A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures

Magnus Själander
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Andrei Terechko and Marc Duranton
Embedded Modem and Media Subsystems
NXP Semiconductors
High Tech Campus 32,
5656 AE Eindhoven, Netherlands

Abstract

Efficient utilization of multi-core architectures relies on the partitioning of applications into tasks and mapping the tasks to cores. In some applications (e.g. H.264 video decoding parallelized at macro-block level) these tasks have dependencies among each other. Task scheduling, consisting of selecting a task with satisfied dependencies and mapping it to a core, is typically a functionality delegated to the Operating System. In this paper we present a hardware Task Management Unit (TMU) that looks ahead in time to find tasks to be executed by a multi-core architecture. The look-ahead functionality is shown to reduce the task management overhead by 40-50% when executing a parallelized version of an H.264 video decoder on an architecture with up to 16 cores. In overall, the TMU-based multi-core architecture reaches a speedup of more than 14x on 16 cores running H.264 video decoding, assuming CABAC is implemented in a dedicated coprocessor.

1 Introduction

The current trend in computer architecture is to add more cores to increase performance. This is also true in the embedded domain, where multi-core solutions are common and for which the general trend is even towards many-core architectures. In order to improve performance on a multi-core architecture, it is necessary that applications are partitioned into tasks, which can be executed in parallel on separate cores.

As the number of cores increases, it becomes necessary to partition applications into more and smaller tasks, to keep all the cores busy and accelerate overall application performance. The creation and distribution of tasks (henceforth called *task scheduling*) has commonly been handled in software. However, as tasks become smaller and increase in number, a software solution will introduce overheads and will not be efficient. Kumar *et al.* [1] showed that by adding hardware support for scheduling of fine grained parallel applications, speedups of 1.7-2.1 can be achieved compared to full software task scheduling.

In the multimedia domain, partitioning of an application will commonly introduce dependencies between tasks, forcing tasks to be executed in a certain order. Examples of such dependencies will be shown for a H.264 decoder in section 2 of this article. For this kind of application workloads, task scheduling becomes a *task management* problem. Together with task creation and distribution, it is also necessary to introduce algorithms that keep track of dependencies and that determine when tasks are ready to be executed. The algorithms do not have to be complex, but they still can introduce large overheads. For a parallelized decoder for Super HD H.264 resolution (3840 pixels x 2160 lines), the code for keeping track of when a task can be executed accounts for 9% of the execution time of parallelized sections.

Code for managing tasks is generally simple, consisting of arithmetic operations (such as integer addition, subtraction, and comparison), branching and atomic loads and stores. These operations can be efficiently implemented in a *task management unit* that will offload the dependency computation from conventional cores. This will allow for a more efficient use of resources, since a tailored unit can execute dependency checks more efficiently, in terms of power and area, than executed on a conventional core.

A task management unit can wait until a task executed by a core has finished its execution before updating its dependencies. In this way, the task management unit will have the current state of dependencies and will know exactly which task(s) that can be executed next. However, if there is no previous task that is ready for execution when the core finishes its current execution, it has to wait for the task management unit to update the dependencies and see if there are new tasks ready for execution. This introduces extra delays, since the check is performed in between two executed tasks.

It is in general not possible to update dependencies before a task has finished its execution. However, while a task

is being executed, it may be possible to find what dependencies will be solved by the currently executed task. If there are tasks that only depend on the currently executed task, then these tasks will be ready for execution, once the currently executed task is finished. These ready tasks can be prepared for execution by a task management unit, such that once the core has finished the current execution it can immediately start the execution of the next task. Updates of dependencies can then be executed by the task management unit in parallel with the execution of the task. An added benefit is that tasks dependent on each other are executed by the same core, which can improve data locality.

The proposed architecture consists of look-ahead Task Management Units (TMUs), capable of executing task-dependency checks in parallel with the execution of tasks. Each TMU offloads dependency checks/updates from a number of conventional cores and tries to schedule dependent tasks onto the same core, thus preserving data locality. Distribution of tasks between TMUs is done through a task queue. By executing the task-dependency checks on the TMUs in parallel with the conventional cores, an execution-time speedup of 4.5% for a multi-core architecture running a H.264 decoder for Super HD resolution is achieved, compared to a multi-core architecture solely based on a task queue for task distribution. This constitutes a 50% reduction of the overhead for managing the tasks. Furthermore, offloading work from conventional cores to the TMUs, improves the energy efficiency of a multi-core architecture.

This paper is organized as follows. In section 2 parallel applications with inter-task dependencies are presented. Section 3 describes the general problem of managing tasks for parallel applications with inter-task dependencies. We then present the task management unit and the look-ahead functionality used for accelerating task management in section 4. The evaluation setup and results are presented in section 5 and 6. The paper is concluded in section 9.

2 Parallel Applications with Task Dependencies

For this work, we target application workloads where the partitioning of an application into tasks introduces dependencies between the tasks (e.g. H.264 parallel video decoding with macro-block level parallelism [2] and spatio-temporal motion prediction 3DRS [3]). This kind of applications differs from other parallel workloads, such as server workloads with multiple incoming requests, desktop workloads consisting of multiple programs, and scientific workloads, where the tasks are commonly independent of each other and can be executed in random order. For applications with inter-task dependencies, the execution order is crucial for correct application behavior. The execution order can not always be totally statically determined at compile time, because of variations in task execution time. Hence, our approach manages task dependencies dynamically at run time.

2.1 H.264 Video Decoder

The driving application for the proposed architecture is Super HD (3840 x 2160 pixels) H.264 video decoding [4] and will be used throughout the article as a representative application with task dependencies. The H.264 advanced video compression standard dominates the embedded markets in both high-resolution video (e.g. BluRay players) and low-bandwidth (e.g. Sony PSP) segments. H.264 video decoding in Super HD requires a multi-core architecture, to reach the performance necessary for decoding in real-time (30 frames per second). With the most powerful single-core solutions currently on the market, it is possible to decode HD (720p) H.264 in real time [5]. However, Super HD requires about nine times more performance, which with the current trend of more cores instead of one faster core, makes a multi-core architecture for decoding Super HD H.264 a necessity.

Fig. 1 shows the main components of a H.264 video decoder. Processing of a frame starts with entropy decoding, consisting of either Context-Adaptive Binary Arithmetic Coding (CABAC) or a Context-Adaptive Variable Length Coding (CAVLC) and both are sequential by nature. The frame is then passed on to the picture prediction stage where it is divided into macro-blocks of maximum 16 times 16 pixels. At this stage, also inter-picture prediction and motion-vector estimation are calculated for each macro-block. The frame is then filtered through a deblocking filter to reduce artifacts at block boundaries introduced by the picture-prediction stage. The resulting frame has then been decoded and can be transferred to the display. The decoded frame is also buffered for inter-picture prediction.

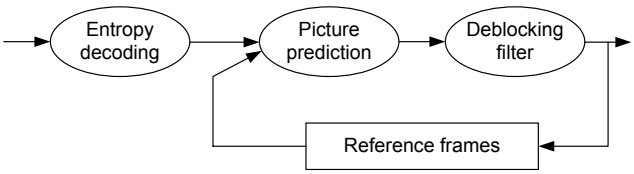

Figure 1. The main components of a H.264 decoder.

The picture prediction and deblocking filter is suitable for parallelization, where the execution of a macro-block can be treated as a task. However, a macro-block cannot be executed before certain neighboring macro-blocks have been executed [2]. A generalized illustration of the macro-block dependencies is shown in Fig. 2. From the figure, it can be seen that before the macro-block marked in gray can

be executed, the macro-block to the left and the three at the top have to be executed.

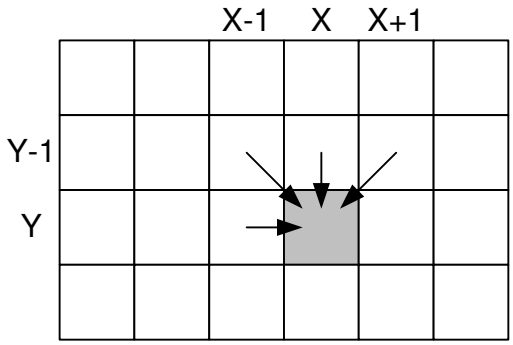

Figure 2. Dependencies between macro-blocks in the H.264 video compression standard.

A task dependency graph can be constructed from the macro-block dependencies, as shown in Fig. 3. The graph starts with the macro-block in the upper left corner, since all its dependencies are solved. When the first task (0/0) has been executed, the second task (1/0) can start. Each of the new tasks can potentially start the execution of one or two other tasks, for example, after task (1/0) both (2/0) and (0/1) can start.

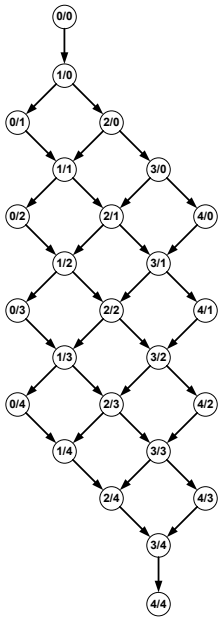

Figure 3. A task dependency graph for a H.264 decoder, with 25 (5x5) macro-blocks in a frame.

Task dependencies can be tracked by storing the number of tasks that each task depends on. For H.264 this value is zero, one, or two. For each finished task, the array with dependencies is updated, decrementing the values of the tasks that depend on the finished task. A task can execute once its value in the array becomes zero [6]. The update of the dependencies for the H.264 decoder can be described by the pseudo code shown in Fig. 4. Here X and Y are the indices of the macro-block that just finished its execution. The first if statement checks that the dependent macro-block is within the frame. If the macro-block is within the frame, its value of number of dependencies is decremented with one. Once the value becomes zero the macro-block has no more dependencies and is ready to be executed. Note that the decrement operations must be executed atomically.

```
if X<FrameWidth-1 then
  decrement(dependencies[X+1][Y])
  if dependencies[X+1][Y] = 0 then
    ready_to_execute(macro_block[X+1][Y])
  end if
end if

if X>0 and Y<FrameHeight-1 then
  decrement(dependencies[X-1][Y+1])
  if dependencies[X-1][Y+1] = 0 then
    ready_to_execute(macro_block[X-1][Y+1])
  end if
end if
```

Figure 4. Pseudo code for inter-macroblock dependency checks.

3 Task Management

In an architecture based on task queues for task scheduling, the execution of a task is followed by a piece of code (Fig. 5) that updates dependencies and checks for tasks ready to be executed. Fig. 5 illustrates the execution of twelve different tasks that are followed by the task dependency execution. Each task is, for simplicity of the drawing, assumed to be identical in execution time, which is not the case in practice. The code for the task dependency check is often simple, but still it can be a significant part of the total execution time. For a Super HD H.264 decoder, the dependency check has been found to increase a tasks' execution time with 9%, on average. Or in other words, the compute power of a complete core is required for managing tasks for every 11 cores in the architecture.

To increase the execution speed of an application with task dependencies and increase silicon efficiency, the dependency checks could be accelerated by a Task Manage-

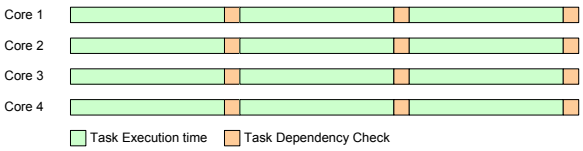

Figure 5. Illustration of the execution for a conventional architecture.

ment Unit (TMU) that keeps track of task dependencies and determines when tasks are ready to be executed. A straightforward implementation of such a TMU is simply to offload the computation of the task dependencies from the core executing the task. Once a task is done, it signals to the TMU that the task is finished and the TMU then starts to look for a new task to be executed by the core. The TMU can be designed to execute task dependency code more efficiently than a conventional core and therefore will execute the code faster. However, if the computation of dependencies is started right after the core finishes its current execution, it will have to wait until the TMU finishes updating dependencies and finds a new task to be executed. Even worse, there will be communication overheads in terms of the core signaling the TMU and TMU instructing the core which task to execute. Therefore, even if the TMU executes the dependency check code more efficiently than a conventional core, the added communication overhead could even degrade the overall performance. Furthermore, if several cores finish at the same time, they will have to wait for each others' dependency code to be executed or the TMU would have to be multi-threaded. Fig. 6 illustrates the execution of twelve tasks executed on an architecture with a naive TMU implementation.

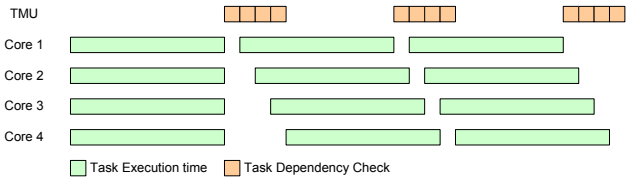

Figure 6. Illustration of the execution for a naive implementation of a task management unit (TMU).

For an efficient solution, it is necessary to execute as much as possible of the task dependency code in *parallel* with the execution of tasks. In the case where there are more tasks than cores, it is possible to fetch tasks from a task queue and have a core to execute it as soon as the core is done with its current task. The TMU will have to keep track of which task finishes and, when it gets spare time, execute the task dependency code for each task and populate the task queue with new tasks. This implementation choice has two major drawbacks: *i)* If there are fewer tasks than cores, the TMU will still have to execute the task dependency code before it can find new tasks for a core to execute, thus introducing delays similar to the straight-forward solution. *ii)* By always storing tasks in the task queue, data locality might be lost, since tasks depending on each other are more likely to be executed on different cores. This would cause extra pressure on the memory subsystem and have a negative impact on execution time.

It is possible to find all tasks that depend on the currently executed task during its execution. All tasks found are then candidates for execution by the core. Of all the found candidate tasks, there can be tasks that only depend on the task that is currently executed by the core. One of these tasks can immediately be executed by the core, once the core is finished with its current task. For example, in Fig. 3 while processing (1/0) the TMU can find out that two tasks, (2/0) and (0/1), can start immediately after (1/0) has been executed. Updating dependencies of the tasks not chosen for execution can then be done in parallel by the TMU. This allows a minimum amount of time between the execution of tasks (henceforth called *turn-around time*) and dependent tasks are executed on the same core, thus improving data locality. For the case where the TMU is not able to find a task ready for execution, it still can improve the turn-around time between tasks. By identifying all dependent tasks and have them readily available, the update of dependencies can be made as fast as possible. Fig. 7 shows an example where twelve tasks are being executed on four different cores, with their look-ahead code being executed in parallel on a TMU. For the first executed task on the first core and the second executed task on the second core, a task is found with its dependencies fulfilled and can be started immediately, once the core has finished the current execution. For the other tasks, the dependencies need to be updated before a task is found for execution.

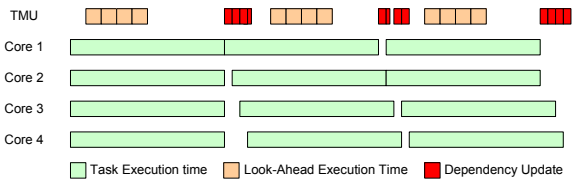

Figure 7. Illustration of an execution of tasks, where the task management unit (TMU) in parallel with the tasks executed on the cores, looks ahead to find tasks with solved dependencies.

4 Task Management Unit

The purpose of the Task Management Unit (TMU) is to offload the management of tasks from conventional cores in a multi-core system. The TMU is therefore closely connected to a number of cores that signals to the TMU each time they have finished a task. While tasks are being executed on the cores, the TMU tries to find tasks that are ready to be executed and have them prepared, so that a core can directly start executing a new task when it finishes its current execution.

For each task being executed, the TMU executes a small function that looks ahead in time, in order to try to find tasks that will be ready for execution. The more cores there are in a system the more look-ahead functions need to be executed at a single point of time. In order to be scalable to more than a few cores, several TMUs are instantiated, with each TMU connected to a limited number of cores. The ratio between TMUs and cores depend on the ratio of task load versus control load. The TMU offloads the control load from several cores and execute them sequentially. A TMU should therefore be faster than the combined control-load for the cores. With one TMU offloading four cores the ratio between the task load and control load can be as high as four to one (25% of the application load). Task queues are then used to distribute tasks between the TMUs. The task queues stores ready tasks that are currently not being executed by a core. Fig. 8 shows an architecture with four TMUs, with each TMU connected to four cores. In this case the TMUs share one task queue. However, more advanced queuing systems can be used to improve performance.

The TMU should be programmable to allow for complex task dependencies, be capable of executing simple arithmetic operations, and have fast branching. Furthermore, a programmable TMU can support irregular task dependencies, even not known statically at compile time. For greater flexibility, each TMU also has an associated hardware mailbox that can be used for message passing. This allows, for example, a programmer to send messages from within the execution of a task to a TMU.

4.1 Look-Ahead Task Management

To keep track of tasks and speed up the turn-around time between executed tasks, each core has a dedicated hardware list. The hardware list holds necessary information about each candidate task. A candidate task is a task that could become ready for execution, once the core has finished with its current task. Fig. 9 shows the list for storing candidate tasks and the information stored for each task.

Task Pointer: The task pointer holds the instruction address of the first instruction of the task.

Argument(s) Pointer: Holds the address to where arguments for the task are stored.

Look-Ahead Pointer: The look-ahead pointer tells the TMU what look-ahead function to execute while the task is executed by the core.

Dependency Pointer: The dependency pointer holds the address to the memory location that stores the number of dependencies that still have to be solved before the task can be executed.

Flags: A set of flags are used for synchronization of the core and TMU and to tell which state the task is in.

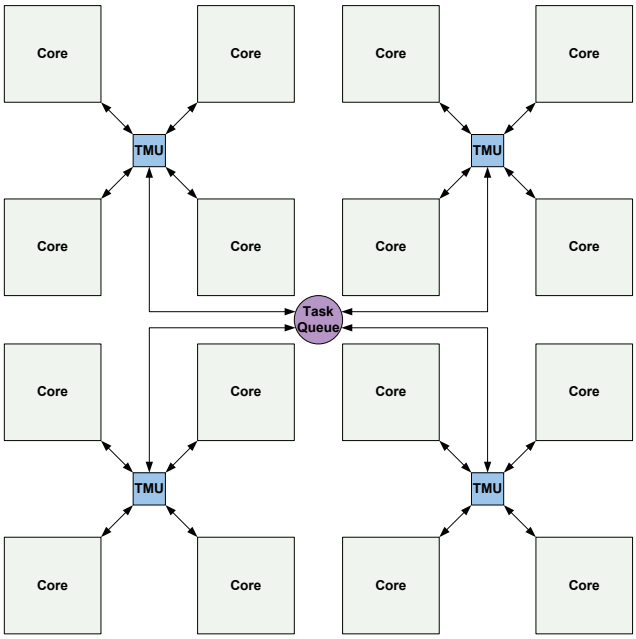

Figure 8. Architecture with 16 cores and 4 task management units (TMU), which share one task queue.

Flags	Look-Ahead Pointer	Dependency Pointer	Task Pointer	Argument(s) Pointer

Figure 9. The candidate list that stores candidate tasks for a core.

The main purpose of the candidate list is to speed up the turn-around time between tasks being executed by a core. This is achieved by allowing direct access to the list from the core. The list is controlled by a simple state machine that inspects the state of the flags of each task in the list. If there is a task ready for execution, then the core reads the task and argument pointer from the candidate list and starts

to execute the new task. Then, in parallel with the execution of the task, the TMU will decrement the value given by the dependency pointers for the tasks not being executed. In case there is no ready task the core will have to wait until the TMU has updated the dependencies to see if a task becomes ready for execution.

The TMU is governed by a simple state machine that checks the state of the task queue, the flags of the candidate list for each core, and for incoming messages. If there is an idle core and a task is in the task queue, a task will be fetched from the queue and the candidate list will be updated. This instructs the core that a task is ready for execution. A small routine is called if a core has finished its execution of a task. The routine first checks for tasks that are ready for execution, but are not executed by the core. These task(s) are stored in the task queue for execution at a later time. Then dependency values for tasks not ready to be executed are decremented. Therefore, only the dependencies of candidate tasks not found ready for execution need to be updated. This reduces the total number of atomic accesses and the pressure on the memory subsystem. Finally, the look-ahead and argument pointer are read for the task currently being executed and the look-ahead function is executed by the TMU, which updates the candidate list with new tasks. Each TMU has also a mailbox for message passing to improve programmability. The state machine checks for incoming messages and will execute a routine for each new message.

The candidate list is a hardware structure with a fixed number of entries. This limits the number of dependent tasks (children) that a particular task (parent) can have. For the H.264 decoder there are no tasks with more than two children, but this might not be the case for other applications. To allow tasks with unlimited number of children, it is possible to add tasks with the only purpose of updating other task's dependencies. This will introduce overheads, so the candidate list should have enough entries for the most common number of children of the applications to be run on the system. However, Stavrou *et al.* [7] performed code analysis that showed that the majority of data-driven multithreading tasks have at most two dependencies, so a candidate list with four entries should be sufficient in most cases.

5 Evaluation Setup

The evaluation of the architecture has been conducted through emulating the architecture in the TTISim simulator [8]. TTISim is a multi-core simulator for the TriMedia VLIW processor family [9]. Each core and Task Management Unit (TMU) is modeled by a TriMedia 3270 processor [10], with the behavior of the TMU implemented in software. The software implementation of the TMU is of course not as efficient as a hardware implementation would be. To emulate the performance of a hardware implemented TMU a stalling technique has been used to remove software overheads.

The *stalling* technique works as follows. For each functionality implemented in software, an estimate has been made of how many cycles it would take to execute the same functionality in hardware. The software implementation is then allowed to run for as many cycles as estimated before all other cores in the system are being stalled. This allows the execution of the remaining part of the software implementation to be executed without influencing the behavior of the rest of the parallel system. The execution time for an application can then be calculated by counting the number of cycles that are being stalled and deducting them from the total execution time. An example of when the TriMedia processors are stalled is when a TMU is updating one of its candidate lists with new tasks. The candidate lists will be implemented in hardware, but for the simulation, they are software structures stored in shared memory. To emulate fast access to the candidate lists parts of the update is therefore stalled.

The TMU is assumed to execute the look-ahead functions at least as fast as the TriMedia processor. This is a reasonable assumption, since the inherent nature of scheduling/control code is that it lacks instruction level parallelism and contains many branches. A specialized datapath with simple arithmetic operations and fast branching should have similar performance or be even faster than the TriMedia processor for this type of code. Furthermore, a specialized datapath should dissipate less power and have a smaller area footprint.

5.1 TriMedia H.264 Video Decoder

A H.264 video decoder [11] that decodes a Super HD (3840 x 2160 pixels per frame) video stream, was selected as application workload. The original H.264 decoder was manually optimized for a single TriMedia TM3270 processor. The optimizations included adding SIMD operations, restrict pointers, loop unrolling, code restructuring, etc.

5.2 H.264 on the Reference Architecture

The H.264 decoder has been manually parallelized to be executed on an architecture based on a shared software task queue [6]. The parallelization of the H.264 decoder consisted of identifying the nested loops and functions for computing the picture prediction and deblocking filter of a frame. The functions for computing a single macro-block were augmented with code for checking what macro-blocks are ready to be executed. The augmented code is similar to the code described in Fig. 4. If only one macro-block is found to be ready, the indices X and Y are updated and a

simple jump back to the beginning of the function is made. In the case where two macro-blocks are found to be active, one of them gets stored in the task queue before updating the indices and jumping back to the beginning of the function. In the case where no macro-block is found to be ready, the function is exited and a new task is fetched from the task queue. The jump to the beginning of the function means that dependent tasks are executed on the same core. This preserves data locality between macro-blocks and removes task creation overhead. Before the first task of each frame is called, an array storing the number of dependencies each macro-block has is initialized. To accurately capture the state of the dependencies, all subsequent updates of the dependency array have to be made atomically.

The reason why an architecture based on a software implementation of a shared task queue has been chosen is because only 0.8-1.7% of the tasks where found to be stored in the task queue when decoding a Super HD H.264 stream, on systems with four to 16 cores. For 98.5-99.3% of the tasks executed, a new macro-block was found ready to be executed with a simple jump to the beginning of the function as a result. The architecture is simulated using the same simulator and 3270 TriMedia processors, which allows for accurate performance comparisons with the proposed TMU architecture. For a different application workload, with more accesses to the task queue, both the architecture based on the task queue and the TMU would benefit from a hardware implementation of the queue(s).

The execution time of each task was measured to improve the understanding of the behavior of the H.264 decoder. The distribution of tasks is shown in Fig. 10. The picture prediction stage of the H.264 decoder has been implemented as two separate functions, PicturePrediction and VectorPrediction. Further, frames can be of P or B type and the distribution for each frame type is shown.

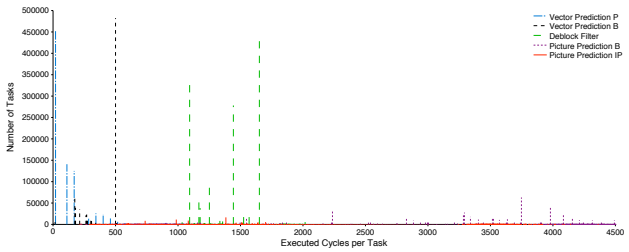

Figure 10. Task execution cycle time for different computations in the H.264 decoder. Tasks larger than 4500 are not shown because they are so few that they would not be visible in the graph.

Table 1 shows a summary of the task distribution and the percentage of the total number of executed cycles that is contributed by each function. The average length of a task was found to be 1270 cycles.

Table 1. Task execution cycle time range and percentage of total execution cycle time for Super HD H.264 decoding.

	Cycle Times	% of Total Cycles
Vector Pred.	20-1153	6.65%
Picture Pred.	208-5326	57.31%
Deblock Filter	978-2017	36.04%

The execution time of the scheduling for the task queue implementation was also measured and was found to be on average 114 cycles. This means that, on average, the overhead for scheduling the next task to be executed is 9%. For tasks with the distribution of the VectorPrediction of frame type P, where many of the tasks are as short as 20 cycles, the average scheduling overhead is 109%. A cycle execution time of 20 cycles is far less than what is commonly considered as the smallest task length for scheduling. Kumar *et al.* [1] considers tasks with at least a few hundred cycles as the smallest task for their carbon architecture.

5.3 H.264 on the TMU Architecture

For the TMU architecture, the parallelized H.264 video decoder was manually modified. The modifications are done such that the code that checks for ready macro-blocks is removed from the tasks and is executed by the TMU instead. The code executed by the TMU only inspects the value stored in the dependency array and if the value is found to be one, then the task is marked as ready in the candidate list.

A single look-ahead function was used to parallelize the H.264 decoder, since all dependencies between macro-blocks can be described by the code given in Fig. 4. However, before the execution of vector prediction, picture prediction, and deblocking filter, initialization functions are needed for setting up the dependency array. The TMU specific code of the H.264 decoder consists of less than 300 lines of C++ code.

The TMU and task queue architectures were simulated with 4, 8, and 16 cores. For the TMU based architecture, a configuration with four cores per TMU was chosen. This configuration is not based on an empirical evaluation and will require further investigation. However, a configuration with four cores per TMU allows for a regular floorplan, similar to the illustration shown in Fig. 8. All simulations are for a perfect memory subsystem. However, the execution order of macro-blocks is similar for both architectures and we should see the same performance degradation for a more realistic memory subsystem.

6 Results

The result from the simulations is shown in Fig. 11. The speedup is given for the parallelized sections of the H.264 decoder. The CABAC encoding for the Super HD video stream is inherently sequential and is assumed to be executed by dedicated hardware. Therefore, its contribution to the total execution time is not considered in the presented speedups. As a reference, the ideal linear speedup is shown in the graph.

As shown by Fig. 11, the Task Management Unit (TMU) architecture outperforms the task-queue architecture. The simulations showed that for 82% of the tasks executed, a task was already waiting to be executed when the current task finished its execution. This is translated into a performance improvement of 4.5% for the eight core configurations and a lowest performance improvement of 3.4% for the 16 core configurations. This constitutes a 40-50% reduction of the task management overhead seen for the task queue simulations.

When the number of cores increases, there are fewer tasks to execute per core and hence, in some cases, there will be idle cores. This explains why there is a drop in the speedup as the number of cores increase.

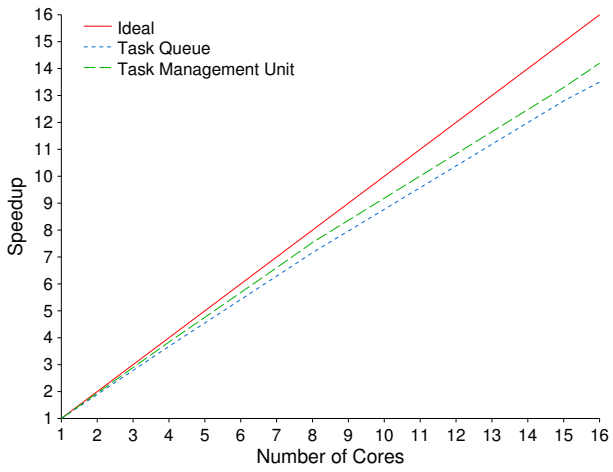

Figure 11. The graph shows the speedup as a relation to the number of cores, for executing a Super HD H.264 decoder on two different architectures based on task management unit(s) or a task queue.

Our simulations show that the execution of VectorPrediction tasks is on average 5% slower when executed on the TMU than if solely computed by the conventional cores. This suggests that the TMU becomes a bottleneck when tasks are extremely small. The TMU sequentializes the control and when the control load offloaded from each processor, is higher than 25% of the cores application load, the TMU becomes overloaded. The overloaded TMU introduces delays, since the cores have to wait for the TMU to execute the look-ahead functions. To avoid this, a different configuration with fewer cores per TMU can be used or tasks could be combined to form slightly larger tasks, thus reducing the number of look-ahead functions executed by the TMU. As mentioned in the previous section, VectorPrediction has many small tasks (20 cycles) and tasks of such small size are commonly not considered for task management.

The deblocking filter stage of the H.264 decoder also has a small section that can be parallelized without introducing dependencies between the tasks. For this kind of tasks the TMU does not bring an added value, since there is no need to execute any look-ahead functions. The simulations have shown that the TMU architecture executes these as fast as the task pool architecture. These tasks are not shown in Fig. 10, but is included when calculating the speedups for Fig. 11.

7 Related Work

Carbon [1] by Kumar *et al.* accelerates creation and distribution of tasks by the introduction of hardware task queues. However, updates and checks of task dependencies are executed by the cores themselves. The Task Management Unit (TMU) offloads the update and check code from the cores, which allows for a more efficient execution. Carbon could be used in conjunction with the TMU to speed up distribution of tasks between different TMUs.

Stavrou *et al.* describes a thread synchronization unit for data-driven multithreading [7] that resembles the presented TMU. Their method is efficient as long as there are more tasks ready for execution than the number of cores. However, the look-ahead technique presented in this paper allows new tasks to be found even in the case where all currently ready tasks are being executed. The look-ahead functionality also allows prefetching to be started earlier for tasks found ready for execution by the TMU.

The synchronization and scheduling unit proposed by Bayer *et al.* [12, 13] is based on a centralized unit and an associated network for task distribution. Centralization of the unit will incur delays as the network scales with more cores. The scheme relies on duplicable tasks that can be packaged into a single package, which is then split into individual tasks and distributed by the network. The TMUs on the other hand, are tightly coupled to each of their cores, thus reducing communication overhead and instead centralize the storage of the task dependency values. The update and check of dependencies are not speed critical for the case when the TMU finds a task to be executed, since the update

and check is decoupled from the execution of the task by the look-ahead technique. Furthermore, the TMU does not rely on duplicable tasks, which are not always present in an application (e.g. the H.264 decoder).

8 Future Work

Future work includes detailing the instruction set architecture and micro architecture of the Task Management Unit (TMU) and to evaluate power and area benefits of offloading control load from conventional cores to the TMUs. Further evaluation is also needed to find the right configuration of the number of cores per TMU for various workloads. An interesting extension to the TMU would be the possibility to perform data prefetching for tasks found ready for execution by the TMU.

9 Conclusions

The Task Management Unit (TMU) outperforms the task pool based multi-core architectures due to its look-ahead capability. 82% of the executed tasks benefited from the look-ahead functionality, thereby reducing the time to compute dependencies for next tasks. Overall, the TMU-based multi-core reaches a speedup of more than 14x on 16 cores running Super HD H.264 video decoding. The overall performance is improved by 4.5% for the eight core configuration. This equals to a 50% reduction of the task management overhead of 9% seen for the task queue simulations. Further, the TMU-based multi-core architecture can execute non-dependent tasks as fast as the task queue architecture.

10 Acknowledgments

We thank the HiPEAC Network of Excellence for financing the internship during which this research was done. We also thank our colleague Jan Hoogerbrugge at NXP for his help with TTISim and the parallelization of the H.264 decoder.

References

[1] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *IEEE International Symposium on Computer Architecture*, June 2007, pp. 734–740.

[2] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture," in *Image and Video Communications and Processing*, 2003, pp. 707–718.

[3] G. de Haan, P. W. A. C. Biezen, H. Huijgen, and O. A. Ojo, "True-motion estimation with 3-D recursive search block matching," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 3, pp. 368–379, October 1993.

[4] T. Wiegand, G. J. Sullivan, G. Bjø:ntegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transaction on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, July 2003.

[5] Apple Inc., "QuickTime HD Gallery System Requirements," http://www.apple.com/quicktime/guide/hd/recommendations.html.

[6] J. Hoogerbrugge and A. Terechko, "A Multithreaded Multicore System for Embedded Media Processing," *To Appear in Transactions on High-Performance Embedded Architectures and Compilers*, 2008.

[7] K. Stavrou, C. Kyriacou, P. Evripidou, and P. Trancoso, "Chip Multiprocessor Based on Data-Driven Multithreading Model," *International Journal of High Performance Systems Architecture*, vol. 1, no. 1, pp. 24–43, 2007.

[8] NXP Semiconductors, "Nexperia Development Kit for TriMedia processors," http://www.nxp.com.

[9] S. Rathnam and G. Slavenburg, "An architectural overview of the programmable multimedia processor, TM-1," in *In Proceedings of Compcon*, 1996, pp. 319–326.

[10] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. van Antwerpen, "The TM3270 Media-Processor," in *IEEE/ACM International Symposium on Microarchitecture*, November 2005, p. 12pp.

[11] J.-W. van de Waerdt, "The TM3270 Media-processor," Ph.D. dissertation, Delft University of Technology, 2006.

[12] N. Bayer and R. Ginosar, "High Flow-Rate Synchronizer/Scheduler Apparatus and Method for Multiprocessors," United States Patent 5 202 987, April, 1993.

[13] P. Avieli, O. Rubenov, and N. Bayer, "Designing a Central Synchronization/Scheduling Unit For Multiprocessors," in *IEEE Convention of the Electrical and Electronic Engineers*, April 2000, pp. 495–498.