Multi-Threading Applications on the MIPS32® 34K® Core

This paper introduces the hardware multi-threading features of the MIPS32 34Kc processor core. The main focus of discussion in the paper is to present performance benefits from executing certain classes of micro-benchmarks as well as a real-world application on such multi-threaded processors. The paper also describes techniques to use the multi-threading support built into the core to reduce the overall execution time on legacy as well as new applications.

Document Number: MD00547
Revision 01.01
August 22, 2011

1 Introduction

Reducing a software application's execution time is a primary concern of software developers as a means of improving end-user experience and adding value. Reducing execution time allows for a range of possible benefits, including:

- Increased throughput, e.g., decoding more 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.

The MIPS32® 34K® and 1004K™ families of cores implement the MIPS® Multi-Threading (MT) ASE [4], which provides hardware support for *multi-threading* 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 core in fewer cycles than on single-threaded MIPS cores, such as those from the MIPS® 24K™ family of cores. In this application note we discuss the use of multi-threading on the MIPS32® 34K™ core [1] as a means of effectively reducing an application's execution time with less developer effort than traditional *single-threading* approaches.

We begin with a brief definition and comparison of single- and multi-threading operation in Section 2 "Single- and Multi-Threading", introducing concepts useful to the remaining discussion, results, and examples. Section 3 "MIPS32® 34Kc™ Core Multi-Threading Architecture" introduces the 34Kc core architecture, with a focus on the architectural features that support multi-threading operation. Section 4 "Multi-Threading Application Performance" presents execution time benchmark results for the MIPS32 34Kc core and MIPS32 24Kc core on several different types of application. Section 5 "Creating Multi-Threaded Applications" discusses how to use the multi-threading support built into the MIPS 34Kc core to reduce the execution time of both legacy and new applications.

2 Single- and Multi-Threading

Before examining the MIPS32 34K processor core's approach to multi-threading, it will be helpful to review a few important concepts and terms. Readers familiar with the concept of multi-threading may wish to skip directly to Section 3 "MIPS32® 34Kc™ Core Multi-Threading Architecture".

We will begin by defining the opposite of multi-threading: a *single-threaded application* is an application that executes a single stream of instructions. This is a natural fit to traditional processor architectures that physically support decoding of a single instruction stream. The execution of the instruction stream in a single-threaded application may essentially halt while the processor waits for some condition to be met, such as data to be loaded into cache or for a floating-point operation to complete. (The processor itself is not necessarily completely halted in this state; it might be interruptible, for example, depending on the processor architecture.) In this condition the application is said to be *blocked*: the processor is doing no useful work and the application's execution time is increasing.

The efficiency of multiple single-threaded applications may be improved using an approach known as *multitasking.* Multitasking is usually implemented at the operating system (OS) level by a *task scheduler* that switches execution periodically among the currently executing applications, giving the appearance of true parallelism and allowing useful work to be performed even when one or more of the applications is blocked. In *co-operative multitasking* the application and the OS must work together to enable task switching. In *pre-emptive multitasking*, the OS may choose when to switch tasks without requiring the programmer or program to be aware of the multitasking environment.

In either case, a set of information about each application—the *application state*—must be maintained to allow the execution to be started and stopped without error. The application state is typically maintained by the operating system, and the size of the state determines the cost of a *context switch*, which ultimately impacts the execution-time performance of the system. Typically, the state is large, meaning that the OS does not switch applications as often as it might if the cost to switch were lower. The result is that while the processor is doing useful work more of the time compared to the non-multitasking case, there is room for improvement.

Further improvement can be made through the use of multiple threads of execution within an application, where a thread is some stream of instructions within the application. Threads can share much of the application state, with the result that far less state is required per thread than for the complete application—e.g., instruction pointer, stack pointer, copies of some registers, but little more—significantly lowering the cost to switch between threads. The lower switching cost can reduce the amount of time the processor is stalled, in turn reducing application execution time relative to the case where threads are not used. Figure 1 depicts the conceptual advantage that can be realized if a processor supports multiple threads of execution through the reduction or elimination of the lengthy process swap.

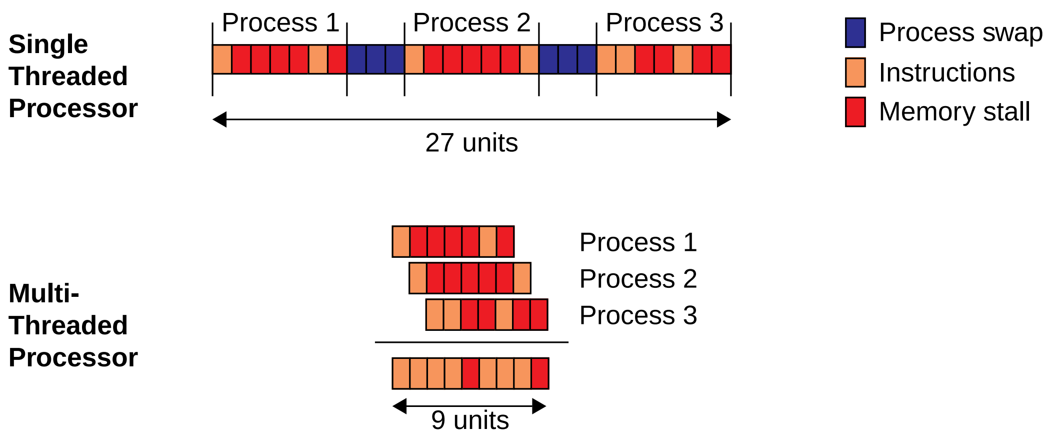

Figure 1 Using multiple threads of execution can improve application execution-time performance by allowing useful work to be done under conditions that would stall a single-threaded application.

Threads can also be used for automatic dataflow synchronization within an application, by allowing one thread to block on the output of another. For example, one thread might read data from an external device, pre-process it, and write it to a buffer managed by a second worker thread for computation. If the external device is busy or slow, the first thread may block (no data to write), causing the second to also block (no data to read), with both resuming when there is data available.

Software threads, like tasks, are also typically managed by the operating system. Software applications executing on POSIX®-compliant operating systems can make use of a special library, the POSIX threads library (Pthreads), to explicitly create and use software threads.

2.1 Reducing Application Execution Time

With the brief discussion of tasks and threads behind us, we now discuss approaches to reducing application execution time. We start by considering the approaches used to reduce the execution time of single-threaded applications, and then consider the impact of adding threads.

2.1.1 Single-Threaded Applications

Traditionally, four general techniques are used to reduce execution time for single-threaded applications. First, the algorithm and implementation can be optimized to reduce the work that must be performed or to minimize run-time stalls or latencies. The application is typically profiled to learn the locations of the application's "bottlenecks" or "hotspots"—the functions or loops that are responsible for the bulk of execution time or application inefficiency—and what causes them. Once identified, there are many approaches to hotspot optimization, including re-implementing algorithms by hand, changing the fundamental data types used for improved performance, and carefully specifying the placement of data in memory. Such optimizations can yield impressive results, but not without cost: the optimization process is generally labor-intensive, and can lead to the creation of special case code that doesn't perform well over the desired range of inputs and operating conditions.

Second, the speed of the processor or memory (or both) can be increased. This may not decrease the absolute number of cycles required to execute the application, but by executing instructions at a faster rate the end result is the same: a lower execution time. This approach can be effective, particularly if a faster processor variant can execute the software without modification (as is true of MIPS family processors). However, it may not be a practical approach due to a variety of logistical and physical factors. For example, there may not be a faster processor family member available given a specific project or product budget. Faster generally means greater energy consumption and this may result in an increase in the cost of the system processor due to factors such as managing thermal dissipation.

Third, fixed-function hardware may be added to the system. This is possible in modern system-on-chip (SoC) designs, particularly those designs that are organized around a standard on-chip bus interface and protocol. Fixed-function hardware is typically both more energy efficient that a programmable solution and has a lower execution time. Disadvantages include availability and cost (i.e., silicon area and licensing), and if the fixed-function IP is not readily available, creating the IP may be impractical or too time consuming.

Finally, compromise may be used: features discarded or output quality sacrificed. Naturally, compromise can be the least attractive of all the options listed here.

2.1.2 Multi-Threaded Applications

As defined earlier, a multi-threaded application utilizes multiple threads of execution. The use of threads allows the application as a whole to continue doing useful work whenever a single thread becomes blocked, provided that there is another thread able to do useful work and that the thread switch can occur. Adding multi-threading support to application software can be relatively straight-forward, e.g., through the use of well-defined calls to the Pthreads library, and require less effort than some of the options discussed above.

Before discussing how threads are created and used, we will explore how threads can be applied to the application software. As an example, consider a web server application, which is the back end that provides data to the familiar web browser application. The web server application has much to do; although not an exhaustive list, we can imagine that at any given time the web server must:

- Listen for new TCP/IP connections, and establish new connections when requested.
- Process requests for pages (HTML source, images, and so on) over the TCP/IP connections already established.
- Tear down (i.e., free the resources for) old TCP/IP connections that are no longer in use.

A web server is a natural candidate for a multi-threaded implementation: it is expected to serve hundreds if not hundreds of thousands of connections and transactions per second, and the web server performance is expected to scale linearly with the number of connections and transactions. From the short list of tasks above it is clear that making a thread to handle each task might be a reasonable approach. For example, a thread might listen for new TCP/IP connections, and launch a new worker thread to handle any page requests related to the connection. Another thread might iterate over the list of connections, looking for inactive connections to free.

The use of multi-threading works well for the web server because important web server resources—the network connection and disks to store the content—have relatively high latency compared to the computational requirements of the transactions being processed. Thus, web server threads can be expected to block frequently while waiting for data transfers to complete, presenting opportunities for other threads to be executed even with software thread scheduling (multitasking).

By utilizing threads, the application developer can improve the performance of the application by making use of the time the processor would have otherwise been stalled. This can be achieved with less effort than required for the optimization techniques described in Section 2.1.1 “Single-Threaded Applications”. In Section 5 “Creating Multi-Threaded Applications”, we will discuss the general approach to implementing a multi-threaded application using a POSIX-compliant thread library (Pthreads, as used in Linux operating systems) for thread creation and use.

3 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 4 “Multi-Threading Application Performance”.

3.1 Architecture

The 34Kc core implements the MIPS Multi-Threading ASE (MT ASE) [4], and extends the MIPS32 24Kc core 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 2 depicts the MIPS 34Kc core architecture.

A 34Kc core implementation may include 0, 8, 16, 32, or 64 KBytes of instruction and data caches. Implementations of the MIPS 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].

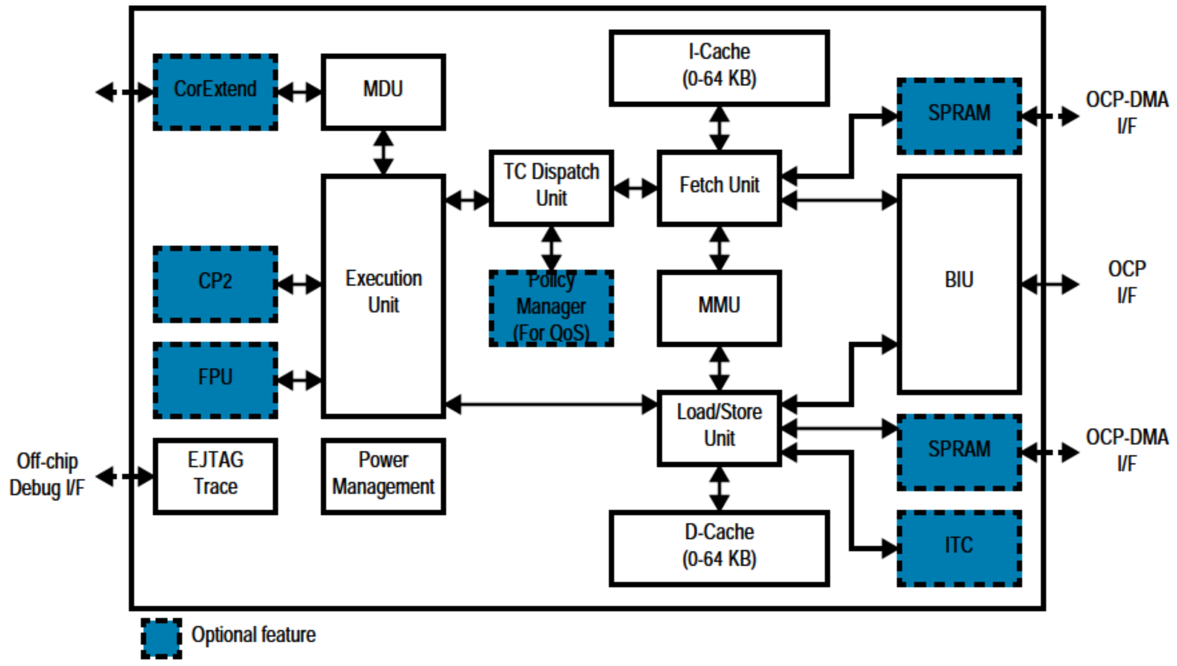

Figure 2 The MIPS 34Kc Core Architecture.

3.2 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.

In 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.

3.3 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, one thread, running OS code, needs to manipulate the contents of another, as may be the case when 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 multi-threaded 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 [4] 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 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.

3.4 New Instructions

The 34Kc core implements the eight instructions of the MIPS MT ASE [4]:

- `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.

4 Multi-Threading Application Performance

In Section 2 "Single- and Multi-Threading", we asserted that multi-threaded applications are expected to have lower execution times than single-threaded applications. In this section we present the results of benchmarking a small selection of multi-threaded applications for both the 34Kc core and 24Kc core. We first introduce the system used to perform the benchmarking, and then present benchmark data for four different explicitly multi-threaded applications: implementations of two different sorting algorithms, and an MPEG-2 encoder and decoder. Each of these applications is benchmarked on emulators for both the 34Kc core and 24Kc core. In addition, results are presented for a single-threaded DSP kernel executing within a multi-threaded test harness on the 34Kc core (the 24Kc core lacks the MIPS DSP ASE implementation on which the DSP kernel depends). A discussion of the results is presented in Section 4.5 "Analysis".

4.1 Benchmarking Configuration

The benchmark results presented here were obtained on a MIPS Malta™ FPGA emulation platform emulating either a 34Kc core executing at 32 MHz and configured with 2 VPEs and 5 TCs enabled, or a 24Kc core executing at 32 MHz. The emulation platform used the ROC-it high-bandwidth system/memory controller configured for a 4:1 ratio of CPU to external memory bus cycles, resulting in a cache miss penalty of approximately 50 cycles. The 34Kc core executed the SMTC Linux kernel in a TimeSys implementation of the Linux operating system. The SMTC kernel is 34Kc-aware and provides access to the on-core performance counters using the Linux `/proc/perf` file system interface. (Performance counters are discussed briefly in Section 5.2 "Performance Profiling" and in more detail in [3].)

The 24Kc core emulation executed a regular MIPS32R2 Linux kernel and also executed the same TimeSys Linux distribution. This distribution includes a POSIX Pthreads library, allowing software compiled for the 24Kc to be used for both 34Kc and 24Kc benchmarking.

```
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
```

Figure 3 YAMON Information Display for MIPS 34Kc Emulator.

4.2 C Functions and Small Applications

Multi-threading may be applied to small applications, single functions, or kernels. Figure 4 shows the result of applying multi-threading to two small applications for sorting data: the mergesort and the bitonic sort.

The input to the bitonic and mergesort benchmarks is a file of random binary data, which may be generated using the Linux `dd` command. For example, to generate a 1 megabyte file of random data called `random`:

```
% dd if=/dev/urandom of=random bs=1024 count=1024
```

The bitonic benchmark is executed using the following command:

```
% bitonic <random file> <size> <threads> [<print>]
```

The number of threads must be a power of two and the size argument must be a multiple of 2,048 and the number of threads, e.g., a size of 4,096 may be used with 2 threads. An optional non-zero print argument causes the sorted output data to be displayed.

The mergesort benchmark is executed using the command:

```
% mergesort <random file> <size> <blocksize> <threads> [<print>]
```

The blocksize and threads arguments must be powers of two, and the size argument must be a multiple of both blocksize and the number of threads, e.g.,

```
% mergesort random 2048 64 2
```

As with the bitonic search benchmark, an optional non-zero print argument causes the sorted output data to be displayed.

To obtain the relative speedup results shown here the cycle count for both benchmarks was measured using the `pc_sweep` profiling tool. A random data set of 256 kilobytes was used as the input to the sort implementations; the mergesort implementation used a blocksize of 64 bytes. Figure 4 shows the relative speedup for multi-threaded sort implementations on both the 34Kc core and 24Kc core. For example, the two-thread mergesort implementation for the 34Kc core is about 35% faster than the single-threaded implementation for the 34Kc core, and about 25% faster than the dual-threaded implementation for the 24Kc core.

Table 1 shows performance profiling data for the mergesort application. The ratio of stalls to instructions is about 90% for the single-threaded case, and drops rapidly as the number of threads is increased.

Metric	Threads		
	1	2	4
Cycles	130,146,350	96,219,537	90,243,830
All instructions	69,609,320	69,949,206	69,949,342
Instructions per cycle (IPC)	0.535	0.727	0.775
All stalls	52,861,043	21,497,272	15,906,789
I-cache accesses	53,071,697	49,722,762	48,584,846
I-cache misses	8,326	9,137	9,270
D-cache accesses	14,677,318	14,947,113	14,953,918
D-cache writebacks	403,763	430,738	432,459
D-cache misses	2,295,647	2,030,454	1,727,431
Replays	30,909	423,917	512,538
LSU replay stalls	25,878	94,363	2,988,110
No instructions available	11,500,820	5,868,088	2,524,588
ALU stalls	42,183,257	15,653,026	13,541,699
L1 I-$ miss stalls	494,945	488,698	501,778
L1 D-$ miss stalls	35,535,572	11,251,526	5,241,305
D-$ miss pending	39,479,636	37,271,257	39,130,953
MDU stalls	15,038	16,281	15,941
Load-to-use stalls	3,892,513	1,770,677	588,844
ALU to AGEN stalls	584,467	174,671	63,678
Branch misprediction stalls	52,608	1,126,636	668,250

Table 1 Performance Metrics for mergesort operating on 256 KB of random data, obtained using the pc_sweep tool.

4.3 MPEG-2 Encoding and Decoding

The ALPBench MPEG-2 encoder/decoder was used to benchmark the execution time of multi-threaded MPEG-2 encoding and decoding software. The ALPBench MPEG-2 software uses the POSIX Pthreads library, creating worker threads for each frame and combining their results prior to writing the bitstream (for the encoder) and displaying each frame (for the encoder). The number of threads to use is specified at run-time, with an upper limit of 16 threads defined at compile time. The ALPBench software is available from `http://rsim.cs.uiuc.edu/alp/alpbench`. The basic source code package was modified slightly to include performance timing (using `gettimeofday()` calls) in the threaded section of code. Note that the benchmark results shown here were obtained using the performance profiling tool `pc_sweep` described in [3].

The MPEG-2 encoder and decoder were built from source under the 24Kc core emulation with the following compiler optimization switches:

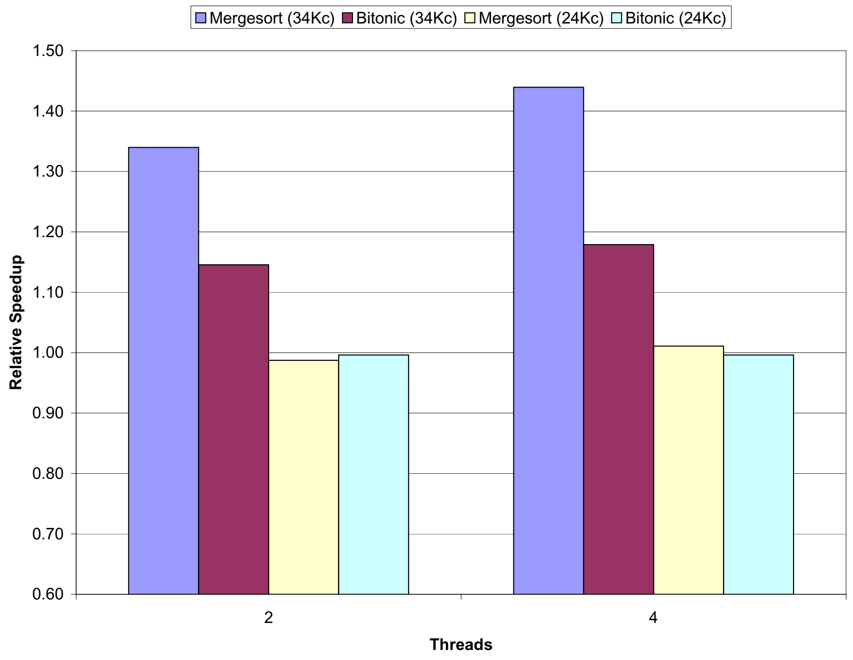

Figure 4 Relative speedup for multi-threaded implementations of mergesort and bitonic sort for MIPS 34Kc and 24Kc core. Higher is better.

```
-O3 -g -mips32r2 -mmad
```

(Although the MIPS 34Kc supports the MIPS DSP ASE, the MIPS 24Kc does not, and use of the compiler's `-mdsp` optimization switch improves the performance of the MPEG-2 decoder by less than 1% on the MIPS 34Kc.)

The MPEG-2 encoder is executed using the command:

```
% mpeg2enc-mips-nptl <par file> <m2v file> <number of threads>
```

Where `<par file>` is an ASCII file containing a list of encoder parameters, `<m2v file>` is the name of the encoded output file, and `<number of threads>` sets the number of threads to use when encoding.

The MPEG-2 decoder using the command:

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

Where `<m2v file>` is the encoded output file created using the encoder, and `<number of threads>` sets the number of threads to use when decoding. The execution-time performance of the video encoder and decoder was measured using the standard "Foreman" video test stream containing 300 CIF resolution (352 × 288 pixels) YUV-format video frames. The application cycle count was measured using the `pc_sweep` profiling tool and the data used to create the relative speedup data plotted in the figures below. Relative speedup is expressed as a percentage and is defined as

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

Thus, a relative speedup higher than 1.0 indicates that the multi-threaded implementation executed in fewer cycles than the single-threaded implementation, and conversely, a value lower than 1.0 indicates that the multi-threaded implementation executed in more cycles. If processor clock speed and instruction execution rates are equal, fewer cycles equates to lower execution time or, equivalently, higher throughput per unit time.

The relative speedup results for the MPEG-2 encoder versus the number of per-frame execution threads for both the 34Kc core and 24Kc core are shown in Figure 5. In this case the encoder is operating on a sequence of 30 frames taken from the longer 300 frame “Foreman” sequence (a standard test sequence for video codecs). Adding a second thread to the per-frame encoding produces a relative speedup of about 10% on the 34Kc core, and produces a slight slowdown for the 24Kc core. That is, the single-threaded 24Kc core gains no advantage from multi-threading in this application implementation.

The columns in Figure 6 shows the relative speedup versus threads of execution for single- and multi-threaded MPEG-2 decoder implementations for both the 34Kc core and 24Kc core. The line plots the relative speedup of the multi-threaded implementation for the 34Kc core compared to the single-threaded implementation for the 24Kc core. The cycle count of the single-threaded implementation for the 34Kc core is higher than that of the single-threaded implementation for the 24Kc core, but the 34Kc core cycle counts drop quickly as the number of decoder threads is increased. In contrast, the 24Kc core implementation becomes slower as the number of threads is increased.

An interesting question is: why does the relative speedup of the MPEG-2 decoder peak at three threads? Using the `pc_sweep` profiling tool, we can examine specific performance counters within the 34Kc processor to learn why. (The set of available 34Kc performance counters is described in [3] and in [4]). We examine the stall counters, specifically, the counters for all stalls (counter 18), IFU stalls (even counter 25), and ALU stalls (odd counter 25).

We use the profiling tool `pc_sweep` to examine the 34Kc core stall cycles in the MPEG-2 decoder for the three- and four-thread case, producing the results shown in Table 2. The data shows *why* the relative speedup drops when four threads are used instead of three—there is a 24% increase in the total number of stall cycles. We can see that misses from the data and instruction caches are a tiny fraction of the total stall cycles, while stalls in the ALU contribute about 70% of the total stall cycles. If the ALU stalls could be eliminated in the three-thread case (unlikely, but useful for the purpose of this discussion) the speedup relative to the single-threaded implementation would be a little over 1.20×, a significant gain in throughput.

Metrics	Threads		
	1	3	4
Cycles	553,453,176	501,513,160	508,720,118
All instructions	457,728,249	457,975,955	458,102,491
Instructions per cycle (IPC)	0.827	0.913	0.901
All stalls	77,674,195	28,347,789	35,247,363
Replays	29,719	366,498	488,771
LSU replay stalls	741,151	2,390,949	4,055,509
No instructions available	31,992,071	8,904,795	9,990,366
ALU stalls	46,545,739	19,822,328	25,370,881
MDU stalls	8,453,274	1,627,133	1,246,536
Load-to-use stalls	7,736,459	1,413,712	1,315,972
ALU to AGEN stalls	8,795,296	1,476,329	1,396,921
Branch misprediction stalls	471,300	608,079	510,979
I-cache accesses	259,021,090	249,131,089	250,028,234
I-cache misses	120,031	179,394	197,243
D-cache accesses	130,923,849	131,017,261	131,060,656
D-cache writebacks	155,201	189,727	313,566
D-cache misses	1,519,772	1,097,434	1,572,165
L1 I-$ miss stalls	6,715,894	3,863,166	5,038,711
L1 D-$ miss stalls	11,565,354	1,516,873	1,811,861
D-$ miss pending	23,595,826	36,310,831	45,571,011

Table 2 Stall cycle measurement results for the MPEG-2 decoder obtained with the pc_sweep performance counter tool.

4.4 Hand-Coded DSP Kernels

The execution time of hand-coded DSP kernels can also be reduced using POSIX threads to “wrap” execution of the kernel. Each thread is used to execute the target kernel some number of times in sequence. Here, we show relative speedup results from benchmarking a multi-threaded test an 8 × 8 IDCT used in MPEG decoding. The hand-optimized kernel is executed 100,000 times, divided by the number of threads used. The benchmark uses the Pthreads library to create the specified number of worker threads (between one and five), with each thread executing the hand-optimized implementation of the IDCT kernel thousands of times.

Note that this benchmark could only be executed on the 34Kc core as it has MIPS DSP ASE support: the MIPS DSP ASE is unsupported on the 24Kc core.

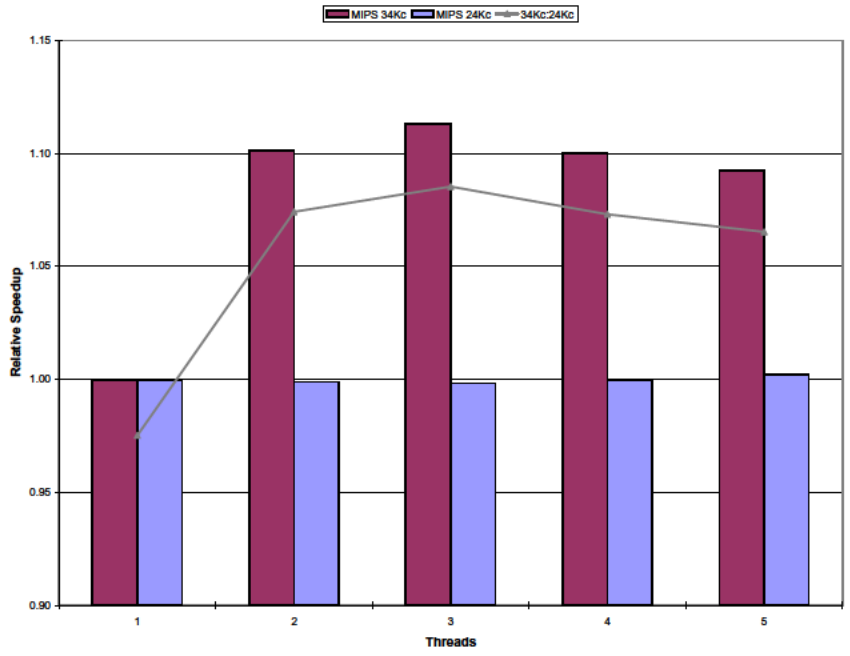

Figure 5 Relative speedup for MPEG-2 encoding of a 30 frame CIF-resolution test sequence vs. number of encoding threads. Higher is better.

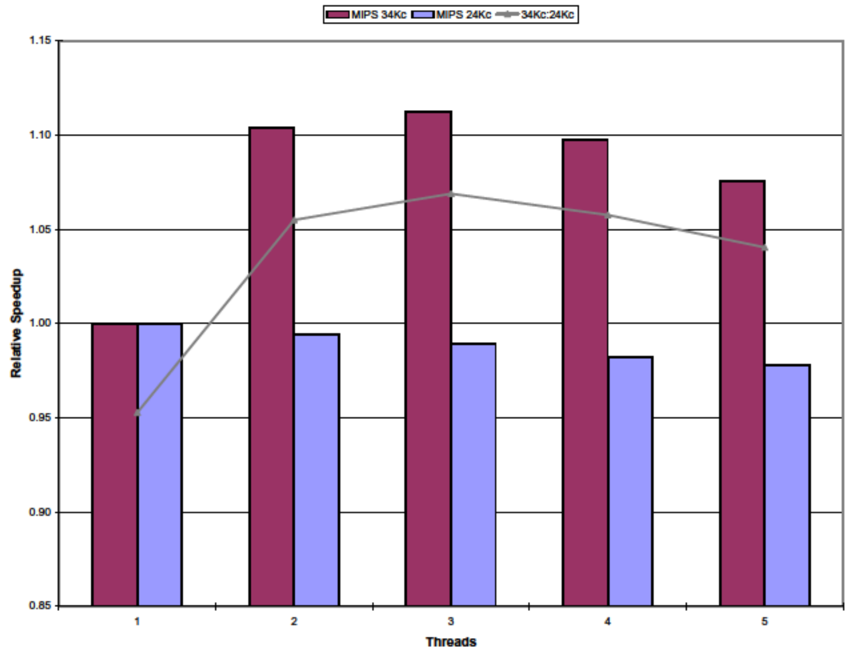

Figure 6 Relative speedup vs. threads of execution for MPEG-2 decoding of a 30 frame CIF-resolution test sequence. Higher is better.

Figure 7 shows the relationship between relative speedup and the number of execution threads for the 8×8 IDCT on the 34Kc core: using more threads increases the relative speedup, with the most significant gain made when moving from single- to dual-threaded operation. Table 3 summarizes selected performance counter measurements obtained from runs of the IDCT test harness. The data show that the primary source of stall cycles are ALU stalls, and that adding threads significantly reduces ALU stall cycles. The maximum relative speedup is approximately 1.20×, given by the ratio of cycles to instructions for the single-threaded case. The actual relative speedup obtained is about half this: as the stall cycles from the primary source diminish rapidly, stall cycles from multiple secondary sources gain greater significance.

Metric	**Threads**		
	1	**3**	**5**
Cycles	101,953,042	93,585,704	92,429,635
All instructions	85,584,564	85,608,770	85,626,162
Instructions per cycle (IPC)	0.839	0.915	0.926
All stalls	13,498,555	5,898,274	5,081,664
Replays	7,262	10,710	11,603
No instructions available	3,732,078	914,791	875,507
ALU stalls	9,567,956	5,143,263	4,050,332
L1 I-$ miss stalls	433,544	517,541	546,439
L1 D-$ miss stalls	273,788	300,338	301,354
Load-to-use stalls	207,974	15,688	12,839
ALU to AGEN stalls	147,252	50,233	48,757
Branch misprediction stalls	238,962	41,655	22,563

Table 3 Detail of stall cycles in the multi-threaded 8x8 IDCT kernel test bench.

4.5 Analysis

As shown in the figures presented in this application note, enabling the use of a small number of threads of execution can reduce the cycle count of the 34Kc applications presented here by up to 40% relative to the same application executed with a single thread, increasing throughput by the same amount. In contrast, adding more threads to the 24Kc implementations typically increased the cycle count, lowering throughput. As identical application code was executed on both emulations, the gain in throughput must be due to the specific hardware support for multi-threading in the 34Kc (differences in the two Linux kernels used are not expected to contribute significantly to the outcome).

Profiling using the `pc_sweep` tool shows that ALU stall cycles are a dominant factor in execution time for the applications benchmarked here, and that adding multiple threads of execution can reduce ALU stall cycles significantly, even in cases where careful optimization may have been performed to reduce stall cycles, such as in the IDCT kernel. Note that the underlying cause of the ALU stalls is not identified by the profiling tool and that further work would need to be done to isolate the source. Clearly, the availability of multi-threading does not change the general approach to optimization discussed in Section 2.1 “Reducing Application Execution Time”. However, it can provide meaningful reductions in execution time “for free.”

For the developer, the important questions are: how can I determine if multi-threading will accelerate my application, and if it will, how many threads are optimal for my application?

To answer the first question we consider the application’s instructions-per-cycle ratio (IPC). The data presented here indicates that single-threaded applications that have a low IPC benefit more from the application of multi-threading

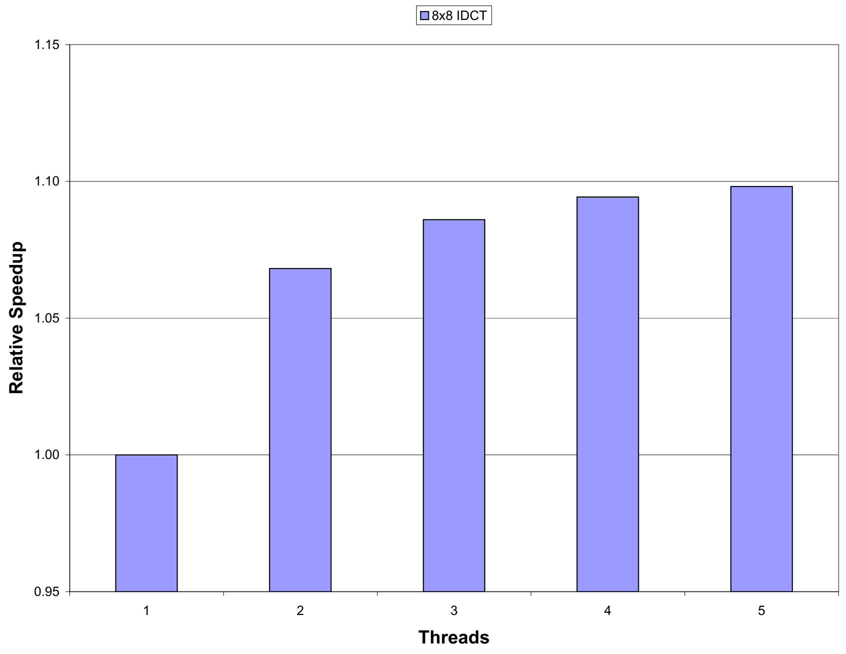

Figure 7 Relative speedup of a hand-optimized 8x8 IDCT kernel when multiple threads of execution are used. Higher is better.

than single-threaded applications that have a high IPC ratio. (Note that because the 34K core family has a single-issue pipeline, the maximum achievable IPC for any application is 1.0.) A low IPC indicates that the application spends a considerable proportion of cycles waiting, either for data or events to occur. Multi-threading helps replace these unproductive cycles with productive cycles, lowering the overall cycle count and increasing the IPC. For example, in the applications benchmarked here the MPEG-2 decoder had a single-thread IPC of 0.827 (Table 2) whereas the single-threaded mergesort had a single-thread IPC of 0.535 (Table 1). As can be seen from the data, enabling more threads of execution in the MPEG-2 decoder resulted in a modest reduction in cycle count, while in contrast the mergesort cycle count was reduced significantly.

To answer the second question, a definition of optimality is needed. One might choose the fewest threads that result in the largest single gain (to preserve TCs for other applications, which may also be multi-threaded), in which case the applications benchmarked here indicate “fewer than five software threads.” All of the applications tested showed the largest relative speedup when the application used two threads of execution. Applications with a high single-threaded IPC may be best served with two threads; applications with a low single-threaded IPC may benefit from up to four threads.

5 Creating Multi-Threaded Applications

The 34Kc core supports multi-threading in hardware and software: a 34Kc-aware operating system can allocate hardware threads to single-threaded applications, allowing a reduction in the overall execution time of a collection of single-threaded applications without further effort on the part of the application developer. However, a developer can also add explicit multi-threading to new and legacy applications through the use of software libraries, as discussed here.

5.1 Migrating POSIX® Pthreads-based Applications Written in C

The POSIX Pthreads C library is used to create and manage a POSIX-compliant application's threads of execution. MIPS has created MIPS32® 34K™-family-aware versions of the Linux kernel that utilizes MIPS MT ASE instructions to create and destroy threads directly on the hardware TCs. From the developer's perspective, migrating multi-threading applications written in C that use the POSIX Pthreads library to the 34Kc core can be as simple as executing the application. However, for best results the developer should also review the number and use of threads in the application, and the application performance profile (discussed in the next section). For example, an application originating from a significantly different architecture and run-time environment (e.g., a server-class application being ported to a deeply embedded 34Kc-core-based product) may have been written to use many more threads than is efficient for the 34Kc core. As seen from the results presented in Section 4 "Multi-Threading Application Performance", low numbers of application threads typically provide the best relative speedup.

Using the POSIX Pthreads library on the 34Kc core is straightforward: Figure 8 shows a fragment of code from the `getpic.c` file in the MPEG-2 decoder used to obtain the benchmark results shown earlier. The worker thread's function is `Thrd_Work`, and the data required by the thread is held in the user-defined structure `thread_data_array`. This code fragment is executed (once per thread) at the start of per-frame decoding to create the new worker threads that do the actual work of decoding a frame slice. The per-thread decoded slices are combined into a single output frame, with each thread terminating when its work is done. When using the MIPS 34Kc-core-aware SMTC Linux OS, the `pthread_create` call will result in the allocation of a single hardware TC to the thread. When the number of threads is higher than the number of TCs supported in the specific MIPS 34Kc core implementation, software scheduling will be used to multiplex the software threads onto the available hardware TCs. In either case the developer does not have to significantly modify the application source code or design, an important saving of effort.

```
/*Thread Create */
t = thrd_num - 1;
thread_data_array[t].id = t;
thread_data_array[t].num_slices = num_decode_slice[t];
thread_data_array[t].framenum = framenum;
thread_data_array[t].MBAmax = MBAmax;
thrd_ptr[t] = tb[t].frame_buf;
Thrd_Initialize_Buffer(t);

#if (NUM_THREADS>1)
if (t!=num_thrds-1) {
   rc = pthread_create(&thread[t], NULL, Thrd_Work,
                        (void*) &thread_data_array[t]);
#endif
if (rc) {
   printf("ERROR; return code from pthread_create() is %d\n", rc);
   exit(-1);
}
```

Figure 8 The POSIX Pthreads library simplifies the creation and use of threads, as shown in this example from the MPEG-2 decoder used for thread benchmarking.

5.2 Performance Profiling

Simply adding software threads is not a guarantee that an application will meet the required or desired execution time targets. As a general rule, the cost *to add* a single software thread on the 34Kc core is low, but the execution-time cost *of adding* another software thread can be high, depending on a range of factors.

As a simple example, consider a multi-threaded application executing on a 34Kc core implementation with a 32 KB data cache and where each thread operates on 8 KB data blocks from disjoint regions of memory. The performance gain (the ratio of execution time for a single-threaded application implementation vs. a multi-threaded implementation) is likely to be lower with 5 execution threads than with 4 due to the impact of data cache thrashing on thread execution (see, for example, the MPEG-2 encoder results presented in Figure 5). By making detailed profiling measurements, the developer can identify this condition, and take action—either limiting the number of threads, or changing the size of each thread's data set.

The underlying factors affecting execution time in a multi-threaded application can be subtle and hard to diagnose without help. The developer can use the wealth of profile counters provided by the 34Kc core and the `pc_sweep` profiling tools discussed in [3] to measure application performance directly and guide any modifications necessary to meet overall application performance goals.

6 Conclusion

Reducing an application's execution time can bring many benefits in terms of increased throughput and reduced energy consumption. The hardware-based multi-threading capabilities of the MIPS32 34Kc core provide the application developer with a nearly transparent means to reduce application execution time with relatively little effort.

MIPS Technologies has enabled an implementation of the Linux kernel and POSIX Pthreads library that combine to support the multi-threading 34Kc core's architecture natively and relatively transparently for the developer. For example, applications compiled for the MIPS32 24Kc and written to use the POSIX Pthreads library can achieve cycle count reductions of between 10% and 40% simply by being executed on the 34Kc core. Further reductions in an application's cycle count can be made using the 34Kc performance counters and a performance profiling tool, `pc_sweep`, to obtain the detailed information needed to fine-tune application performance.

7 References

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

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

3. *Using the MIPS32® 34K™ Core Performance Counters*, MIPS Document Number: MD00548

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

7 References