Considering Quality-of-Service for Resource Reduction using OpenMP

Artur Podobas[1], Mats Brorsson[1], Vladimir Vlassov[1], Chi Ching Chi[2], and Ben Juurlink[2]

[1] KTH - Royal Institute of Technology , Stockholm, Sweden
[2] TUB - Technische Universität Berlin , Berlin, Germany

Abstract. Not caring about resources means wasting them. Current task-based parallel models such as Cilk or OpenMP care only about execution performance regardless of the actual application resource *needs*; this can lead to over-consumption resulting in resource waste.We present a technique to overcome the resource un-awareness by extending the programming model and run-time system to dynamically adapt the allocated resources to reflect the expected *Quality-of-Service* of the application. We show that by considering tasks' timing constraints and the expected quality-of-service in terms of real-time behavior, one can reduce the number of resources and temperature compared to a greedy work-stealing scheduler. Our implementation uses a feedback controller that continuously samples the application-experienced service and dynamically adjusts the number of resources to match the quality required by the application.

1 Introduction

General purpose processors typically have two to eight cores (multicore) and this core count is increasing. The increase in core count is motivated by power and temperature issues [11]. Prior to multicore, processors had only a single core where performance came from increased clock frequency. Higher clock frequency leads to higher temperature known to cause soft- and hard-errors [21], which decrease the mean time to failure (MTTF). To gain performance one must now use several cores which leads to the need to program in parallel.

However, increased core count does not mean that an application can effectively *utilize* all the cores.

Modern parallel programming models such as OpenMP and Cilk [9] support different number of cores without change in the program. Typically cores are allocated at startup and the allocation remain static throughout execution[3]. To avoid resource waste some models uses fast user-space mutexes (`futex`) or OS system calls (`usleep`) to suspend threads (libGomp in GNU C). These *reactive* and sporadic methods to conserve resources work when applications are executed in isolation, something that is often not the case in a general purpose

[3] Although OpenMP allows for different core allocations between different parallel regions, this is rarely done in practice.

environment. The future will require the parallel run-time system to interact with the OS [1], where the OS expects each user-level scheduler to estimate the amount resources it needs and request them from a *system scheduler* [22] or CPU broker [7, 17].

Our problem is defined as: If we know the required Quality-of-Service of the application, can we reduce the amount of resources used while still satisfying the application? A solution would reduce both resource waste and power-consumpion[4]. Solving this problem would also reduce the temperature.

For the purpose of the discussion in this paper, a resource is a thread; each thread is a proxy for a core or a hyperthread. We solved the problem by adding Quality-of-Service (QoS) to the OpenMP tasking model; these QoS extentions allows the user to put timing constraints (soft-real time) on tasks. Our run-time system uses tasks' timing information to dynamically adjust resources to *match* the that required by the application; a **PID** feedback controller connects the history of the application (in terms of experienced quality) to the resource history and tunes the resources to reduce waste. We demonstrate our method on a H.264 capable decoder and a Volumetric RayCaster for medical imaging. Our techniques work on both fork/join- and data-flow-type of parallelism.

Related work has considered OpenMP (and similar) models as platforms for real-time computing, often focused on hard real-time. Ferry et al. [8] developed RT-OpenMP which schedules periodic data-parallel for-loops with hard-real time constraints on multicore systems. Laksmanan et al. [15] focused on theoretical properties of hard real-time *tasks* within OpenMP using the fork/join programming model. Nogueira et. al [19] proposed two different EDF strategies for implementing real-time features in a work-stealer. Our work differs since we focus on resource-usage and QoS in task-based models rather than meeting hard real-time tasks. Furthermore, we allow data-flow computing and are not restricted to only fork/join.

2 A Motivating Example

We motivate our work with the H.264 video decoder [2] which require a form of QoS. The application is programmed with OmpSs [5], a state-of-the-art run-time system that supports *data-flow* parallelism. With version 4.0 data-flow synchronization between tasks was also introduced in OpenMP.

For a video decoder, the QoS can be expressed as a certain desired frame rate per second (FPS). The computational requirement to achieve a specific frame rate varies, not only with the frame rate, but also with the contents of the video. To achieve the desired frame rate, one solution is to allocate all- we over-supply the cores - available cores to the application.

For a particular movie with QoS target of 240 FPS we achieved 301 FPS when the program was free running on a test system with four cores, eight threads thus using all available resources. However, this solution is not very

[4] Note, power-consumption and not necessarily the energy consumed by the application

resource friendly; the application can probably achieve the frame rate with fewer resources. We can manually try out the required number of cores until we have the smallest number of cores that satisfies 240 FPS. In this case, we found that the application required somewhere between three and four cores for 240 FPS. Doing this manually for all possible workloads and target platforms is not a portable solution. This situation motivated us to *dynamically* allocate and release resources on-line in the run-time system.

3 Programming mode extensions

OpenMP is a directive-driven parallel programming model. The programmer annotates source code to expose explicit parallelism using directives. We have focused on the `task` directive in OpenMP which exposes asynchronous work to be scheduled onto worker threads in the system. The general syntax of the OpenMP task construct is shown below. The directive starts using `omp task` followed by clauses that further defines the properties of the task:

```
#pragma omp task [clause1, clause2, ...]
```

Listing 1.1. OpenMP general task format

Our work works well with both *fork-join* and *data-flow* parallelism. *Data-flow* is easier to use and often yields a high parallel *span* (in for example Wavefronts [18]); it has been embraced by several task-based run-time systems [5, 20, 10]

We added three new OpenMP clauses to allow timing constraints on tasks: `deadline`, `onerror` and `release_after`. The `deadline`*(time)* clause specifies a timing constraint on the task. It requests that the task *must not* start after the specified time. The argument *time* is related to OpenMP's `omp_get_wtime()` that returns the current time in the application (in seconds). The `release_after`*(time)* clause is similar to the `deadline`-clause except it sets a timing requirement *when* a task is allowed to start. This is useful to ensure that tasks do not execute before a certain time, such as displaying frames in a movie decoder. The `onerror`*(prop)* clause conveys information regarding the *drop-ability* of the task. Tasks can be dropped by the run-time system to improve the Quality-of-Service. We provide an example of a task exposure with timing requirements below:

```
#pragma omp task inout(data[i][j]) \
        deadline(omp_get_wtime()+1.0) \
        onerror(OMP_SKIP)
   draw();
```

Listing 1.2. Exposing a task with deadline using OpenMP with proposed clauses

The example creates a task with a modify (`inout`) dependency on `data[i][j]`, and is expected to execute `1.0` seconds after it was created. If the task is not execute before the given time the run-time system is allowed to drop the task(`on_error(OMP_SKIP)`). The proposed timing clauses have been implemented into the Mercurium compiler [3] which is a transcompiler for OmpSs [5].

4 Run-Time system support

A *PID* controller is an invention from the early twentieth century [16] and is used (even today) in e.g. ship steering, controlled flight, owens... Resources in the run-time system are controlled using a **P**roportional **I**ntegral **D**erivative feedback controller (*PID*). The **PID** output controls the amount of resources needed to meet the QoS. The formula for the output(o_n) is:
$o_n = K_p * e_n + K_i * \sum_{i=0}^{n} e_i + K_d \frac{e_n - e_{n-1}}{\delta t}$

The formula is a weighted-sum of three fractions, all based on the error (e_n). The error e_n is in our case the difference between the sampled amount of deadlines missed and the expected (the *quality* set by the user). The first fraction is the proportional part that multiplies the error with a weight K_p. The second fraction (the **I**ntegral) accumulates past errors and multiplies it with a weight K_i. The final fraction calculates the *slope* of past errors multiplied by a weight K_d. The sign of the output(o_n) determines the increase or decrease in resource changes, a positive sign requires resource to increase. The amount of resource change is determined by the magnitude of the output ($|o_n|$).

Our implementation is limited because the *coefficients* (K_e,K_d,K_i) need manual tuning. Further work would automatically tune the **PID** during installation (similar to ATLAS [4]) or during/post execution [23, 12]. We implemented our **PID** controller using `POSIX` signals that preempts a thread in user-level mode. We sampled the amount of deadlines missed every $\frac{1}{10}$[5] second and activate the **PID** controller during the sampling.

Apart from the **PID** controller, our scheduler performs the following:

– If a task's deadline is violated (missed) and the task *can* be dropped, then the task is unconditionally dropped to prevent sub-sequent tasks to miss their deadlines
– If a task's deadline is violated and the task *cannot* be dropped, then the task will execute
– Child task's will inherit their parent's timing constraints (but not their dropability)
– A thread *may* not be de-activated when waiting for child tasks' to complete. This avoids *dead-locks*.

The scheduling algorithm consists of a global Earliest-Deadline-First (EDF) queue where all timing constrained tasks are inserted to [6]. The EDF queue sorts them according the earliest deadline using a binary heap (*O(log n)* push/pop complexity). Each resource also has a private queue used for tasks without timing requirements, and the work distribution scheme is a random work-stealer to maintaining performance in mixed (timing and non-timing constraints tasks) workloads.

[5] This time can vary. Higher interruption frequency gives quicker convergence at the cost of overhead, something we have not evaluated in the present study

[6] Unless they are rejected(dropped)

4.1 Example

The following example illustates how our scheduler reflects the quality-of-service requirements set by the application and how resources are conserved. Consider an *fork/join* application where tasks have different timing constraints (deadlines) and can be dropped. Our example uses two Quality-of-Service configuration. `Configuration#1` will allow that at-most 8% of all tasks can be dropped while `Configuration#2` allows 4%. This means that `Configuration#2` requests a more constrained service than `Configuration#1`. Sample code for the example is shown below:

```
double app_start = omp_get_wtime();

while (1)
 {
    for (int i=0;i<1000;i++)
      #pragma omp task \
          deadline(app_start + 0.05) \
          onerror(OMP_SKIP)
     do_work();
    app_start+=0.1;

    #pragma omp taskwait
 }
```

Listing 1.3. "Micro-benchmark overall structure for simulation of periodic tasks

Figure 1 shows the execution history of the two configurations. Figure 1:a show how our schedulers maintain the expected amount of deadline misses for the two configurations. Both configuration receives the expected amount of service from the scheduler (8% and respectively 4%). Also note the **PID** characteristics that oscillates with decreasing amplitiude until it converges near the expected quality. Figure 1:b shows the resource usage for the two configurations and how our scheduler varies it with time. Note how the scheduler detects the configuration with more relaxed constraints (`Configuration#1`) and automatically use less resources in it. The average number of resource used are 2.78 threads for `Configuration#1` and 3.04 threads for `Configuration#2`.

4.2 Relation to Multiprogrammed scenarios through CPU brokers and Frequency scaling

Our proposed methods *should* work well in a multiprogrammed scenario (although, future work would verify this). Let us assume that we have a CPU broker that distributes resource according to some application need. Each running application request a number of resource at certain, predetermined interval and the CPU broker either *gives* or *rejects* an application the requested resources. In such a scenario, the application will need to predict the needed resources in advance, something our PID-based solution already does. Theoretically, the

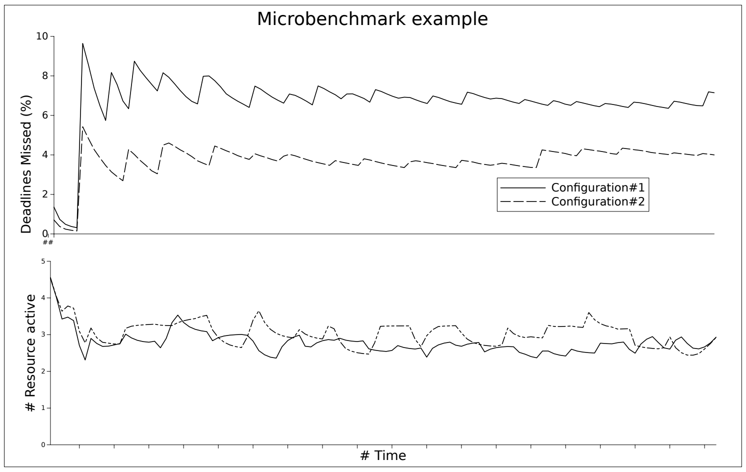

Fig. 1. Micro-benchmark example showing how our scheduler adapts to the application expected QoS. Top-most plot (**a**) shows the *application experienced* deadline missed during application execution while the bottom-most plot (**b**) shows the dynamic resource allocation made by the scheduler

CPU broker would use the amount of deadlines missed for each application and compare it with the *expected service* (the amount of deadlines allowed to miss) and distribute the resource *fairly* to the applications. Variations to the *fair* policy could be based on the amount of tasks that an application *can* dropped (as annotated by the programmer using `onerror`). Frequency scaling would have little effect on our solution. To include frequency scaling, we would provide the run-time system with an *a-priori* profiled scaling function $p(f)$ that shows the performance gains as a function of frequency on either application- or task-granularities. Knowing how the frequency transfers to performance in a certain application, the algorithm now is a constraint problem where the amount of *gain* (as mapped through o_n) in processing power now relates to both for how long we require an resource and at what frequency domain. Integrating frequency into the algorithm yields several new interesting problems: Is it better to predict a resource for a long time with a low frequency, or is it better to use a resource for a short time with a higher frequency. To solve this requires information on the task *injection rate*- how often does a new task arrive-, which in itself a function of the how many resources are currently used (for data-flow parallelism). The algorithm can also be extended to *further* reduce power consumpton if the number of discrete voltage levels associated with the frequencies are more than one.

5 Methodology

5.1 Experimental platform

We have used a general purpose x86-64 Intel Nehalem processor with four cores (eight hyperthreads) with 8 GB of RAM. All eight hyperthreads were enabled and *turbo boost* disabled in order to disable frequency scaling. Temperature was measured with the *CoreTemp* kernel module.

5.2 Applications

We evaluated two applications written in the OmpSs programming model: an volumetric raycaster and a H.264 decoder [2]. The volumetric raycaster uses *fork/join* type of parallelism where each task handles an area of 4x4 rays. In the raycaster, we allow all tasks except the task which outputs the raycasted image to be dropped. Our benchmark's input is a 256x256x256 volume voxel [14]. The H.264 decoder uses *data-flow* type of parallelism to allow pipelined parallelism. Originally the code is from the *ffmpeg* package. The video used for the H.264 evaluation is the BigBuckBunny adventure movie [13] running at picture quality of 720p.

6 Evaluation

We executed the benchmarks five times and took the median QoS performance to represent the result. We measured temperature both for our resource-aware algorithm and the pseudo-random work-stealer [6] existing in the OmpSs framework. Since the work-stealer is resource- and QoS-unaware, temperature measurement strenghten our argument for the neeed of resource management. Benefits would likely be more pronounced if coupled with kernel-aware middleware that helps our scheduler's decisions. We used the average temperature throughout the runs to illustrate that our algorithm reduces both resource usage and temperature.

We varied the number of deadlines the application is "allowed" to drop; this "allowance" is what we call *expected service*. The scheduler will try to satisify the *expected service* but also minimize resources; the service that is actually experienced by the application (and our primary metric for evaluation) is the *provided service*. The resources our scheduler uses is provided as R_{usage} and is read as the average number of threads (including hyperthreads). The thermal effects ($T_{decrease}$) are given as *decrease* over the work-stealing algorithm.

Table 1 shows the result for the H.264 decoder running the BigBuckBunny adventure movie. Tasks are set to have a timing requirement according to 240 respectively 260 frames per second. We varied the *expected service* from 2% to 32% (how many tasks are allowed to miss their deadline). Comparing the *expected service* to the *provided service* we saw that the scheduler is accurate in neither over- nor under-providing resources. Deviations exist because some tasks not droppable (called reference or P-frames in H.264 terminology). As we increase

FPS	Requested (%)	Provided (%)	R_{used}	$T_{decrease}$
240	2%	2.7%	3.8	-9.7%
	4%	5.1%	3.5	-10.1%
	8%	8%	2.9	-10.4%
	16%	15%	3.1	-11.1%
	32%	25%	2.8	-10.5%
260	2%	2.2%	5.3	-4.9%
	4%	3.7%	4.9	-5.3%
	8%	8.7%	3.5	-7.8%
	16%	17.5%	3.6	-8.0%
	32%	27.7%	3.3	-17.7%

Table 1. Scheduler performance of the H.264 BigBuckBunny movie running an expected 240 respectively 260 FPS

the *expected service* (allowing more missed deadlines), both the resource usage and the average temperature decreases. Also note the difference between the FPS levels; when deadlines are set for higher FPS while keeping the same expectance of service, our scheduler detects the increased demand and automatically reflects this in the amount of resources used.

FPS	Requested (%)	Provided (%)	R_{used}	$T_{decrease}$
8	2%	2.2%	3.4	-11.2%
	4%	4.15%	3.3	-10.7%
	8%	8.3%	3.0	-5.4%
	16%	16.2%	2.6	-8.5%
	32%	30.5%	2.5	-14.2%

Table 2. Scheduler performance of the RayCaster running at (expected) 8 FPS

Table 2 show similar results in terms of accuracy of the *requested service*. We see that the *requested service* is satisfied and the temperature is reduced (compared to random work-stealer). The resource usage is also continuously pulled down as the *requested service* becomes more relaxed (more tasks allowed to be dropped). Note that our scheduler performs (in terms of *provided service*) better using the Volumetric Raycaster compared to the H.264 decoder. This is because *all* tasks in the Volumetric Raycaster are droppable, something not the case in the H.264 decoder. Future work would investigate prediction of droppable tasks based on history. This prediction would guide the scheduler in knowing when to expect tasks that can be dropped.

7 Conclusion

We presented a resource-aware scheduler and details how to incorporate it into a task-based programming model, preparing the task-based paradigm for multi-programmed scenarios. Our solution controls resources dynamically according to required Quality-of-Service in the application. We showed that our scheduler supplies the application with enough resources to execute at expected quality, and detects when changes in resources are needed. The preservance of resources was shown by recording both the by-scheduler experienced resource-usage as well as the physical thermal sensors; both indicate positive results. Future work would calibrate the **PID** controller either online or post-execution, and evaluate multi-programmed scenarios where our scheduler would interact with a kernel middleware that maintains system resources. A continuation of this paper would also include sampling the error term, o_n, over a period of time; autocorrelation the samples could provide insight on repeating patterns within the application and could help reduce and predict periods of increased activity, such as an incoming reference frame in a movie decoder.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (`www.encore-project.eu`), grant agreement nr. 248647. The authors are members of the HiPEAC European network of Excellence (`www.hipeac.net`). Additional thanks goes to the Barcelona Supercomputer Center for help in integration of real-time extensions into the Mercurium compiler.

References

1. K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson. Adaptive scheduling with parallelism feedback. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 100–109. ACM, 2006.
2. M. Andersch, C. C. Chi, and B. Juurlink. Programming parallel embedded and consumer applications in OpenMP superscalar. *ACM SIGPLAN Notices*, 47(8):281–282, 2012.
3. J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*, volume 8, 2004.
4. R. Clint Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.
5. A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
6. A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *OpenMP in a New Era of Parallelism*, pages 100–110. Springer, 2008.

7. E. Eide, T. Stack, J. Regehr, and J. Lepreau. Dynamic cpu management for real-time, middleware-based systems. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 286–295. IEEE, 2004.
8. D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu. A real-time scheduling service for parallel tasks. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–272. IEEE, 2013.
9. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.
10. T. Gautier, F. Lementec, V. Faucher, and B. Raffin. X-Kaapi: a Multi Paradigm Runtime for Multicore Architectures. Rapport de recherche RR-8058, INRIA, Feb. 2012.
11. D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.
12. M. Güzelkaya, I. Eksin, and E. Yeşil. Self-tuning of PID-type fuzzy logic controller coefficients via relative rate observer. *Engineering Applications of Artificial Intelligence*, 16(3):227–236, 2003.
13. http://www.bigbuckbunny.org/. Big buck bunny adventure.
14. http://www.osirix viewer.com/datasets. Dicom sample image sets.
15. K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 259–268. IEEE, 2010.
16. N. Minorsky. Steering of ships. 1984.
17. K. Nahrstedt, H.-h. Chu, and S. Narayan. QoS-aware resource management for distributed multimedia applications. *Journal of High Speed Networks*, 7(3):229–257, 1998.
18. R. S. Nikhil, K. K. Pingali, et al. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632, 1989.
19. L. M. Nogueira, L. M. Pinho, J. Fonseca, and C. Maia. On the use of Work Stealing Strategies in Real Time Systems. Technical Report CISTER-TR-130110, 2013.
20. A. Pop and A. Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53, 2013.
21. J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *ACM SIGARCH Computer Architecture News*, volume 32, page 276. IEEE Computer Society, 2004.
22. Varisteas, Georgios and Brorsson, Mats and Faxen, Karl-Filip. Resource management for task-based parallel programs over a multi-kernel.: BIAS: Barrelfish Inter-core Adaptive Scheduling. In *Proceedings of the 2012 workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE12)*, pages 32–36, 2012.
23. Z.-W. Woo, H.-Y. Chung, and J.-J. Lin. A PID type fuzzy controller with self-tuning scaling factors. *Fuzzy Sets and Systems*, 115(2):321–326, 2000.