Increasing Application Throughput on the MIPS32® 34K® Core Family with Multithreading

Document Number: MD00535
Revision 2.02
August 22, 2011

Contents

1 Overview

The MIPS32® 34K® core family implements the MIPS® Multithreading ASE, which enables fine-grained hardware multithreading. Cores in this family are capable of issuing an instruction from one of up to nine hardware thread contexts every cycle. Thus, when one thread is blocked from running due to an event such as a cache miss, instructions from another thread context (TC) can be issued instead, recovering cycles that would otherwise be wasted. With the right combination of threads and the appropriate choice of scheduling policy, multithreading can significantly increase the efficiency of a single core. There are different ways to measure the increase in efficiency, but in this paper we will focus on increase in system throughput as the key metric.

System throughput, broadly defined, is the amount of useful work done per unit time. In some cases, a system must maintain a minimum level of throughput for correct behavior. For example, in an audio processing system we can define throughput as the number of frames processed per second. The correct behavior of this system requires a minimum throughput to meet real-time constraints; otherwise, audible errors will result as frames are dropped. A system capable of greater than real-time throughput has frequency headroom that can be used for other tasks.

For a system-on-chip (SoC) designer, the need to maintain a certain level of throughput often leads to a multi-core design where real-time tasks, such as audio processing, are partitioned from the OS and its associated services on a separate core. In other cases, usually for low-cost solutions, a single core is used, and significant effort is required to balance the various workloads. The 34K core family offers more options. By increasing the efficiency of a single core and giving greater control to thread scheduling policy, multithreading can in some cases allow a two-core solution to be reduced to a one-core solution, saving significant die area and power. For single-core solutions, increasing the throughput of real-time tasks can allow higher-complexity or even multiple streams to be processed. And regardless of the number of cores used, the flexible scheduling options of the 34K core family make it easier to balance the sharing of processor resources by multiple workloads.

In this paper, we will look at how multithreading can be used to improve throughput on a MIPS32® 34Kc core relative to a single-threaded MIPS32 24KEc core. Since the 34Kc core pipeline is based on the 24KEc core pipeline, this comparison provides a good indication of the benefits that come directly from multithreading. Focusing on audio processing and file I/O, two workloads common in set-top box (STB) and digital television (DTV) applications, we will show that multithreading can deliver a ***31% increase in audio throughput***. In addition, we will show that when audio processing is run concurrently with file I/O, multithreading can deliver a ***226% increase in audio throughput*** without significant impact on file I/O throughput.

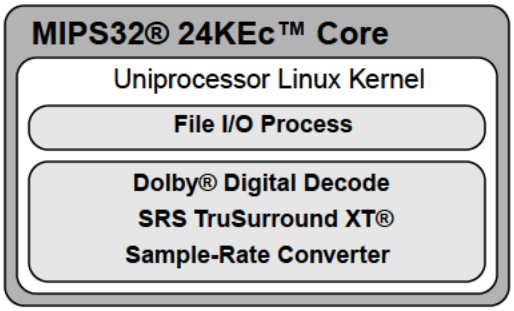
2 Single-Threaded Workloads

The first workload of interest is audio processing. We selected processing likely to be performed in a STB or DTV application: a cascade of a Dolby® Digital (AC-3) decoder, an SRS TruSurround XT® post-processor, and a sample-rate converter. A single block (six per frame) of audio passes through each codec sequentially, and we measure average throughput for the audio-processing cascade in frames per second (fps), for the processing of 100 frames. Intermediate data buffers are used, allowing for producer-consumer memory access locality. So, the same data written by a producer codec are read by a consumer codec, making it likely that the data are still in the cache.

The second workload is a file transfer task. We measure the average throughput, in megabytes per second (MB/s), for transferring a 100 MB file from one disk location to another. The file is transferred with a user-mode program that makes system read() and write() calls on 16 KB chunks. Both workloads run as processes under a uniprocessor Linux kernel on a 24KEc core, as illustrated in Figure 1, below. Note that for this experiment the core is implemented on a

Xilinx® Virtex™-4 FPGA running at ***33 MHz***. Therefore all measured throughputs are lower than they would be in a real implementation running at a higher speed.

Figure 1 Single-Threaded Workloads



3 Using Multithreading to Increase Throughput

In this section we will discuss how the single-threaded workloads can be parallelized and threaded to take advantage of the 34Kc core. First, we will focus on increasing the throughput of the audio workload alone, and then we will look at how multithreading improves the throughput of audio and file I/O workloads running simultaneously.

3.1 Increasing Audio Throughput

In order to increase the throughput of the audio workload, we need to decompose it into parallel threads of execution. Unlike multiprocessing, where threads execute in parallel on multiple processors, on the 34Kc core all threads share a single processor and its execution resources. Therefore, parallelism is a necessary but insufficient condition for throughput increase. The actual increase will also depend on the quantity and nature of stalls in individual threads. Most workloads are likely to be stalled at least some of the time, though, especially as the processor-memory performance gap increases at higher processor frequencies. Nevertheless, some threads overlap more efficiently than others. When considering parallel decomposition strategies, we offer several guidelines for choosing threads to run on the 34Kc core:

- ***Run enough threads concurrently.*** Increasing the number of concurrent threads increases the probability of having a thread available to schedule when another blocks. Keep in mind, though, that at some point adding more threads will likely result in diminishing returns.

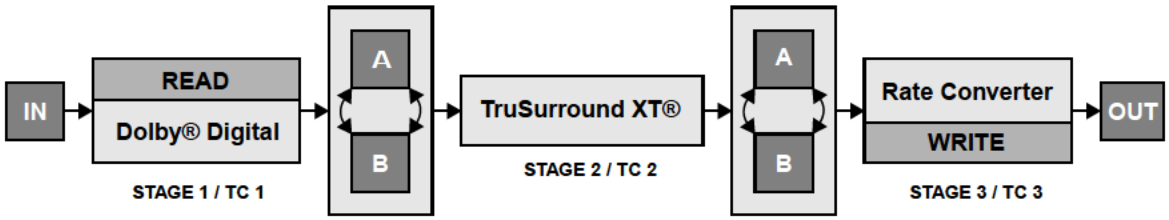
- ***At least some of the threads should have low IPC.*** Threads with low IPC frequently block, allowing other threads to be scheduled during pipeline stalls and bubbles. Many common tasks such as file I/O inherently have a low IPC.

- ***Overlap threads that stall for different reasons.*** Threads that efficiently share mutually-exclusive resources, like the multiply-divide unit (MDU) or memory, are less likely to interfere with each other.

There are many ways to decompose the audio-processing cascade into parallel threads of execution. Each individual codec could be decomposed to exploit either task or data parallelism. This would require a substantial programming effort and expert knowledge of codec structure. Alternatively, coarse-grain data parallelism could be exploited by processing multiple audio frames simultaneously. Because this method requires multiple instances of each codec, it would also require a substantial programming effort to make each codec reentrant. The simplest solution, which requires the least amount of programming effort, is to parallelize the audio-processing cascade as a coarse-grained

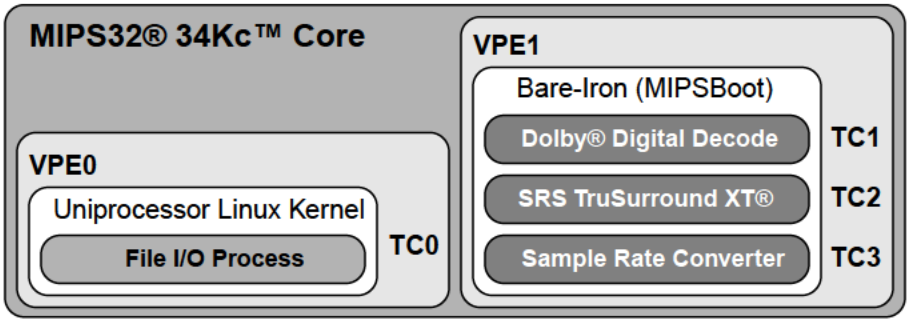
pipeline. Each pipeline stage is assigned to a different hardware thread context, and the threads are decoupled using double buffers, allowing for concurrent processing, as illustrated in Figure 2. Note that the double-buffering scheme used also maintains producer-consumer memory-access locality, helping to make efficient use of the data cache. Of course, double-buffering also increases the memory footprint of the audio workload, but this trade-off is required for the performance gain that results from parallelism.

Figure 2 Three-Stage, Task-Parallel Pipeline



In addition to breaking the audio cascade down into multiple threads, we can further increase its throughput by partitioning it from the OS on a separate virtual processing element (VPE). On the 34Kc core, there are two VPEs, and each is a full instantiation of the MIPS32 ISA and privileged resource architecture (PRA), thus appearing to software as an independent processor. In contrast, a thread context (TC) is just the architectural state required to support a single thread of execution, including a program counter and set of general-purpose registers. The first VPE runs the Linux kernel in a single thread context, and the second VPE runs the three audio-processing thread contexts in a bare-iron (no OS) run-time environment called MIPSBoot, although other environments could be used (such as ThreadX® RTOS). This configuration has two benefits. First, it frees the audio workload from the context-switch overhead of running within a Linux process. Second, it adds a fourth concurrent thread context, increasing the opportunity for overlapping stall cycles. This benefit will become especially apparent later, when we introduce the file I/O workload to the multithreaded system. The final system configuration is illustrated in Figure 3, below.

Figure 3 Multithreaded Workloads



As shown in Table 1, below, pipeline parallelization and VPE partitioning results in a 31.4% increase in audio throughput, from 6.43 fps to 8.45 fps on a 33 MHz core (as a point of comparison, on a 266 MHz 24Kc LV core, the single-threaded throughput is 54.2 fps). We can also see the increased utilization of the single-issue pipeline of each core, as IPC goes from 0.720 to 0.898. Another way to interpret these results is to look at the minimum processor clock speed required to maintain a real-time throughput of 31.25 fps. For a 24KEc core the minimum clock speed is 160 MHz, while for the 34Kc core it is 122 MHz. So, multithreading gives 38 MHz of additional frequency headroom. Note that the absolute throughput on both cores could further be reduced by taking advantage of the MIPS32 DSP ASE, an additional set of instructions intended to accelerate signal-processing code.

Table 1 Audio Throughput Increase[1]

Core	Speed	Configuration	Throughput	IPC
24KEc FPGA	33 MHz	Single-Threaded Audio in Linux Process	6.43 fps	0.720
34Kc FPGA	33 MHz	Multi-Threaded Audio on VPE1, Linux on VPE0	8.45 fps	0.898

1. Test Conditions: (a) input stream was 100 frames, 5.1 channels, 256 Kbps, 48 KHz; (b) TruSurround XT configured with TruSurround®, TruBass® and Dialog Clarity™ enabled and 5.1 channel input; (c) 24KEc core bitfile on CoreFPGA™ 3 at 33 MHz CPU with SOC-it® system controller, 16KB/16KB write-back/write-allocate caches and 4:1 CPU to SOC-it clock ratio; (d) 34Kc core bitfile options same as 24KEc core with addition of inter-thread communication (ITC) memory; (e) Linux kernel had 100Hz tick rate, 32-entry TLB and 120 MB memory size.

3.2 Increasing Throughput in the Presence of a File I/O Workload

Having looked at the benefit multithreading can deliver to the throughput of a single workload, we now turn our attention to the interaction of two workloads, each competing for the resources of a single processor to complete their respective tasks. Our single-threaded and multi-threaded setups are the same. On the 24KEc, the audio processing and file I/O workloads each run in their own respective processes under a uniprocessor Linux kernel. On the 34Kc core the uniprocessor OS and file I/O workload run within a single thread context on one VPE, and the multi-threaded audio workload runs in a bare-iron environment on the other VPE.

As shown in the third row of Table 2, when both workloads are run on a 24KEc with default Linux priorities, the audio throughput is 1.87 fps with a corresponding file I/O throughput of 1.34 MB/s. This is a 71% decline in audio throughput relative to the 6.43 fps we measured without file I/O (row 1). On the other hand, file I/O throughput is only 16% lower than the value measured without the audio workload of 1.59 MB/s (row 2). Again, keep in mind these numbers are for a 33 MHz core, and on a 266 MHz 24Kc LV file I/O throughput starts out at 10.5 MB/s. What happens if we use the Linux scheduler to try and return audio throughput to its previous level? The result can be seen in row 4. Even when the audio process receives maximum priority, its throughput is only 5.82 fps, and this increase comes at great cost to file I/O throughput, which drops to 0.20 MB/s. The inability to improve the throughput of one workload without starving the other would make it very difficult to run both on the same core while still meeting real-time constraints.

Table 2 Audio and File I/O Throughput[1]

Core	Speed	Scheduling Policy	Workloads	Audio Throughout	File I/O Throughput	IPC
24KEc FPGA	33 MHz	Linux, equal-priority	Audio	6.43 fps	--	0.720
24KEc FPGA	33 MHz	Linux, equal-priority	File I/O	--	1.59 MB/s	0.253
24KEc FPGA	33 MHz	Linux, equal-priority	Both	1.87 fps	1.34 MB/s	0.388
24KEc FPGA	33 MHz	Linux, highest-priority audio	Both	5.82 fps	0.20 MB/s	0.655
34Kc FPGA	33 MHz	Hardware, equal-priority round-robin	Both	6.70 fps	0.95 MB/s	0.847
34Kc FPGA	33 MHz	Hardware, raised-I/O-priority round-robin	Both	6.11 fps	1.26 MB/s	0.804

1. Test Conditions: all conditions identical to those of Table 1.

When the same workloads are run on the multithreaded 34Kc core, we see dramatic improvements in efficiency. With the default equal-priority round-robin scheduling of the 34Kc core's policy manager, audio throughput is 6.70 fps and file I/O throughput is 0.95 MB/s (row 5). This audio throughput is higher even than the single-threaded case without file I/O, and it represents a 258% increase over the 1.87 fps observed on the 24KEc. Of course the 0.95 MB/s is less

than the single-threaded file I/O throughput, but this is largely due to the default round-robin scheduling. There are a total of four hardware thread contexts, each with equal priority, but these are disproportionately divided between the two workloads. So with round-robin, on average the audio workload gets more scheduling opportunities.

We can compensate for this by switching to a fixed-priority scheduling policy and raising the priority of the file I/O thread. The effect is that any time the file I/O thread is able to run, the processor runs it exclusively. The results are in row 6. File I/O throughput increases to 1.26 MB/s, within 6% of its single-threaded level, and audio throughput goes to 6.11 fps. This is still a 226% increase in audio throughput, relative to the single-threaded system, with only a 6% decrease in file I/O throughput. Another way of looking at the results is that at the same clock speed required to process one real-time audio stream on a 24KEc core, a 34Kc core could process three without impacting file I/O throughput.

4 Discussion

In part, the large disparity between single-threaded and multithreaded system throughput stems from the inability of Linux to interleave process execution at a fine enough grain to mask the stalls of the file I/O workload. The result is that processor time is always divided with some proportion between the two workloads, but there is never an increase in core efficiency. By moving context-switching between threads to the hardware level, the 34Kc core can switch out a blocked thread in only one cycle, a level impossible to achieve with software.

As we saw, hardware scheduling policy can be used to alter the balance of processor time allotted to running threads. In fact, the 34K core was designed to allow even more flexibility in scheduling policy than was used in this experiment. A more sophisticated policy manager called weighted round-robin can be used to assign threads to groups which, on average, get unequal fixed fractions of processor time. For even more control, the core even allows for closed-loop feedback control. A software layer on top of the policy manager could monitor the behavior of individual threads and dynamically adjust policy manager parameters to make quality-of-service (QoS) guarantees. This is particularly useful for real-time tasks, when throughput is absolutely critical for correct system behavior. For more information on 34K core scheduling policies, consult the *"Programming the MIPS32® 34K Core Family"* (MD00427) guide.

The reader should keep in mind that all of the absolute throughputs reported in this paper are for 33 MHz FPGA implementations of the cores. A higher speed processor will obviously increase absolute throughput. Less obviously, the speed will also have an impact on the relative speedup from multithreading. Due to the complexity of the interactions of the workloads, Linux, and the hard disk drive, though, it is impossible to extrapolate the expected behavior for a system of arbitrary speed. However, so long as the file I/O workload remains a low-IPC task, inefficiently utilizing the processor, and so long as the real-time audio workload continues to be impacted by heavy disk activity, the 34K core will offer significant advantages for improving overall throughput.

5 Summary

In this paper we have shown that the multithreading capabilities of the MIPS32 34K core family can be used to deliver significant increases in core efficiency and utilization. For an example STB or DTV audio-processing workload, this led to a 31% increase in throughput compared with a single-threaded 24KEc core. When this workload was run concurrently with a file I/O workload we showed a 226% increase in audio throughput with only a 6% decrease in file I/O throughput. We discussed how this increase in efficiency and throughput could be used to process higher-complexity or multiple real-time streams simultaneously, or how it could be used to reduce a two-core SoC design to

a one-core design. Additionally, we discussed how the flexible scheduling policies of the 34Kc core could be used to balance the throughputs of various workloads and make real-time guarantees.

6 Additional Reading

- *"Programming the MIPS32® 34K® Core Family"* (MD00427)
- *"MIPS® MT Principles of Operation"* (MD00452)
- *"Getting Started with the VPE-loader and AP/SP API"* (MD00453)
- *"MIPS32® Architecture for Programmers Volume IV-f: The MIPS® MT Application-Specific Extension to the MIPS32® Architecture" (MD00378)*