A Timed-Automata Approach for Critical Path Detection in a Soft Real-Time Application

Bugra M. Yildiz, Christoph M. Bockisch, Arend Rensink, Mehmet Aksit
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
Netherlands
{b.m.yildiz, c.m.bockisch, arend.rensink, m.aksit}@utwente.nl

In this paper, we report preliminary ideas from our project called "Time Performance Improvement With Parallel Processing Systems" (TIPS). In the TIPS project, we plan to take advantage of multi-core platforms for performance improvement by parallelizing a complex soft real-time application implemented in Java.

Manual parallelization does not generally yield optimal performance since its results depend on the designer's intuition and experience; it is not feasible to manually inspect all possible parallelization alternatives and their impact on the performance for large-scale and complex software systems.

Even if we are able to find possible parallelization alternatives with tool support, not all parallelization efforts end up with a performance improvement. One needs to detect the execution paths of the application which are semantically relevant and have higher timing costs. By the term semantically relevant, we mean the execution paths that play an important role from the perspective of requirements of the system. These paths are called "critical paths" and an improvement in these paths is likely to lead to better overall performance of the application whereas the improvements for non-critical paths might not guarantee this [3][4][5][6][7][8]. By examining the critical paths, we can find the bottlenecks along them [3].

The problem we are tackling is that: How can we find the critical execution paths of the application that violate the timing constraints? As sub-problems, we need to answer the following questions: (i) how can we model the application formally so that we can search for critical paths effectively and (ii) how can we represent critical path definitions so that these definitions can give us critical paths when they are applied to the model. One can choose to apply source code analysis directly to get a formal model of the application. However, this approach has two shortcomings. First, source code analysis does not scale for such a large system as we have in the TIPS project. And second, one cannot derive the relevance of an execution path just by observing the source code; relevance has to be derived from the requirements to decide how relevant a critical path is from the requirement analysis perspective.

The approach we propose for critical path detection is shown in Fig. 1. It is based on the creation and usage of a formal timed-automata model of the application which contains the information related to the execution steps with timing properties. Then, we query this model to check existence of critical paths. We have chosen the UPPAAL[1] environment for this purpose.

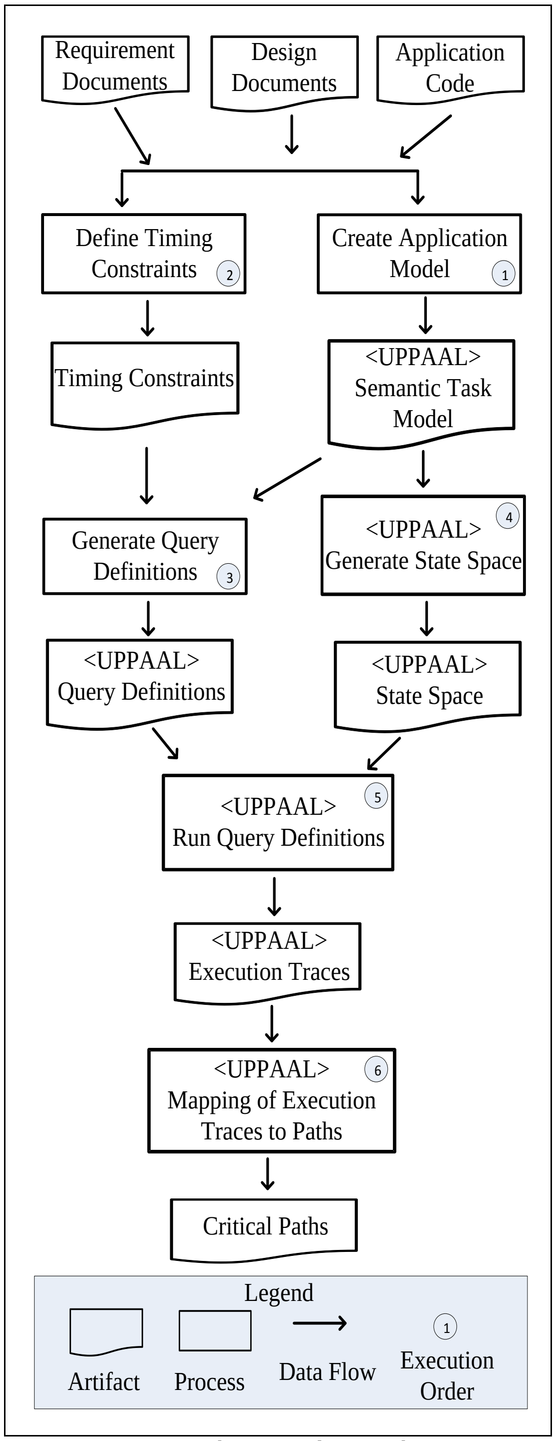
In *step 1*, the approach starts with the creation of the application's semantic task model in timed-automata using UPPAAL. We use the application code, design and requirements documents as an input to this process. The timed-automata model of the application presents the system as a state-machine annotated with timing properties. The model also includes the semantic information coming from requirements documents.

We will map a selected set of entities in the source code and in the design to the semantic task model. In this mapping, deciding on the abstraction level is an important challenge. We must consider that the more concrete the model is, the more accurate critical path decisions will be; but also that the volume of the state space in *step 4* increases dramatically as the detail in the model increases. So, we need to find an optimal abstraction level so that the accuracy is as high as possible while the state space's size stays manageable.

In *step 2*, we define the timing constraints of the application using the application code, requirements and design documents. The timing constraints are already stated as non-functional requirements from the perspective of the end user in requirements documents. What we will do for this step is to re-define these constraints from the viewpoint of the application code and design document in a formal way with an appropriate domain-specific language so that these formally defined constraints can be automatically executed in *step 3*.

In *step 3*, we generate the query definitions to be run on the semantic model from the timing constraints formally defined in *step 2*. In *step 1*, we have created a mapping from the application (code, design and other documents) to the semantic task model, which shows how to go from the source code and design documents to the created model. This mapping helps us detect which parts of the semantic model correspond to the application's parts in the timing constraints. After detecting relevant parts in the model, we generate the query definitions in UPPAAL from the formally defined timing constraints in *step 2*.

This work has been published in the proceedings of the "Work in Progress Session" held in connection with the 22nd EUROMICRO International Conference on Parallel, Distributed and Network-based Processing (PDP 2014). The proceeding has ISBN Number 978-3-902457-39-4. © 2014 SEA-SR-39-4 Johannes Kepler University Linz.



Requirement Documents
Design Documents
Application Code
Define Timing Constraints
2
Create Application Model
1
Timing Constraints
<UPPAAL> Semantic Task Model
Generate Query Definitions
3
<UPPAAL> Generate State Space
4
<UPPAAL> Query Definitions
<UPPAAL> State Space
<UPPAAL> Run Query Definitions
5
<UPPAAL> Execution Traces
<UPPAAL> Mapping of Execution Traces to Paths
6
Critical Paths
Legend
Artifact
Process
Data Flow
1
Execution Order

Fig. 1. The Proposed Approach

In *step 4*, UPPAAL automatically generates the state space from the semantic task model implicitly, which shows all possible execution flows that can be derived from the model. Then, the query definitions are run on this state space in *step 5*. Our query definitions are in the form: “Is there an execution path in the model from … to … which takes longer than timing limit?”. If such query is successful, then UPPAAL provides a diagnostic execution trace showing which execution flow violates the timing constraint [2]. In that case, we examine the execution trace and map it back to the application’s actual execution flow as *step 6*. This reveals which execution path of the application is in fact appears as a critical path.

Back-mapping from the model to the application is not a trivial task. To facilitate this back mapping, the transformation in *step 1* must be reversible. Otherwise, we may not get any execution paths in the application which correspond to a trace provided by UPPAAL.

Another important point to point out is that UPPAAL provides only one example diagnostic trace in case of the violation of the timing property stated in the query. But we are interested in finding multiple critical paths. To solve this issue, we can do a systematic search by modifying the query. We can search through the time dimension by changing the timing threshold in the query in some range (for example, between the timing value stated in the requirements and the maximum execution time). We can also constraint the query to look for the paths that visit desired locations in the model, which can be used to get a diagnostic trace different than the previous ones.

There are still some points to be solved in the approach. For example, how exactly the mapping from the source code entities to the timed-automata model entities should be done.

The abstraction level of the model defined in timed-automata should be optimized so that the state space volume stays manageable. For that purpose, we may consider dividing the model into modules so that the state space does not grow in an uncontrolled manner.

References

[1] UPPAAL web site, http://www.uppaal.org/ , accessed in August 2013.

[2] K G. Larsen, P. Pettersson, and W. Yi. Diagnostic model-checking for real-time systems. In *Proceedings of Workshop on Verification and Control of Hybrid Systems III, Lecture Notes in Computer Science, Berlin Heidelberg New York: Springer-Verlag*, vol. 1066, pp. 575-586, October 1995.

[3] C.-Q. Yang and B.P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 366-373, 1988.

[4] J.K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. In *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 1029–1040, 1998.

[5] Y. Kwok and I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems, IEEE*, vol. 7, no. 5, pp. 506-521. 1996.

[6] R.D. Blumofe. Executing multithreaded programs efficiently. Doctoral dissertation, Massachusetts Institute of Technology, 1995.

[7] M. Schulz. Extracting critical path graphs from MPI applications. In *Proceedings of IEEE International Conference on Cluster Computing 2005*, pp. 1–10, Sep. 2005.

[8] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: the second generation of a parallel program measurement system. In *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 206–217, Apr. 1990.