Using the MIPS32® 34K® Core Performance Counters

The MIPS32® 34K® core provides four performance counters and over 100 countable events. The developer can access and use these counters using a number of different interfaces and tools. In this application note, we present the performance counters and countable events, discuss the means to access and use the counters, and provide examples of use to obtain information useful for application profiling and optimization.

Document Number: MD00548
Revision 01.01
August 22, 2011

Contents

1 Introduction

The MIPS32® 34Kc™ core [1] implements the MIPS® Multi-Threading (MT) ASE [3], which provides hardware support for multi-threading of software through the implementation of virtual processor elements (VPEs) and thread contexts (TCs). Through the use of multi-threading, software applications can be executed by the MIPS 34Kc in fewer cycles than on single-threaded MIPS cores such as the MIPS32® 24K® core. Reducing execution time allows for a range of possible benefits, including:

- Increased throughput, e.g., decoding more output MPEG-2 video frames per second,
- Using the extra available processor time to add functionality,
- Lower energy consumption, through the ability to reduce processor clock speed for the same throughput, and,
- Lower cost, if lower frequency and hence less costly system components can be used.

Application software support for multi-threading is provided at the operating system level in SMTC Linux for the 34Kc core. Typically, existing multi-threaded application software compiled for the 24K family of cores may be executed without change on the 34Kc core, and will execute in fewer cycles due to the MIPS 34Kc core's hardware-based multi-threading support. However, to obtain the best performance extra development work may be required to optimize the application's execution time performance. In this application note, we discuss the hardware and software tools available for making detailed measurements of the execution of a software application, and using that information to improve the performance of the application.

We begin with an overview of the 34Kc core architecture in Section 2 "MIPS32® 34Kc™ Core Multi-Threading Architecture", focusing on the hardware support for multi-threading. In Section 3 "Performance Counters" we introduce the performance counters and the aspects of the processor that may be measured with them. Section 4 "Using the Performance Counters" discusses how to make measurements using the performance counters, and introduces an application for simplifying and automating the use of the performance counters, `pc_sweep`. In Section 5 "Examples of Use" we present examples of using the performance counters to understand the behavior of two different types of application.

We begin with an overview of the 34Kc core architecture in Section 2 "MIPS32® 34Kc™ Core Multi-Threading Architecture", focusing on the hardware support for multi-threading. In Section 3 "Performance Counters" we introduce the performance counters and the aspects of the processor that may be measured with them. Section 4 "Using the Performance Counters" discusses how to make measurements using the performance counters, and introduces an application for simplifying and automating the use of the performance counters, `pc_sweep`. In Section 5 "Examples of Use" we present examples of using the performance counters to understand the behavior of two different types of application.

2 MIPS32® 34Kc™ Core Multi-Threading Architecture

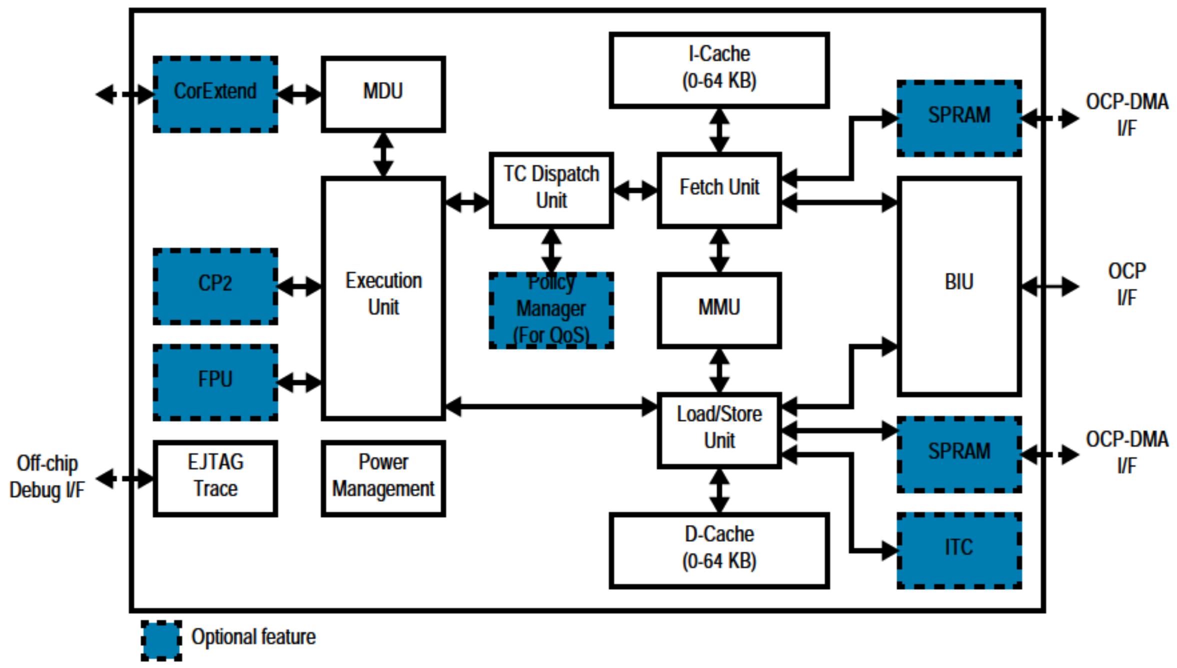
But first, we will discuss a new approach to multi-threading at the hardware level that offers high performance on threaded and unthreaded code: the 34K family of cores and the MIPS Multi-Threading ASE (MT ASE).

In the following sections we present an overview of the 34Kc™ core architecture and instruction set features, with a focus on the features directly relevant to multi-threading. We will then discuss the thread switching mechanism before presenting benchmark results for a set of multi-threaded applications. Readers familiar with the 34Kc core may wish to skip this section and move directly to Section 3 "Performance Counters".

2.1 Architecture

The 34Kc core implements the MIPS Multi-Threading ASE (MT ASE) [3], and extends the 24Kc core architecture with additional hardware and instruction set support for up to two *virtual processing elements* (VPEs) and up to nine hardware *thread contexts* (TCs). Each TC stores the state for a single thread of execution. The definitions of both VPEs and TCs are briefly expanded upon in the following sections. Figure 1 depicts the 34Kc core architecture

Figure 1 MIPS32® 34Kc™ Core Architecture.



A 34Kc core implementation may include 0, 8, 16, 32, or 64 KBytes of instruction and data caches. Implementations of the 34Kc core may also include a floating-point unit, CorExtend™ user-defined instructions, scratchpad memory, a front-side level two (L2) cache, and a fixed-mapping memory management unit (MMU). These optional features are not directly relevant to the use of multi-threading. Readers who wish to learn more about the 34Kc core architecture are referred to [1] and [2].

Implementations of the 34Kc core may also include inter-thread communication storage (ITC), a means of synchronizing thread execution and data flow using gating storage. An ITC is a set of 64-bit cells that exist in the memory map and are accessed using load and store operations. The address used to access a 64-bit cell defines the cell's view, which in turn defines access-based behavior. For example, access to a particular view may block or not depending on whether the ITC is full or empty, allowing fine-grained thread synchronization. Note that this document is principally concerned with the use of multi-threading with high-level software such as the POSIX® Pthreads library, which at the time of writing did not support ITC. Applications executing on bare iron may be expected to make heavy use of ITC to implement synchronization mechanisms typically provided by the OS. For further information on ITC consult [1].

2.2 Virtual Processing Element (VPE)

A virtual processing element (VPE) contains at least one thread context and is an instantiation of the full MIPS32® ISA and privileged resource architecture (PRA), sufficient to run a per-processor OS image. A VPE can be thought of as an “exception domain”, as exception state and priority apply globally within a VPE, and only one exception can be dispatched at a time on a VPE.

n a processor with multiple VPEs, one VPE can be assigned to be a “master” VPE that has privileged access to the resources of all VPEs using `MTTR`/`MFTR` instructions and can suspend or resume other VPEs. This allows one OS to operate as a master, dispatching tasks to slave OSes. Other VPEs may assume (or be given) the role of master at run-time.

This application note does not consider VPE-related issues, but assumes execution on a single processor or multi-threaded VPE.

2.3 Hardware Threads and Thread Context

The 34K family of cores provide a rich set of controls to govern the behavior of VPEs and thread contexts (TCs). Only a brief summary of TC operation is provided here, to provide background for the efficiency and capabilities of the 34Kc core.

From the perspective of the 34Kc core hardware, a software thread is a stream of instructions executed in the order the instructions were written. The 34Kc core provides hardware support, the thread context, that preserves the processor state sufficient to describe the state of a software thread. The hardware for each TC is identical, i.e., there are no TCs reserved specifically for Kernel Mode or User Mode threads. A TC preserves at least the following processor state:

- The program counter and 32 integer general-purpose registers
- The *HI*/*LO* register pair and the additional accumulator registers defined in the MIPS DSP ASE
- The privilege mode bits from the *Status* register
- The address space ID (the ASID field of the Coprocessor 0 *EntryHi* register), to allow the software thread assigned to the TC to operate in a separate address space
- Coprocessor access state bits from the *Status* register

To minimize the hardware required to hold and maintain the TC, many registers are not stored in the TC. This includes, for example, Coprocessor 1 registers.

A TC can be *free* or *activated*. Only free TCs may be allocated to new threads and only activated TCs may be scheduled by the thread scheduler. An activated TC may be *running* or *blocked*. Instructions are fetched and executed for running TCs according to the implementation-dependent scheduling policy in effect. One such policy, provided by MIPS, is a round-robin policy that issues one instruction from each running TC in turn. An instruction from a running TC may be momentarily stalled, for example, due to functional unit delays. A blocked TC is one that has issued an instruction that depends on an explicit synchronization event that has not yet occurred. For example, a thread assigned to a TC may request synchronization with an external event via the external yield manager. The thread will be in a blocked state until the external condition is met, at which point the TC may be rescheduled for execution (i.e., will return to the running state). This feature can be used to automatically and efficiently synchronize software with fixed-function hardware.

Independent of being free or activated, a TC can be *halted*. A halted TC cannot be allocated, even if marked free, and cannot execute instructions, even if activated. The state of a halted TC is stable, making this state useful when, for example, software must support more threads of execution than there are hardware TCs. An activated TC can also be placed *offline* by code executing in the EJTAG Debug Mode, preventing the TC from being scheduled.

An activated TC may cause an exception during execution, causing the remaining activated TCs within the parent VPE to be *suspended* while the exception is handled. Activated TCs may also be suspended if privileged software

deactivates multithreaded execution. A suspended TC can have active instructions in the pipeline, but cannot take any action that would cause an exception or otherwise change VPE state. (Exception handling is complicated in a multi-threading system; the reader is encouraged to review [3] to learn more.)

It is important to note that apart from initial configuration, the process of switching execution among a pool of activated threads is automatic and independent of software intervention. Hardware state allows a thread switch on the MIPS 34Kc core to complete in a single cycle, significantly faster than the tens to thousands of cycles of thread switching latency in an operating system executing on a single-threaded processor.

2.4 New Instructions

The 34Kc core implements the eight instructions of the MIPS MT ASE [3]:

- `FORK` and `YIELD` control thread allocation, deallocation, and scheduling, and are available in all execution modes if implemented and enabled.
- `MFTR` and `MTTR` (*move from thread context* and *move to thread context*, respectively) are system coprocessor (Coprocessor 0) instructions available to privileged system software for managing thread state.
- `EMT` and `DMT` are privileged Coprocessor 0 instructions for enabling and disabling multi-threaded operation of a VPE.
- `EVPE` and `DVPE` are privileged Coprocessor 0 instructions for enabling and disabling multi-VPE operation of a processor.

These instructions will cause a Reserved Instruction exception if executed by a processor not implementing the MIPS MT ASE.

Ideally, each software thread is mapped to one TC. However, the user—or more usually, the operating system—can allocate more software threads than there are hardware TCs, creating and managing thread state through the use of the instructions listed above. In this application note we assume that there are always free hardware TCs that can be used to execute the software application's threads, which offers the lowest thread switching cost and the lowest execution time.

In general, application software need not know or use the instructions listed above. Operating systems such as the SMTC Linux kernel may use the new instructions to efficiently provide multi-threading support without further application programmer intervention.

3 Performance Counters

Performance counters are provided in a processor to count each occurrence of user-specified events. The 34Kc core extends the performance counters of the 24Kc core family to provide:

- Four performance counters shared between the two VPEs
- The ability to count events per TC, per VPE, or per processor
- Additional countable events, e.g., `fork` and `yield` instructions and ITC accesses.

In Section 3.1 “Configuration” we present details of the performance counter configuration registers and list the different events that can be counted. In Section 3.2 “Events Useful to Software Developers” we describe which events are the most useful to a software developer, and why.

Note: Implementation of the performance counters is optional, and software should first check for the presence of the performance counters before use. When performance counters are present, the PC bit of the *Config1* register is set to 1.

3.1 Configuration

The four performance counters and four associated control registers are mapped to the system coprocessor (Coprocessor 0) register 25. The select field of the `MTC0`/`MFC0` instructions are used to select the specific register accessed by the instruction, as shown in Figure 2.

Each counter is a 32-bit read/write register (interpreted as an unsigned 32-bit value) that is incremented by one each time the countable event specified in the associated control register occurs. Each counter can independently count one type of event at a time. An interrupt signal, `SI_PCI`, is sent to the core when any counter has both the interrupt enable (IE) bit of the corresponding control register set and has bit 31 set (i.e., the counter is half-full). The interrupt allows software to take action when a counter value exceeds the half-full mark, such as extending the range of the counter to avoid overflow, writing a zero to the counter (which may clear the interrupt), or even changing to counting a different type of event.

Figure 2 shows the format of the Performance Counter Control register; Table 1 describes the Performance Counter Control register fields. The available 34Kc core implementations may provide more than the minimum set of four counters. When only four counters are implemented they are grouped into two pairs: the even pair (counters 0 and 2) and the odd pair (counters 1 and 3

Figure 2 Performance Counter Register

31	30	29 22	21 20	19 16	15 12	11 5	4	3	2	1	0
M	0	TCID	MT_EN	VPEID	0	Event	IE	U	S	K	EXL

Table 1 Performance Counter Register Selects Used in MTC0 and MFC0 Instructions

Select[2:0]	Register
0	Register 0 Control
1	Register 0 Count
2	Register 1 Control
3	Register 1 Count
4	Register 2 Control
5	Register 2 Count
6	Register 3 Control
7	Register 3 Count

Table 2 lists the countable events. The type field indicates whether the event can be counted per-TC (T), per-VPE (V), or per-processor (P). TC countable events can also be counted in VPE or Processor modes, and VPE countable events can also be counted in Processor mode. The operation of a counter is **UNPREDICTABLE** for events which are specified as Reserved. All events not specified in the table are reserved.

Table 2 Performance Counter Control Register Field Descriptions

Name	Bits	Description	Read/ Write	Reset State
M	31	If this bit is one, another pair of Performance Control and Counter registers is implemented at a `MTC0` or `MFC0` select field value of ‘n+2’ and ‘n+3’. (There may be more than four performance counters available. There may also be no counters available—software should check the PC bit of the *Config1* register before attempting to use the performance counters.)	R	1 for counter 0-2 0 for counter 3
0	30	Must be written as zero; returns zero when read.	0	0
TCID	29:22	Specifies which TC events should be counted if per-TC counting is enabled	R/W	Undefined
MT_EN	21:20	Specifies which events should be counted: 00: Count events from all TCs and VPEs 01: Count events from all TCs of the VPE specified in VPEID 10: Count events from the TC specified in TCID 11: Reserved	R/W	Undefined
VPEID	19:16	Specifies which VPE events should be counted if per-VPE counting is enabled	R/W	Undefined
0	15:12	Must be written as zeros; returns zero when read.	0	0
EVENT	11:5	Counter event enabled for this counter. Possible events are listed in Table 3.	R/W	Undefined
IE	4	Counter Interrupt Enable. This bit masks bit 31 of the associated count register from the interrupt exception request output.	R/W	0
U	3	Count in User Mode. When this bit is set, the specified event is counted in User Mode.	R/W	Undefined

Table 2 Performance Counter Control Register Field Descriptions (Continued)

Name	Bits	Description	Read/ Write	Reset State
S	2	Count in Supervisor Mode. When this bit is set, the specified event is counted in Supervisor Mode.	R/W	Undefined
K	1	Count in Kernel Mode. When this bit is set, the specified event is counted in Kernel Mode.	R/W	Undefined
EXL	0	Count when in exception handling mode. When this bit is set, count the event when EXL=1 and ERL=0	R/W	Undefined

Table 3 Performance Counter Event Code Listing

Event	Counter 0/2	Type	Counter 1/3	Type
0	Cycles	P	Cycles	P
1	Instructions completed	T	Instructions completed	T
2	Branch instructions	T	Branch mispredictions	T
3	`jr $31` (return) instructions	T	`jr $31` mispredictions	T
4	`jr $xx` (not `$31`) instructions	T	`jr $31` not predicted (stack mismatch)	T
5	ITLB accesses	T	ITLB misses	T
6	DTLB accesses	T	DTLB misses	T
7	JTLB instruction accesses	T	JTLB instruction misses	T
8	JTLB data accesses	T	JTLB data misses	T
9	Instruction cache accesses	T	Instruction cache misses	T
10	Data cache accesses	T	Data cache writebacks	T
11	Data cache misses	T	Data cache misses	T
12	External intervention requests	P	External intervention requests	P
13	External intervention hits dirty	P	External intervention hit clean	P
14	Integer instructions completed	T	FPU instructions completed	T
15	Loads completed	T	Stores completed	T
16	`j`/`jal` completed	T	MIPS16 instructions completed	T
17	No-ops completed	T	Integer multiply/divide completed	T
18	All stalls (no action in RF pipeline stage)	P	Replay traps (other than uTLB)	T
19	`sc` instructions completed	T	`sc` instructions failed (because link bit cleared)	T
20	Prefetch instructions completed	T	Prefetch instructions completed with cache hit	T
21	L2 cache writebacks	P	L2 cache accesses	P
22	L2 cache misses	P	L2 cache misses	P

Table 3 Performance Counter Event Code Listing (Continued)

Event	Counter 0/2	Type	Counter 1/3	Type
23	Exceptions taken	T	Reserved	
24	Cache fixup	T	Refetches	T
25	IFU stalls	P	ALU stalls	P
26	DSP instructions completed	T	ALU-DSP saturations done	T
27	Reserved		MDU-DSP saturations done	T
28-31	*Available to count implementation-specific events signalled by wires from configurable interfaces.*			
28	Implementation-specific PM event	T	Implementation-specific CP2 event	T
29	Implementation-specific ISPRAM event	T	Implementation-specific DSPRAM event	T
30	Implementation-specific CorExtend event	T	Implementation-specific system event	T
31	Implementation-specific XYM event	T	Implementation-specific ITC event	T
32	ITC loads	T	ITC stores	T
33	Uncached loads	T	Uncached stores	T
34	`fork` instructions completed	T	`yield` instruction completed	T
35	CP2 arithmetic instructions completed	T	CP2 to/from instructions completed	T
36	Reserved			
37-46	*Count number of cycles (most often “stall cycles”, i.e. time lost), not just number of events.*			
37	Instruction cache miss stall cycles	T	Data cache miss stall cycles	T
38	L2 cache instruction miss stall cycles	P	L2 cache data miss stall cycles	P
39	Data cache miss cycles	P	L2 miss cycles	P
40	Uncached stall cycles	T	ITC stall cycles	T
41	MDU stall cycles	T	FPU stall cycles	T
42	CP2 stall cycles	T	CorExtend stall cycles	T
43	ISPRAM stall cycles	T	DSPRAM stall cycles	T
44	`cache` instruction stall cycles	P	Reserved	
45	Load to Use stalls	T	ALU to AGEN stalls	T
46	Other interlock stalls	T	Branch mispredict stalls	Pi
47	Relax bubbles	V	Reserved	
48	IFU FB full refetches	T	FB entry allocated	P
49	Reserved			
50-55	*Monitor the state of various FIFO queues in the load/store unit: FSB (“fill/store buffer”), LDQ (“load queue”) and WBB (“write-back buffer”). Some count events, others count stall cycles. None can be filtered per-TC or per-VPE.*			
50	FSB < 1/4 full	P	FSB 1/4-1/2 full	P
51	FSB > 1/2 full	P	FSB full pipeline stalls	P
52	LDQ < 1/4 full	P	LDQ 1/4-1/2 full	P
53	LDQ > 1/2 full	P	LDQ full pipeline stalls	P
54	WBB < 1/4 full	P	WBB 1/4-1/2 full	P

Table 3 Performance Counter Event Code Listing (Continued)

Event	Counter 0/2	Type	Counter 1/3	Type
55	WBB > 1/2 full	P	WBB full pipeline stalls	P
56-127	Reserved			

Most of the events in Table 3 are self-explanatory although some events specific to multi-threading support require greater explanation, such as:

- Indirect jump mispredictions (events 3 and 4). Only one Return Prediction Stack (RPS) resource is available in the 34Kc core, and is dynamically mapped to one TC at any point in time. Thus, the count of RPS mispredictions can be inaccurate or misleading.

- Pipeline events (events 18 and 24). Long stall conditions can cause a thread's instructions in the pipeline to be nullified and rolled back to the thread's instruction fetch unit (IFU) instruction buffer, a condition known as a replay. In some cases, an instruction may no longer be in the IFU buffer when a replay condition is detected, requiring the instruction to be re-fetched. And, it is possible for the load/store unit (LSU) to be in replay (i.e., re-fetching data) and for subsequent ALU operations to be stalled.

- Relax stalls (event 46). The number of cycles that the low-power thread is executed as requested by the policy manager.

Many event types count complementary events in the even/odd pairs of counters. For example, event 37 counts instruction caches misses in counters 0 and 2, and data cache misses in counters 1 and 3. Some event types count events that are also counted by other event types, e.g., event 25 counts IFU stalls in counters 0 and 2 and ALU stalls in counters 1 and 3. The logical OR of both conditions can be counted in counters 0 and 2 under event 18, Stalls. That is, in some cases a single event may not be sufficient to uniquely identify the cause of all stalls and the software must be executed multiple times to obtain a clearer picture of cause. (Which is a feature of `pc_sweep`, described in Section 4.3 "Measurement Tools".)

3.2 Events Useful to Software Developers

Of the events listed in Table 3, the two events of most immediate relevance to a software developer are *Cycles* (event 0) and *Instructions completed* (event 1). Measuring these two events allow the computation of an application's instructions-per-cycle (IPC) metric, which is an important figure of merit for optimization or development. A high IPC indicates that the machine is already executing the application efficiently, and gains, while possible, may come at too high a cost to be justified. A low IPC indicates that performance can be improved, possibly through the use of multi-threading or other optimizations.

If a low IPC is measured, indicating inefficiency, the developer can count other events to discover where the application is inefficient. Among the approximately 100 events that can be measured the most two useful groups are:

- Events 9, 10, 11, 37, and 39, which measure cache-related inefficiencies such as misses and cycles spent waiting for data.

- Events 18, 24, 25, 41, 45, and 46, which measure stalls, such as stalls waiting for instructions or within the ALU.

Many of the remaining events in Table 3 are special-purpose events with a very narrow focus that are unlikely to be used in day-to-day software development.

In the following sections we show how to access performance counters individually or collectively from application software or from the operating system. We then introduce a tool that assists the developer to specify events to count, and to make and report measurements.

4 Using the Performance Counters

4.1 Application Software Access

Kernel-mode software executing on the 34Kc core may use `MTC0` and `MFC0` instructions to move values to and from the performance counter and performance counter control registers. The performance counters are accessible at Coprocessor 0 register 25, with individual performance counter and performance counter control registers accessed using the `MTC0` and `MFC0` instructions `sel` field (as shown in Table 1). For example, to use performance counter 0 to measure instructions completed during some part of application execution, the following C code might be used:

```
int32_t count=0, control=0x28;
asm volatile("mtc0 %0, $25, 0" : "=r" (control)); // configure counter 0
asm volatile("mtc0 $0, $25, 1");                  // reset counter 0

... // Perform task

asm volatile("mfc0 %0, $25, 1" : "=r" (count));   // read counter 0
```

Note that the code example above is susceptible to execution and instruction hazards. Hazard barriers should be used to clear and remove hazards when reading and writing the Coprocessor 0 registers. For example, the following assembly code might be used to configure and read the performance counters:

```
    addiu  $t1, $0, 0x28              ; load control word into $t1
    move   $t2, $0                    ; zero counter
    la     AT, 1f                     ;
    mtc0   $t1, $25, 0                ; configure performance counter 0
    ehb                               ; clear execution hazard
    jr.hb  AT                         ; clear instruction hazards
    nop
1:
    la     AT, 2f
    mtc0   $0, $25, 1                 ; reset performance counter 0


    ehb                               ; clear execution hazards
    jr.hb  AT
    nop
2:

...                                   ; Perform task

    la    AT, 3f
    mfc0  $t2, $25, 1                 ; read counter 0
    jr.hb AT                          ; clear instruction hazards
    nop
3:
    nop
```

User-mode software may not access Coprocessor 0 directly (doing so will cause an exception), but may read the Coprocessor 0 *Count* register to obtain a high-resolution cycle count using the `RDHWR` instruction. The *Count* register is updated once every one or more CPU cycles; the update frequency can also be read using the `RDHWR` instruction. For example,

```
int32_t countStart=0, countEnd=0, countRes;
asm volatile("rdhwr %0, $2" : "=r" (countStart));

... // Perform task

asm volatile("rdhwr %0, $2" : "=r" (countEnd));
asm volatile("rdhwr %0, $3" : "=r" (countRes));
```

Accessing the cycle counter from within the application allows for very precise measurement with as little error as possible due to counter start/stop latencies. In addition, if the software application under test can temporarily prevent context switches at the application level, variance due to a multitasking OS environment can also be minimized or eliminated.

4.2 Linux Operating System Access

The performance counter registers and the performance counter control registers are available to the user through the Linux `/proc` filesystem interface at `/proc/perf`. The interface may be read at any time; an example is shown in Figure 3. The interpretation of the control registers is implementation-specific. Note that when reading the `/proc/perf` interface, the values of the control registers are presented in hexadecimal format, while the values of the counters are presented in decimal format.

Figure 3 Output From /proc/perf Performance Counter Interface

```
% cat /proc/perf
PerfCnt[0].Ctl : 0x80000528
PerfCnt[0].Cnt : 108399
PerfCnt[1].Ctl : 0x800005a8
PerfCnt[1].Cnt : 1171512
PerfCnt[2].Ctl : 0x800005a8
PerfCnt[2].Cnt : 285070
PerfCnt[3].Ctl : 0x000005c8
PerfCnt[3].Cnt : 779389
```

The root user may also configure the performance counters using this interface, by writing simple ASCII-format strings to `/proc/perf` in the format *num control value*, where

- *num* is the number of the performance counter and associated control register to set, in decimal format,
- *control* is the value to write to the associated control register, in hexadecimal format, and
- *value* is the value to initialize the performance counter with.

The example below shows the command used to configure performance counter 0 to count User Mode instruction completions while initializing the counter to an initial value of zero. The example assumes that the user is root; less-privileged users are not allowed to write the `/proc/perf` interface.

```
% echo "0 0x00000028 0" > /proc/perf
```

More than one counter can be updated or read at one time. For example, to set the properties for all four counters at once using values in the file `perfcnt_config`, the following commands can be used:

```
% more perfcnt_config
0 0x8000001f 0
1 0x8000003f 0
2 0x8000025f 0
3 0x800002ff 0
% cat perfcnt_config > /proc/perf
  ... run program to profile here ...
% cat /proc/perf > log_perfcnt_results
% cat log_perfcnt_results
```

It is also possible to suspend and resume event counting, as shown in the example below:

```
echo "3 0x1f 0"  > /proc/perf # clear counter 3, set to count all cycles, and
                              # enable overflow interrupt
echo "3 0x0 -1"  > /proc/perf # suspend counting
echo "3 0x1f -1" > /proc/perf # resume counting
```

An advantage of the `/proc/perf` interface is its simplicity and its availability to scripts and applications. A disadvantage is possible inaccuracy caused by (a) start/stop latency, and (b) context switches to other applications while measurements are made. In general, by repeating measurements under a variety of operating conditions, these sources of inaccuracy can be calibrated or otherwise made small relative to the measurements of interest. This is discussed further in Section 4.5 “Making Reliable and Repeatable Measurements”, below.

4.3 Measurement Tools

As stated earlier, the 34Kc core provides four performance counters used to make measurements of an application. However, in many cases, it is necessary to take more than four measurements in order to create an accurate picture of application behavior. A command-line tool for Linux, `pc_sweep`, simplifies the measurement of multiple events and is scriptable. The tool must be executed as the root user, to allow read/write access to the `/proc/perf` performance counter interface under Linux.

The basic syntax of the `pc_sweep` command is:

```
% pc_sweep <mode> [<options>] <command>
```

Where `<command>` is the application (or sequence of applications) for which performance measurements are to be made. The execution mode (User, Supervisor, Kernel, or Exception) under which measurements will be made must also be specified. Table 5 lists all command line options for `pc_sweep`, including the options to define the application execution mode. Both short- and long-format options are supported, and may be used interchangeably. Several short options may combined into one, provided that the result is not ambiguous. For example, the common options `-u -i -p` may be combined into one option, `-uip`.

Table 6 shows the mapping of command-line options to counted events. Options may be combined; the union of events is used for the actual measurement. For example, giving the command-line options `--misses` and `--tlb` results in an even counter event list of 5, 6, 7, 8, 9, 10, 11, 21, 22, and 39.

Figure 4 shows an example invocation of `pc_sweep` to measure the instructions per cycle of the command-line tool `grep`. User commands are shown in bold face. In this example, only two counter events are specified, and the target command is executed once. Note that the `--parse` option has been specified, which causes the raw counter data collected by `pc_sweep` to be “pretty printed” by a helper application supplied with `pc_sweep`, `pc_parse`. When the `-p` or `--parse` options are not specified, `pc_sweep` output looks similar to that shown in Figure 3; this format is harder to read and understand and use of parsing is recommended.

Table 4 Command Line Options for pc_sweep

Option	Event Type
Mandatory Arguments	
`-u, --user`	User-mode events
`--supervisor`	Supervisor-mode events
`--kernel`	Kernel-mode events
`--exl`	Exception-mode events
Optional Arguments	
`-p, --parse`	Post-process `/proc/perf` dump with `pc_parse`
`--output <filename>`	Save counter output to <filename>
`--append <filename>`	Append counter output to <filename>
`--all|-a, --ipc|-i, --stalls|-s, --queues|-q, --cache|-c, --branches|-b, --stalls_all, --instructions, --tlb, --l2, --itc`	Event selection
`--evens "[events numbers]"`	Even-counter events to measure
`--odds "[events numbers]"`	Odd-counter events to measure
`--balance`	Sweep TCs and VPEs
`--modes`	Sweep user/supervisor/kernel/exception cycles

Table 5 Mapping of pc_sweep Command Line Options to Counted Events

Option	Events Assigned to Counters	
	Event	Odd
`--ipc`	0	1
`--stalls`	18,25,41,45,24	18,25,41,45,46
`--all_stalls`	18,24,25,37,38,40,41,42,43,44,45,46,47,48	18,24,25,37,38,40,41,42,43,45,46,51,53,55
`--queues`	50,51,52,53,54,55	50,51,52,53,54,55
`--misses`	5,6,7,8,9,10,11,21,22,39	5,6,7,8,9,10,11,21,22,39
`--instructions`	1,2,3,4,14,15,16,17,19,20,26,35,32,34	2,3,4,14,15,16,17,19,20,26,27,35,32,34
`--cache`	9,10,37,39	9,10,11,37
`--branch`	2,3,4,16	2,3,4,16
`--tlb`	5,6,7,8	5,6,7,8
`--l2`	21,22,38	21,38,39
`--ipc`	0	1

Figure 4 shows a minimal run of pc_sweep to gather the instructions per cycle (IPC) ratio of the grep command executed in User mode. The *mode* field displays the mode (user mode, kernel mode, etc.) for which the counter was programmed to count events. The *VPE* and *TC* fields show which thread contexts were counted when running on the 34Kc core. The *event* field shows whether the counter displayed is an even-numbered counter or an odd-numbered counter, and which event number was specified. The pc_parse tool converts the event number into a human-readable description. It also computes percentage ratios of certain statistics, most notably stalls, and pseudo-statistics, such as instructions per cycle (IPC, a measure of efficiency) and cycle sharing overhead (a measure of the impact of the OS on the application). The IPC is calculated as the ratio of all instructions to the sum of all instructions and all stalls. The cycle-sharing overhead is computed as the ratio of the sum of all instructions and all stalls to the cycle count.

Figure 4 Example pc_sweep

Table 4.1

```
# pc_sweep --parse -ui grep root /etc/passwd
Even-counter events: 0
 Odd-counter events: 1
root::0:0:root:/root:/bin/bash
operator:*:11:0:operator:/root:/sbin/nologin

[mode] [VPE  TC ] (event )              event description:      value
---------------------------------------------------------------------
[U   ] [        ] (0 /  0)                          cycles:    1241355
[U   ] [        ] (1 /  1)               all instructions:     695424 m 100.0%

[U   ] [        ]                                      IPC:      0.560
```

When more than four events are to be collected, pc_sweep will invoke the target command repeatedly until all of the data is collected. Then, the counter results are summarized and displayed. This is shown (again, for the grep command) in Figure 5.

4.4 Avoiding Measurement Pitfalls

The purpose of the performance counters is to provide the developer with fine-grained, quantitative information on factors that affect execution time performance, either to confirm performance or to allow the performance to be improved (if possible). Access to a large number of countable source events—there are over 100 separate countable events—allows the developer many different avenues of investigation. However, care must be taken when making measurements with the performance counters for three reasons.

First, many events track conditions that are similar or overlap other events. For example, event 25 tracks either IFU stalls or ALU stalls, and event 18 tracks pipeline stalls, which includes both IFU and ALU stalls. A developer must take care to understand the sources of the events that are being counted, and create operational conditions that allow a single source to be identified.

Figure 5 Using pc_sweep to Measure Cache Activity While Executing grep Command in User Mode

```
# pc_sweep --parse -cui grep root /etc/passwd
Even-counter events: 0 9 10 37 39
 Odd-counter events: 1 9 10 11 37
root::0:0:root:/root:/bin/bash
operator:*:11:0:operator:/root:/sbin/nologin
root::0:0:root:/root:/bin/bash
operator:*:11:0:operator:/root:/sbin/nologin
root::0:0:root:/root:/bin/bash
operator:*:11:0:operator:/root:/sbin/nologin

[mode] [VPE  TC ] (event )              event description:      value
----------------------------------------------------------------------
[U   ] [        ] (0 /  0)                          cycles:    1235557
[U   ] [        ] (1 /  1)               all instructions:     699255 m 100.0%
[U   ] [        ] (0 /  9)               I-cache accesses:     497656
[U   ] [        ] (1 /  9)               I-cache misses  :       8227
[U   ] [        ] (0 / 10)             D-cache accesses  :     242055
[U   ] [        ] (1 / 10)             D-cache writebacks:       3591
[U   ] [        ] (1 / 11)             D-cache misses    :      18236
[U   ] [        ] (0 / 37)             L1 I-$ miss stalls:      87527
[U   ] [        ] (1 / 37)             L1 D-$ miss stalls:     108114
[U   ] [        ] (0 / 39)               D-$ miss pending:     140076

[U   ] [        ]                                    IPC:      0.566
[U   ] [        ]                          I-$ miss rate:       1.7%
[U   ] [        ]                          D-$ miss rate:       7.5%
```

Second, there are only four performance counters. To gain an accurate understanding of the cause of performance issues, the developer must execute the application repeatedly while measuring different sets of events. Typically, this means repeatedly executing the application with the same input data set and initial conditions. However, the developer may have little control over the behavior of other applications in the system, such as the operating system. For example, a regularly scheduled, but infrequent, housecleaning task may execute during the middle of a series of measurement runs, distorting the results. This can be a significant source of error when the software under test executes in relatively few cycles, e.g., as might be the case when testing a single kernel written in hand-optimized assembly. A useful strategy to minimize the error due to external events is to simply re-run the tests multiple times and discard outlier data sets.

Third, each performance counter is only 32 bits wide. It is possible for a counter to overflow and wrap around during a single execution run. For example, if the 34Kc core executed instructions at a clock rate of 500 MHz, an application experiencing an average instruction cache miss rate of 1 per 8 instructions would cause the counter tracking this event to wrap around after approximately 68 seconds, which may be too brief for the application(s) and event(s) under test. To offset this possibility, the performance counters may interrupt the core when any or all of the counters is over “half-full,” allowing the counters to be extended and the probability of actual overflow reduced or eliminated. (A 64-bit wide counter tracking the same event under the same conditions would overflow after 9,300 years of continuous operation!) However, extending the dynamic range of the counters may not be possible if the developer is not able to enable or use the interrupt.

In the following sections, we will discuss strategies and tools for dealing with each of the issues summarized above.

4.5 Making Reliable and Repeatable Measurements

The `pc_sweep` tool executes the target application repeatedly until all requested measurements have been made. To make the results useful the target application must exhibit deterministic behavior, i.e., each run must be identical to the previous run. Many applications will have deterministic behavior—modulo the behavior of the operating system, discussed next—when processing prepared files, e.g., MPEG-2 decoders operating on an encoded bitstream stored as a file. Applications processing data collected from other, “live” sources (such as from a sensor feed or when processing user input) will generally not exhibit deterministic behavior. In the latter case, multiple runs, possibly of long durations, can be used to determine performance statistically.

The behavior of the operating system can also skew results: the OS may execute a disruptive task irregularly while measurements are being made and the OS may cache files used by the application (and the application itself). In general, the impact of the OS can be measured and then factored out from the target application’s results by making repeated test runs and discarding outliers—i.e., data that is clearly far from the average. If desired, OS file system caches can be flushed with respect to the application under test either by using explicit OS commands or by running another application.

Currently, not all SMTC Linux kernels implement performance counter range extensions in the `/proc/perf` interface, making it possible for a counter to overflow and wrap around during a test run. This is most likely to occur when counting events that are triggered with highest frequency, such as event 0 (cycles). Note that when overflow occurs, statistics derived from the overflowed counter are inaccurate. Check with MIPS Sales or Technical support staff for the availability of the performance counter range extension in SMTC Linux.

5 Examples of Use

In the following sections we present three examples of `pc_sweep` use: a mergesort application and two examples using an MPEG-2 decoder.

The results presented here were obtained on a MIPS Malta™ FPGA emulation platform containing an emulated MIPS 34Kc processor executing at 32 MHz, configured with 2 VPEs and 5 TCs enabled. The emulation platform used the ROC-it high-bandwidth system/memory controller configured for a 4:1 ratio of CPU:Bus cycles, resulting in a cache miss penalty of approximately 50 cycles. The MIPS 34Kc executed the SMTC Linux operating system, which is MIPS 34Kc-aware and provides access to the on-core performance counters using the Linux `/proc/perf` filesystem interface.

Figure 6 YAMON Description of the MIPS 34Kc FPGA Emulation Board

```
Compilation time =                Sep  1 2006  16:34:30
Board type/revision =             0x02 (Malta) / 0x00
Core board type/revision =        0x09 (CoreFPGA-3) / 0x01
System controller/revision =      MIPS ROC-it / 0.0   FW-1:1 (CLK_unknown)
FPGA revision =                   0x0001
MAC address =                     00.d0.a0.00.04.cd
Board S/N =                       0000000991
PCI bus frequency =               33.33 MHz
Processor Company ID/options =    0x01 (MIPS Technologies, Inc.) / 0x00
Processor ID/revision =           0x95 (MIPS 34Kc) / 0x44
Endianness =                      Little
CPU/Bus frequency =               32 MHz / 7968 kHz
Flash memory size =               4 MByte
SDRAM size =                      64 MByte
First free SDRAM address =        0x800b6fb0
```

5.1 Reducing Data Cache Misses in mergesort

Our first example shows how to use `pc_sweep` to count more than four events by executing a target application multiple times in the same way. Our target application is the mergesort, a simple sorting algorithm with a relatively small program size (76 kilobytes as compiled for the 24Kc), that may operate on quite large data sets that do not fit completely within the data cache. The multi-threading implementation used here takes four command-line arguments: an input random data file, the number of elements to sort, a blocksize, and number of threads. We suspect that data cache misses will lower performance and that the choice of blocksize will affect data cache misses. We start with a hypothesis that increasing the blocksize parameter can reduce data cache misses by allowing the sort algorithm to spend more cycles operating on data already loaded in the cache rather than waiting for new data to be loaded into the cache.

We will use `pc_sweep` to determine the best blocksize when using four threads to sort 1 MB of random input data, executing the mergesort application multiple times to collect performance data related to the instruction and data caches. When collecting cache performance data, `pc_sweep` will count 10 events (five even and five odd), requiring multiple runs of the target application.

We invoke `pc_sweep` in a loop from the shell, requesting User mode counting (`-u`), with IPC computation (`-i`), returning important cache statistics (`-c`), and parsing the output through `pc_parse` (`-p`):

```
for ((i=64; i<2048; i+=i)); do \
    echo "Blocksize $i"; \
    pc_sweep -uicp ./mergesort ../random 1048576 $i 4; \
done;
```

Table 6 shows selected output from from `pc_sweep` profiling the mergesort application, iterating over six values of blocksize. (Relative execution time is computed after data collection, and is the ratio of the cycle count for blocksizes above 64 to the cycle count with blocksize equal to 64.) Data cache stall cycles and relative execution time results from the table are plotted in Table 7. We can see that as expected, data cache miss cycles trend down as blocksize increases. We can also see that the number of instructions executed increases with blocksize while ALU and total stall cycles are roughly constant with blocksize. The increase in instructions issued is expected due to the way the mergesort operates: by recursively partitioning of the block and sorting each half of the new blocks created. Hence, larger blocks result in more instructions being executed as more blocks are sorted.

Table 6 pc_sweep Output from mergesort

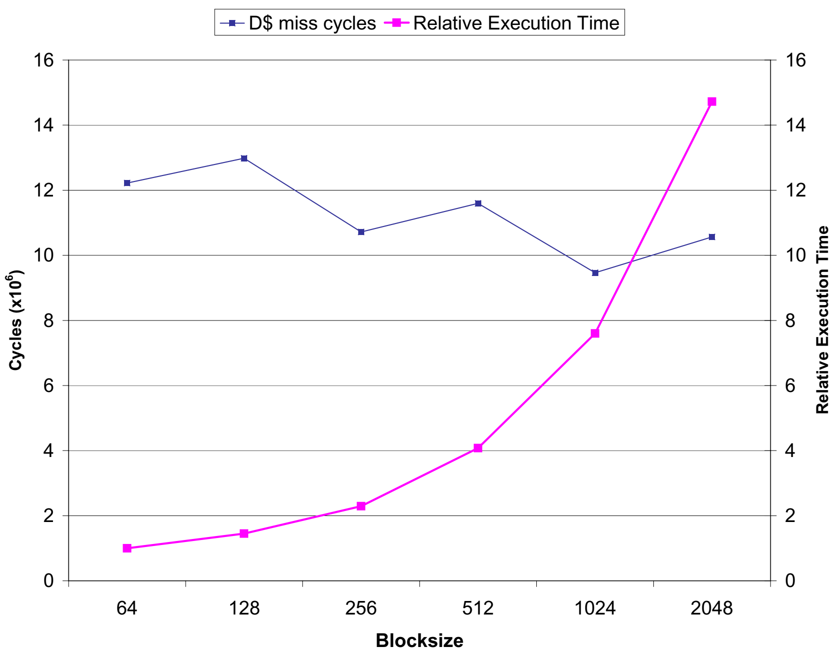
Metric	Blocksize					
	64	128	256	512	1024	2048
Relative execution time	1.00	1.45	2.29	4.08	7.60	14.72
Cycles	522,337,380	754,283,302	1,198,406,121	2,137,115,472	3,967,414,208	7,689,066,234
All instructions	446,849,798	667,638,291	1,110,112,990	2,012,639,686	3,812,539,153	7,431,331,191
All stalls	51,943,984	61,686,395	50,990,956	64,044,425	63,407,402	91,081,096
ALU stalls	44,430,415	52,139,381	43,594,682	56,612,052	55,545,441	79,673,813
D-cache accesses	73,681,290	98,705,929	144,750,486	245,363,821	442,501,474	845,222,541
D-cache writebacks	1,944,965	1,929,496	1,673,532	1,686,965	1,453,707	1,520,521
D-cache misses	12,220,059	12,980,219	10,716,190	11,597,488	9,464,412	10,564,993
L1 D-$ miss stalls	25,759,417	32,208,075	23,478,699	30,071,077	24,398,339	32,766,275
D-$ miss pending	169,377,268	175,177,187	157,146,579	171,146,721	194,420,395	256,755,966

5.2 Measuring a Specific Thread in an MPEG-2 Decoder

The ALPBench MPEG-2 decoder uses the POSIX Pthreads library, creating worker threads for each frame and combining the results prior to outputting each decoded frame. The number of threads to use is specified at run-time, with an upper limit defined at compile time. The ALPBench software is available from `http://rsim.cs.uiuc.edu/alp/alpbench`.

Figure 7 shows using `pc_sweep` to profile the mergesort application to determine the best blocksize to use to sort 1 MB of random data. Relative execution time is execution time relative to the time for a blocksize of 64 bytes.

Figure 7 mergesort Profile



The MPEG-2 decoder is invoked with the following command-line syntax:

```
% mpeg2dec-mips-nptl -b <m2v file> -n <number of threads>
```

The application is profiled while decoding the standard “Foreman” video test stream containing 300 CIF resolution (352 × 288 pixels) YUV-format video frames. The MPEG-2 decoder application takes command-line arguments for the location of the input encoded video file (`-b <m2v file>`) and for the number of threads to create per frame (`-n <number of threads>`). To prevent `pc_sweep` from parsing these arguments as its own, insert the end-of-switches switch, “`--`”, prior to the MPEG-2 decoder command, as shown below:

```
% pc_sweep -uicp -- ./execs/mpeg2dec-mips-nptl -b ../video/foreman10.m -n 4
```

The `pc_sweep` application allows measurements can be made from any single TC or VPE. It is possible to specify a specific TC for use by the application using the SMTC Linux utility `taskset`, and to specify that `pc_sweep` only collect events from the specific TC using the `-t` command-line switch. An example is shown in Figure 8; the output from `pc_sweep` shows that only data for TC3 was collected. A similar approach can be used to obtain data for a single VPE through the use of the `-v` switch.

Data can be collected for a specific TC using the Linux utility taskset in conjunction with the pc_sweep command, as shown in Figure 8.

Figure 8 Using taskset and pc_sweep

```
# pc_sweep -uicp -t3 -- taskset -c 3 \
     ./execs/mpeg2dec-mips-nptl -b ../video/foreman10.m -n 1
Even-counter events: 0 9 10 37 39
 Odd-counter events: 1 9 10 11 37
ropebench: kernel: '10_mpeg2dec'
ropebench: 1 threads
height 288 width 352
ropebench: realtime 5.470546
ropebench: kernel: '10_mpeg2dec'
ropebench: 1 threads
height 288 width 352
ropebench: realtime 5.463928
ropebench: kernel: '10_mpeg2dec'
ropebench: 1 threads
height 288 width 352
ropebench: realtime 5.462602

[mode] [VPE  TC ] (event )              event description:      value
---------------------------------------------------------------------
[U   ] [     TC3] (0 /  0)                          cycles:  182220114
[U   ] [     TC3] (1 /  1)              all instructions:  150634834 m 100.0%
[U   ] [     TC3] (0 /  9)              I-cache accesses:   85094070
[U   ] [     TC3] (1 /  9)              I-cache misses  :      43897
[U   ] [     TC3] (0 / 10)            D-cache accesses  :   43144409
[U   ] [     TC3] (1 / 10)            D-cache writebacks:     106040
[U   ] [     TC3] (1 / 11)            D-cache misses    :     427095
[U   ] [     TC3] (0 / 37)            L1 I-$ miss stalls:     434017
[U   ] [     TC3] (1 / 37)            L1 D-$ miss stalls:    1323273
[U   ] [     TC3] (0 / 39)              D-$ miss pending:    3364544

[U   ] [     TC3]                                    IPC:      0.827
[U   ] [     TC3]                         I-$ miss rate:       0.1%
[U   ] [     TC3]                         D-$ miss rate:       1.0%
```

Care must be taken when interpreting single-TC measurements made using `pc_sweep` when the target application uses multiple threads. The work being done by the other application threads will not be visible to the `pc_sweep` run, which may be confusing. If the application's behavior is completely deterministic, multiple `pc_sweep` runs examining different threads can be used and the developer can then combine the results to obtain a clearer picture of performance.

5.3 Tracking Stall Cycles in an MPEG-2 Decoder

Although `pc_sweep` provides command-line switches to target specific groups of events in a single run, it can be convenient to specify a customized list of events to measure. In this example, we will use the `--evens` and `--odds` options to `pc_sweep` to specify specific stall events of interest. These two options take a list of events to apply to the even and odd counters, respectively. We first choose events 18 (all stall cycles and replay traps), 37 (instruction cache miss cycles), and 39 (data cache miss cycles) from Table 3, then execute `pc_sweep` as follows:

```
% for i in 1 2 3 4 5; do \
    pc_sweep -uip --evens "18 37 39" --odds "18 37 39" -- \
       ./execs/mpeg2dec-mips-nptl -b ../video/foreman10.m2v -n $i \;
  done;
```

Selected results from the measurement are shown in Table 7. Relative speedup (not computed by `pc_sweep`, but computed from the cycle counts returned by `pc_sweep`) is defined as

$$S_{\Delta} = \frac{\text{single-threaded cycle count}}{\text{multi-threaded cycle count}}$$

Table 7 Selected Results of Customized Stall-cycle Measurement

Metric	Threads				
	1	2	3	4	5
Cycles	183,624,622	168,450,653	167,250,453	169,070,742	172,441,901
All instructions	151,020,049	151,086,135	151,120,799	151,162,631	151,206,089
IPC	0.822	0.897	0.904	0.894	0.877
All stalls	26,770,910	11,939,387	10,932,458	12,451,991	15,394,697
Replays	13,949	108,549	132,180	142,431	202,719
L1 I-$ miss stalls	2,642,765	1,528,107	1,686,401	2,003,892	2,133,641
L1 D-$ miss stalls	3,961,518	710,077	720,560	845,133	817,302
Cycle sharing overhead (%)	3.20%	3.20%	3.10%	3.20%	3.40%
Relative speedup	1.00	1.09	1.10	1.09	1.06

From the table we see that three threads of execution offers the best mix of relative speedup and fewest L1 instruction cache miss cycles. For example, total stall cycles are more than halved when three threads are used compared to the single-threaded case. Additional worker threads offer diminished returns: with four and five threads of execution, L1 instruction cache miss cycles continue to grow while the execution time remains relatively constant.

The table includes the instructions per cycle ratio (IPC) and cycle sharing overhead, as computed by `pc_sweep`. The IPC is maximized and the cycle sharing overhead is minimized when three threads are used. As described in Section 4.3 "Measurement Tools", IPC is calculated as the ratio of all instructions to the sum of all instructions and all stalls. An IPC of 1.0 indicates that the machine is performing work every cycle; values lower than one indicate less-than-ideal performance. The cycle sharing overhead is the ratio of the sum of all instructions and all stalls to the cycle count and is a measure of the overhead of other tasks (principally the OS) on the measured data.

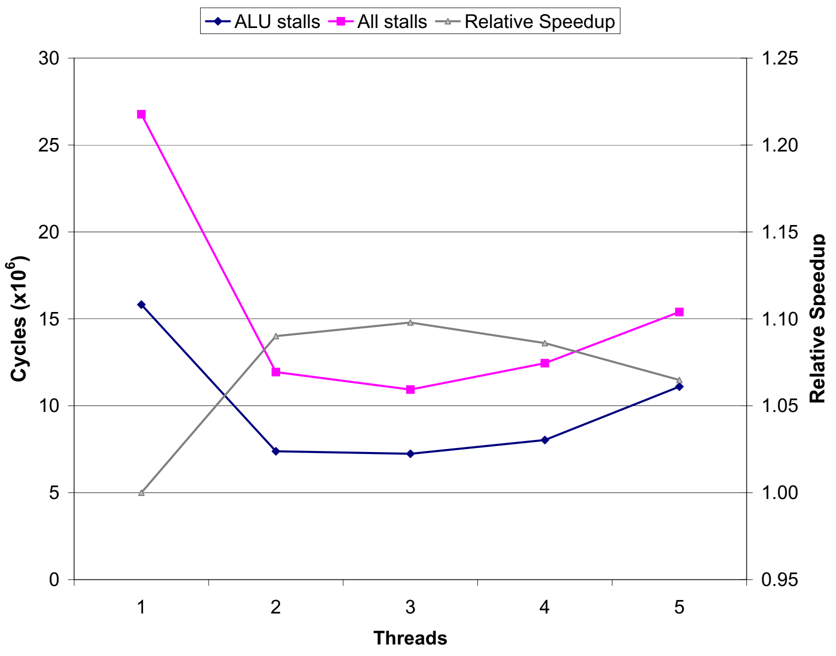
We also note that, independent of the number of threads, the total stall cycle count is generally four to five times higher than the sum of replay stall cycles and L1 cache miss cycles. What is the source of the extra stall cycles? To find out, we add more events to the list of events to count and run `pc_sweep` once more. We add events 24 (cache fixup and re-fetches) and 25 (IFU stalls and ALU stalls) to both the `--evens` and `--odds` lists. The results are shown in Table 8 below and plotted in Figure 9, and indicate that ALU cycles are the primary source of stall cycles in this application.

.

Table 8 Stall Cycle Measurement Results

Metric	Threads				
	1	2	3	4	5
Cycles	183,624,622	168,450,653	167,250,453	169,070,742	172,441,901
All instructions	151,020,049	151,086,135	151,120,799	151,162,631	151,206,089
IPC	0.822	0.897	0.904	0.894	0.877
All stalls	26,770,910	11,939,387	10,932,458	12,451,991	15,394,697
LSU replay stalls	248,040	441,140	804,660	1,265,543	2,380,531
Refetches	2,937	8,210	6,861	8,972	15,458
No instructions available	11,057,833	4,310,231	3,534,592	3,819,503	4,321,247
Replays	13,949	108,549	132,180	142,431	202,719
ALU stalls	15,819,932	7,382,784	7,240,907	8,032,199	11,102,178
L1 I-$ miss stalls	2,642,765	1,528,107	1,686,401	2,003,892	2,133,641
L1 D-$ miss stalls	3,961,518	710,077	720,560	845,133	817,302
Cycle sharing over-head (%)	3.20%	3.20%	3.10%	3.20%	3.40%
Relative speedup	1.00	1.09	1.10	1.09	1.06

Figure 9 PC_sweep Profiles



Note that for three threads, the sum of the individual stall cycle counters (14,126,161 cycles) is not equal to the reported total of all stalls (10,932,458 cycles): if two stall events occur at the same time, they are counted as one. This can make finding the exact cause of stalls confusing. However, in this case, we have a clear indication that ALU stalls are the most significant source of stall cycles. MPEG-2 decoder applications typically spend most cycles in relatively few functions, implying that the majority of the ALU stalls are caused by software in a few, or possibly just one, of the kernel functions within the MPEG-2 decoder. Armed with the data from the performance counters, the developer can proceed to identify the source and cause of the ALU stalls, and seek to reduce them.

6 Conclusion

The MIPS32 34Kc core's performance counters can be a powerful tool to help developers understand the *reasons* for the performance of their application. Developers can access the 34Kc core performance counters in several ways. The counters are easily used with a simple, yet powerful command-line tool, `pc_sweep`, and also through the `/proc/perf` filesystem interface and from the Linux command line. Privileged software may read and write the performance counters from C or from assembly using Coprocessor 0 register read and write operations; unprivileged software may only access a cycle count register.

The performance counters must be used with care to ensure that measurements are accurate and repeatable, and that counter overflow and wrap around does not obscure useful data. Developers using the performance counters should review the 34Kc core performance counter documentation to understand which events can be counted, and which events share counters (e.g., as is the case with the total stall cycles counter and the IFU and ALU stall counters).

The 34Kc core enables hardware-based support for multi-threading in a way that is almost completely transparent to the software developer. However, optimizing the execution time and throughput performance of a multi-threading software application requires special insight into complex application behavior. The wealth of 34Kc core performance counters can provide the developer with quantitative data about application performance, and so help the developer to make informed decisions about how and where to best apply optimization efforts.

7 References

1. *MIPS32® 34Kc™ Processor Core Datasheet*, MIPS Document Number: MD00418

2. *MIPS® MT Principles of Operation*, MIPS Document Number: MD00452

3. *MIPS32® Architecture Reference Manual VolumeIV-f: The MIPS® MT Application-Specific Extension to the MIPS32® Architecture*, MIPS Document Number: MD00376