Using Petri-Net Modelling to Support the Case for HW-Assisted Task Scheduling

Lucas H. Morais[1], **Alfredo Goldman**[1], **Guido Araujo**[2]

[1]Instituto de Matemática - Universidade de São Paulo (USP)

[2]Instituto de Computação - Universidade Estadual de Campinas (Unicamp)

{moraislh,gold}@ime.usp.br, guido@ic.unicamp.br

Abstract. *Given the pervasiveness of multi-core processors in systems from various domains, the need for efficient parallelization tools has only increased during the last decade. Among the paradigms built to answer this demand, Task Parallelism stands out as a highly productive tool for leveraging data parallelism with minimum code altering. Nonetheless, its current supporting runtimes cannot efficiently execute workloads involving tasks in the fine 1-100us range, limiting its applicability. That said, by performing a thorough Petri-Net-based analysis of task parallel systems with several degrees of HW-assistance, we show that the development of Native CPU support for Task Parallelism is the key for efficiently serving these challenging workloads.*

1. Introduction

Since the dawn of the dark silicon era around 2005, designers have relied more and more on multi-core processor architectures to keep delivering aggregate performance improvements over time [Borkar and Chien 2011]. Even though the switch to multi-core architectures could not fully overcome the challenges posed by the disruption of Dennard Scaling, the benefits of such architectures on power efficiency and computational throughput have paved their way towards becoming the standard for modern processor design. As a matter of fact, multi-core processors can now be found everywhere from low-cost mobile devices to the HPC realm, with 16+ core processors being already available in the desktop market.

Nonetheless, if software engineers from the golden age of Dennard Scaling could always hope for their programs to get automatic performance improvements at the release of every new processor generation, single-threaded performance increase rate has gone down considerably since then [Borkar and Chien 2011]. In order to leverage the still increasing computing power of current processors, programmers are now demanded to exploit their multiple cores, vectorization features, superscalar pipelines, etc., in ways that may not be trivial [Asanovic et al. 2009]. In fact, the sole task of developing and maintaining programs capable of benefiting from all cores available on a system may be very daunting, given that:

1. Constructs provided by some parallel programming frameworks can obfuscate program logic and require extensive code reorganization.
2. The concurrency inherent to parallel software makes it harder do debug.
3. Some kinds of computations - like MD5 computation, LZMA compression, etc - are inherently serial, limiting parallelization or requiring algorithmic relaxation.

Among the various existing parallelization paradigms, *Task Parallelism* stands out as a generic and very productive tool for solving the first of these issues for programs from a great variety of domains [Gupta and Sohi 2011]. On the other hand, current software support for task parallelism is not able to efficiently handle task parallel workloads involving tasks in the small 1-100us size range [Kumar et al. 2007, Meenderinck and Juurlink 2010]. That being the case, by modelling systems with several degrees of HW-support for this paradigm, our work shows that adding native CPU support for Task Parallelism could enable even these fine-grained task workloads to be efficiently executed. This finding suggests that the Task Parallelism paradigm should boast even broader applicability if proper HW-support is provided.

This paper is organized as follows: in Background and Related Work (2), we give a brief overview of the Task Parallelism paradigm, define a three-class taxonomy of Task Scheduling Systems that should be useful for understanding prior work on this area and also define some relevant terminology; in Evaluating the Limitations of Current Task Scheduling Systems (3) we analyze the shortcomings of the current Task Scheduling Systems that do not rely on any HW-assistance for Task Scheduling; in The Case for Native Task Scheduling (4), by modelling Task Scheduling Systems with Petri Nets, we show how adding native support for Task Parallelism to CPUs could dramatically improve the performance of fine-grained task workloads with respect to all systems displaying a lower degree of HW-assistance; at last, in Conclusion (5), we draw our final remarks.

2. Background and Related Work

Task Parallelism frameworks allow programmers to require a runtime system to execute certain basic code blocks or function calls in an asynchronous manner. The code regions allowed to be run in such way are called *tasks*. By specifying how these tasks depend on variables available at the moment they were created, the programmer lets the runtime to automatically infer, at execution time, the dependence relationships between these tasks and run them in parallel. Support for this paradigm is currently offered by several frameworks, such as OpenMP 4.0, Intel TBB, Intel Cilk, BSC's OmpSS, etc.

The performance limitations of Software-based Task Scheduling Runtimes led to the development of several Task Scheduling Systems that, by incorporating some degree of hardware acceleration, allowed for improved scheduling performance. This section thus begins by presenting a three-class taxonomy that groups relevant Task Scheduling Systems from the literature according to their degree of HW-assistance for scheduling computation, while also highlighting the main shortcomings and advantages of each organization. Afterwards, we finish the section by displaying a glossary of relevant terms related to Task Scheduling.

2.1. Software-based Task Scheduling (SW-TS)

In a Software-based Task Scheduling system, task dependence inference is performed by a CPU-based software runtime, of which the Intel OpenMP Runtime [Intel 2013] and the GNU OpenMP Runtime [GNU 2013] are examples.

This is the simplest Task Scheduling configuration, since it only depends on the availability of a software runtime. Nonetheless, as the analysis of Section 3 shall demonstrate, the performance of these systems is severely degraded when they are used to serve

task applications generating fine-granularity tasks - that is, tasks with execution times in the range from 1 to 100us.

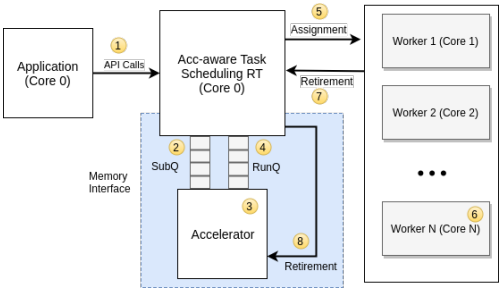

Fig. 1. ACC-TS organization

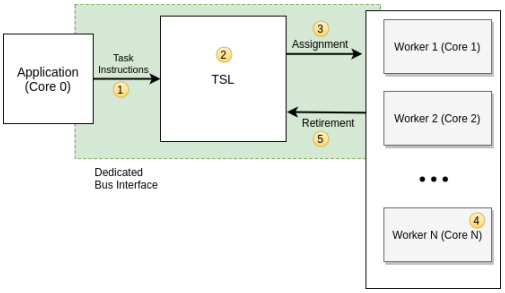

Fig. 2. Native-TS organization

2.2. Accelerator-based Task Scheduling (ACC-TS)

Accelerator-based Task Scheduling systems aim to improve Task Scheduling performance by implementing several scheduling actions in an FPGA-based accelerator, which interacts with task applications through the API provided by a lightweight SW Runtime [Yazdanpanah et al. 2015, Dallou et al. 2013, Wang et al. 2013, Bamnote and Nerkar 2015, Dallou et al. 2016]. Such organization is depicted in Fig. 1.

For these systems, maximum task throughput is expected to exceed that of SW-TS systems. Yet, as it is usual for computation-offloading systems [Vuduc et al. 2010], their performance is heavily dependent on the characteristics of the data paths connecting the CPU hosting the runtime and the FPGA holding the accelerator. Such systems excel at serving workloads related to denser task dependence graphs, which are more demanding of dependence resolution, but display poor performance for workloads related to sparse task dependence graphs, due to the significant cost of CPU-FPGA communication.

2.3. Native Task Scheduling (Native-TS)

In Native Task Scheduling systems, HW-support for Task Parallelism is provided by the processor itself, which makes it available as an ISA extension. Tasking constructs of high-level task parallel programs may thus be translated to tasking instructions during compilation time. In such systems, since Task Scheduling Logic dwells within the processor itself, task scheduling actions do not involve the traversal of data through data paths external to the CPU, effectively eliminating the main bottleneck of ACC-TS systems.

In the diagram of Fig. 2, we see how several elements involved in the execution of a task parallel program interact with each other in such a system. The application generating tasks engages the CPU's Task Scheduling Logic (TSL) through task-creation instructions. The TSL is then left responsible for (1) assigning tasks to available cores in agreement with the restrictions imposed by the inferred task dependence graph and (2) taking notice of cores signaling the conclusion of their tasks, triggering a task graph update and potentially making more tasks ready for assignment. The figure also depicts the fast dedicated bus interface that connects all those elements - the Application, the TSL and the cores running tasks (worker cores) - within the processor.

Given that the execution of task parallel programs by such systems does not require interaction with any software runtime, not only RT-ACC communication overheads

are eliminated, but also all overheads that arise from the communication between the application and such a runtime through its software API.

Although some works on ACC-TS systems have pointed before to the benefits that solutions like this could bring [Dallou and Engelhardt 2015], to the best of our knowledge, no Native-TS system has yet been implemented.

2.4. Relevant definitions

Task granularity refers to the mean execution time of a task. High granularity corresponds to long execution times, while low granularity relates to short execution times.

Task retirement is the action by which a core finishing the execution of some task informs the Task Scheduling Runtime about its conclusion. Once a task is retired, its corresponding node is removed from the task dependence graph and new tasks may become ready to be executed.

Task submission is the action by which the task parallel program creates tasks and sends their descriptors to a Task Scheduling Runtime.

In-flight task is a task that is currently referenced by the Task Dependence Graph - that is, a task that has already been submitted but that has not yet been retired.

Ready task is a task T such that there is no in-flight task T' such that T depends on T'.

Task assignment is the action by which the Task Scheduling Runtime assigns a ready task to an available processor, which immediately starts to execute it.

Task allocation is the action by which the Task Scheduling Runtime allocates enough space in memory for holding a single task metadata structure.

Run queue is a software queue for holding descriptors of ready tasks.

3. Evaluating the Limitations of Current Task Scheduling Systems

This section aims to assess the performance limitations of current Task Scheduling systems. In order to do so, we devise performance metrics (Sec. 3.1) that allow us to formulate performance upper bounds (Sec. 3.2) for systems of such kind. Once in place, we apply such tools to the analysis of a representative SW-TS Runtime (Sec. 3.3), leading us to conclusions about the intrinsic limitations of the SW-TS organization (Sec. 3.4).

3.1. Performance metrics

Mean Overhead Time (Eq. 1) is the sum of all average scheduling overheads related to the processing of a single task.

Maximum Dispatch Rate (Eq. 2) is the maximum task completion rate that the whole set of worker cores would be able to achieve if no scheduling overheads existed.

Mean In-flight Time (Eq. 3) is the average lifetime of the tasks generated by an application, considering both mean execution time and scheduling overheads.

Maximum Task Throughput (MTT) (Eq. 4) is the maximum number of tasks that a Task Scheduling system may be able to retire during an interval of time.

Overhead ratio (Eq. 5) is the mean fraction of in-flight time due to scheduling overhead.

$$T_{ovh} = T_{addition} + T_{RunQDeq} + T_{alloc} + T_{retirement} \quad (1) \qquad MDR = \frac{N_{wcores}}{T_{exec}} \quad (2)$$

$$T_{inflight} = T_{ovh} + T_{exec} \quad (3) \qquad MTT = \frac{1}{T_{ovh}} \quad (4) \qquad O_{ratio} = \frac{T_{ovh}}{T_{inflight}} \quad (5)$$

3.2. Devising Theoretical Limits to Application Speedup

3.2.1. From Overhead Ratio

Let us consider a task scheduling system with N cores for which an Overhead Ratio equal to O_{ratio} was measured for a certain workload W. In such situation, *assuming that all cores may perform scheduling actions* and given that W has enough intrinsic parallelism not to let cores starve for work, the maximum program speedup is attained when all cores are fully utilized. In such optmal case, any core has a chance of O_{ratio} of being, at a certain moment, performing scheduling actions. That being the case, the average number of cores that will actually be executing tasks will be:

$$N_{executing-tasks} = (1 - O_{ratio}) \times N, \tag{6}$$

which corresponds to the maximum application speedup MS_{ratio} for that scenario.

3.2.2. From MTT

Let us consider a task scheduling system with N cores for which an MTT of K was measured for a certain workload W. Tasks contained in W display mean task execution time equal to T_{exec}. Assuming that in this case *only one core is allowed to perform scheduling actions*, the Task Scheduling Runtime may not serve more than K tasks per unit of time. If we then assume that worker cores are able to keep running tasks without any scheduling overhead, each of these cores should be able to run as much as T_{exec}^{-1} tasks per second. From these assumptions, Ineq. 7 can be derived, leading to the application speedup upper bound of Eq. 8.

$$\frac{N_{executing-tasks}}{T_{exec}} \leq K, \tag{7}$$

$$MS_{MTT} = K \times T_{exec} \tag{8}$$

3.3. Evaluating SW-TS overheads

In this subsection, we instrument a software Task Scheduling Runtime running on a dual-core x86 machine to analyze its performance while serving several scheduling-related actions: (1) processing newly created tasks, which demands a node addition to the task graph; (2) task allocation; (3) task retirement, which demands a node to be removed from the task graph; (4) run-queue queuing.

For simplicity, the Task Scheduling Runtime of choice was MTSP [César 2016], a light-weight open-source implementation of the tasking provisions of OpenMP 4.0. Similar work could be done with the Intel OpenMP Runtime or with the GNU OpenMP Runtime. Experiments were performed on an Intel Core i5-3230M dual-core CPU @2.60GHz, with 16GB of DDR3 RAM running on Ubuntu 16.04.

In order to instrument MTSP's code, Intel's RDTSC and RDTSCP cycle-counting instructions in GCC Asm were used to enclose code fragments for each scheduling-related action of interest. The instrumented runtime was then evaluated while serving two task-parallel applications from the Kastors Benchmark Suite [Virouleau et al. 2014]. Each measurement value is the average resulting from ten experiment replications.

Task Granularity	Jacobi		SparseLU	
	E. Params	Cycles	E. Params	Cycles
small	-n 8192 -b 64	8.0e+03	-n 128 -m 64	9.4e+02
medium	-n 1024 -b 64	6.9e+04	-n 128 -m 16	2.3e+04
large	-n 128 -b 64	3.8e+05	-n 128 -m 4	1.5e+06

Table 1. Execution parameters for each benchmark

The overhead ratios measured for all six experiments according to Eq. 5 are summarized in Fig. 3. We can see that for fine-granularity workloads - for which, according to Table 1, mean task execution time is not higher than 8K cycles - overhead ratios are over 80%. On the other hand, as task granularity increases to about 380K cycles, overhead ratios become lower than 10% for both test programs.

These findings support our hypothesis that, although current task scheduling systems may be fast enough to serve coarse-granularity task workloads, they fail at serving applications generating smaller tasks, which is indicated by the high overhead ratios measured for both the *Small* and *Medium* workloads.

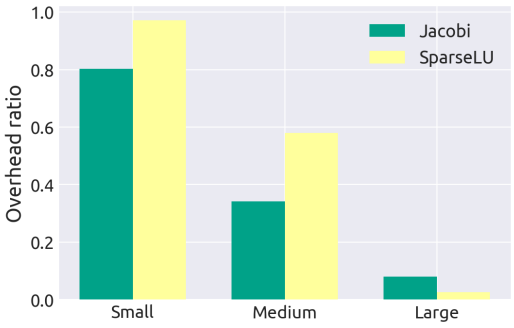

Fig. 3. Relationship between task granularity and overhead ratio

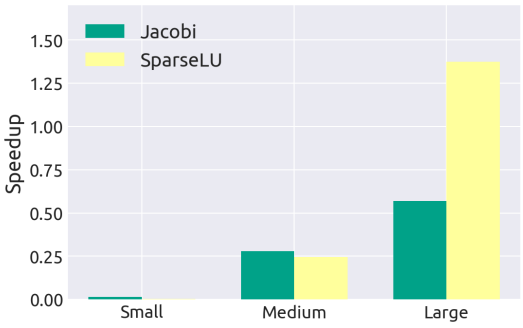

Fig. 4. Relationship between task granularity and program speedup

Fig. 5 showcases the components of scheduling overheads for all six experiments. The amount of overhead cycles reported for each (scheduling action, experiment) pair is an average value calculated from the set of all tasks produced by the experiment. This breakdown shows that actions related to task graph management (addition and retirement) are the most taxing, while allocation and run-queue dequeuing never take more than 10% and 5% of total per-task overhead, respectively. Such figure also suggests that both (1) relative weights of scheduling latency components and (2) mean total per-task overhead do not vary much depending on task granularity or task graph topology, since they remain very similar across all six experiments. One major implication of this is that one should not expect total per-task overhead to be lower than about 30K cycles for any task parallel application with data dependencies, regardless of its mean task granularity.

The speedup graph of Fig. 4 shows that application speedup will usually improve as workload granularity increases, going up to 1.37x for SparseLU-large (T_{exec} of about 1.5×10^6 cycles), the most coarse-grained workload. Nonetheless, slowdowns as high

Fig. 5. Overhead components for each experiment

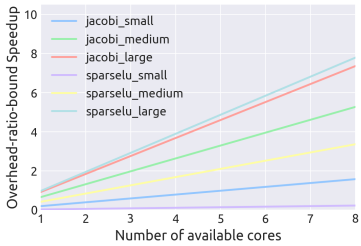

Fig. 6. Overhead-ratio-driven Speedup Bounds

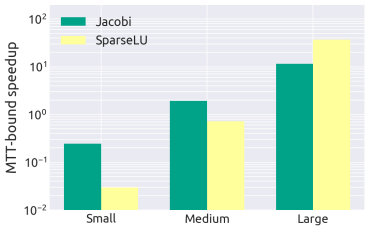

Fig. 7. MTT-driven Speedup Bounds

as 100x for Jacobi-small (T_{exec} of about 8.0×10^3 cycles) and 200x for SparseLU-small (T_{exec} of about 9.4×10^2 cycles) were also measured, illustrating the inability of SW-TS systems to handle fine-grained workloads.

Moreover, as shown by the optimistic theoretical speedup upper-bounds of Fig. 6 - which considers a perfectly parallelized TS Runtime - supposing a system with 8 cores, the optimal speedup figures for the fine-grained workloads would not be higher than 1.6 for Jacobi-small and 0.2 for SparseLU-small. By its turn, the more realistic evaluation of Fig. 7 - which supposes a serial TS Runtime - shows that even if an unbounded number of cores was available, one would not be able to achieve speedups higher that 0.25 and 0.029 for Jacobi-small and SparseLU-small, respectively.

3.4. Conclusions about SW-TS performance

From the data just showed and analyzed, one can come to the following conclusions regarding SW-TS performance:

1. SW-TS systems perform best for task applications generating tasks larger than about 1×10^6 cycles, which take around 500us in a 2GHz processor. For finer workloads, the performance of the parallel program may be considerably worse than that of its serial counterpart.
2. Mean total per-task overhead does not vary much depending on task granularity for task applications generating tasks with data dependencies.
3. The most computationally taxing actions performed by a SW-TS RT are those related to task graph management - namely, addition and retirement.

4. The Case for Native Task Scheduling

The theoretical analysis of Section 3 demonstrated the limitations of current Task Scheduling Systems relying on software runtimes. This section, in turn, aims to show how these shortcomings may be avoided by the introduction of enough hardware acceleration. Moreover, we show that while ACC-TS systems from the literature already displayed substantial gains with respect to organizations without any HW-assistance, Native Task Scheduling should represent a major step forward from this organization, since it circumvents its communication-related bottlenecks.

This section hence starts with the description of our proposed Petri Net Model for Task Scheduling Systems and finishes with the evaluation of the three organizations from

Section 2 according to this model, from which we can gather conclusions regarding their relative performance and their intrinsic limitations.

4.1. Modelling Task Scheduling Systems with Petri Nets

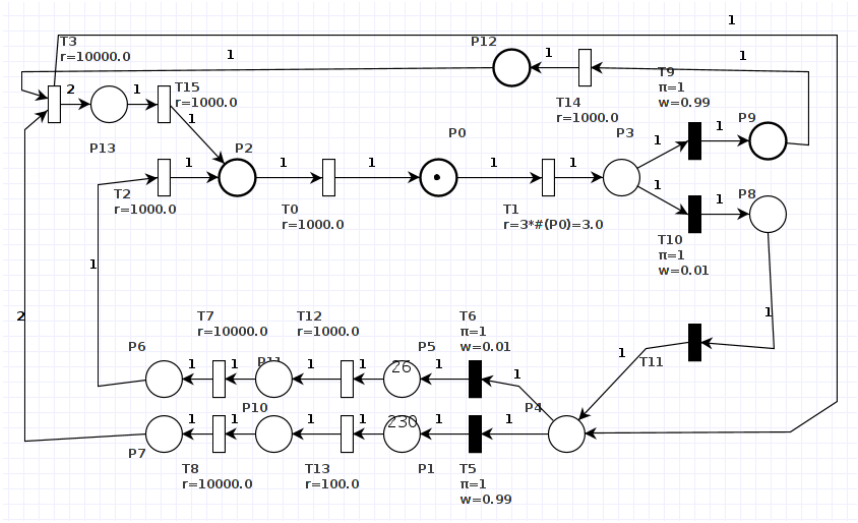

Fig. 8. ACC-TS Petri Net model

In this section, we present a Petri Net model for an ACC-TS system with realistic characteristics, which may be used to evaluate system bottlenecks and resource utilization as well as to predict application speedup. We start by discussing the reasons that lead us to organize the net topology and set the transitions parameters as they are. Then, we show how one can use the same net for simulating SW and Native TS systems by setting its transition rates accordingly. All experiments involving Petri Nets were performed using the PIPE4 simulation tool [Dingle et al. 2009], and the developed models can be found at a public Git [1]. Table 2 describes the parameters that can be varied across simulations.

High-level simulation feature	Dependent features
System characteristics (SW, ACC, Native)	System organization (SW, ACC, Native)
	number of CPU cores; timing characteristics
	presence/absence of communication queues
Workload characteristics	Task granularity; parallelism degree
	percentage of dependence-less tasks

Table 2. Tunable simulation features.

4.1.1. System topology analysis

Since their generation at the task parallel code, tasks go through the various sub-systems of a HW-accelerated Task Scheduling system according to the flow depicted in the system block diagram of Fig. 1. In the following lines, we show how each of the indicated steps were mapped to place-transition combinations in our model.

[1] https://bitbucket.org/lucashmorais/task-scheduling-petri-net-modeling

(1-2) Task-creating API calls and writes to the Submission Queue are modeled by places {P1, P5, P10, P11} and transitions {T5, T6, T12, T13}. The number of marks in P5 represent the number of dependence-less tasks that the system was required to schedule, while the number of marks in P1 represent the number of tasks with dependencies that the application would like the system to manage. Additionally, the number of marks in P11 represents the number of task descriptors of dependence-less tasks that have already been enqueued to the Submission Queue. Analogously, the number of marks in P10 encodes the number of tasks with dependencies that the system may already start to process. The transition weights of T5 and T6 define the simulated workload ratio between dependence-free and dependence-full tasks, while the transition rates of T13 and T12 define the simulated rate at which the RT may fill the Submission Queues with task descriptors for tasks with or without data dependencies, respectively.

(3) Task dependence inference at the accelerator is represented in our model by places {P6, P7} and transitions {T7, T8}. The number of marks in P6 represent the number of dependence-less tasks on the task graph, while P7, the number of dependence-full tasks on the graph. The transition rate for T7 represents the rate at which the accelerator may add dependence-less tasks to the graph, while the rate for T8, the rate at which it may process dependence-full tasks.

(4) Writing to the Runnable Queue is represented in our model by P2 and the transitions T2 and T15. The number of marks in P2 represents the number of tasks on the Runnable Queue, while the identical rates of T2 and T15 represent the rate at which the queue may be written to.

(5) Assignment of tasks to the processor cores is represented by P0 and T0. The number of marks in P0 represent the number of processors that are currently running a task. The rate for T0 represents the rate at which task descriptors may be read from the Runnable Queue and assigned to an available core.

(6) Task execution is represented by transition T1. Its rate represents the rate at which tasks are executed, which is a function of the number of currently active cores and of task granularity. Such relationship is depicted in Fig. 8 below T1.

(7-8) Task retirement is modeled by places {P3, P8, P9, P12, P13} and transitions {T1, T3, T9, T10, T14}. The number of marks in P9 represent the number of tasks on which other tasks depended that have recently finished executing, while the number of marks on P8 represent the number of tasks on which no other task depended that were recently finished. The number of marks in P12 represent the number of task IDs that have been recently written to a HW-register that receives IDs of retiring tasks. The rate for T14 represents the rate at which packets may be written to such register. The rate for T3 represents the rate at which the system may detect two tasks to be ready to execute as a result of the acknowledgement that one more task has finished executing.

4.1.2. Reasoning behind used rates and place occupancy restrictions

All rate parameters of our Petri Net model may be changed for better reflecting its target (TS System, workload) pair. In the specific configuration depicted in Fig. 9, we consider that tasks are $10\text{K}/3 \approx 3.3\text{K}$ cycles long. The amount of cycles NC_{action} related to each

of the remaining modeled actions may be calculated with the following formula:

$$NC_{action} = \frac{10000}{R_{action}}, \tag{9}$$

where R_{action} is the rate of the related timed transition.

The rates for transitions representing memory operations correspond to reasonable memory access latencies for modern x86 machines. The SW and ACC models abstract several sources of overheads that would make their actual implementations less efficient, while pessimistic latency values are set for the Native-TS model. Furthermore, we even consider the Runtimes behind SW and ACC systems to be able to perform several different scheduling actions in concurrent fashion, which, although desirable, would be difficult to implement for real systems. By doing so, our comparison between the Native and the other configurations is made more favorable towards the latter ones, making any perceived advantages of the Native TS system more reliable. The weights for the instantaneous transitions, on their turn, are set in such a way that a workload comprising 99% of tasks with dependencies and 1% of dependence-less tasks was simulated. Such ratio between the two task classes correspond to a very taxing task parallel workload, which goes in line with our goal of comparing the performance of the three system organizations under the most stressful circumstances. Finally, we set the limit on the number of marks on P0 to 64, which goes to represent that our system has a 64-cores CPU.

4.2. Models for SW and Native systems

We now showcase two petri nets that may be used to simulate the execution of the workload of Fig. 9 by SW and Native task scheduling systems. They share the same topology of the former model, differing only by the rates of some of their timed transitions.

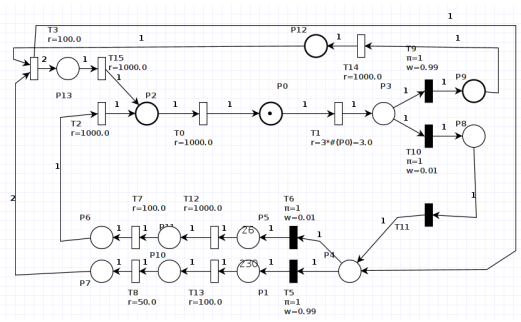

Fig. 9. SW-TS model

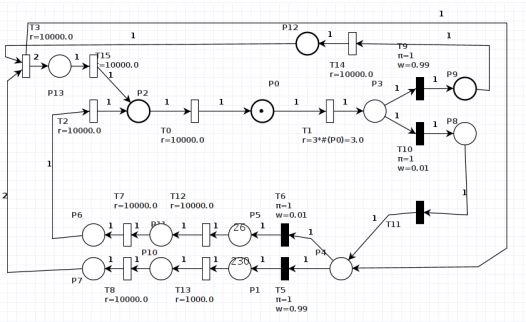

Fig. 10. Native-TS model

4.3. Experimental evaluation

We now present information regarding the number of cores that each of these solutions was found to be able to feed, as well as the subsystems that we discovered to be the bottlenecks of each organization. Results for both analysis are in Table 3.

Organization	TCO	System Bottleneck	C. Places
SW	17	Task-scheduling kernel	P11
ACC	33	SW-Queues Comm. interface	P1, P13, P12
Native	64	Limited number of CPU cores	P7, P0, P13, P2

Table 3. Typical Core Occupancy, System Bottlenecks and Congested Places for each organization

As discussed in Section 4.1.1, the number of cores currently running tasks during simulation relates to the number of marks in P0. Hence, the Typical Core Occupancy (TCO) of each scenario corresponds to most frequent value of marks of P0 after steady state has been achieved. Such metric represents a close upper-bound for program speedup.

System bottlenecks were evaluated by identifying, for each simulation model, the modelled system elements corresponding to places with highest typical occupation.

The conducted set of experiments suggests that Native-TS systems should bring a large leap in performance with respect to SW and ACC systems. In fact, the simulated ACC system was only able to increase TCO with respect to the SW system by a factor of 1.9 times, while the Native system was able to effectively saturate the available processor cores with work. These findings agree with our expectations that, by (1) circumventing the high overhead ratios of SW-TS systems and (2) avoiding the communication latencies of the FPGA-CPU buses and the queue interfaces of ACC-TS systems, the Native-TS organization should enable significantly higher performance.

Similar conclusions can be derived from experiments involving a higher ratio between dependence-less and dependence-full tasks (not depicted). These scenarios lead to significantly better performance for SW-TS systems. Concretely, if the dependence-less/dependence-full task ratio is increased to 50/50, Typical Core Occupation rises by about 50% for the latter organization. All the same, the performance of other configurations is not considerably altered by this change - given that the bottlenecks of ACC and Native TS systems are not related to slow task-graph management - and the SW-based systems keep their place at the bottom of the performance scale.

5. Conclusion

In this work, we have shown that the development of Native CPU support for Task Parallelism should enable efficient execution of task parallel workloads comprising tasks on the fine 1-100us size-range, for which existing task scheduling systems have presented only poor results. In order to do so, we developed a parametric Petri Net model and theoretical upper bounds for predicting performance characteristics of Task Scheduling systems under arbitrary workloads. With that, we believe to have motivated further research on Native Task Scheduling Systems - that is, systems where full support for Task Parallelism is provided by the CPU itself - for realizing the potential of the Task Parallelism Paradigm as not only a very productive, but also as a highly efficient parallelization tool.

6. Acknowledgements

The authors thank the support of FAPESP (2017/02682-2) for this research undertaking.

References

Asanovic, K., Wawrzynek, J., and Patterson, D. (2009). A view of the parallel computing landscape. In *Communications of the ACM*, pages 56–67.

Bamnote, R. and Nerkar, R. P. (2015). Review on Dynamic Task Scheduling to Support OoO Execution in an MPSoC Environment. In *Int'l Journal of Computer Applications*.

Borkar, S. and Chien, A. (2011). The Future of Microprocessors. In *Comm. of the ACM*.

César, D. (2016). MTSP: Multicore Task Scheduling Platform. https://bitbucket.org/lgeunicamp/mtsp/.

Dallou, T., Elhossini, A., and Juurlink, B. (2013). FPGA-Based Prototype of Nexus++ Task Manager. In *6th Wksh. on Many-Task Computing on Clouds Grids and Supercomputers*.

Dallou, T. and Engelhardt, N. (2015). Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models. In *IEEE 29th Int'l Parallel and Distributed Processing Symposium (IPDPS)*.

Dallou, T., Lucas, D. S., Araujo, G., Morais, L., Frank, M., and Ferreira, E. (2016). Task Parallel Programming Model + Hardware Acceleration = Performance Advantage (poster). In *Hot Chips*.

Dingle, N. J., Knottenbelt, W., and Suto, T. (2009). PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets. In *ACM SIGMETRICS Performance Evaluation Review*, pages 34–39.

GNU (2013). An OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp.

Gupta, G. and Sohi, G. (2011). Dataflow Execution of Sequential Imperative Programs on Multicore Architectures. In *Proc. 44th IEEE/ACM Int'l Symp. on Microarchitecture*.

Intel (2013). Intel OpenMP Runtime Library. https://www.openmprtl.org.

Kumar, S., Hughes, C., and Nguyen, A. (2007). Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. 34th annual Int'l Symp. on Computer architecture*.

Meenderinck, C. and Juurlink, B. (2010). A Case for Hardware Task Management Support for the StarSS Programming Model. In *Proc. 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*.

Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., and Gautier, T. (2014). Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *10th Int'l Workshop on OpenMP*, pages 16 – 29.

Vuduc, R., Chandramowlishwaran, A., and Choi, J. (2010). On the Limits of GPU Acceleration. In *Proc. 2nd USENIX conference on Hot topics in parallelism*, page 13.

Wang, C., Li, X., and Zhang, J. (2013). MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs. In *ACM Transactions on Architecture and Code Optimization (TACO)*, pages 1–9.

Yazdanpanah, F., Álvarez, C., and Jiménez-González, D. (2015). Picos: A hardware runtime architecture support for OmpSs. In *Future Generation Computer Systems*.