勾选，勾选特定采样 CPU，并勾选特定的采样事件“CPU_CLK_UNHALTED”和“RETIRED_INSTRUCTIONS”，表示基于这两个事件进行采样，如图 7-10：

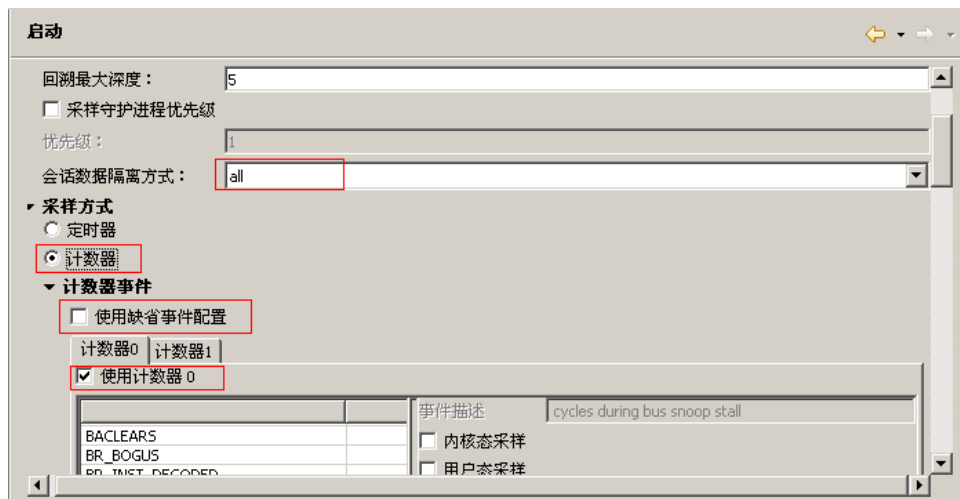


图 7-10 启动选项配置

其他选项页保持默认缺省配置即可。完成 KProfiler 配置后，在 KProfiler 视图中对此配置依次执行【控制】->【挂载】，【控制】->【启动】，待启动采样后，在目标端运行用户态程序，运行一段时间后，执行【控制】->【停止】，中断采样，之后 KProfiler 会自动打印出解析结果。其输出结果同 7.2.2.5 介绍。

8. 精确计时功能

KProfiler 除了提供打点采样的性能记录外，还提供了精确记录函数执行时间的功能。

8.1. 原理简介

当前的主流 CPU，大多都拥有一个频率与 CPU 主频相关的 count 寄存器，用来作为稳定且高精度的时间源。在测量某函数的精确执行时间时，只需在函数入口处访问该 count 寄存器，从而得到开始时刻的 start_stamp，在函数出口处再次访问该 count 寄存器得到结束时刻 end_stamp，就可以通过 end_stamp-start_stamp 得到函数精确执行时间。


8.2. 使用说明


8.2.1. 生成性能日志

精确计时功能需要工具链配合，请使用最高版本的成研工具链。目前支持该功能的体系架构包括 X86、X86_64、PPC、MIPS、MIPS_64 和 ARM（仅 452 系列工具链支持，且需内核打开 Performance Monitoring Unit 的访问权限，通过将 8.4 的代码编译为内核模块插入即可完成）。

使用步骤如下：

- 1) 编译用户态程序，添加 **-kprof** 编译参数。需要注意，**请使用 gcc 而不是 ld 作为链接器**；
- 2) 运行用户态程序，获得性能记录日志（可使用“**kill -s SIGURG 用户进程号**”生成快照日志文件，或者待用户态程序运行结束后自动生成日志）。性能日志命名格式为“进程名_进程号_snap.log”（快照生成的日志）或者“进程名_进程号_final.log”（运行结束后生成的日志）。

 提示：对于动态链接的程序，请注意将工具链的 libkprof.so 动态库放置到运行环境中。

 提示：为提高被测程序的运行速度，默认开启了 **-kprof,min** 选项来编译被测程序。该选项使得少于 150 条寄存器传递指令的小函数不被测量。该选项会导致测量信息不完整，若需要完整的测量信息，可使用 **-kprof,min:0** 来开启完整测量。如果默认情况下的性能损失仍然过大，可使用 **-kprof,min:num** 编译被测程序，num 表示被测函数的指令条数下限，升高 num 可以减少性能损失。

8.2.2. 解析性能日志

将用户态程序和相关动态库拷贝到本地目录 XX 下，使用 KProfiler 中提供的 kprof-tool 工具，执行如下命令解析日志：


kprof-tool -l 日志文件

解析工具还提供如表 8-1 所示的其他选项，用于控制输出。关于解析输出的详细描述见 8.2.3。

表 8-1 kprof-tool 参数列表

参数定义	功能与含义	默认值
-l <log-path>	指定需要解析的日志文件（带路径），日志所在的路径自动作为符号表文件的搜索路径	无
-s <log-path>	指定一个基准日志，解析器将显示从基准日志获取时间起到 -l 指定的日志获取时间止的时间段内，各个函数的执行情况（最大最小时间除外，这里不改变其值，若两	无

	次日志中的最大最小时间有变动，则性能记录前将标记“*”符号)。注意基准日志与-l 指定的日志必须来自同一个进程，并且基准日志应该先获取。	
-p <image-path,image-path>	指定额外需找符号表的路径	NULL
-T <top-number>	指定显示排序后的前多少个函数	5
-S <sort type>	指定排序的类型，其排序类型有： <ul style="list-style-type: none"> ● total 按照函数总的耗时来进行排序 ● max 按照函数最大耗时来进行排序 ● min 按照函数最小耗时来进行排序 ● avr 按照函数平均耗时来进行排序 ● self_t 按照函数自身耗时来进行排序 ● self_a 按照函数平均自身执行时间来进行排序 ● called 按照函数执行次数来进行排序 ● var 按照“最大执行时间/平均执行时间”比值排序 	self_t
-c	显示函数调用树信息（显示父函数）	NULL
-C	显示函数调用树信息并附带源代码行信息（显示父函数）	NULL
-x	显示函数调用树信息（显示子函数）	NULL
-X	显示函数调用树信息并附带源代码行信息（显示子函数）	NULL
-a	显示线程函数信息	NULL
-t <top-number>	在调用记录显示中的最大回溯层数（不受限制的话，循环调用可能导致程序混乱）	5
-f	过滤参数指定需要查看的线程号	all
-i	过滤参数指定需要查看的函数名	all
-h	显示帮助信息	NULL

 提示：-p、-f、-i 参数均允许使用“,”来传递多个参数。-f 和-i 参数一旦指定，输出中将只包含两个参数指定的内容。

8.2.3. 配置信号

精确计时内部需要使用一些信号。在所有体系上使用了 **SIGURG** 作为写日志的触发信号，在 MIPS 体系上使用了 **SIGPROF** 作为内部溢出检查的触发信号。这些信号若被用户占用，则精确计时功能可能

会发生异常。为避免信号使用冲突，可以使用“**export KPROF_SIG_LOG=数字**”来配置写日志触发信号，用“**export KPROF_SIG_PROF=数字**”来配置溢出检查信号。注意配置环境变量后，需要重新启动被测应用，配置才能生效。

8.2.4. 精确计时接口

如果难以通过配置 **KPROF_SIG_LOG** 的方式来更改写日志的触发信号，则可以通过下面的两个开放接口来完成写日志功能。

int __kprof_write_log_full(char * file, int line_no) 这个函数可以在代码中直接调用，用于生成定点日志，方便进行特殊流程的定点测量。生成的日志名为“进程名_进程号_文件名（来自入参 file）_行号（来自入参 line_no）”。

int __kprof_write_log(void) 这个函数为简化版本，方便用户生成快照日志。生成的日志文件名为“进程名_进程号_kprof_snap.log”。

上面的两个开放接口，可以在代码中调用，也可用调试器直接调用生成日志。

8.2.5. 使用精确计时功能辅助阻塞点定位

精确计时功能可以用于辅助定位阻塞点。具体使用方法如下：

- 1) 以“最大执行时间/平均执行时间”的比值作为排序依据（-S var 参数），排名靠前的为可疑函数。部分函数正常设计时可能有较大的波动率，这类干扰需由用户排除。子函数的时间波动会传递到父函数中，因此父函数可能也会挤入 top 列表中，这部分干扰会由工具自行过滤（过滤规则：选用“最大执行时间/平均执行时间”的比值作为排序依据时，当子函数出现在 top 列表并且其比值高于父函数时，过滤掉父函数）；
- 2) 用自身平均执行时间-S self_a 进行排序，排序靠前的函数均为可疑函数。（单次执行或者执行次数不多的函数将会在这里出现）；
- 3) 经过 1、2 两步缩小函数排查范围，用户可以通过-X 显示的调用关系图进一步排查可疑函数的子函数（如果子函数为阻塞点，前面两步已经将其列出）。如果分析和排查的工作量还是较大，用户可尝试挂接更多的测量钩子（通过调低-kprof,min:n 中的 n 值，或者为更多模块加入-kprof 选项进行编译）。

8.3. 解析结果说明

解析工具默认显示进程汇总信息，加入-a 选项会将各个线程的记录数据分开显示，各个线程的显示

结构是相同的。

对每一个线程来说，解析结果主要包括三个部分：线程头记录、函数精确时间测量信息、调用关系信息。进程汇总信息的结构与线程相似不再赘述。

需要注意的是，精确计时的函数时间计入了阻塞时间。因此，测量结果可能与打点采样测量获取有所不同。另外，由于开启了小函数过滤功能，当小函数成为热点时（如一些字符串处理函数），精确计时不能直接给出热点小函数，而只能给出其父函数。并且，由于精确计时需要编译时插入测量桩，没有插桩的库函数都无法被测量。

部分多核 cpu 上，多个核心的时钟源不同步，当同一个函数出口和入口的时钟来自不同的核时，计时将会出现混乱。尽管这种函数在执行过程中被调度到其他核心上的概率很小（实测在万分之一以下），但是一旦出现，由于时钟源不同步，函数的执行时间在两个数值相减后会变成负值，而这个负值将被解析为很大的正值，最终导致某个时间数据变得异常的高。

为了解决该问题，计时工具内部已经做了过滤处理，对这类错误数据进行丢弃（由于这类数据的出现概率很低，丢弃并不影响测量的整体精确性）。但是对于一些关键的时间点和关键数据，由于无法丢弃，最终的结果仍可能出现这类异常数值。用户可以选择忽略该数据，也可选择重新测量一次。

用户也可以主动采取措施来避免该情况。最为简单和有效的方式就是绑定被测代码所在的线程到特定核心上。

■ 线程头记录

每个线程的数据会由如下标识来进行区分：

```
#####
thread id: 22134
thread name: test.out
thread started at 2011_09_15-22:22:00 used 1.328 seconds
#####
```

该线程头表明当前显示的为线程号 22134、线程名 test.out 的线程所得数据。

该线程开始时间是 2011 年 9 月 15 日晚上 22 点 22 分 0 秒。线程总共执行了 1.328 秒。

■ 函数精确时间测量信息

Prof result:								
self_t	%	max	min	avr	self_a	total	called	symbol
3.0s	50%	3.0s	3.0s	3.0s	3.0s	3.0s	1	B
2.0s	33.3%	2.0s	2.0s	2.0s	2.0s	2.0s	1	A

1.0s	16.7%	6.0s	6.0s	6.0s	1.0s	6.0s	1	main
------	-------	------	------	------	------	------	---	------

其各参数的含义分别是：

- self_t：函数总的自身执行时间（不含其子函数的执行时间）
- %：函数自身总执行时间占线程所用时间（在进程汇总中分母为进程中所有线程用时之和）的比例
- max：函数单次最大执行时间
- min：函数单次最小执行时间
- avr：函数单次执行的平均时间（函数消耗的总时间除以函数执行次数）
- self_a：函数平均自身执行时间
- total：函数消耗的总时间（包含其子函数）
- called：函数执行次数（第一行表示 B 函数被调用的次数）

➡ 提示：如果不用 -S 选项指定排序类型，则解析器将过滤掉性能记录中自身执行时间占比低于 1% 的函数，调用记录中自身执行时间占比低于 5% 的函数。

■ 调用关系信息（显示父函数）

Call Graph(From Bottom to Top):(TOP 5,max backtrace level 5)				
self_t	%	total	called	symbol
3.0s	50%	3.0s	1	B
				(100%:3.0s:1)main <源文件:行号>
2.0s	33.3%	2.0s	1	A
				(100%:2.0s:1)main <源文件:行号>
1.0s	16.7%	6.0s	1	main

其各参数的含义分别是：

- total：当前显示函数（包含子函数）的总时间消耗；
- %：自身执行时间与线程执行时间之比；
- self_t：当前显示函数自身消耗的时间（不包含其子函数的耗时）；
- called：当前显示函数的总执行次数。

红色部分的文字，**100%**表示 B 被 main 调用消耗的时间，占 B 总执行时间的比例。<>括号内显示函数来自的源代码文件和行号，此内容仅在-C 选项开启后才会显示。**3.0s: 1**表示 mian 函数调用 B 函数 1 次，main 函数消耗在 B 函数上的时间为 3 秒。

■ 调用关系信息（显示子函数）

Call Graph(From Top to Bottom):(TOP 5,max backtrace level 5)

self_t	%	total	called	symbol
3.0s	50%	3.0s	1	B
2.0s	33.3%	2.0s	1	A
1.0s	16.7%	6.0s	1	main

|_(50%:3.0s:1)B <源文件:行号>

|_(33.3%:2.0s:1)A <源文件:行号>

其各参数的含义分别是：

- total : 当前显示函数（包含子函数）的总时间消耗；
- %: 自身执行时间与线程执行时间之比；
- self_t : 当前显示函数自身消耗的时间（不包含其子函数的耗时）；
- called : 当前显示函数的总执行次数。

红色部分的文字，**50%**表示 B 被 main 调用消耗的时间，占 main 总执行时间的比例。<>括号内显示函数来自的源代码文件和行号，此内容仅在-X 选项开启后才会显示。**3.0s: 1**表示 mian 函数调用 B 函数 1 次，main 函数消耗在 B 函数上的时间为 3 秒。

8.4. 用于开启时钟寄存器访问权限的内核模块代码

如下内核代码将用于开启时钟寄存器访问权限：

```
#include <linux/kernel.h>
#include <linux/module.h>

MODULE_DESCRIPTION("ZTE module example");
MODULE_LICENSE("GPL");

static int init_perfcounters ()
{
    /* enable user-mode access to the performance counter*/
    asm ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));
}
```

```
/* disable counter overflow interrupts(we check overflows in userspace now) */
asm ("MCR p15, 0, %0, C9, C14, 2\n\t" :: "r"(0x8000000f));

}

static int mod_init(void)
{
    printk( "<1> Module init:user-mode access to performance registers enabled!\n");
    init_perfcounters();
    return 0;
}

static void mod_exit(void)
{
    printk( "<1> Module exit:user-mode access to performance registers disabled!\n");
    asm ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(0));
}

module_init(mod_init);
module_exit(mod_exit);
```

9. FAQ

9.1. 如何判断当前环境的采样类型

- 关键字： 采样类型
- 版本： all
- 问题描述：

如何判断当前环境是基于定时器方式的采样还是基于性能计数器的采样？

- 分析及处理：

在 oprofile 模块加载成功，oprofilefs 文件系统挂接成功之后，查看/dev/oprofile/cpu_type 的值即可知道。

使用定时器方式返回值如下：

```
#cat /dev/oprofile/cpu_type
#timer
```

使用性能计数器返回值为当前环境的 CPU 类型，如 x86 虚拟机上的结果为：

```
#cat /dev/oprofile/cpu_type
#i386/p4
```

9.2. 设置调用图深度不成功

- 关键字： 调用图深度 设置失败
- 版本： all
- 问题描述：

使用 **kprofiler --start -c 5** 设置图深度时，报如下错误：

```
“Count 500 for event DL1_RELOADS is below the minimum 7500”
```

这是为什么？

- 分析及处理：

出现此种情况，是由于使用 **callgraph** 时要求计数器阈值为最小值 **x15**，此时只需将计数器阈值设置为文档规定的最小值的 15 倍即可解决此类问题。

9.3. 多个事件采样时报错

- 关键字： 多个事件采样 设置失败
- 版本： all
- 问题描述：

当使用 **kprofiler --start -e** 设置多个事件采样时，报如下错误：

```
Couldn't allocate hardware counters for the selected events.
```

这是为什么？

- 分析及处理：

出现此状况，是由于基于性能计数器的采样需遵循两个原则：

- 1) 设置的事件个数不能超过性能计数器总数；

- 2) 使用同一个性能计数器的多个事件互斥。例如：事件 1 和事件 2 都只使用编号为 3 的性能计数器记数，那么在一次采样过程中不能同时设置事件 1 和事件 2。

9.4. 评测结束无结果显示

- 关键字： 统计结束 无结果显示
- 版本： all
- 问题描述：

当评测结束显示统计结果时，却无结果显示，报错如下：

```
Xxxxxxx(File name too long)
```

这是为什么？

- 分析及处理：

出现此种情况，是由于 KProfiler 可执行文件/待评测用户程序放置的绝对路径较深，导致 KProfiler 采样生成的采样文件会放置在很深的路径下，特别是 KProfiler 使用回溯采样的方式进行采样的话，放置的回溯采样文件的路径会更深。后续解析的时候解析工具在使用 stat 函数查看文件信息时，发现路径很深超出了本环境的最大路径长度，无法处理，失败报错。

解决方法：采样时不要将 KProfiler 可执行文件/内核符号表 vmlinux/待评测用户程序放置在很深的目录中。

9.5. 开始采样执行清除采样数据操作报错

- 关键字： 开始采样 清除前期采样数据 出错
- 版本： all
- 问题描述：

以 NFS 方式启动目标端环境，使用 **kprofiler --stop -k** 命令结束采样关闭守护进程后，立即执行 **kprofiler --start -r** 命令启动采样，并执行清除前期采样数据操作，却报“无权限删除文件”等类似问题，这是为什么？

- 分析及处理：

出现此种情况，是由于网络原因导致。可在执行完结束采样的命令后，延迟几秒后再进行清除采样数据的操作，这时问题就可解决。以 CPIO 方式启动目标端环境，将不会出现类似问题。

9.6. 结束采样后打包数据报错

- 关键字： 结束采样 打包数据 出错

- 版本: all
- 问题描述:

使用 **kprofiler --stop -k** 结束采样关闭守护进程, 提示 “Daemon stuck shutting down; killing !”, 之后进入 `var/lib` 目录下去打包 `oprofile` 文件, 报如图 9-1 所示错误, 这是为什么?

```
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/oprofiled/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/oprofiled/TIMER.0.0.all.all.all
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/oprofiled/<cg>/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/oprofiled/<cg>/<root>/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/oprofiled/<cg>/<root>/oprofile_result/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/oprofiled/<cg>/<root>/oprofile_result/bin/
./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>/opr
oprofile_result/bin/oprofiled/<cg>/<root>/oprofile_result/bin/oprofiled/
tar: ./oprofile/samples/current/<root>/oprofile_result/bin/oprofiled/<dep>/<root>
/<root>/oprofile_result/bin/oprofiled/<cg>/<root>/oprofile_result/bin/oprofiled/TIMER.
0.0.all.all.all: File name too long
./oprofile/samples/oprofiled.log
tar: error exit delayed from previous errors
#
```

图 9-1 打包文件报错

- 分析及处理:

使用 **kprofiler --stop -k** 结束采样, 出现 “Daemon stuck shutting down; killing !” 提示, 是由于当我们执行 **kprofiler --stop -k** 命令后, 程序停止守护进程 (kill -TERM) 后并不会立即结束, 而是还有一段循环处理, 每循环一次睡眠 1 秒, 直到找不到 `oprofiled` 的进程号或者是循环次数大于或等于规定次数为止。若出现循环次数大于或等于规定次数时, 就会打印信息 “Daemon stuck shutting down; killing !”, 然后再杀死守护进程 (kill -9)。这个提示与采样后的数据打包无任何关联。打包失败是由于在所使用的目标端环境中 `tar` 命令压缩文件名超出了其规定的字符数, 所以报错为 “Filename too long”。

9.7. 计数器采样, 不评测内核依然可采集到内核数据

- 关键字: 计数器 不评测内核 采集到数据
- 版本: all
- 问题描述:

采用基于计数器方式进行采样, 在事件配置时设置为不评测内核, 采样后, 解析数据, 依然可以解析出内核数据, 这是为什么?

➤ 分析及处理:

出现此种情况,是由于只要当状态寄存器的两位(MIPS 架构下为 EXL 位和 ERL 位)都被置为 0 时,就会对内核进行计数采样。而当设置不评测内核,即设置 kernel 为 0 时,在完成状态寄存器的 EXL 位和 ERL 位设置并生效的这段置位的时间内,可能会对内核进行采样,所以会出现上述情况。这也是为什么设置不评测 user,但仍可能会采集到 user 的一点数据的原因。各个架构关于计数器 kernel、user 置位生效的情况可能会有所不同,需要具体情况具体分析。

9.8. 设置堆栈深度后无堆栈回溯信息显示

➤ 关键字: 堆栈深度 无回溯信息显示

➤ 版本: all

➤ 问题描述:

成功启动内核后,插入 KProfiler 模块,设置堆栈深度并启动采样,结束采样后关闭,进行采样解析,却无法对采样回溯信息进行查看,这是为什么呢?

➤ 分析及处理:

出现此种情况,是由于采样符号表文件的放置路径过深,从而导致包含采样符号表文件放置路径信息的采样的回溯信息文件的文件名过长,超出了所支持的最大路径长度,使得无法读取到该采样回溯信息文件,因而无法予以显示。因此,若是希望对采样回溯信息进行查看,请勿将待采样的符号表文件放置在过深的路径下。

9.9. 查看解析结果 overflow 值较大

➤ 关键字: overflow 值较大

➤ 版本: all

➤ 问题描述:

查看解析结果,发现 event_overflow 值为 0 或数值很小,而某个 CPU 下的 overflow 值较大, received 值大小也近似于 overflow,但采集结果中无该 CPU 的数据文件,这是为什么呢?

➤ 分析及处理:

统计信息中的 overflow 文件主要用于指明,当指定 CPU 上保存采集样本时,内核缓存的剩余空间不够。引起此种状况的原因是: sync_buffer 任务同步的速率小于样本采集的速率。另外最终的数据文件中没有该指定 CPU 的数据文件,说明用户态工具没有从/dev/oprofile/buffer 中获取到该 CPU 的数据,结合 received 和 overflow 数据相差不多,可以判断该 CPU 上的 sync_buffer 任务没有被调度。

解决方法:

出现此问题通常都是由于指定 CPU 上存在高优先级死循环进程,导致 sync_buffer 任务永远得不到调

度，可以通过工具将该 CPU 上的 event 任务绑定到有空闲的 CPU 上，使得 sync_buffer 任务可以被调度。

如果采集结果中有该 CPU 的数据，但是 overflow 值依然存在，则在使用性能计数器的情况下，可以调整计数器溢出值，降低采样频率，以使同步进度大于等于采样进度。

9.10. 查看解析结果 event_overflow 值较大

➤ 关键字： event_overflow 值较大

➤ 版本： all

➤ 问题描述：

查看解析结果，发现统计信息中 overflow 值很小或者为 0，而 event_overflow 值很大，这是为什么呢？

➤ 分析及处理：

event_overflow 有记录表明 KProfiler 用户态工具从/dev/oprofile/buffer 读数据的速度小于从内核缓存向文件缓冲同步的速度，从 received 和 overflow 的值来看，内核模块部分的处理是正常的，问题在用户态工具这边。通常的原因是：用户态工具的优先级太低，得不到调度。

解决方法：

使用优先级调整工具将 KProfiler 用户态工具的优先级调高或者使用 KProfiler 的采样工具 kprofiler 的 -P 参数调整 KProfiler 用户态工具的优先级。

9.11. 两种采样方式结果差异很大

➤ 关键字： 采样方式不同 结果差异大

➤ 版本： all

➤ 问题描述：

在 x86 系列的处理器上，使用 KProfiler 进行性能调优，在相同的环境下，且保证基于性能计数器采样频率等于基于定时器方式采样频率的前提下，发现 KProfiler 基于这两种不同采样方式所得到的采样结果有较大差异：使用定时器方式采样的结果中 99% 的样本都在 mwait_idle（cpu 的 idle 进程）中，而基于性能计数器的采样结果中却几乎看不到 idle 函数的样本，这是为什么呢？

➤ 分析及处理：

对于 x86 系列的处理器，我们在使用基于性能计数器采样时，如果使用的事件是 CPU_CLK_UNHALTED，其含义是非 halt 状态的处理器时钟周期，当处理器处于非 halt 状态时，每一个处理器时钟周期都会产生一个事件。在默认情况下，出于在 CPU 空闲时节能的目的，idle 进程会发出 HLT 指令使 CPU 进入挂起状态，直到有新的调度需求出现。此时不会产生 CPU_CLK_UNHALTED 事件，KProfiler 无法进行采样。

9.12. 计数器采样，结果大多落在软中断

- 关键字： 计数器采样 采集软中断
- 版本： all
- 问题描述：

采用计数器的方式进行采样，发现 59% 的采样结果都落到了软中断上，现象如下：

```
# ./kprofiler
Using 2.6+ OProfile kernel interface.
Daemon started.
Start sample time : 01:09:36
Profiler running.
# ./kprofiler --stop
Stopping profiling.
Stop sample time : 01:09:42
Sample time : 5.963440s
Killing daemon.
# ./kprofiler --show
Daemon not running
Event 0: CPU_CLK:1199000:0:1:1
NR_CHOSEN=1
Session directroy=/var/lib/oprofile/
Separate options:  library kernel cpu thread
vmlinux file: /vmlinux
Call-graph: 0
# ./opreport cpu:1 -l /vmlinux
Sample Statistical Information :
```

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7
received	5964	5964	5963	5964	5963	5964	5963	5964
overflow	0	0	0	0	0	0	0	0
%(overflow)	0	0	0	0	0	0	0	0
backtrace(aborted)	0	0	0	0	0	0	0	0
%(backtrace)	0	0	0	0	0	0	0	0
received(total)	no-mapping(backtrace)	% event-overflow	% no-mapping(sample)	% error				
47709	0	0	0	0	0	0	0	0

```
Start Sample Time : 01:09:36
```

```
Stop Sample Time : 01:09:42
Sample time : 5.963440s
image=/vmlinux
CPU: e500, speed 1199.99 MHz (estimated)
Counted CPU_CLK events (Cycles) with a unit mask of 0x00 (No unit mask) count 1199000
Running Samples:
  CPU_CLK:1199000|
  samples|      %|
  -----
samples  %      symbol name
3487    59.0917  ____do_softirq
2404    40.7389  cpu_idle
8        0.1356  run_timer_softirq
1        0.0169  _spin_unlock_irq
1        0.0169  find_next_bit
```

出现这种情况，这是为什么呢？

➤ 分析及处理：

经分析，出现此种状况是由于用户设置的默认事件的采样频率很接近当前内核的时钟频率（但并不相等），从而导致采样点大部分落在中断里面。要避免此种状况，用户需手动调整采样频率，使其明显区别于内核的时钟频率，具体方式参见 [3.1.2.2](#)。

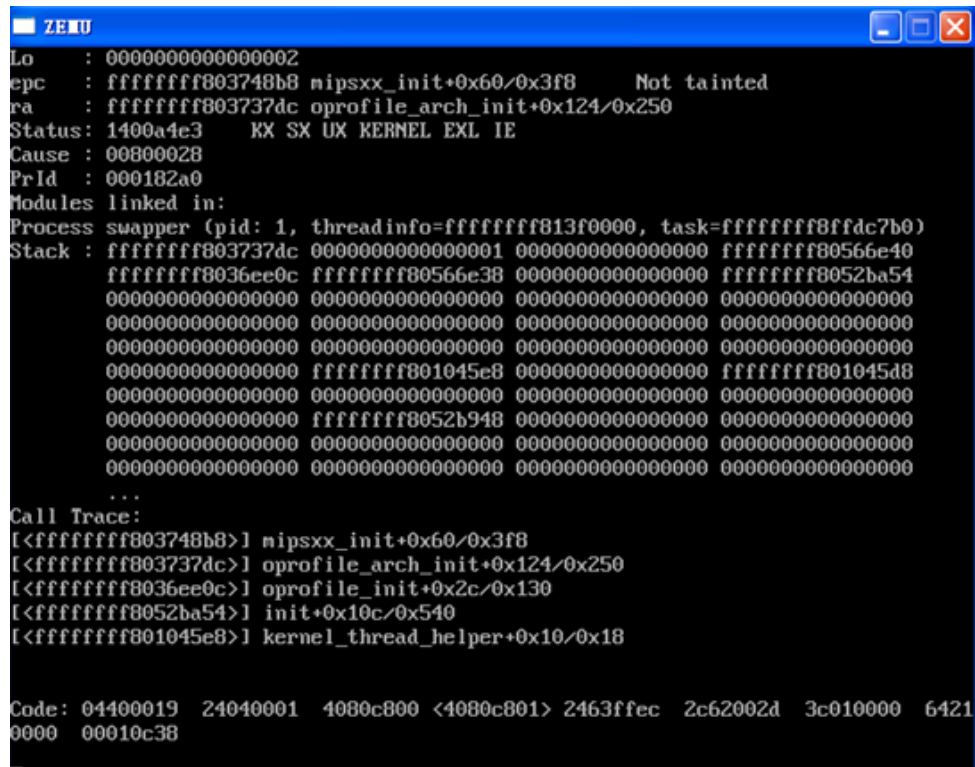
9.13. 在 zemu mips64 仿真环境下配置 oprofile 后内核启动异常死机

➤ 关键字： mips64 zemu oprofile 配置进内核 启动异常

➤ 版本： all

➤ 问题描述：

在 mips64 的 zemu 仿真环境中，将 oprofile 模块配置进内核，内核启动时异常挂起；将 oprofile 配置成模块时，插入模块同样也会导致系统异常挂起。



```
ZEMU
Lo : 0000000000000002
epc : ffffffff803748b8 mipsxx_init+0x60/0x3f8 Not tainted
ra : ffffffff803737dc oprofile_arch_init+0x124/0x250
Status: 1400a4c3 KX SX UX KERNEL EXL IE
Cause : 00800028
Prid : 000182a0
Modules linked in:
Process swapper (pid: 1, threadinfo=fffffffff813f0000, task=fffffffff8ffdc7b0)
Stack : ffffffff803737dc 0000000000000001 0000000000000000 ffffffff80566e40
        ffffffff8036ee0c ffffffff80566e38 0000000000000000 ffffffff8052ba54
        0000000000000000 0000000000000000 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000 0000000000000000
        0000000000000000 ffffffff801045e8 0000000000000000 ffffffff801045d8
        0000000000000000 0000000000000000 0000000000000000 0000000000000000
        0000000000000000 ffffffff8052b948 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000 0000000000000000
        ...
Call Trace:
[<ffffffff803748b8>] mipsxx_init+0x60/0x3f8
[<ffffffff803737dc>] oprofile_arch_init+0x124/0x250
[<ffffffff8036ee0c>] oprofile_init+0x2c/0x130
[<ffffffff8052ba54>] init+0x10c/0x540
[<ffffffff801045e8>] kernel_thread_helper+0x10/0x18

Code: 04400019 24040001 4080c800 <4080c801> 2463ffec 2c62002d 3c010000 64210000 00010c38
```

9-2 内核启动异常

这是为什么呢？

➤ 分析及处理：

出现该状况是由于 mips64 的 zemu 环境下暂时不支持对性能计数器寄存器的操作，导致 oprofile 初始化的过程中读这些寄存器的时候异常死机。因此，在构建内核时不能将 oprofile 模块配置进内核，同样也不能在内核启动后将 oprofile 以模块的形式插入内核。

9.14. 图形界面 GUI 控制目标端采样遭遇干扰导致采样失败

➤ 关键字： x86 zemu 图形界面 GUI 采样失败

➤ 版本： all

➤ 问题描述：

采用图形界面 GUI 对目标端 x86 zemu 上的用户态程序进行采样解析，采样结束后，通过对已挂载的 KProfiler 配置点击【控制】->【停止】，中止数据采样，停止采样失败，这是为什么？

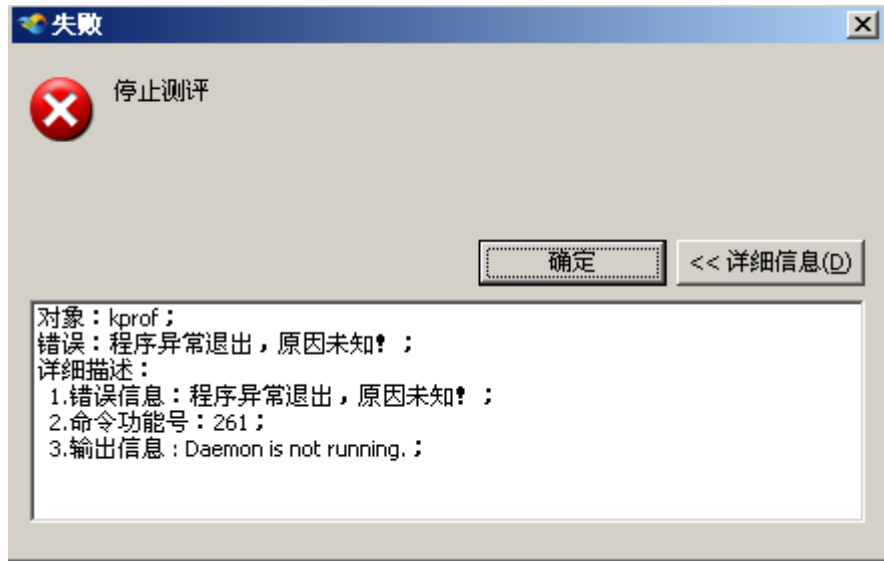


图 9-3 停止采样失败

➤ 分析及处理:

经查看, 在由 KProfiler GUI 发送停止采样命令之前, 已经有其他用户登录目标机, 并手动停止了采样, 从而导致图形界面下停止采样失败, 整个采样过程被第三方干扰。该问题也会出现在, 当其他用户登录目标机启动采样后, 再由图形界面开启采样, 启动采样失败。从而, 为避免此类问题发生, 需保证同一台目标机在某个时间段内进行数据采样解析过程中, 采样的开启、结束命令由同一个操作端发送, 即仅由一台 KProfiler GUI 所连接、控制或者仅由同一个用户登录目标端控制, 保证采样过程的完整。

9.15. 图形界面 GUI 进行数据采样无法获取采样结果

➤ 关键字: x86 zemu 图形界面 GUI 无法获取采样结果

➤ 版本: all

➤ 问题描述:

采用图形界面 GUI 对目标端 x86 zemu 上的用户态程序进行采样解析, 采样结束后, 解析数据输出有误, 并未采集到任何数据。

Sample Statistical Information :	
CPU0	
received	0
overflow	0
%(overflow)	0

```

backtrace(aborted)0
%(backstrace)      0
                    received(total)  no-mapping(backtrace)%          event-overflow  %
no-mapping(sample)%          error          %
                    0                0                0                0                0
0                            0                0                0
Start Sample Time : 10:14:08
Stop Sample Time : 10:14:13
Sample time : 5.33413s

Running sample number : 0

error: no sample files found: profile specification too strict ?

```

这是为什么？

➤ 分析及处理：

经查看，此 KProfiler 配置保持了默认的采样方式“计数器采样”，而目标端为 x86 架构的虚拟机，目前虚拟机都不支持计数器采样方式，但由于 KProfiler 无法区分是运行于真实硬件，还是在虚拟机上运行，所以某些架构的虚拟机在界面上显示可以选择计数器采样方式，从而导致采样失败。用户在虚拟机上采样，请选择“定时器采样方式”，以确保正常采样。

9.16. 当 Linux 环境配置了 nmi_watchdog 时如何采样

➤ 关键字：nmi_watchdog 采样

➤ 版本：all

➤ 问题描述：

KProfiler 使用性能计数器采样时，产生的中断是 NMI 中断，如果环境配置了 nmi_watchdog，就会产生冲突，在启动 KProfiler 采样时，报如下错误：

```

nmi_watchdog using this resource ? Try:
echo 0 > /proc/sys/kernel/nmi_watchdog

```

➤ 分析及处理：

在采样结束后，使用 opreport 解析时会发现没有采到样本数据。

此时，应按照如下步骤使用 KProfiler 采样：

- 1) 卸载 oprofilefs 文件系统，使用命令 `umount /dev/oprofile`
- 2) 卸载 oprofile 模块，使用命令 `rmmod oprofile`
- 3) 关闭 nmi_watchdog，使用命令 `echo 0 > /proc/sys/kernel/nmi_watchdog`
- 4) 加载 oprofile 模块，参考 3.1.1 节
- 5) 使用 KProfiler 进行采样
- 6) 结束采样
- 7) 解析数据
- 8) KProfiler 工具使用结束后，重复 1 和 2，卸载 oprofile 模块
- 9) 开启 nmi_watchdog，使用命令 `echo 1 > /proc/sys/kernel/nmi_watchdog`

如果 oprofile 链入内核，在内核启动参数中增加 `nmi_watchdog=0` 然后重起系统。

10. 开源 LICENSE 说明

10.1. GPL 简介

GPL 是 GNU 通用公共许可证的简称，为大多数的 GNU 程序和超过半数的自由软件所采用，Linux 内核就是基于 GPL 协议而开源。GPL 许可协议对在 GPL 条款下发布的程序以及基于程序的衍生著作的复制与发布行为提出了保留著作权标识及无担保声明、附上完整、相对应的机器可判读源码等较为严格的要求。GPL 规定，如果将程序做为独立的、个别的著作加以发布，可以不要求提供源码。但如果作为基于源程序所生著作的一部分而发布，就要求提供源码。

成都研究所 OS 平台基于开源社区研发提供适用于各产品线的嵌入式 Linux 产品及相关工具，从总体而言，主要应该遵循 GPL 许可协议的条款。Linux 是基于 GPL2.0 发布的开源软件，CGEL 是基于 Linux 内核进行修改完成的新作品，因此，根据 GPL 协议第 2 条的规定，CGEL 属于修改后的衍生作品，并且不属于例外情况。因此，CGEL 在发布时需开放源代码，具体形式需要符合 GPL 协议第 3 条的规定。

10.2. 开发指导

为尽可能地保护公司在 Linux 方面的自有研发成果，特别是产品线的应用程序，同时顺从于 GPL 协议条款的规定，这里为大家提供一些开发指导原则供参考，以便有效地规避由于顺从 GPL 条款所带来的风险。

■ 应用程序尽可能运行于用户态

应用程序尽量放到用户态运行，应用程序对于内核的访问、功能使用主要通过信号、数据、文件系统、系统调用，基本属于松耦合，而且绝大部分系统调用都是通过 C 库封装进行，能比较充分地证明应用的独立性、可移植性，避免开源。

■ 以动态链接方式链接开源库

以 GLIBC 库为例，应用程序对于库的链接方式应尽量采用动态链接，这样可遵循 LGPL，避免公开源码。如果需要静态链接，则可以设计一个封装容器，与开源库静态链接在一起，这样只需开放封装容器源码。其余应用程序以动态链接方式与封装容器链接，只需要提供二进制文件。

■ 内核应用采用模块加载方式

内核应用可以采用静态链接、模块加载方式加入内。

静态链接方式是将所有的模块（包括应用）都编译到内核中，形成一个整体的内核映像文件。这种情况下，比较难以鉴别应用程序与内核的独立性、可移植性，按照 GPL 的规定，就很可能被要求开发源代码。

模块加载方式是指将上层应用独立地编译连接成标准 linux 所支持的模块文件 (*.mod)。运行时，首先启动 Linux 内核，然后通过 insmod 命令将各模块按照彼此间的依赖关系插入到内核。该方式下应用程序相对独立于 Linux 内核，有可能被证明为独立的作品，与 OS 无关，认为是纯粹的聚集行为，从而可以不用开放源码。

因此，对于运行于内核态的驱动和应用软件应尽量采用模块加载方式进行独立编译，通过模块的可动态装载、卸载，以及跨 OS 的可移植性来证明应用与 OS 的独立性。

10.3. 源码获取方式

广东中兴新支点所发布的 CGEL 操作系统是基于开源软件 Linux 2.6.21 版本开发的，遵从 GPL 协议开放源代码。如果你需要源码，请发送源码申请邮件到 cgel@gd-linux.com 联系获取源码。

附录 A

典型事件列举

表 10-1 I386/piii 典型事件

Name	Description	Counters usable	Unit mask options
CPU_CLK_UNHALTED（默认）	clocks processor is not halted	all	
ITLB_MISS	number of ITLB misses	all	

表 10-2 I386/p4 典型事件

Name	Description	Counters usable	Unit mask options
GLOBAL_POWER_EVENTS（默认）	time during which processor is not stopped	0, 4	0x01: mandatory
ITLB_REFERENCE	translations using the instruction translation lookaside buffer	0, 4	0x01: ITLB hit 0x02: ITLB miss 0x04: uncachable ITLB hit

表 10-3 X86_64/hammer 典型事件

Name	Description	Counters usable	Unit mask options
CPU_CLK_UNHALTED（默认）	clocks processor is not halted	all	

L1_DTLB_AND_L2_DTLB_MISS	L1 and L2 DTLB misses	all
L1_ITLB_MISS_AND_L2_ITLB_MISS	L1 ITLB miss and L2 ITLB miss	all

表 10-4 Ppce500 典型事件

Name	Description	Counters usable	Unit mask options
CPU_CLK（默认）	Cycles	all	

表 10-5 Ppce500mc 典型事件

Name	Description	Counters usable	Unit mask options
CPU_CLK（默认）	Cycles	all	

表 10-6 Mips732 典型事件

Name	Description	Counters usable	Unit mask options
CYCLES（默认）	Processor clock cycles	all	