ABSTRACT
Not caring about resources mean 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 core count increase 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 [20], decreasing 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 execution1. 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 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 need and request them from a system scheduler [21] or CPU broker [7, 16].

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-consumption2. 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 to provide the user means of specify timing requirements on tasks (soft-real time). 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 [18] proposed two different EDF strategies for imple-

---

1 Although OpenMP allows for different core allocations between different parallel regions, this is rarely done in practice.
2 Note, power-consumption and not necessarily the energy consumed by the application.
menting 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 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 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 start by #pragma omp task followed by clauses that further defines the properties of the task:

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

Listing 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 Wave-fronts [17]); it has been embraced by several task-based run-time systems [5, 19, 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 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

Resources in the run-time system are controlled using a Proportional Integral Derivative 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 Integral) 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_p,K_d,K_i) need manual tuning. Further work would automatically tune the PID during installation (similar to ATLAS [4]) or during/post execution [22, 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 \(\frac{1}{\text{FPS}}\) 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 drop-
ped 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 \(^4\). 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.

4.1 Example

The following example illustrates 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:

```c
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 3: Micro-benchmark overall structure for simulation of periodic tasks

Figure 1 shows the execution history of the two configurations. Figure 1a 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 amplititude until it converges near the expected quality. Figure 1b 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 \(^4\)Unless they are rejected(dropped)

Figure 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 (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.

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 strengthen our argument for the need 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.
Raycaster compared to the H.264 decoder. This is because (in terms of provided service allowed to be dropped). Note that our scheduler performs requested service as the requested service re-

Table 2 show similar results in terms of accuracy of the amount of resources used.

The increased demand and automatically reflects this in the FPS levels; when deadlines are set for higher FPS while kee-

Also note the difference between the 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 kee-

We varied the number of deadlines the application is “al-

Table 1 shows the result for the H.264 decoder running the BigBuckBunny adventure movie. Tasks are set to have a ti-

ming 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). Compar-

ing the expected service to the provided service we saw that the scheduler is accurate in neither over- nor under-

providing resources. Deviations exists because some tasks not droppable (called reference or P-frames in H264 termi-

nology). As we increase 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 kept-

ning the same expectance of service, our scheduler detects the increased demand and automatically reflects this in the amount of resources used.

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 workstealer). 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, nothing 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.

We varied the number of deadlines the application is “al-

owed” to drop; this “allowance” is what we call expected service. The scheduler will try to satisfy the expected service but also minimize resources; the service that is actually experience by the application (and our primary metric for evaluation) is the provided service. The resources our schedule 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 exists because some tasks not droppable (called reference or P-frames in H264 terminology). As we increase 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.

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 workstealer). 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 change 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.

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 (urlwww.hipeac.net). Additional thanks goes to the Barcelona Supercomputer Center for help in integration of real-time extensions into the Mercurium compiler.

8. REFERENCES


