

# A Multithreaded Multicore System for Embedded Media Processing

Jan Hoogerbrugge

Andrei Terechko

NXP Semiconductors, Eindhoven, The Netherlands  
{jan.hoogerbrugge, andrei.terechko}@nxp.com

**Abstract.** We describe a multicore system targeting media processing applications where the cores are multithreaded. The multithreaded cores use a new type of multithreading that we call Subset Static Interleaved (SSI) multithreading. SSI multithreading combines the advantages of blocked multithreading and a simple form of interleaved multithreading called static interleaved multithreading. SSI multithreading divides threads into foreground and background threads and performs static interleaving among the foreground threads. A foreground thread is swapped with a runnable background thread whenever the foreground thread is stalled. SSI multithreading achieves reduced operation latencies, memory latency tolerance, fast context switching, and compared to traditional dynamic interleaving, a relatively low design complexity of the register file.

We use a task scheduling unit (TSU) to dispatch tasks to the cores. The TSU is aware of the fact that the cores are multithreaded. This makes a more efficient mapping of tasks to cores possible by scheduling tasks on the least loaded cores.

We evaluate the system on an optimized Super HD H.264 decoder where the macroblock decoding and deblocking has been parallelized. The complexity of the H.264 standard and the high resolution makes this a challenging and performance demanding application. We achieve speedups of up to 17.7 times for 16 cores with four threads per core relative to a single-threaded single core. Furthermore, the proposed SSI multithreading achieves a speedup of 1.52 times relative to no multithreading, while blocked multithreading achieves only 1.38 times and a restricted form of interleaved multithreading achieves only 1.37 times speedup.

## 1 Introduction

Until recently multiprocessor systems were used for supercomputing and server applications, but with the introduction of single chip multiprocessors, called multicore, they are entering the personal computing market as well. Soon we will also see many examples of embedded computing where multicores are used to execute a single application. Note that although many embedded systems on chips contain several processor cores, an application (e.g. video decoding or channel decoding) typically runs only on one core.

This paper describes a cache coherent multicore system targeting media processing in embedded applications. We evaluate the system with a parallelized H.264 video decoder on Super HD input streams (3840x2160 at 30 frames per second). The complexity of the H.264 standard and the high resolution makes this a challenging and performance demanding application. One of the novelties of the system is the multithreading of the cores. We apply a type of multithreading that we call Subset Static Interleaved (SSI) multithreading that tolerates memory latency, improves instruction level parallelism, and enables a cost-effective implementation of the register file. Another contribution of this paper is the task scheduling unit which knows about the utilization of each core. This makes it possible to make better scheduling decisions by trying to balance the load over the cores equally.

Our experimental multicore system is composed of TriMedia TM3270 VLIW [1] cores modified to support cache coherence and multithreading. TM3270 is designed for media processing (e.g. video decoding, audio decoding, graphics, and content analysis) and H.264 is one of the key applications for it. One TM3270 is capable to decode SD resolution H.264 in real-time. By instantiating multiple TM3270 cores and modifying them for multithreading we obtain a system which is capable to decode SHD resolution at a comparable clock frequency (+/- 400MHz). Notice that the resolution of SHD is 24 times higher than SD resolution.

The remainder of the paper is organized as follows. Section 2 categorizes multithreading and introduces the subset static interleaved multithreading. Section 3 describes the task scheduling unit (TSU) that balances the processing load over the multithreaded cores. Section 4 discusses the parallelization of the H.264 decoder which we will use for benchmarking purposes. Section 5 evaluates the performance of SSI multithreading compared to existing multithreading techniques. Furthermore we evaluate the effectiveness of the load balancing technique of the task scheduling unit. Section 6 discusses related work. Finally, Section 7 concludes the paper.

## 2 Multithreading

In a multithreaded core the execution data path is shared by multiple threads of control so that when a thread is not able to utilize the data path because of a stall or data dependences, the data path can be utilized by other threads. Each thread owns its own execution context consisting of a program counter and a register file. As memory latencies become longer and the data path provides more parallelism, it becomes harder to keep the wide issue data path utilized with one instruction stream, and therefore multithreading becomes more attractive.

### 2.1 Classification

Multithreaded cores can be classified based on whether the hardware is able to detect independent threads or the programmer is responsible for identifying independent threads [2]. The first class is called implicit multithreaded architectures

which are still in an experimental stage. Explicit multithreaded architectures are much more mature and can be classified in three classes: blocked multithreading interleaved multithreading, and simultaneous multithreading [2].

- **Blocked multithreading**, also known as coarse-grain multithreading, executes instructions from a thread until it encounters a stall which takes more than a few cycles (see Figure 2a). A typical example is a cache miss where a processor stalls until data is fetched from memory. On such a stall the pipeline is flushed and restarted with instructions from another thread that is able to execute. This new thread is then executed until it also encounters a long stall.
- **Interleaved multithreading**, also known as fine-grain multithreading, dynamically selects every cycle another thread to enter the pipeline (see Figure 2b). Successive pipeline stages are therefore typically holding instructions from different threads. When a thread stalls on, for example, a cache miss, that thread is flushed from the pipeline and not selected anymore until the cache miss has been handled. A thread is flushed by canceling the instructions from it that come after the instruction that caused the stall.
- In the case of **simultaneous multithreading** (SMT), instructions from different threads might be present in the same pipeline stage simultaneously. This type of multithreading is typically applied for scalar ISAs in superscalar designs where multiple instructions streams are fed into an out-of-order execution pipeline [3]. However, it has also been attempted to apply this to VLIW ISAs where VLIW operations from different instructions are merged during execution [4, 5].

In the sequel of the paper we refer to software threads as *tasks* or *software threads*, whereas hardware contexts in a multi-threaded core are termed *threads* or *hardware threads*.

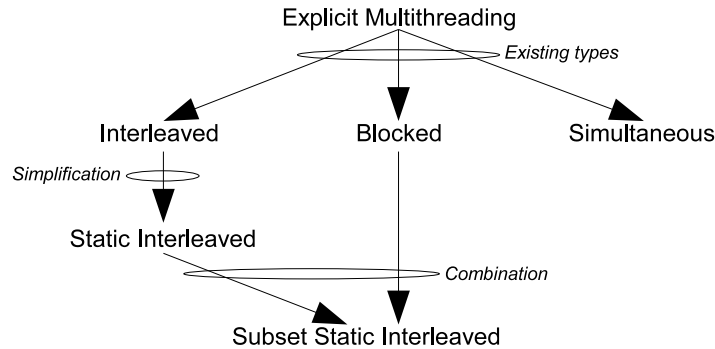
## 2.2 Selection

When selecting a type of multithreading for an embedded VLIW mediaprocessor, interleaved multithreading seems to be most suitable candidate. Simultaneous multithreading is not attractive because of its high complexity that is necessary to merge the VLIW instructions. It is also the question how effectively VLIW instructions can be merged when the merging algorithm has to be kept simple. For example, if it is not allowed to issue the operations of one VLIW instruction into multiple cycles.

When comparing interleaved and blocked multithreading, the most attractive technique is interleaved multithreading. Because instructions from different threads are executed in an interleaved fashion, the operation latencies become shorter in terms of executed instructions from the same thread (not in terms of cycles). Shorter operation latencies make it easier to fill issue-slots of the VLIW instructions so that more instruction-level parallelism (ILP) can be exploited [6].

The register file complexity for interleaved multithreading is high. First we discuss a simplified version of dynamic interleaving, called static interleaving

(SI), in Subsection 2.3 We can combine this with blocked multithreading resulting in a technique we term Subset Static Interleaving (SSI), which we will introduce in Subsection 2.4. Subsection 2.5 describes a register file for SSI multithreading. Subsection 2.6 compares SI and blocked multithreading and motivates SSI multithreading as an interesting combination. Figure 1 shows how we arrive at SSI multithreading starting from existing techniques.



**Fig. 1.** Relations between described types of multithreading.

### 2.3 Static Interleaved Multithreading

The interleaving technique described so far is dynamic. The order in which the threads are selected to feed an instruction into the pipeline depends on their availability and is therefore dynamic. A less complex variant would be static interleaved (SI) multithreading, opposite to dynamic interleaved (DI) multithreading, where the order in which the threads are executed is fixed. In an  $N$ -way SI multithreaded core (a core with  $N$  threads), an instruction from thread  $i$  is executed in cycle  $i \bmod N$  (see Figure 2c). A bubble is inserted into the pipeline when a thread is stalled. SI multithreading becomes even less complex relative to DI multithreading if all operation latencies in an  $N$ -way core are a multiple of  $N$  cycles. This can be achieved by inserting empty stages to increase the operation latency of a functional unit up to the next multiple of  $N$  cycles. The result is that all writes to the register file of a particular thread are happening in the same cycle modulo  $N$ . This means that write backs can be statically scheduled and we can use a simplified register file design as described in Section 2.5. Instead of inserting empty stages, one can also use the additional cycles to relax the timing of the operations or reduce the power consumption of an operation. An example of the latter is to replace a data cache with parallel tag and data access by a sequential design where only one way in the data array has to be accessed.

SI multithreading achieves the operation latency reduction that made (DI) interleaved multithreading attractive. If normally an operation takes  $C$  cycles



while in the case of  $M = N$  it corresponds to SI multithreading. Both blocked and SI multithreading has distinct advantages. Blocked multithreading better tolerates memory latency while SI multithreading reduces memory latency as well as operation latency. By choosing  $M$  between one and  $N$  we obtain a solution that performs better than the two extremes as shown by our experiments.

## 2.5 Register File Design

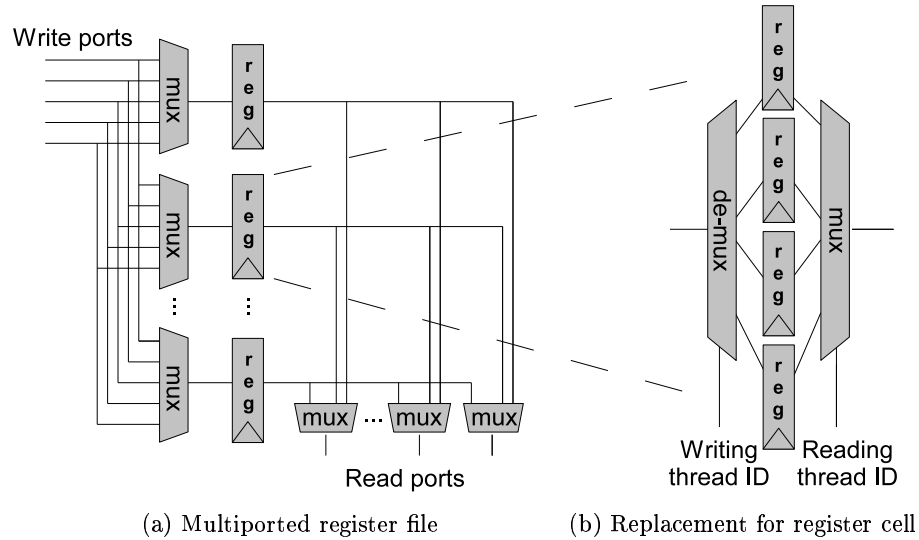
The register file is one of the most complex parts of wide-issue VLIW cores, which require a large number of ports and registers to feed multiple FUs with data. Processor cores for embedded systems often employ a RF (Register File) constructed of standard cells in contrast to many non-embedded processors with custom-layout RFs. Figure 3a shows the structure of a standard cell RF composed of write multiplexers, registers in flip-flops, and read multiplexers. Each register has a multiplexer tree at the input to select a value from one of the write ports. Furthermore, register’s outputs are connected to multiplexers that select a register for each of the read ports.

**Table 1.** Maximum number of accessed contexts for the multithreading schemes.

	<b>SMT</b>	<b>Blocked</b>	<b>Dynamic Interleaved</b>	<b>Static Interleaved</b>	<b>SSI</b>
<i>reading</i>	all	one	one	one	one
<i>writing</i>	all	one	all	one	one

An  $N$ -way multithreading core needs  $N$  times more registers. Let us consider access requirements of different multithreading schemes in Table 1. The *reading* row shows how many hardware threads can read source operands from their respective RFs each cycle. The *writing* row specifies how many hardware threads can write to their respective RFs each cycle. With  $N$  total threads in the cases of blocked and SI multithreading, as well as with  $N$  foreground threads in the case of SSI multithreading with all operation latencies being a multiple of  $N$  cycles, all reads from the same thread as well as all writes from the same cycle are happening in the same cycle modulo  $N$ . Thus, the Blocked and (Subset) Static Interleaved schemes each cycle read from only one register file and write to only one register file. This suggests that the read and write port multiplexer trees from Figure 3a can be shared by all the threads. Note, that the shared read and write multiplexers for multi-ported RFs often dominate implementation complexity. In fact, one can simply replace the register cell with multiple registers for different hardware contexts as shown in Figure 3b. The new multithreaded register cell consists of  $N$  regular registers and multiplexers around it that are controlled by the thread IDs of the reading and writing threads. Note, that the read (write) multiplexers in the multithreaded register cell are shared by all read (write) ports. This way, register file complexity for multithreaded schemes Blocked and (Subset) Static Interleaved can be limited. Our experimental layout exercises in

CMOS 65nm technology indicate that the 4-way multithreaded RF designed as shown in Figure 3, is only 2.35 times larger than the single-threaded RF.



**Fig. 3.** Adapting the register file for multithreading. (a) Multi-ported register file for single threaded core. (b) Replacement for register cell to create a register file for a 4-way multithreaded core.

## 2.6 Multithreading Comparison

In order to understand the benefit of a combination between blocked and SI multithreading we should understand the advantages and disadvantages of both techniques.

1. The *single thread performance* of an  $N$ -way SI multithreaded core can be up to  $N$  times lower than the single threaded core, because an instruction from a thread is issued once every  $N$  cycles. Single thread performance is important when TLP is low.
2.  $N$ -way SI multithreading *reduces operation latency* by a factor of  $N$  which improves the ILP that can be exploited.
3.  $N$ -way SI multithreading *reduces memory latency* by a factor of  $N$  whereas blocked multithreading is able to reduce it to zero provided that at least one thread stays runnable.
4.  $N$ -way SI multithreading *needs  $N$  threads* to keep the core utilized, while blocked multithreading needs only one in the absence of stalls.
5. For blocked multithreading *the switch penalty* is determined by the pipeline stage where the switch is triggered. This stage and all earlier stages have to be

flushed. For  $N$ -way SI multithreading only the stages holding instructions of the same thread have to be flushed, which is a factor of  $N$  lower, and therefore the switch penalty is also  $N$  times lower.

So both techniques have their merits and by selecting a proper value for the number of foreground threads for SSI multithreading we hope to maximize performance.

### 3 Task scheduling

Our programming model relies on the programmer to extract parallelism from the application and define concurrent tasks. Besides the well-known Pthread API, the system can be programmed by short-running tasks that can be submitted and dispatched with low overhead. This makes fine-grain TLP possible. Submission of a task involves its registration in a shared pool of ready-to-execute tasks whereas dispatching allocates a task to a core and starts it up. The low overhead ( $\pm 15$  cycles for submit and dispatch) is enabled by a hardware device, called the task scheduling unit (TSU), that is shared by the cores.

#### 3.1 Task Scheduling Unit

Our task scheduling unit is based on the Carbon design [7]. It is basically a pool of tasks in a hardware structure to which task can be submitted and retrieved with special operations. This makes it an order of magnitude faster than an optimized software implementation would be. A task is typically a function pointer of the function that performs the task and arguments for it.

The TSU implements distributed task stealing, which also is used for software task pool implementations [8]. The TSU maintains a double-ended queue of tasks for every core in the system. Task submissions and retrievals are done on one side of the queue so that the order is first-in-last-out. Choosing the newest instead of the oldest is typically better for cache locality because the newest task often has data in common with the latest executed task. When a core wants to retrieve a task from its queue and finds it empty, it steals a task from a randomly chosen queue at the opposite side from which task are submitted, i.e., the oldest task in the queue.

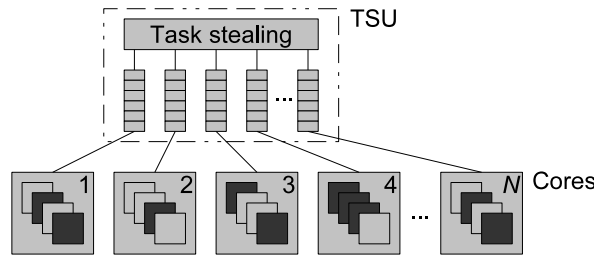
Because the TSU has a finite capacity, it generates an interrupt when a queue gets nearly full. An interrupt handler will then spill tasks to an overflow area in memory. When later the queue gets nearly empty, again an interrupt is generated to copy back tasks from the overflow area.

#### 3.2 Improvement for Multithreading

Whenever a software thread running on a core requests the TSU for a task and there is no task in any of the queues, the TSU blocks the requesting thread until a task becomes available. Then typically the longest blocked thread or a

randomly selected blocked thread gets the task. In a multithreaded multi-core system this can easily lead to imbalance where, for example, some cores have only one runnable thread while others have all threads runnable. We can reduce such imbalance by making the TSU aware of the multithreaded cores. The heuristic, which we call most-blocked-first, is to assign a task to a blocked hardware thread on a core which has the most threads blocked on the TSU, i.e., the least loaded core. This avoids much of the load imbalance problem.

Figure 4 shows a multicore system with a TSU. It illustrates that a submitted task will be allocated to core 4 because this core has the most threads blocked.



**Fig. 4.** Task scheduling unit. There is a queue in the TSU for every core. The cores have four threads shown as overlaid squares. Blocked threads are shown in dark color. Let us assume that core 1 submits a task to the TSU. In the case of the most-blocked heuristic, the task stealing module will send this task to the queue for core 4 because this core has most threads blocked (3 threads).

The improvement is relevant at the moments when there is insufficient task-level parallelism to keep all threads utilized. Handling of these moments efficiently is very important to obtain close to linear speedups.

## 4 Parallel H.264 Decoding

The H.264 / MPEG-4 Advanced Video Coding (AVC, part 10 of MPEG-4) standard for video compression can be seen as an improved successor of MPEG-2 and other parts of MPEG-4 [9, 10]. It achieves a higher compression performance by more advanced coding techniques that are also much more compute intensive, and in particular, more control intensive. An example is sub-pixel motion compensation, which has 1/2 pixel resolution in MPEG2 and 1/4 pixel in H.264. The higher resolution makes motion compensation more expensive for H.264.

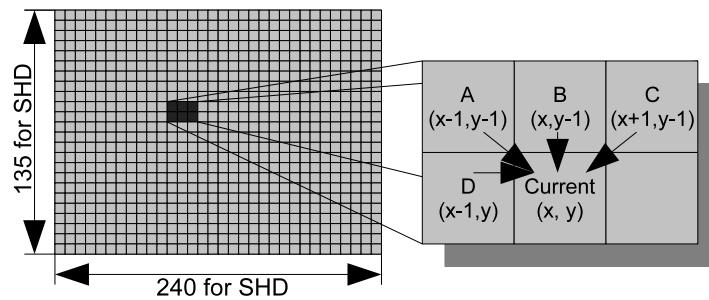
The high computational requirements make programmable implementations of H.264 decoding very challenging. While SD resolution (Standard Definition,  $720 \times 480$ ) is possible on DSPs, HD (High Definition,  $1920 \times 1080$ ) is only possible on high clock frequency cores or configurable cores [11], and SHD (Super HD,  $3840 \times 2160$ ) is out of reach for single core programmable designs. Therefore, the only programmable solution appears to be a multicore one.

A H.264 decoding consists of three steps: entropy decoding, macroblock decoding, and deblocking. Entropy decoding translates the input bitstream into a sequence of encoded macroblocks. Macroblock decoding is the heart of the decoding process. It translates encoded macroblocks into pixel blocks of 16 times 16. Deblocking is a post pass filter that tries to remove artifacts in the output that are introduced due to block based compression. The decoded frames that are coming out of deblocking step are stored in memory and are used as reference in future frames.

#### 4.1 Parallelization

Parallelization of H.264 decoding could be done by running the three decoding steps in parallel in a pipelined fashion. However, there is a feedback of reference frames from deblocking to macroblock decoding which limits parallelism. Therefore, we parallelized the three steps individually. Unfortunately, parallelization of entropy decoding for H.264 does not appear to be possible. Therefore, we still have to rely on dedicated hardware to do this in real-time for H.264 at SHD resolution. Fortunately, solutions for this exist. For example [12] describes a hardwired solution that needs 45MHz for HD resolution.

Parallelization of macroblock decoding and deblocking is very well possible and can be done in a similar manner [13]. We parallelized these two steps at the macroblock level by decoding macroblocks in parallel. Which macroblocks can be decoded in parallel is determined by dependences between macroblock that are described by the standard. Figure 5 shows these dependences. Macroblock decoding of block *current* in Figure 5 depends on blocks *A*, *B*, *C*, and *D*. Effectively, it depends on only two blocks *C* and *D* because these dependences cover the dependence on *A* and *B* as well. For deblocking, block *current* depends on *B* and *D*. We can decode/deblock macroblocks in parallel as long as we respect the dependences.



**Fig. 5.** Dependences between macroblocks.

The parallelization is implemented by associating a reference counter with each macroblock. The value of the reference counter associated with macroblock

$M$  corresponds to the number of macroblocks on which  $M$  depends that have not been processed yet. This is a value between zero and two, where zero indicates that the macroblock is ready for processing. When a macroblock becomes ready for processing, a task is submitted to the TSU that performs the processing. Such a task does the actual processing, i.e. decoding or deblocking, and decrements reference counters to blocks on which the block depends afterward. Whenever a reference counter becomes zero, a task is submitted to the TSU to do the processing of the macroblock corresponding to it. The resulting parallel execution is a wave-front of macroblock processing from the top-left of the frame to the bottom-right. Parallelism ramps up in the beginning and ramps down at the end of a frame. In between the parallelism depends on the number of macroblocks on the wave-front, which is 120 at the peak for SHD for macroblock decoding and 135 for deblocking.

Note, that the scheduling of the decoding and deblocking tasks is dynamic. A static schedule would not be efficient as the execution time for each macroblock varies a lot.

## 4.2 Tail Submits

A task with task submits at the end of the task resembles a tail recursive function. Similar to recursive functions, we can replace a recursive task submit at the end of a task by a jump back to the beginning of the task. Figure 6 shows this optimization for macroblock processing in simplified pseudo code. In the original code, shown in Figure 6a, we see the actual decoding work followed by decrements of reference counters of macroblocks that depend on the just decoded macroblock. This has to be performed atomically to prevent inconsistencies. Depending on the outcome, up to two macroblocks become ready for processing, which are submitted to the task pool.

The optimized version of the code where tail submits have been replaced by jumps (goto's) to the beginning of the task is shown in Figure 6b. After decrementing the reference counters we determine how many macroblocks become ready for processing. If this is one macroblock, then we adjust the  $(x, y)$  position and jump back to the beginning of the task. If there are two ready macroblocks, we submit one and process the other one directly after that by jumping back to the beginning.

This optimization gives us two benefits. First, there is overhead involved in task submission and dispatching, which occurs less frequently after this optimization has been applied because one task performs the work for multiple macroblocks. Furthermore, there is task invariant code that can be moved out of the loop that is created by this optimization. The second benefit is more subtle. Because we have more control over the order in which the macroblocks are executed, we can obtain better data cache locality. This is achieved by executing macroblock  $(x+1, y)$  directly after  $(x, y)$  when possible and  $(x+1, y)$  instead of  $(x-1, y+1)$  when both are possible. Because the pixel lines of macroblocks  $(x, y)$  and  $(x+1, y)$  are adjacent in memory, many data that is needed for  $(x+1, y)$  is

<pre> decode_mb(int x, int y) {     ... decoding (x, y) ...      ready1 = atomic_decrement(...);     ready2 = atomic_decrement(...);     if(ready1)         submit(decode_mb, x+1, y);     if(ready2)         submit(decode_mb, x-1, y+1); } </pre>	<pre> decode_mb(int x, int y) {     L: ... decoding (x, y) ...      ready1 = atomic_decrement(...);     ready2 = atomic_decrement(...);     if(ready1 &amp;&amp; ready2) {         submit(decode_mb, x-1, y+1);         x += 1;         goto L;     } else if(ready1) {         x += 1;         goto L;     } else if(ready2) {         x -= 1; y += 1;         goto L;     } } </pre>
(a) Original code	(b) After optimization

**Fig. 6.** Tail submits.

already in the data cache because it was necessary for  $(x, y)$ . Therefore, more data will be reused.

## 5 Evaluation

### 5.1 Experimental Setup

Figure 7 shows the simulated architecture composed of multithreaded TriMedia cores, hardwired H.264 entropy decoder, TSU, synchronization unit, and shared memory. We used the following L1 data cache parameters values: 64KB size, 64B line size, 4 way set associative, fixed 40 cycles reload penalty, write back, MESI cache coherence, and allocate-on-write-miss policy [14]. The latencies, functional units, and issue-width (5 issue) are the same as the TM3270. We chose a fixed 40 cycle miss penalty reflecting a 40 cycle average miss latency. We do not model the shared L2 cache, L1 instruction caches and contention on interconnect and off-chip SDRAM memory. However, we believe that the high average miss latency represents the unmodelled parts in modern embedded systems on a chip.

For synchronization we modeled a synchronization unit that provides locks to implemented atomic operations. Among other purposes, they are necessary to perform atomic decrements on reference counters (see Section 4).

Our multithreading implements a priority scheme similar to the one of the IBM RS64 IV [15]. This scheme exchanges a foreground and background thread also when the former has a lower priority than the latter. One of the purposes of priorities is to reduce priority during the idle loop of the Pthread scheduler

so that it does not consume resources when there are other threads that are executing useful code.

The H.264 decoder that we have parallelized and use for our experiments is optimized including extensive usage of intrinsics, and ILP exposing transformations.

We compiled the decoder with the TriMedia production compiler. For the runs with more than one foreground thread we compiled the application and the supporting libraries with reduced operation latencies and branch delay slots as described in Sectionsec-ssi.

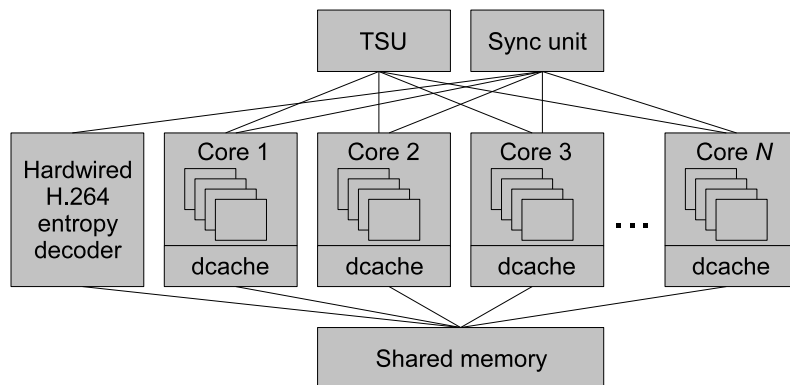


Fig. 7. Architecture of the simulated system.

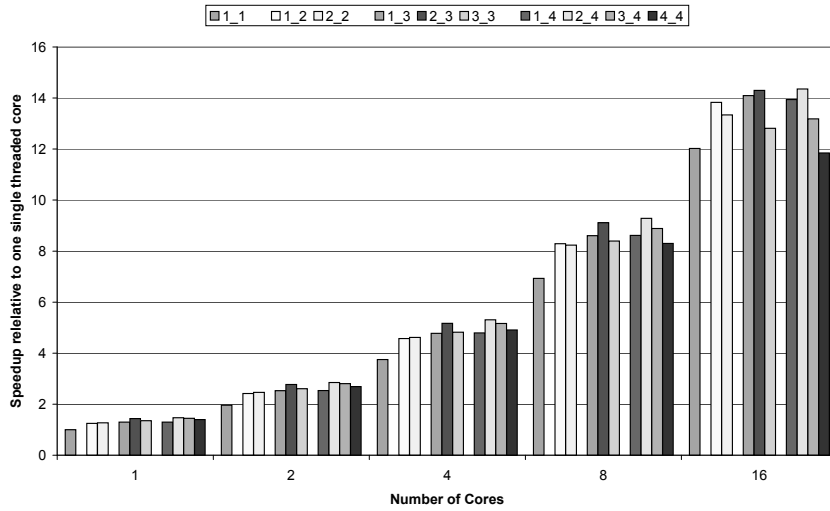
## 5.2 Performance of SSI Multithreading

For evaluation of the described system and parallelized decoder, we use a video stream from the SVT High Definition Multi Format Test Set called 'CrowdRun'. We encoded this into a main profile H.264 bitstream. Its resolution is  $2160 \times 3840$ , which is the closest to SHD ( $1920 \times 3840$ ) that we could find. Due to long simulation times, execution is restricted to 14 frames.

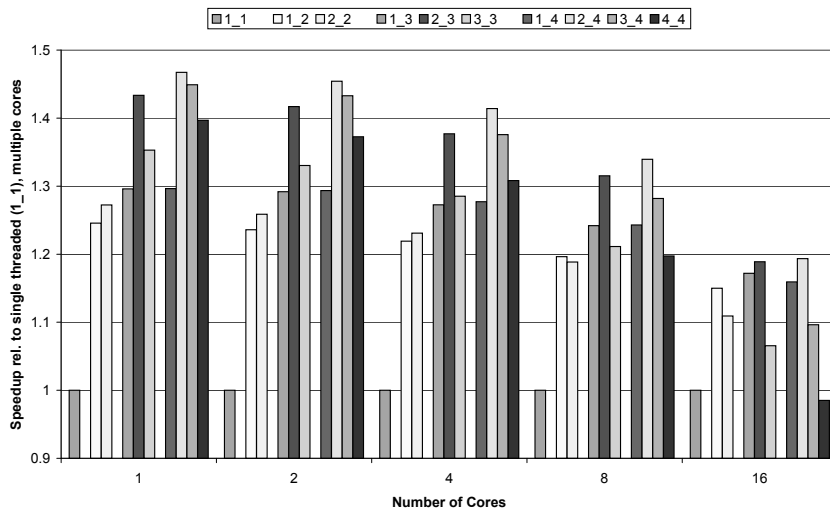
We vary the number of cores from 1 to 16 in powers of two. Furthermore, we experiment with 1 to 4 threads per core, and vary the number of foreground threads from one (blocked) to the number of threads (SI).

We evaluated two versions of the decoder, with and without the tail submit optimization (see Section 4.2), benchmarking different levels of manual optimization. As discussed in Section 4.2, the two versions differ in cache locality where the optimized version has a better locality.

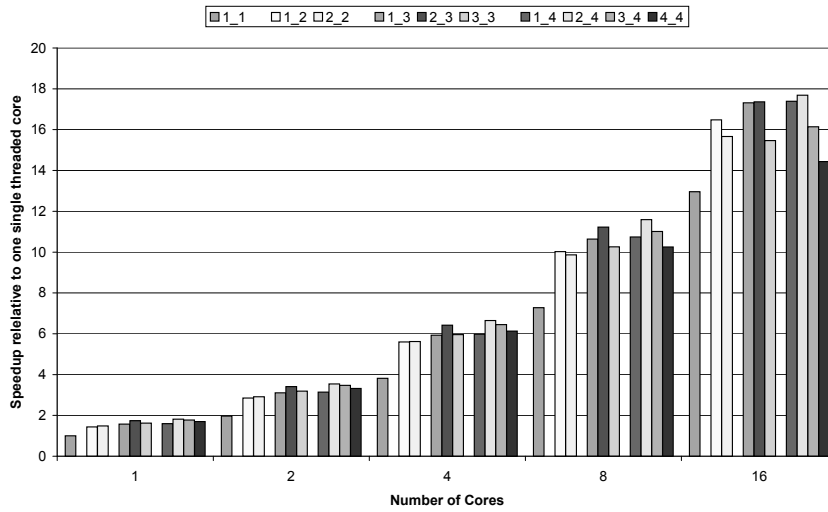
Figures 8 to 11 show the outcome of our measurements where Figures 8 and 9 are for the tail submit optimized version and Figures 10 and 11 for the version without the optimization. Figures 8 and 10 show speedup relative to one single threaded core, while Figures 9 and 11 show the speedup relative to a single



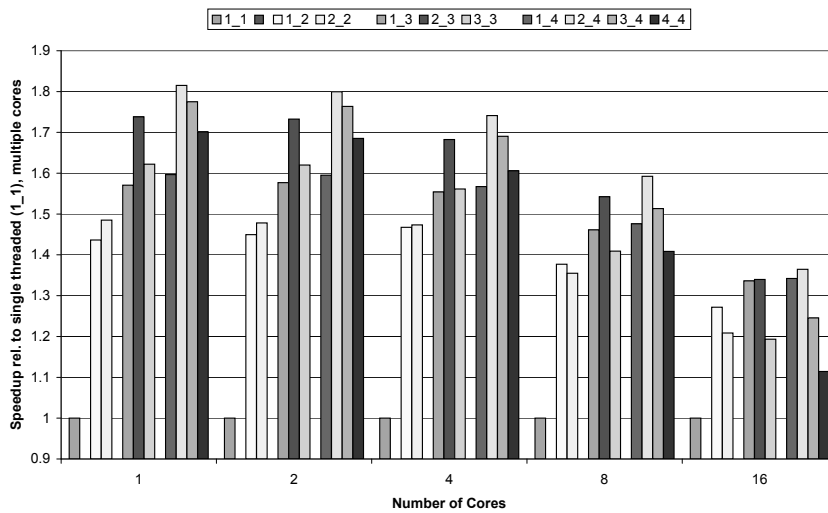
**Fig. 8.** Speedup for 1, 2, 4, 8, 16 cores relative to single threaded, single core. The bars are grouped in five groups; each group for a certain number of cores. The groups are further grouped in subgroups; each subgroup for a number of threads. The bars within a subgroup vary in the number of foreground threads. The bars are labeled  $F_T$ , where  $F$  is the number of foreground threads and  $T$  the total number of threads per core. Hence, blocked multithreading is at the left ( $1_N$ ) and SI ( $N_N$ ) multithreading at the right. Tail submits have been optimized.



**Fig. 9.** The same as Figure 8 but speedup relative to single threaded multi-cores.



**Fig. 10.** The same as Figure 8 but without tail submit optimization. Note that Figure 8 has a different vertical scale. Also notice that although the speedup relative to one single threaded core is better than the version with the tail submit optimization, the absolute running times are longer.



**Fig. 11.** The same as Figure 9 but without tail submit optimization. Note that Figure 9 has a different vertical scale.

threaded system with the same number of cores, illustrating the performance advantage of multithreading.

It is important to note that the two versions of the H.264 decoder differ in execution time even on one single threaded core. The optimized version runs 1.31 times faster on one single threaded core. Hence, the baselines for Figures 8 and 10 are different. Results show speedups of up to 14.4 and 17.7 times for the optimized and non-optimized versions of the code, respectively. These numbers demonstrate the effectiveness of the H.264 decoder parallelization on our architecture.

Results also show that multithreading becomes less effective as the number of cores increases (see Figures 9 and 11). This is because the application does not provide sufficient parallelism for the hardware, which provides a TLP of 64 (for 16 cores and 4 threads per core).

The bars in Figures 8 to 11 are gathered in groups with the same number of cores and threads where the number of foreground threads vary from one (left, blocked) and all threads (right, SI). This shows that SSI multithreading (bar(s) in the middle) performs in all cases better than SI and blocked multithreading. The configuration with two foreground threads provides the best results. Averaged over 1 to 16 cores and the two versions of the code, the speedup of SSI multithreading relative to no multithreading is 1.52 times. SI multithreading and blocked multithreading achieve speedups of 1.37 and 1.38 times, respectively.

Comparing the two versions of the code, we see that multithreading is more effective on the version without the tail submit optimization than the version with. The speedups due to SSI multithreading for the first version are in the range of 1.36 to 1.81 while for the latter version it is 1.19 to 1.47. As discussed in Section 4.2, this can be explained by the data locality improvements of the tail submit optimization. This demonstrates the importance of employing optimized benchmarks, because non-optimized benchmarks exaggerate the performance benefit from multithreading.

It is also interesting to see that SI multithreading performs better than blocked multithreading with a low number of cores and becomes worse with a higher number of cores. As discussed in Section 2.6, SI needs more application TLP to keep hardware threads utilized. With more than eight cores the application is not able to provide sufficient TLP for SI multithreading. When the number of cores is low, though, the SI multithreading outperforms blocked multithreading due to reduced operation latencies.

### 5.3 Multithreading Aware Task Scheduling

In Section 3.2 we described how to make the TSU aware of the fact that the cores are multithreaded by means of the most-blocked-first heuristic. We argued that this is most important when the application utilizes less parallelism than the system provides. To evaluate this, we run the decoder on systems with 8 and 16 cores, 4 threads, and 1 to 4 foreground threads. We evaluate it on a SHD and HD streams. The latter has two times lower macroblock-level parallelism.

Table 2 shows the improvement of the most-blocked-first heuristic relative to selecting the longest blocked thread for a new task. It shows that the importance rises as the parallelism of the system increases (8  $\rightarrow$  16 cores) or the parallelism of the application decreases (SHD  $\rightarrow$  HD). Results also show that multithreading aware task scheduling is more effective with fewer foreground threads. This is because threads get more cycles assigned to them on a partially loaded core if there are fewer foreground threads. The reason that there is still a slight speedup for SI multithreading is likely due to reduction in data cache sharing. Without caching effects, multithreading aware task scheduling should not be effective on SI multithreading.

**Table 2.** Speedups of the most-blocked-first heuristic.

Cores	Stream	Speedup (%)			
		number of foreground threads			
		1 (blocked)	2 (SSI)	3 (SSI)	4 (SI)
8	HD	4.92	2.86	1.18	0.03
8	SHD	1.97	0.41	-0.23	0.14
16	HD	17.61	8.68	2.50	0.36
16	SDH	6.48	3.23	1.36	0.27

#### 5.4 Discussion

The optimal number of foreground threads is application dependent. It depends on the importance of the items listed in Section 2.6 for a particular application. For the optimized H.264 decoder we found that two foreground threads is optimal for three and four threads. However we also experimented with SPLASH-2 benchmarks [16] that are not optimized for TM3270. For most of the SPLASH-2 benchmarks, the ILP on a single threaded core is low because floating point operation latencies are long and there are many memory references that the compiler can not disambiguate. Therefore, for most SPLASH-2 benchmarks the optimal number of foreground threads equals the number of threads, i.e., SI multithreading.

We can design cores such that the number of foreground threads is a run-time parameter. For example, we can construct a core with four threads where the number of foreground threads can be two or four. At run-time, the application can switch between the modes whenever it switches between parts of the application with different characteristics. These parts should be compiled (scheduled) with different operation latencies.

Real-time performance is typically very important for media processing applications. Although we did not experiment with it, multithreading might improve real-time performance because variation in the input data that translate in variation of memory access locality can be suppressed by the latency tolerance that multithreading provides.

## 6 Related Work

SSI multithreading is a combination of blocked multithreading and SI multithreading. With one foreground thread it corresponds to blocked multithreading and if all threads are foregrounds threads it is equivalent to SI multithreading.

Zuberek describes enhanced interleaved multithreading which is also a combination of blocked multithreading and SI multithreading [17]. He bases his performance analysis on Petri net models instead of executing application code. Moreover, Zuberek does not address implementation issues such as register file design.

Balanced multithreading, as proposed by Tune and et al., combines simultaneous multithreading and blocked multithreading in firmware so that long stalling threads are removed from the pipeline to free resources [18].

Examples of interleaved multithreaded machines are Horizon, MIT M-machine, Tera/Cray MTA, Denelcor HEP, Sun Niagara, Sandbridge Sandblaster [19], and MicroUnity MediaProcessor [20]. From what we know it appears that the MediaProcessor and the Sandblaster apply SI multithreading.

Van der Tol et al. pioneered parallel H.264 decoding [13]. They motivated parallelization at the macroblock level and described the dependences between macroblocks as well as the wave-front parallelism that results from it. In comparison with this work, van der Tol et al. do not report performance measurements, apply their work on HD streams, and do not use highly optimized code.

Ramadurai et al. describe a parallelized H.264 decoder for the Sandblaster multithreaded DSP [21]. They use intra macroblock parallelism for macroblock decoding, and, like [13] and our work, inter macroblock parallelism for deblocking. A disadvantage of intra macroblock parallelism is that it does not scale with the frame resolution. Ramadurai et al. do not report performance figures.

Parallelization of MPEG-2 decoding is easier to realize in comparison to H.264. An example of this is the work of Bilas et al. [22]. MPEG-2 has slices that can be decoded independently of at most one row of macroblocks. Furthermore, slices are easy to recognize in the bitstream by means of start codes. H.264 has the concept of independent slices as well but in the case of H.264 the slices could cover a whole frame, which would mean no parallelism.

## 7 Conclusions

We demonstrated the applicability of a multithreaded multicore for parallel H.264 decoding. The main contribution of this paper is the novel type of multithreading combining blocked multithreading and a restricted form of interleaved multithreading. This type, that we call subset static interleaved (SSI) multithreading, achieves reduced operation latencies, memory latency tolerance, fast context switching, and a relatively low design complexity of the register file. Results show a speedups of up to 17.7 times for 16 cores with four threads relative to a single core, single threaded system on a manually parallelized H.264 on a SHD bitstream. Furthermore, the proposed technique, SSI multithreading,

achieves a speedup of 1.52 times relative to no multithreading, while blocked multithreading achieves 1.38 times and a restricted form of interleaved multithreading achieves 1.37 times speedup.

Furthermore, we use a task scheduling unit that is able to balance the load over the multithreaded cores. This improves performance in parts of the application where there is insufficient work for all threads. We observed the performance increase of 6.84% from the improvement of the task scheduling unit for multithreading. On an HD stream we measured speedups of up to 17.61%.

Future work includes optimization and benchmarking of other media applications (e.g., 2D to 3D conversion and frame-rate conversion), more accurate modeling, and RTL implementation to obtain accurate latency, area, and clock frequency figures. Furthermore, we would like to benchmark SSI multithreading against DI multithreading.

**Acknowledgments:** We would like to thank Arno Glim for providing the optimized H.264 decoder, Jan-Willem van de Waerdt for discussions, Chris Yen and Magnus Sjölander for back-end support in register file layout exercises.

## References

1. van de Waerdt, J.W., Vassiliadis, S., Das, S., Mirolo, S., Yen, C., Zhong, B., Basto, C., van Itegem, J.P., Amirtharaj, D., Kalra, K., Rodriguez, P., van Antwerpen, H.: The TM3270 Media-Processor. In: MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2005) 331–342
2. Ungerer, T., Robič, B., Šilc, J.: A Survey of Processors with Explicit Multithreading. *ACM Comput. Surv.* **35**(1) (2003) 29–63
3. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous Multithreading: Maximizing On-chip Parallelism. In: ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture, New York, NY, USA, ACM Press (1995) 392–403
4. Keckler, S.W., Dally, W.J.: Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In: ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture, New York, NY, USA, ACM Press (1992) 202–213
5. Özer, E., Conte, T.M., Sharma, S.: Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. In: Proceedings of the 8th International Conference on High Performance Computing (HiPC'01), Hyderabad, India (2001) 1520–6149
6. Jouppi, N.P., Wall, D.W.: Available Instruction-level Parallelism for Superscalar and Superpipelined Machines. In: ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems, New York, NY, USA, ACM Press (1989) 272–282
7. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. In: ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture, New York, NY, USA, ACM Press (2007) 162–173

8. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. In: PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM Press (1995) 207–216
9. Wiegand, T., Sullivan, G.J., Bjntegaard, G., Luthra, A.: Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. Circuits Syst. Video Techn* **13**(7) (2003) 560–576
10. Richardson, I.E.: H.264 and MPEG-4 Video Compression. John Wiley and Sons (2003)
11. Sci-Worx: MSVD-HD, Multi-Standard High Definition Video Decoder (2006) [www.sci-worx.com](http://www.sci-worx.com).
12. Chen, J.W., Lin, Y.L.: A High-Performance Hardwired CABAC Decoder. In: IEEE International Conference on Acoustics, Speech and Signal Processing, Santa Clara, California, United States (2007) 1520–6149
13. van der Tol, E.B., Jaspers, E.G., Gelderblom, R.H.: Mapping of H.264 Decoding on a Multiprocessor Architecture. In: Image and Video Communications and Processing, Santa Clara, California, United States (2003) 707–718
14. van de Waerd, J.W., Vassiliadis, S., van Itegem, J.P., van Antwerpen, H.: The TM3270 Media-Processor Data Cache. In: Proceedings of the IEEE International Conference on Computer Design. (2005) 334–341
15. Borkenhagen, J., Eickemeyer, R., Kala, R., Kunkel, S.: A Multithreaded PowerPC Processor for Commercial Servers. *IBM Journal of Research Development* **44**(6) (2000) 885–898
16. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture, New York, NY, USA, ACM Press (1995) 24–36
17. Zuberek, W.M.: Performance Analysis of Enhanced Fine-Grain Multithreaded Distributed-Memory Systems. In: Proc. IEEE Conference on Systems, Man, and Cybernetics, Tucson, Arizona, United States (2001) 1101–1106
18. Tune, E., Kumar, R., Tullsen, D.M., Calder, B.: Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy. In: MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2004) 183–194
19. Schulte, M., Glossner, J., Jinturkar, S., Moudgill, M., Mamidi, S., Vassiliadis, S.: A Low-Power Multithreaded Processor for Software Defined Radio. *J. VLSI Signal Process. Syst.* **43**(2-3) (2006) 143–159
20. Hansen, C.: MicroUnity's MediaProcessor Architecture. *IEEE Micro* **16**(4) (1996) 34–41
21. Ramadurai, V., Jinturkar, S., Moudgill, M., Glossner, J.: Multithreading H.264 Decoder on Sandblaster DSP. In: Proceedings at the 2005 Global Signal Processing Expo (GSPx) and International Signal Processing Conference (ISPC), Santa Clara, California (2005)
22. Bilas, A., Fritts, J., Singh, J.P.: Real-Time Parallel MPEG-2 Decoding in Software. In: IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing, Washington, DC, USA, IEEE Computer Society (1997) 197–203