Autonomic Thread Scaling Library for QoS Management[*]

Gianluca C. Durelli
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
gianlucacarlo.durelli@polimi.it

Marco D. Santambrogio
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
marco.santambrogio@polimi.it

ABSTRACT

Over the last years embedded system industry faced a revolution thanks to the introduction of multicores and heterogeneous devices. The availability of these new platforms opens new paths for these devices that can be nowadays used for more high demand tasks, exploiting the parallelism made available by the muticore processors. Nonetheless the progresses of the HW technology are not backed up by improvements of the SW side, and runtime mechanisms to manage resource allocation and contention on resources are still lacking the proper effectiveness. This paper tackles the problem of dynamic resource management from the application point of view and presents a user space library to control application performance. The control knob exploited by the library is the possibility of scaling the number of threads used by an application and seamlessly integrates with OpenMP. A case study illustrates the benefits that this library has in a classic embedded system scenario, introducing an overhead of less than 0.5%.

1. INTRODUCTION

Over the last years embedded system industry faced a revolution with the introduction of multicores and heterogeneous devices. The representative platform in the field has then moved from the classic micro-controller or single core processor with relatively low performance and low power consumption to a more high performance family of chips, featuring a high degree of parallelism. An example of this revolution is represented by the mobile world which moved in few years from a single core processor working at few hundred megahertz, to nowadays systems featuring octa-cores asymmetric processor working at more than 2 GHz. This increased computational capability led to a drastic change in the applicability of this devices; if once they were used only for simple functions, now they can be used to perform highly demanding tasks such as multimedia, video processing, or other kinds of high performance computations. As a further example of this wide applicability of embedded system devices the EU project Mont Blanc [1] is aimed at developing the first High Performance Computing (HPC) cluster based on ARM processors.

If on one hand the updates on the hardware side marked this big revolution, on the other the software side didn't keep up and, generally, applications in this domain rarely exploit parallelism (as for instance mobile device applications), and, more importantly, there is a big limitation in how the system performs online resource management. Taking in consideration a wide-spread programming model for parallel applications, such as OpenMP [2], we have that applications written following this standard generally try to exploit all the available cores in the system, conflicting with the other running applications.

Academia and industries tackled this problem by devising algorithms and solutions to minimize interference among colocated applications [3] and to perform dynamic resource management in multiple contexts, ranging from embedded devices to multi-tenants infrastructures [4]. Many works proposed to substitute the current programming models with a resource on-demand oriented model where the applications ask the system the resources they need at runtime (Section 5).

This paper tackles the problem of dynamic resource management from the application point of view and presents a user space library to control application performance. The control knob exploited by the library is the possibility of scaling the number of threads used by an application and this is carried out by seamlessly integrating with OpenMP (Section 2). To achieve this goal, the proposed solution monitors application performance at runtime through a heartbeat-like API [5] and builds an estimation of the application scalability considering the contention caused by other applications concurrently running. To summarize this work proposes:

- a library to automatically infer the right number of threads to use to respect the application Quality of Service (QoS)
- a model to infer the scalability of an application at runtime considering resource contention

Details about the implementation and experimental results are presented in sections 3 and 4 respectively. The library, being implemented in user space, can be used as is on every system without further system modification, and the code is available as open source ([6]).

2. PROBLEM DEFINITION

The availability of multicore architectures in embedded systems, allows these systems to execute computational intensive tasks taking advantage of the available parallelism. As a

[*]EWiLi'15, October 8th, 2015, Amsterdam, The Netherlands. Copyright retained by the authors.

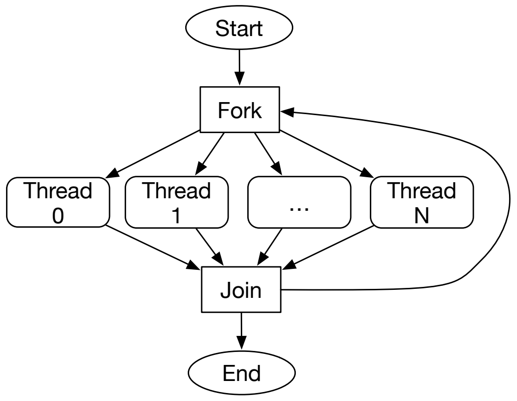

Figure 1: Task graph model of supported applications.

consequence multiple applications will be executed, or consolidated, on the same system to increase its utilization. Consolidation, as we learned from the utility computing scenario, will lead to conflict among the concurrent applications. The reference programming languages to achieve high performance (at least for low level libraries) are with no doubt C and C++, and *pthread* and *OpenMP* are the two main solutions to implement parallel applications. This section presents the problem tackled by our work and illustrates the different components that participate at its solution.

2.1 Application Model

This work focuses on the management of CPU intensive, data parallel applications; these applications have the advantage of achieving an high speedup when parallelized using multi-threaded implementations.

A simple model of these applications is the one presented in Figure 1. This model is a cyclic graph that represents the execution of a hot loop of a program. Each iteration of a loop executes a computational kernel over a set of data and it can be parallelized following a fork-join model. This might seem a fairly simple model to consider but it finds real application in a wide range of fields, such as image analysis where the loop may work on different images or frames, or physical simulations where each iteration represents a time step of the simulation or the analysis of a partition of the input dataset. In the target programming language, either C or C++, this structure is represented by a for loop executing a *computational intensive kernel* that can be parallelized by means of *pthread* or *OpenMP*.

```
for(i=0; i<NUM_ITERATIONS; i++){
  execute_kernel(...);
}
```

Listing 1: Main loop structure

```
void execute_kernel(...){
#pragma omp parallel for
  for(d=0; d<DATA_SIZE; d++){
    // Do Computation...
  }
}
```

Listing 2: Parallel kernel structure

2.2 Application Performance Requirements

The model for the applications so presented allows to define application's performance both in terms of constant throughput and deadline. In particular we can consider two cases, whether the size of the input dataset is known a priori or not. In the first situation supposing that D is the size of the dataset and each iteration computes a partition P_D of it, we can express on the application two kinds of QoS levels. If we want to express a deadline, T_{QoS}, for our application we might express it as a time constraint or as an average throughput goal as well; in fact a deadline of T_{QoS} second is the same as an average throughput constraint of $D/T_{QoS} = Thr_{QoS}$ actions/s (note that we used the term actions because this metric is application dependent). To enforce such deadline we have to guarantee that the desired $D\ /\ Thr_{QoS}$ ratio is greater than the one really achieved by the application $D\ /\ T_{real}$. Considering the structure of the described applications, we have that if every single iteration of the loop respects the throughput constraint then the whole application respects the throughput constraint as well:

$$\frac{P_D}{T_{iteration}} < \frac{D}{T_{QoS}} \implies \frac{D}{T_{real}} < \frac{D}{T_{QoS}}$$

Otherwise if we do not know the amount of data to be computed we can not set a deadline, but we can instead set a minimum average throughput for our application. In this second case we can express the goal over the execution of a single iteration of the loop and so we have to guarantee that $P_D/T_{iteration} < Thr_{QoS}$. As we can see, we can enforce deadline and throughput constraints in the same way in our applications; this means that we can use a single control policy to manage both requirements at the same time.

The performance in our application model is expressed as performance of the computational intensive loop of the application. Expressing quality of service constraints in this way allows the library to be application independent and more importantly allows the final users to express performance in a high level manner. For instance if we consider a video processing application, we have that each iteration of the loop will compute one frame; in this case the performance can be set to 30 Frame/s which means that the loop has to execute at least 30 times per second. This concept is similar to the one expressed by the Application Heartbeat framework [5] from which we derived our monitoring component. Application Heartbeat proposes to instrument the application with proper function calls that allow to express progresses of the application in terms of high level metrics. In the same manner, our application will be then instrumented to provide this information to the autonomic library; Section 3 details how the monitoring has been implemented and will clarify the steps that have to be taken to instrument an application.

2.3 Performance Resource Assignment

Given a set of applications that express requirements in terms of deadline or throughput, the goal of the proposed work is to perform a dynamic resource assignment such that each application gets the resources it needs to fulfill its requirement. The resource management policy envisioned in this work aims at managing the number of threads of an application and it works on a per application basis, meaning that each application will try to reach the solution to the resource management problem on its own. The reason behind this choice is that it solves many of the issues present in state of the art works without losing in effectiveness. For instance it permits to run on a given system both application exploiting this dynamic control and legacy applications without any problem. Furthermore distributing the control allows to execute the control step only on given events that cause the availability of new data to tune the applications (i.e. when a single iteration of the loop terminates), instead of having an external entity that executes with a given period. This will drastically reduce the impact of the controlling mechanism over the system since now the computation happens only when it is needed and it happens inside every application impacting only its performance. These presented here are few hints of the benefit that can be achieved with this distributed approach while a full comparison with the state of the art is drawn in Section 5.

2.4 Thread Scaling

The reason behind the choice of thread scaling instead of controlling other resources such as the number of cores assigned to one application relies on the fact that controlling the number of threads solves many management issues while allowing at the same time for better results.

The biggest problem to solve in order to find the correct number of threads to use to execute a kernel is to infer how the application scales when it is executed with a different number of threads. A first solution would be to exploit offline profiling to know the performance profile of the application. However we do not think this is the right way to approach the problem since it would require a different profiling of the application for all the possible target machines and for the different size of input data used in each iteration of the for loop. Furthermore the scaling curve observed with offline profiling might change at runtime depending on other applications running in the system. Since profiling all the possible applications combinations would not be possible, we decided to exploit the same monitoring information used for detecting application performance, and then build online a scalability model of the application.

The method to predict the performance, which is better explained in Section 3, starts supposing that the performance of an application scales linearly with the number of the available threads. The first run of the loop will then provide an expectation of the actual scalability of the applications and an hint for the policy on the number of threads to use in the next iteration. After each iteration the performance measured for the number of threads used in that execution are trained using an exponential moving average that allows the prediction to be adapted to runtime conditions. Points not yet tested are predicted interpolating the model starting from available values. Threads configurations that are not monitored for a certain number of iterations are progressively forgotten so that the model does not get stuck with older estimations that are no longer valid.

3. IMPLEMENTATION

This section provides implementation details for the different components of the proposed library. As a general note all the components here discussed are programmed using C++ and the entire library is implemented as an userspace component. A programmer can write its own application and compile it against the library adding the proper function calls in its program to take advantage of the autonomic functionalities as further described. Interested readers can look at the complete code for this work cloning this GIT repository [6].

3.1 Monitoring

As a first step, in order to make the library aware of the performance of the applications, and be able to adapt to changing runtime conditions, we need a monitoring component that captures application performance. This component is modeled after the Heartbeats framework [5]. The application can send heartbeats through a proper API and the monitor collects these heartbeats and summarizes them in high level metrics which are proper of the application. As previously highlighted, since the policy controls the number of threads, the application should emit a heartbeat only when the parallel section of the kernel ends (i.e. at the end of the iteration). With respect to standard Heartbeat framework we allowed the possibility to emit not only the heartbeat signal but also to specify the number of heartbeats to be sent with the signal. This allows one to specify how much computation has been effectively done in an iteration of the loop, allowing the final user to specify a true high level metric that does not change with respect to how the input is partitioned across the iterations. Consider a kernel that has a throughput of Thr_k operations/s and that takes Δt seconds. In the standard library a single heartbeat would be emitted leading to a measured throughput of:

$$Thr_{standard} = \frac{1}{\Delta t} = \frac{1}{N} \times Thr_k = \frac{Thr_k}{N}$$

while if we consider the proposed solution the heartbeat will signal the computation of N units during the iteration leading to a throughput of:

$$Thr_{proposed} = \frac{N}{\Delta t} = \frac{N}{N} \times Thr_k = Thr_k$$

Considering a video processing application N might be a function of the resolution of the input video and a given goal of 30 Frames/s should not vary while changing the input size. The current implementation of the monitoring library exports three different values to the controller: the *instant throughput* (i.e. the throughput of the last iteration), the *window throughput* (i.e. the throughput over a set of iterations), and the *global throughput* (i.e. the throughput from the start of the program). When an application emits a heartbeat signal, the heartbeat value and the emission time are stored in a FIFO buffer having a length defined by the programmer; this length corresponds to the length of the window used to compute the *window throughput.* Supposing a FIFO of length N the three metrics, at iteration i, are so computed:

$$Thr_{instant} = Heartbeats_i/(t_i - t_{i-1})$$

$$Thr_{window} = \frac{\sum_{k=i-N+1}^{i} Heartbeats_k/(t_k - t_{k-1})}{N-1}$$

$$Thr_{global} = \sum_{k \in I} Heartbeats_k/(t_i - t_0)$$

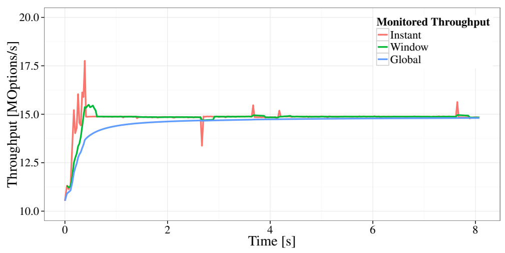

Figure 2: Monitored performance values during the execution of the application.

These three metrics have different meanings, the *instant throughput* captures the actual performance of the application and plays a key role in building the performance model of the kernel. The *window throughput* instead can be used to monitor applications that specify a QoS that consists in having a constant throughput throughout all the execution; finally the *global throughput* can be used to check if an application is meeting its deadline.

Figure 2 illustrates how these throughputs vary across the execution of one of our benchmarks. As expected, given the way these quantities are computed, the *instant throughput* shows spikes due to the change of threads allocation or the interference from other application, the *window throughput* is generally stable across the execution, while the *global throughput* asymptotically tends to a stable value.

3.2 Model Update

To predict which is the right number of threads to use to execute a kernel fulfilling the requirements, the library needs to know how the performance of the application scales with respect to the number of threads used. One approach followed by many state of the art solutions [7, 3] relies on offline profiling information to capture both single application performance and also interference caused by colocated workloads. However this solution is unfeasible since the possible number of application combinations increases exponentially. Our solution tries instead to avoid offline profiling and relies only on online monitoring information, collected by the heartbeat-like API, like other approaches [8, 9].

Since the library controls the number of threads to assign to an application it has to build a model that estimates for each possible number of threads the performance of the kernel. The algorithm to train this model is rather intuitive, and it is based on a linear interpolation of the online monitored *instant throughput*. As we discussed at each iteration the monitor updates the value of the *instant throughput* that represents the real performance the kernel can achieve in a given moment with the number of threads it used. The *instant throughput* values are collected in FIFO buffers as it was for the monitoring data, and the user can set the length of the FIFO.

Listing 3 reports the *updateModel()* routine which computes the new model on the basis of the last observations. The routine is divided in two steps; the first one adjusts the performance estimation for monitored points on the basis of the performance history. The performance history is en-

```
std::map<int, std::vector<double>>
    threadPerformanceHistory;

void LibThreadScale::updateModel()
{
    std::map<int, double> tempThreadToPerformance;

    for(auto &v : threadPerformanceHistory){
        tempThreadToPerformance[v.first] =
            exponentialMovingAverage(v.second,
            forgettingFactor);
    }

    model.createModel(tempThreadToPerformance, std
        ::pair<int, int>(1, maxThreads));
}
```

Listing 3: Model update routine

coded as a structure (a *map*) that connects each possible thread number configuration to a list of performance measurements. This part computes, for each of the data in the performance history, an exponential moving average over the list of performance measurements. Depending on the value of the *forgettingFactor* parameter the last values inserted in the FIFO might impact more than the older ones, meaning that last observations are more meaningful for determining the current performance. This behavior is explained by the fact that if runtime conditions change we want that our model converges quickly to the new estimation. After the estimation for the already monitored point is determined, the function creates the final model (*model.createModel(...)*) interpolating linearly the just computed points over a specified interval (from 1 to the maximum number of threads that can be used). If only one value is present in the *tempThreadToPerformance* map, the point $(0, 0)$ is used as second point to build the linear interpolation function.

For sake of clarity we recall that the exponential moving average over a list of values V using a forgetting factor α is computed as follows.

$$avg_1 = V_1$$

$$avg_i = \alpha \times V_i + (1 - \alpha) \times avg_{i-1} \;,\; for\ i > 1$$

3.3 API

The library exports two APIs that the programmer has to use to instrument its application. All the monitoring and model estimation complexity is hidden behind these APIs, allowing to minimize the impact of the library on the target application. The APIs are the following: (1) *setMinQoS* : allows to set a minimum QoS requirement (measured as throughput) that the library tries to meet; (2) *sendHeartbeat* : communicates to the library that a certain amount of data has been processed. These APIs when invoked connect to the rest of the system and updates the monitored values and the model estimations depending on the runtime conditions. Figure 3 represents a sequence diagram illustrating the interactions between the library components. All the complexity of the devised library is hidden behind the *sendHearbeat* function. Upon the invocation of such function the library registers the monitoring information and updates the model. The model update is triggered by the *getThreadsToUse* call that uses the new information available in the monitoring

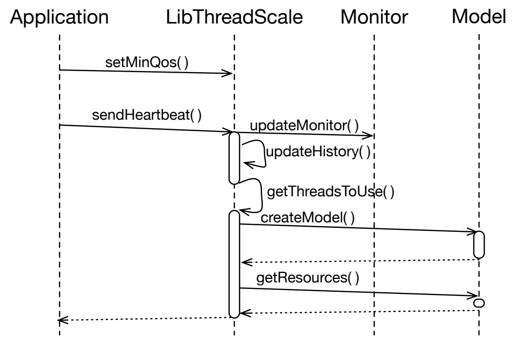

Figure 3: Sequence diagram of function calls invocation initiated by publicly exported library functions.

```
LibThreadScale libQoS;
libQoS.setMinQoS(MIN_QoS);
for(i ...){
#pragma omp parallel for
  for(j ...){
    // Kernel code here
  }
  libQoS.sendHeartbeat(NUM_HEARTBEATS);
}
```

Listing 4: Instrumented Application

history to compute the optimal number of threads to use in the subsequent iterations.

Listing 4 illustrates the instrumentation process of the application. Only 3 lines of code have been added, meaning that the usage of the proposed library does not heavily impact the application code and programmability. Upon the execution of the *sendHeartbeat* function the number of threads to use for the next execution is set automatically if *OpenMP* is used; this number is also returned to the caller if manual implementation of thread dispatching is needed, for instance when *pthreads* are used.

4. RESULTS

This section illustrates the experimental campaign carried out to test the functionalities and performance of the autonomic library proposed in this work. The tests have been performed on the Odroid XU3 development board, which features an ARM big.LITTLE processor mounting 4×A15, and 4×A17 cores. For sake of clarity, we limit the discussion to only one case study; an interested reader can download the source code at [6] and replicate the tests shown here on the other benchmarks we instrumented.

4.1 Case Study

The case study presented here is rather simple, but it is quite effective in illustrating the potential of the proposed solution. Consider the case where an embedded device, as the one we used here, is employed on cameras for surveillance purposes. Considering the hardware available on the device, is quite natural to perform as much computation as possible on board instead of streaming the data to external workstations. The system has to perform the task of monitoring the environment and detecting movements; once the movement

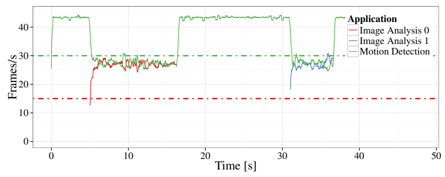

Figure 4: Colocated applications managed by Odroid scheduler.

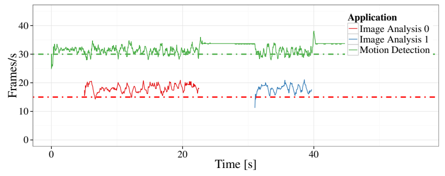

Figure 5: Colocated applications managed by our solution.

is detected the device performs a deeper analysis of the interested frames, while continuing with the motion detection task. Since motion detection is critical to initiate the second phase and to trigger possible alarms, we need for this task to maintain a constant performance of at least 30 frames per second (fps). The subsequent image analyses instead have less priority and it is acceptable for them to have a lower performance, let's say 15 fps. Figure 4 illustrates what happens in the system when the running applications rely on the Odroid native scheduler. As we can see *motion detection* works at higher performance than the one requested but when the *image analysis* is requested its performance drops and stays below the desired limit for all the time the *image analysis* task is running. On the contrary, Figure 5 illustrates how the proposed solution is able not only to keep the *motion detection* task in an acceptable working condition range when colocated with the *image analysis* tasks, but it also keeps its performance near to the one requested using fewer threads to run the task. This decision to scale the number of threads is done automatically, and helps for instance to reconfigure this particular application when the new generation of surveillance camera has to be put on the market. Furthermore, in this case, lowering the number of threads implies a reduction in the number of cores active in the system, and subsequently the power consumption and the overall temperature are reduced.

4.2 Adapting to Different System Loads

In this second experiment we used an instance of the Black and Scholes benchmark, which runs for about one minute, and then we forced on the CPU three different loads using Linux *stress* and *cpulimit* utilities in combination. We recall that *stress* utility forces a 100% CPU load on given number of processors and that *cpulimit* can limit the CPU time of a process to a given percentage.

This experiment has the double goal of illustrating how the library reacts to varying system load and that the devised solution can be used alongside legacy applications without the need for them to be instrumented. We injected the load in the system bounding 8 *stress* workers to the 8 CPUs and

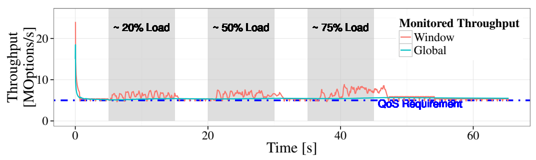

Figure 6: Adaptation of the devised library to external system load.

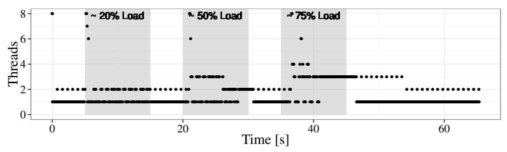

Figure 7: Threads allocation under different loads.

we then limit the CPU time of these workers to the load we desired. The load was kept for 10s and then we left the system at rest (without any external load except for our benchmark) for 5s; the loads used to stress the system where 25%, 50%, and 75%. The goal set for the application was of 5 MOptions/s (i.e. number of stock options to process in one second). Figure 6 illustrates the performance perceived by the application, while Figure 7 illustrates the thread allocation enforced by the devised library. The first allocation at the beginning of the program consists in using 8 threads and then the model of application scalability is built and used to determine the next allocations. If we look at the throughput perceived by the application we see that global throughput is always maintained above the required QoS threshold regardless of the system load injected. To achieve this result a different number of threads are provisioned to the application at each iteration so that for higher loads the library will consistently use a higher number of threads. From the figures we can see that the *window throughput*, even if it is always higher than the application requirement, has an higher variability and it is not kept at the goal level as it was in the previous example. The reason of this behavior relies in the fact that *cpulimit* utility does not force a precise utilization, but rather an average one allowing oscillations in the utilization enforced in any given moment and the policy in turn tries to adapt to these unpredictable changes. Anyhow after the end of the load the library returns at the initial working condition.

4.3 Library Overhead

The last part of this section is aimed at showing the overhead introduced by the proposed library. To measure the overhead we instrumented our library to collect the execution times of the most interesting functions. We want to determine the impact of the library on the application, and for this reason we collected the execution time of the *getThreadsToUse*, *sendHeartbeat* and *updateModel* functions that are executed when a heartbeat is emitted. Table 1 reports the average, standard deviation and maximum execution time measured over 4500 invocation of the library. As it can be noticed from the table the overhead is relatively small and it is about 140.14 μs on average for each iteration[1]. Note that, since the actuation varies the number of threads, the iterations need to be quite long to not incur in excessive overhead while spawning the threads. In our experiment with a ideal kernel duration of $33.33ms$ (corresponding to a requirement of 30 frame/s) the library introduced an overhead of 0.42% on average which is negligible.

[1]The three function calls in Table 1 are nested, only the last one has been taken in account to compute the overhead).

Table 1: Execution time for the most interesting function calls.

Function	Average[μs]	Std. Deviation[μs]
updateModel	64.86	45.01
getThreadsToUse	81.82	51.36
sendHeartbeat	140.14	105.02

5. RELATED WORKS

Autonomic resource management has been a hot research topic over the last few years. The capability of a system to adapt to runtime varying conditions has been explored in different context ranging from embedded systems [10], to workstations [8] and cloud infrastructures using virtual machines [7]. Many works proposed resource management controllers with the goal of managing colocated application trying to optimize their performance [3] or other figures such as performance per watt [11]. Different control techniques have been explored and a fairly comprehensive summary can be found in [12]. To attain a good control, multiple types of actions have been investigated such as CPU cores affinity management [9], DVFS control [10], task mapping [13], CPU time allocation [14], and cache coloring [15].

Some of the related works rely on offline application profiling to determine their characteristics (e.g. I/O or CPU bound) or how colocated workloads interfere one with the other [7]. After the offline profiling phase the online algorithm ensures a given QoS or maximizes a given metric (e.g. performance). However this approach does not scale with the number of possible running applications and in general online profiling solutions are better suited when the set of applications is not known in advance, as done by [9, 14, 12], and the proposed approach. Considering the resource controlled to attain a given QoS the works most similar to the one we propose controls at runtime either the number of CPU cores or the CPU time assigned to applications in order to grant a given QoS [3, 9, 14]. All these solutions, as well as our work, take in account the interference caused by colocation among different applications when they estimate the resource to allocate to the applications. However in their evaluation these researches used a fixed number of threads per application and then constrained these threads to execute on a variable number of cores depending on resource manager decision. This solution causes slowdown due to the useless context switch overhead, which degrades performance even more when the context switch delays thread synchronization; in fact a delay in executing one instruction from one thread might block another thread that is waiting on a synchronization point. In the same way [3, 9] possibly cause slowdown at synchronization point because they change the thread mapping which might be suboptimal with respect to memory hierarchy architecture. Usually the operating system uses some heuristics to optimize threads placement among cores and changing the affinity manually rarely leads to improvements. The approach proposed in this paper aims at changing the number of threads used at runtime by an application avoiding context switches among threads of the same application and relying on operating system heuristics for threads placement.

Finally the last aspect that has to be considered when comparing our approach to the state of the art is the fact that the control policy of our solution is implemented for each application running in the system. Most of the state of the art approaches [14, 3, 7] relies instead on a central entity that controls the resource assignment for all the application running in the system. A central entity has the advantage that it can better manage the resources when the system is overloaded allowing applications with higher priorities to get more resources, but in this situation the system is already not respecting QoS so it is just a fact of mitigating the fact that the system is failing in its goal rather than a real advantage. Embedding the control inside the single application has instead multiple advantages, while maintaining the same effectiveness when the system is not overloaded. A first advantage is the fact that the overhead in which the application incurs is responsible only by the application itself and not on external entities. Another important aspect is the that a central entity can seldom be used with legacy applications running in the system, which the controller is not aware of; instead our approach does not have any issue with legacy application and can be deployed in any system. Connected to this we can argue that implementing the control inside the application will allow a faster adoption of the autonomic control since our approach does not need any change in the system (which might not belong to the final user) and that the action taken by the autonomic library can be customized for each different applications.

6. CONCLUSION

In this paper we presented an autonomic thread scaling library aimed at guaranteeing to an application certain QoS expressed as a throughput. The library is implemented in userspace and compiled against the application, which has to be instrumented with a few function calls, and seamlessly integrates with OpenMP library. The proposed solution allows to adapt at runtime the number of threads used to compute an iteration of an hot loop to guarantee the desired QoS exploiting a model of the scalability of the application with respect to the number of threads used that is built online. A testing campaign illustrated how the library behaves under different working conditions and highlighted how the overhead introduced on the application is in the order of 140 μs which is negligible; furthermore the impact on programmability is very low since the library requires the insertion of only 3 lines of code in the application. The library devised in this work is available at this link [6].

Acknowledgment

This work has been partially funded by the European Commission in the context of the FP7 SAVE project (#610996-SAVE).

7. REFERENCES

[1] N. Rajovic *et al.*, “Supercomputing with commodity cpus: are mobile socs ready for hpc?” in *High Performance Computing, Networking, Storage and Analysis (SC), International Conf. for.* IEEE, 2013.

[2] L. Dagum *et al.*, “Openmp: an industry standard api for shared-memory programming,” *Computational Science & Engineering, IEEE*, vol. 5, no. 1, 1998.

[3] A. Sharifi *et al.*, “Mete: meeting end-to-end qos in multicores through system-wide resource management,” in *Proc. of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems.* ACM, 2011.

[4] D. B. Bartolini *et al.*, “Towards a performance-as-a-service cloud,” in *Proc. of the 4th annual Symposium on Cloud Computing.* ACM, 2013.

[5] H. Hoffmann *et al.*, “Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments,” in *Proceedings of the 7th international conference on Autonomic computing.* ACM, 2010.

[6] G. Durelli, “Source code.” [Online]. Available: https://bitbucket.org/durellinux/libthreadscale-code

[7] R. Nathuji *et al.*, “Q-clouds: managing performance interference effects for qos-aware clouds,” in *Proceedings of the 5th European conference on Computer systems.* ACM, 2010.

[8] D. B. Bartolini *et al.*, “The autonomic operating system research project: achievements and future directions,” in *Proceedings of the 50th Annual Design Automation Conference.* ACM, 2013.

[9] F. Sironi *et al.*, “Metronome: operating system level performance management via self-adaptive computing,” in *Design Automation Conference (DAC), 2012 49th.* IEEE, 2012.

[10] F. X. Lin *et al.*, “K2: a mobile operating system for heterogeneous coherence domains,” in *Proc. of the 19th int. conf. on Architectural support for programming languages and operating systems.* ACM, 2014.

[11] H. Hoffmann and M. Maggio, “Pcp: A generalized approach to optimizing performance under power constraints through resource management,” in *11th Int. Conf. on Autonomic Computing*, 2014.

[12] H. Hoffmann, “Seec: A general and extensible framework for self-aware computing,âĂİ massachusetts institute of technology,” Tech. Rep, Tech. Rep., 2011.

[13] J. R. Wernsing and G. Stitt, “Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing,” in *ACM SIGPLAN Notices*, vol. 45. ACM, 2010.

[14] D. B. Bartolini *et al.*, “Automated fine-grained cpu provisioning for virtual machines,” *ACM Trans. on Architecture and Code Optimization*, vol. 11, 2014.

[15] A. Sharifi *et al.*, “Courteous cache sharing: Being nice to others in capacity management,” in *Design Automation Conf. (DAC), 2012 49th.* IEEE, 2012.