# The Limits of TDMA Based Memory Access Scheduling

Jack Whitham
Real-Time Systems Group
Department of Computer Science
University of York, UK
jack@cs.york.ac.uk

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

***Abstract*—In a multicore system, several CPUs frequently require access to a shared memory. If this access is required to be time-predictable to enable worst-case execution time (WCET) analysis of tasks, a form of TDMA based bus arbitration is usually used. A global TDMA schedule controls when each CPU may make use of the bus. This schedule is normally static.**

**It has been suggested that the TDMA schedule could be specialized to reduce the WCET for particular tasks. In this paper, we show that there is a hard limit to the potential of this technique within a general-purpose system. We simulate single-path tasks running within a multitask, multicore system and apply TDMA slot scheduling on the memory access traces. For medium numbers of CPU cores and low memory latencies, CPU utilization can be improved by up to 25%, but as more cores are used and memory latency increases, the bus gets saturated and the difference between a specialized schedule and a generic schedule disappears.**

## I. Introduction

In a typical multicore embedded system, multiple CPU cores share a single memory bank [3]. This is used for storing code and shared data. As only one CPU may access the shared memory at a time, an arbiter is used to resolve conflicts where more than one CPU wishes to begin an access. The shared memory bus is a bottleneck that may limit the overall *utilization* of the CPU cores, so it is important to choose arbitration algorithms that maximize utilization [5]. Furthermore, for *hard real-time* tasks, it is essential to choose arbitration algorithms that are compatible with *worst-case execution time* (WCET) analysis to provide a means to ensure that hard real-time deadlines will always be met [6].

Generally, maximizing CPU utilization while permitting WCET analysis is a challenging problem. To date, two approaches have been taken for multicore systems. Firstly, a pessimistic assumption can be made for each memory access: the arbiter delays every access for the maximum possible time [5]. This is suitable for any system with a fair arbiter, i.e. one that provides every CPU with equal access to the bus. However, the guaranteed utilization is reduced by the assumption of delay. A second approach uses *time-division multiple access* (TDMA) as the arbitration scheme [6], [11]. This makes it possible to accurately estimate the delay that will be enforced by the arbiter. Furthermore, the sizes of the time slots assigned by the arbiter can be reprogrammed to suit the current task set, creating a *task-set specific* TDMA schedule [2], [12].

This paper uses an idealized system model to demonstrate that, even under ideal conditions, specialized TDMA schedules are only useful for relatively small number of CPUs and relatively low memory latencies. It uses realistic memory access patterns derived from benchmark code and compares CPU utilization using generic TDMA schedules, task-set specific schedules and absolute upper bounds. *Single-path programs* are used for the experiments on the grounds that if utilization cannot be improved with single-path code, it also cannot be improved for conventional code.

The structure of this paper is as follows. Related work is summarized in Section II. Section III describes the experimental environment that is used within Section IV to test the effects of various TDMA schedulers on utilization. Section V discusses the results of these experiments and Section VI concludes.

## II. Background

Shared memory is commonly used for inter-task communication, because of all the possible methods of communication, it maps most easily to programming languages. Conventional multi-threaded programming environments assume that memory is shared between threads.

Shared memory introduces the possibility of timing interference between tasks. This occurs when one task affects the execution time of another. A typical scenario occurs when a shared cache is present, but even a shared bus is sufficient to provoke interference. Two tasks cannot access the bus simultaneously. Unless effort is expended to isolate the tasks from such effects, this is highly unpredictable. The classical approach to WCET analysis is not applicable since it assumes no interference from other tasks [9], although some variants assume maximum interference [5].

One way to eliminate shared bus interference is to adopt *time-division multiple access* (TDMA) arbitration, which grants each CPU access to the bus on a fixed, statically predictable schedule. The CPU with access to the bus at time $t$ is determined only by a function of $t$. TDMA arbitration permits classical WCET analysis on multicore systems [7], [8].

The simplest form of TDMA schedule provides bus access to CPUs in a strict rotation, providing a fixed time-slice to each. This could be called a *generic* TDMA schedule. Andrei et al. have published results demonstrating that *task-set specific* TDMA schedules can substantially improve task WCETs [2], [11]. However, such schedules are difficult to generate due to the large search space created by conventional tasks. At present, the optimal schedule for a task set can only be found by exhaustive search, and the search space is vastly increased by data-dependent control flow within the tasks, as control flow determines when memory access operations are issued.

It follows that the search space can be greatly reduced if *single-path programming* is used. Single-path programming was introduced by Puschner as an approach for generating time-predictable code [10], as the WCET of a single-path program is equal to its execution time. Single-path code contains no data-dependent control flow, as control flow is *if-converted* to predicates [1]. All hard real-time tasks are bounded by definition, and therefore all hard real-time tasks can be translated into an equivalent single-path form.

## III. Experimental Environment

An optimal task-set specific TDMA schedule can only be found by exhaustive search in the general case, but the complexity of the search space can be reduced by considering only tasks composed of single-path code. This is an idealized case, but idealized cases are useful when examining theoretical limits, because realistic cases cannot exceed the idealized bound.

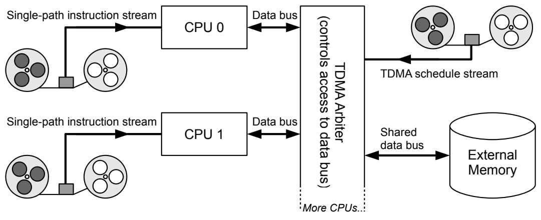
This section describes an experiment to compare generic and task-set specific TDMA arbitration on a system executing single-path code. To date, the only experiment to consider task-set specific arbitration used conventional (multi-path) tasks rather than single-path code [2], and hence made use of heuristics to generate TDMA schedules. In this paper, the use of single-path tasks allows the creation of locally optimal TDMA schedules.

### A. Independent Variables

The variables *num_cpu_cores* and *access_time* define the properties of the architecture under test: the number of CPU cores and the number of clock cycles required to complete a memory access. A third independent variable is the algorithm that is used to generate the task-set specific schedule: these are described in section IV. To avoid dependence on a particular task set, every experiment is repeated with the same collection of 200 randomly generated task sets, each of which is produced as follows:

**Task Set:** A task set consists of 100 tasks. The tasks are assigned priorities in order of size, such that the largest task is executed first.

**Task**: Each task is represented as a linear sequence of instructions. The tasks are derived from the benchmark programs shown in Table I. These benchmarks are well-known programs, widely used in WCET-related research [4]. Although those programs are originally not single-path programs, they are bounded, and can therefore be converted into single-path form by the if-conversion process [1].



Single-path instruction stream
CPU 0
Data bus
Single-path instruction stream
CPU 1
Data bus
TDMA Arbiter (controls access to data bus)
TDMA schedule stream
Shared data bus
External Memory
More CPUs..

Fig. 1. Simulated Architecture.

This paper makes use of execution traces representing the *worst-case execution path* (WCEP) for each benchmark. This is the path that will remain after single-path conversion. As these programs are quite simple, the benchmark creators have been able to force the WCEP to be taken upon every execution [4]. Therefore, the WCEP execution trace can be captured from a single simulated execution. This means there is no need to actually apply the if-conversion process; equivalent results are available from simple execution thanks to the simplicity of the benchmark code.

Within the trace, each executed instruction is classified as "*internal*" or "*memory access*", discarding all other information about its function. *Internal* operations do not make any use of the memory bus, and include arithmetic and control-flow operations. A single-path program may include control-flow operations provided that they are not data-dependent.

The tasks used in the experiment are generated by choosing contiguous slices from these traces. For each task, a benchmark trace is chosen at random from Table I. Then, an array slice containing at most 2000 instructions is taken from that trace. Small task sizes are deliberately used to keep the simulation time reasonable while capturing all relevant properties.

### B. Simulated Architecture

Each task set is executed in a simulator with the architecture as illustrated in Figure 1. A TDMA arbiter controls access to the external memory via a shared data bus. There are *num_cpu_cores* CPUs, each executing single-path code represented as a linear sequence of instructions. An *internal* instruction always requires one clock cycle to execute. A *memory access* instruction also requires one clock cycle, which is always followed by at least *access_time* $- 1$ clock cycles of latency, and a possible *blocking* time if the TDMA arbiter does not immediately give access permission to the CPU. The TDMA arbiter operates according to a linear sequence of scheduling commands, which specify which CPU is permitted to initiate a memory access in each clock cycle. In the simulation an ideal instruction cache and no data caching are assumed. Therefore, accesses to the shared memory occur only when *memory access* instructions are executed.

| Benchmark | Nr. Mem. Access | Nr. Internal | Benchmark | Nr. Mem. Access | Nr. Internal |
|---|---|---|---|---|---|
| adpcm | 1140 | 88704 | fir | 460 | 1741 |
| bs | 7 | 67 | insertsort | 209 | 433 |
| bsort100 | 19901 | 45862 | jfdctint | 320 | 1507 |
| cnt | 405 | 1858 | lcdnum | 20 | 141 |
| compress | 1267 | 1851 | matmult | 26801 | 68413 |
| crc | 1194 | 25646 | ns | 625 | 4919 |
| edn | 10830 | 42810 | nsichneu | 1992 | 3012 |
| fdct | 257 | 1177 | statemate | 455 | 229 |

TABLE I
BENCHMARK PROGRAMS TAKEN FROM [4]. COUNTS FOR *memory access* AND *internal* INSTRUCTIONS ARE DERIVED FROM THE EXECUTION TRACE OF THE MAIN() FUNCTION OF EACH BENCHMARK, CLASSIFIED AS DESCRIBED IN SECTION III-A. EXECUTION TRACES ARE GENERATED BY COMPILING WITH GCC (-O2) FOR THE MICROBLAZE CPU ARCHITECTURE [13], THEN EXECUTING WITHIN A MICROBLAZE SIMULATOR.

### C. Simulation

Simulation is notionally a two-step process that accepts a task set, *access_time* and *num_cpu_cores* as inputs. In the first step, the TDMA schedule is generated. In the second step, tasks are executed with this memory access schedule. Each CPU executes one task from the task set, and when that task finishes, another is removed from the task set according to the priority order until the task set is empty. When all CPUs have run out of instructions to execute, the simulation stops.

### D. Dependent Variable

The simulation determines the processor *utilization* of each simulation. This is the dependent variable of interest. It is the proportion of clock cycles during which a CPU is busy (i.e. not blocked). In this paper, multicore utilization is computed as:

$$U = \frac{\text{number of busy cycles across all CPUs}}{\text{number of clock cycles in simulation}} \tag{1}$$

A CPU is considered *busy* when it is executing an *internal* or *memory access* instruction, and the simulation is considered complete when no tasks remain in the task set. Both types of instruction always require exactly one busy cycle, because the CPU is *not* considered "busy" when it is blocked and waiting for the memory bus. The number of busy cycles is constant for each task set, because the instruction count does not change.

The total number of clock cycles can be reduced by parallelism when more than one CPU core is present. In this way, the utilization can be increased from 1.0 (the value for a single CPU) up to an absolute maximum of *num_cpu_cores*. The effectiveness of parallelism depends on the choice of TDMA schedule. Median utilization values are derived by collecting $U$ for all task sets, sorting them into numerical order, and choosing the midpoint.

## IV. EXPERIMENTS WITH TDMA SCHEDULERS

The choice of TDMA schedule affects the utilization by determining when the tasks get access to the shared memory. To enable experiments with different TDMA schedulers, it is useful to have some notion of the absolute upper bound on utilization.

The following equation for estimating the upper bound will produce an optimistic result that is likely to be greater than

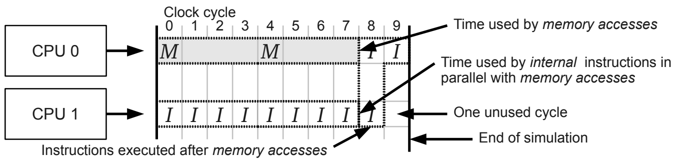

Clock cycle
0 1 2 3 4 5 6 7 8 9
CPU 0
CPU 1
M
M
I
I
I I I I I I I I I
Time used by memory accesses
Time used by internal instructions in parallel with memory accesses
One unused cycle
End of simulation
Instructions executed after memory accesses

Fig. 2. Example to illustrate equation 2. In this example, $I = 11$, $M = 3$, $a = 4$ and $n = 2$. There are $M + I = 13$ busy cycles. Because memory accesses take a total of $Ma = 8$ cycles, 8 *internal* instructions can execute in parallel on the second CPU and 3 others remain. Therefore, a total of 10 clock cycles are required, and $U_{max} = \frac{13}{10} = 1.3$.

the true upper bound, because it assumes that all instructions within the task set can be rescheduled into any order. The parameters are the number of *internal* instructions $I$ and the number of *memory access* instructions $M$ within the task set. *num_cpu_cores* is shortened to $n$ and *access_time* is shortened to $a$:

$$U_{max} = \frac{M + I}{Ma + \lfloor \frac{1}{n} \max(0, I - Ma(n-1) + n - 1) \rfloor} \tag{2}$$

The intuition behind equation 2 is shown in Figure 2. It is the number of busy cycles (a constant, $M + I$, for each task set) divided by the total number of clock cycles. The *memory access* instructions cannot be parallelized, so they always require exactly $Ma$ clock cycles. The *internal* instructions can run in parallel, so they occupy the space alongside the *memory access* operations ($Ma(n-1)$ instructions) and, if that is not sufficient space for $I$ instructions, they overflow into the space after $M$.

Table II shows the idealized utilization across a number of simulations using different values of *access_time* and *num_cpu_cores*. $U_{max}$ is never greater than *num_cpu_cores*, and in general, utilization is always bounded by memory bandwidth (if all *internal* instructions can be executed in parallel with *memory access*) or by *num_cpu_cores*. The latter is seen in Table II wherever $U =$ *num_cpu_cores*.

### A. Generic TDMA Schedules

A static, non-task-set specific TDMA schedule typically falls short of the ideal utilization $U_{max}$, as illustrated by Table III. Where *access_time* and *num_cpu_cores* have large values, static arbitration gets within 10% of the ideal figure, because

| *num_cpu* | *access_time* | | | | | |
|---|---|---|---|---|---|---|
| *_cores* | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 1.00 | 0.82 | 0.69 | 0.60 | 0.53 | 0.48 |
| 2 | 2.00 | 1.64 | 1.39 | 1.14 | 0.91 | 0.76 |
| 3 | 3.00 | 2.28 | 1.52 | 1.14 | 0.91 | 0.76 |
| 4 | 4.00 | 2.28 | 1.52 | 1.14 | 0.91 | 0.76 |
| 5 to 10 | 4.56 | 2.28 | 1.52 | 1.14 | 0.91 | 0.76 |

TABLE II
UPPER BOUND $U_{max}$ FOR SIMULATED ARCHITECTURES: MEDIAN VALUES ACROSS 200 TASK SETS.

| *num_cpu* | *access_time* | | | | | |
|---|---|---|---|---|---|---|
| *_cores* | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 100% | 100% | 100% | 100% | 100% | 100% |
| 2 | 85% | 76% | 71% | 71% | 75% | 81% |
| 3 | 75% | 68% | 78% | 87% | 89% | 91% |
| 4 | 67% | 77% | 87% | 90% | 92% | 93% |
| 5 | 65% | 85% | 89% | 92% | 93% | 94% |
| 6 | 69% | 86% | 90% | 92% | 93% | 94% |
| 7 | 74% | 87% | 91% | 92% | 93% | 93% |
| 8 | 79% | 88% | 91% | 92% | 92% | 93% |
| 9 | 81% | 88% | 90% | 91% | 92% | 92% |
| 10 | 82% | 88% | 90% | 91% | 91% | 91% |

TABLE III
UTILIZATION WITH STATIC TDMA ARBITRATION, EXPRESSED AS A PROPORTION OF $U_{max}$. THE TDMA TIME SLICE IS *access_time*.

the operation of the system is bounded by shared memory contention.

The greatest discrepancies between static TDMA arbitration and the ideal occur with small *access_time* and a medium number of CPUs. A difference of up to 35% is observed versus $U_{max}$ with $num\_cpu\_cores = 5$. This occurs because the CPUs are often still executing internal instructions while the memory bus is available. If the generic TDMA schedule could adapt to the memory requirements of each task, this situation could be avoided.

In order to quantify this effect, the notion of a *memory idle* cycle is introduced. This is any clock cycle where no CPU is waiting to finish a *memory access* instruction. Memory idle cycles may occur for two reasons. Firstly, all CPUs may have *internal* instructions waiting to be executed. Secondly, the TDMA arbiter may have assigned access rights to a CPU that cannot currently use it. Figure 3 illustrates both types of memory idle cycle. Since memory access operations cannot occur in parallel, they set an upper bound on the utilization (equation 2). In many cases, utilization is bounded not by the number of CPUs, but by the quantity of memory accesses (Table II). Therefore, memory idle cycles should be eliminated wherever possible in order to maximize the utilization.

The first type of memory idle cycle can be eliminated in *some* cases by choosing the priority order of memory accesses such that *internal* instructions always occur in parallel with *memory access* instructions. The second type can be eliminated in *all* cases by ensuring that the TDMA arbiter always gives access to a CPU ready for a memory access instruction. Both require the introduction of task-set specific TDMA schedules.

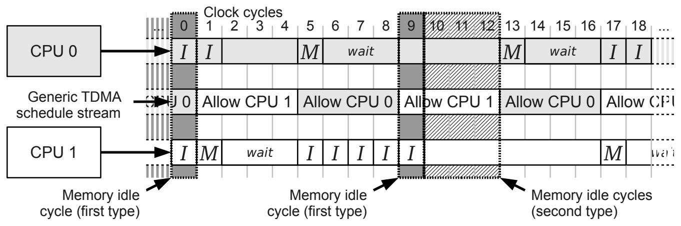


Fig. 3. Memory idle cycles occur whenever the shared memory bus is inactive, either because the CPUs are executing internal instructions (first type) or because the CPU that is permitted to begin a memory access is currently unable to do so (second type).

## B. Greedy Scheduling Algorithm

Memory idle cycles can be significantly reduced by a move to a task-set specific TDMA schedule. Suppose that such a schedule can be of any length. Since the tasks are single-path, it is possible to simulate any form of memory arbitration, including fixed priority, round-robin, or even a fully customized TDMA schedule as proposed by Andrei et al. [2], and it will be easier to search for such a schedule because of the simplicity of the task execution model. The practical disadvantage is that a custom schedule of any length may require additional hardware resources or memory bandwidth, but this is not relevant to a study of the limitations of such techniques.

The results in Table IV were produced using a task-set specific TDMA schedule generated by a *greedy* algorithm that attempts to execute all *memory access* instructions as soon as possible. In order to eliminate the second type of memory idle cycle, the algorithm ensures that *any* CPU executing a *memory access* instruction has a chance of getting access to the shared bus, no matter when the access is initiated. That is, even if CPU 0 currently has priority, CPU 1 can still use the bus if CPU 0 does not need it.

If more than one CPU attempts to execute a *memory access* instruction simultaneously, then a priority order is used to determine which CPU is granted access. Each completed *memory access* causes this priority order to rotate, so that each CPU gets equal time in the high priority state. This scheduling algorithm is greedy because its decisions consider only the current state. It does not perform any sort of lookahead or backtracking, so decisions are only guaranteed to be locally optimal.

The effect is a significant reduction in the number of memory idle cycles (Table V). Table IV shows that the median utilization moves closer to the ideal figures presented in Table II. The largest discrepancy from $U_{max}$ is now observed as just 15%.

## C. Further Improvements

Table IV shows that utilization for small values of *access_time* and *num_cpu_cores* is brought closer to the ideal utilization by task-set specific scheduling. However, a smarter scheduling algorithm may yield further improvements, as memory idle cycles still remain (Table V). If all of these

| *num_cpu* | *access_time* | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *_cores* | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | |
| 1 | 95k | 95k | 92k | 92k | 92k | 92k | 92k | 92k | 92k | 92k | 92k | 92k |
| 2 | 47k | 36k | 41k | 27k | 44k | 20k | 48k | 19k | 44k | 15k | 31k | 13k |
| 3 | 29k | 15k | 26k | 9806 | 19k | 5145 | 13k | 3762 | 18k | 1824 | 14k | 2263 |
| 4 | 20k | 8773 | 16k | 2088 | 11k | 1361 | 11k | 671 | 12k | 259 | 12k | 987 |
| 5 | 13k | 4705 | 8776 | 1599 | 9195 | 648 | 10k | 321 | 8573 | 381 | 9447 | 69 |
| 6 | 12k | 2316 | 8228 | 407 | 7995 | 289 | 7588 | 213 | 10k | 360 | 8292 | 360 |
| 7 | 8504 | 598 | 8611 | 418 | 8662 | 79 | 10k | 123 | 9100 | 47 | 10k | 97 |
| 8 | 7130 | 302 | 8041 | 333 | 8317 | 22 | 10k | 135 | 11k | 51 | 11k | 26 |
| 9 | 6489 | 130 | 8225 | 189 | 5873 | 140 | 8648 | 154 | 14k | 46 | 12k | 152 |
| 10 | 4341 | 238 | 5322 | 212 | 8746 | 100 | 10k | 30 | 13k | 41 | 10k | 9 |

TABLE V
MEDIAN NUMBER OF MEMORY IDLE CYCLES FOR THE STATIC TDMA SCHEDULE (LEFT) AND THE TASK-SET SPECIFIC SCHEDULE FROM THE GREEDY ALGORITHM (RIGHT). THERE IS NO CHANGE FOR *num_cpu_cores* = 1 BECAUSE TDMA ARBITRATION IS UNUSED THERE. FOR ALL OTHER CASES, THERE IS A SIGNIFICANT REDUCTION.

| *num_cpu* | *access_time* | | | | | |
|---|---|---|---|---|---|---|
| *_cores* | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 100% | 100% | 100% | 100% | 100% | 100% |
| 2 | 96% | 90% | 85% | 85% | 89% | 93% |
| 3 | 91% | 84% | 93% | 97% | 98% | 99% |
| 4 | 85% | 94% | 98% | 99% | 100% | 100% |
| 5 | 86% | 98% | 100% | 100% | 100% | 100% |
| 6 | 93% | 99% | 100% | 100% | 100% | 100% |
| 7 | 97% | 100% | 100% | 100% | 100% | 100% |
| 8 | 98% | 100% | 100% | 100% | 100% | 100% |
| 9 | 99% | 100% | 100% | 100% | 100% | 100% |
| 10 | 99% | 100% | 100% | 100% | 100% | 100% |

TABLE IV
UTILIZATION FOR EACH EXPERIMENT WITH A GREEDY SCHEDULE, EXPRESSED AS A PROPORTION OF $U_{max}$.

| *num_cpu* | *access_time* | | | | | |
|---|---|---|---|---|---|---|
| *_cores* | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 100% | 100% | 100% | 100% | 100% | 100% |
| 2 | 97% | 92% | 86% | 86% | 90% | 93% |
| 3 | 93% | 87% | 95% | 98% | 99% | 99% |
| 4 | 88% | 96% | 99% | 100% | 100% | 100% |

TABLE VI
UTILIZATION FOR EACH EXPERIMENT USING THE DEPTH-BOUNDED SEARCH ALGORITHM WITH $n = 6$, EXPRESSED AS A PROPORTION OF $U_{max}$. LARGE VALUES OF *num_cpu_cores* ARE NOT SHOWN HERE AS THE $O(2^n)$-TIME SEARCH PROCESS PROVED TO BE COMPUTATIONALLY INTRACTABLE.

were eliminated, the utilization would reach the practical maximum. The shared memory bus is the bottleneck for the system, and if there are no memory idle cycles, then the system operates as efficiently as it possibly can without reordering instructions. Andrei used a simulated annealing search procedure to determine good TDMA schedules [11]. For single-path tasks, the problem is easier because there is no dynamic control flow. Therefore, a simpler search procedure can be used.

Table VI shows the utilization when the TDMA schedule is determined by a depth-bounded search. This search considers a "decision" to be any occasion where two or more CPUs are both attempting to begin a *memory access*. The search tries out all possible choices for each decision, and will look ahead at up to $n$ future decisions. ($n = 6$ for Table VI.) The choice made is the one that minimizes memory idle cycles, and (in the case of a tie) maximizes the number of executed instructions.

Table VI shows that the degree of improvement from search versus the greedy algorithm is quite poor. The gap between $U_{max}$ and the utilization from the TDMA schedule has narrowed by only a few percent at most. This suggests that Table IV is already very close to the practical maximum available with *any* schedule.

## V. DISCUSSION

The results in section IV confirm that task-set specific TDMA schedules lead to improvements in utilization. For single-path tasks, efficient greedy algorithms can produce good schedules that bring utilization very close to $U_{max}$. For larger *num_cpu_cores* and *access_time*, contention for the memory bus has become the (unavoidable) limiting factor. This is expected, given the well-known *memory bottleneck* phenomenon [3].

However, the findings also illustrate a unintuitive result, which is that task-set specific TDMA schedules are only helpful in reducing the memory bottleneck in a restricted set of special cases. This set is centered upon a medium number of CPUs (4 or 5) and a very low memory latency (*access_time* $= 1$). These cases have the property that CPUs are often still executing internal instructions when the memory bus becomes available. This is less likely to happen when *num_cpu_cores* or *access_time* are increased, because the time between bus availability is also increased within the static TDMA schedule. It is also less likely to happen when *num_cpu_cores* is decreased, because the penalty of missing an opportunity to access memory is less severe. For these special cases, a task-set specific TDMA schedule can boost utilization by around 25% under ideal circumstances. It is no coincidence that the numbers here (4 CPUs, 25%) are approximately equal to the average period between memory access instructions within a task and the inverse of that period respectively.

In general, the benefit of task-set specific TDMA schedules is quite poor. For large *access_time* and *num_cpu_cores*, a specialized schedule can get within 1% of the ideal utilization, whereas a static schedule will only be within 10%.

## VI. Conclusion

In this paper we have evaluated the limits of using task-set specific TDMA schedules for a shared memory bus. Our metric is the combined utilization of the task set. We assumed single-path programs in the evaluation, as their static behavior makes them ideal for TDMA schedule optimization. We found that a greedy algorithm could generate schedules close to the theoretical maximum utilization $U_{max}$, and that depth-bounded searches do not lead to a significant additional improvement.

This paper has highlighted that task-set specific schedules are only worthwhile in the special cases where CPUs are likely to miss their opportunity to access memory. These only occur when the memory latency is very small and the number of CPUs is approximately equal to the average period between memory access operations. However, in general, the benefit of a task-set specific schedule is not very significant. For large *access_time* and *num_cpu_cores*, the improvement in utilization will be less than 10%.

These findings can be stated with confidence because the model used in this paper is idealized. For TDMA memory bus scheduling, single-path tasks enable the use of simple but effective greedy algorithms which come close to the results of depth-bounded search (Table VI). Therefore, the results presented here are close to the practical maximum for *any* task set on *any* system with TDMA arbitration.

## References

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proc. POPL*, pages 177–189, 1983.

[2] Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosen. Predictable implementation of real-time applications on multiprocessor systems on chip. In *Proceedings of the 21st Intl. Conference on VLSI Design*, pages 103–110, Jan. 2008.

[3] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[4] MRTC. WCET Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[5] Marco Paolieri, Eduardo Qui nones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *The 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 57–68, Austin, Texas, USA, 20-24, June 2009. ACM.

[6] Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 115–122, Santa Clara, USA, September 2008. ACM Press.

[7] Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, pages 34–42. IEEE, June 2008.

[8] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

[9] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

[10] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[11] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the Real-Time Systems Symposium (RTSS 2007)*, pages 49–60, Dec. 2007.

[12] Martin Schoeberl and Peter Puschner. Is chip-multiprocessing the end of real-time scheduling? In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.

[13] Xilinx. Microblaze processor reference guide. Manual UG081, Xilinx Corporation, 2005.