Application Of Medium-Grain Multiprocessor Mapping Methodology To Epileptic Seizure Predictor

Elena Hammari[1], Francky Catthoor[2], Jos Huisken[3] and Per Gunnar Kjeldsberg[1]
[1] Norwegian University of Science and Technology, 7491 Trondheim, Norway
[2] IMEC, Kapeldreef 75, 3000 Leuven, Belgium
[3] Holst Centre / IMEC Netherlands, High Tech Campus 31, Eindhoven, The Netherlands

Abstract— In this paper we present a methodology that enables mapping and scheduling of a dynamic real-time medical signal processing application onto an MPSoC platform. We apply the Task Concurrency Management (TCM) methodology on Lyapunov Exponent calculator, which is a part of an epileptic seizure predictor. TCM requires a division of an application into thread frames and thread nodes. In particular, we demonstrate a new technique for thread node splitting so as to reduce execution time variance. This is necessary to meet stringent energy and performance requirements during mapping and scheduling. Through experiments we verify that the resulting model of the Lyapunov Exponent calculator fulfills the requirements of the TCM methodology.

I. Introduction

State of the art advanced Multi-Processor System-on-Chip platforms and applications typically require a dynamic scheduling of instructions and threads to be able to meet stringent performance and low energy requirements [1]. The Task Concurrency Management (TCM) methodology is a systematic approach for mapping of an application onto a multiprocessor platform in real-time embedded systems [2]. The basis of the methodology is a two-step scheduling technique that at design-time performs computation intensive parts of the mapping and scheduling, leaving for run-time the parts that result in less overhead for computing the actual schedule.

An application is divided into thread frames, each consisting of a number of thread nodes, as detailed in Sec. II. At design-time, each thread node is profiled to find its execution time and power consumption for all possible input data and on all possible processors on the platform. These numbers are used to find all thread frame schedules with a pareto-optimal trade-off between execution time and energy consumption. A schedule candidate is pareto-optimal if it, e.g., has the lowest energy consumption for a given execution time. As a result, each thread frame has a set of Pareto-optimal schedules along a pareto-curve in a two-dimensional execution time - energy consumption solution space.

In cases where different sets of input data give a too wide spread in a thread node's execution times, scenarios can be used to find a thread frame pareto-curve for each of the different situations [6]. At run-time, input data is monitored to keep track of the currently active scenario. For this purpose easy to execute detection schemes are composed at design-time. Corresponding schedules are selected for each thread frame in such a way that the global energy consumption is minimized while still meeting the application's execution deadline [2].

Even though scenarios can handle differences in thread node execution times, it is in many situations better to split the thread nodes in such a way that we have a more limited variation. So far, only a set of general guide lines exist for this splitting. In this paper we present a case study to show how to split the thread nodes in an appropriate way. This is our first contribution. We use the Lyapunov Exponent calculator as our running example. The presentation is organized as follows. Sec. II summarizes previous work on gray box model employed for modelling of applications using thread nodes and thread frames. This is followed by a presentation of the Lyapunov Exponent calculator and its initial gray box model. Next we demonstrate how its thread nodes can be split in an appropriate way. This is supported by execution time experiments. The experiments prove the feasibility of the methodology on a realistic application from the biomedical domain. This is our second contribution. Finally we discuss our findings and conclude the paper.

II. Overview of the applied methodology

A. Definition of gray box model

A gray box model has been developed as an appropriate application representation in the TCM methodology [2], [3], [4]. The model combines in one view two layers of abstraction: the bottom and more detailed layer is used by the design-time scheduler, while the top layer is passed to the run-time scheduler. In this paper the way to achieve this is more thouroughly described than in previous publications [3], [5].

At the upper layer the application is represented as an extended Petri net. This is done by splitting the application into a graph of *thread frames*, the basic run-time scheduling units in TCM:

Definition 1. *A thread frame is a maximally sized piece of functionality that can run without rescheduling from the run-time scheduler and having a single thread of control. A thread frame has exactly one entry control port but it may have multiple exit control ports. A thread frame can have any number of data ports associated to it.*

When a token arrives at the entry control port, the thread frame is enabled to fire. After firing, the token is removed from the entry control port and a token is put on all outgoing edges for which the associated guard expressions are true. Three essential edge types exist: control flow edges carrying tokens, data flow edges showing access to shared memory variables and constraint edges specifying timing constraints.

According to Def. 1 no non-determinism, dynamic task creation, event handling, resource contention or synchronisation are allowed inside a thread frame, while at the borders of a thread frame, at least one such construct should exist. The aforementioned constructs constitute thread frame borders in the application.

At the lower layer, each thread frame is decomposed in a graph of *thread nodes*, the basic design-time scheduling units in TCM. Thread nodes hide control and data flow information irrelevant to the scheduler:

Definition 2. *A thread node T is a maximal set of connected operations with a deterministic execution latency $\Lambda(T) = [\delta(T); \Delta(T)]$, where $\delta(T)$ and $\Delta(T)$ are the minimum and maximum execution time of the thread node T respectively. Execution time variance of the thread node T is given by $\sigma(T) = \Delta(T)/\delta(T)$. It is a small number less than some user-defined variance threshold σ_{max}. Associated with the thread node is the CDFG representation of the set of operations it contains.*

B. Guidance rules for extraction of thread nodes

Identification of thread node borders has not yet been formalized. It has been discussed that a finer thread node granularity would lead to more possible optimal schedules to choose from at run-time. On the other hand, run-time scheduler overhead and resource consumption would increase. A resource consumption model of the run-time scheduler is required to consider this effect and make appropriate decisions for this trade-off [5]. Moreover, the following informal guidelines for extraction of thread nodes were specified:

1. *Deciding the thread node granularity range*
   When analyzing different applications from the multimedia, wireless and biomedical domains, it can be observed that the fastest dynamic event never occurs more often than once per few milliseconds. In the wireless domain, the fastest changes are caused by Doppler effects in the channel, while the channel is considered to be constant during coherence time ranging from ten to hundreds of milliseconds. In the multimedia domain, for instance video coding applications, dynamism resides in changing of frames, which happens at a typical rate of 30 frames/second. Inside a frame dynamic changes can occur, but no more than a dozen of them. Biomedical applications are operating on human body signals, which have frequencies below 1 kHz, for example: EMG - 7-20 Hz, ECG - 0.05-100 Hz, EEG - 1-100 Hz.

   This means that the run-time scheduler should be activated at a minimum interval of around 1 ms. A higher response rate would introduce too much overhead at run-time while a lower maximal response rate, like that of a general operating system, cannot always react to the dynamically changing system load, hence cannot manage the system in a guaranteed and energy efficient way.

   Within a thread frame tens to hundreds thread nodes can exist for modern applications. Thus a thread node should have an execution time between 10μs to 100μs on the target platform.

2. *Identifying candidate thread nodes*
   The following types of code segments have been identified as candidates for thread nodes:

   (a) A loop body with an execution time between 10μs and 100μs. On the target platform (200 MHz) this execution will require 2 000 - 20 000 clock cycles. This offers possibilities for parallelism when no dependency exists between loop iterations. This parallelism resides at the medium grain, between the fine grain (instruction-level) parallelism already exploited by compilers and the coarse grain parallelism that is typically exploited at real-time operating system level. If dependencies exist between the loop iterations, pipelining is still possible by applying loop folding and lookahead transformations.

   (b) A loop, of which the bound, i.e., number of iterations, is so large that the total loop execution time becomes dominant, e.g., 20% of the thread frame execution time.

   (c) A data-dependent loop, which first should count for more than 5% of the total thread frame execution time in the worst case to be relevant, and of which the worst case and best case execution times differs by more than a factor of N. N is determined experimentally on the target platform to see which variations lead to significant execution time overhead. For our platform N = 3 has been determined.

(d) Branches in conditional statements, if choosing a particular branch results in a large difference in execution time, e.g., N times difference, when compared to choosing the other branch.

(e) Function calls with an execution time counting for more than 5% of the total thread frame execution time in the worst case to be relevant.

(f) Important data accesses which bring dynamic behavior that is addressed by a task-level data transfer and storage exploration [8].

3. *Checking execution time variance of a thread node*
If the execution time variance of the thread node, $\sigma(T)$, exceeds the threshold N, the thread node T still contains a data dependent construct that needs to be visualized, and it must either be split or be attached with different working scenarios, based on input data. The node is marked for transformation.

4. *Checking dominant execution time*
If regarding the execution time, a dominant thread node exists in the thread frame, this thread node should be split and if possible parallelized. The node is marked for transformation.

5. *Checking dominated execution time*
If inside a thread frame, a thread node accounts for very little execution time, it should be merged with surrounding thread nodes. The node is marked for transformation.

6. *Executing node transformations.*
When all thread nodes to be split or merged are identified, the node transformations are performed in the source code. However, no methodology for performing split of excessively variant nodes exists. Developing such methodology has been the main focus of the current work and the proposed technique for thread node splitting is described in Sec. IV.

7. *Verifying execution time range and variance*
Finally, the execution times and variances of the split nodes are checked, and if they do not satisfy guidance rule 1, 4 and/or 5, additional transformations are performed until they do.

III. Lyapunov Exponent calculator and its initial gray box model

The Lyapunov Exponent calculator is a real-time algorithm used as part of an epileptic seizure predictor [7]. It collects data samples `X[2048]` from a number of EEG electrodes every 10.24 s, presents them in a 7-dimensional phase space `Z[2024][7]` and computes the rate of separation of neighboring trajectories using an iterative procedure that traverses the phase space representation a large number of times. The obtained rate of separation is called the Lyapunov Exponent, a characteristic of a dynamic system, in this case the human brain, that quantifies system predictability. Prior to an epileptic seizure, the Lyapunov Exponent tends to decrease, starting from the electrodes nearest to the brain location where the seizure originates. By analyzing pairwise differences in consecutive Lyapunov exponents between the electrodes, the epileptic seizure predictor can foresee the upcoming seizures and warn the patient. The algorithm is data-dominated and includes dynamic behavior: some calculations are skipped and others are repeated based on variables obtained from the input EEG samples.

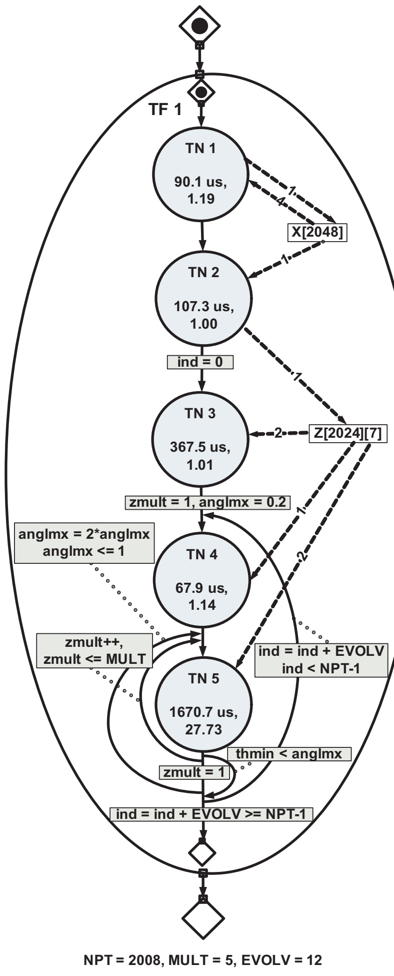

Fig. 1. Thread node candidates for the Lyapunov exponent calculator.

To obtain a gray box model for the Lyapunov Exponent calculator we start from the definitions of the thread frame and the thread node from Sec. II-A. First, thread-frame borders are identified. The Lyapunov Exponent calculation for one electrode has no non-deterministic and operating system behavior and can be put entirely inside a single thread frame, see Fig. 1.

To identify thread nodes, we perform profiling of the Lyapunov Exponent calculator on a reference VLIW processor platform using a publicly available EEG database

```
for(ind = 0; ind < NPT-1; ind = ind + EVOLV){
    ...
    ......... TN 5 start ..................
    for(i = 0; i < NPT; i++){
        difind_check = f(ind,i);
        if (difind_check) continue;
        dnew_check = g(Z,i);
        if (dnew_check) continue;
        B;
    }
    ......... TN 5 end ...................
    ...
}
```

Fig. 2. Initial code structure of TN 5.

of the University of Bonn, Germany [9]. The reference VLIW processor is not optimized for the application, it has three issue slots and no pipeline stages; it is designed in a standard ASIC design flow, 90 nm low-leakage CMOS technology (CLN90LP) and runs at 100 MHz. The obtained execution time data are scaled up by a factor of 10 to include the expected speedup on the target VLIW processor (200 MHz) after optimization.

We then apply guidance rules 1 and 2(a)-(f) for thread node extraction from Sec. II-B and separate the application code inside the thread frame into five parts shown on Fig. 1. Each circle represents a thread node (TN) candidate, and its worst case execution time $\Delta(T)[\mu s]$ and the execution time variance σ_T are indicated within the circle. Thread node candidates TN 1 and TN 4 contain maximum connected code with execution time within the granularity range $[10\mu s, 100\mu s]$ (guidance rule 1). Thread node candidates TN 2 and TN 3 are single loops with large amount of iterations (≈2000) (guidance rule 2b), and TN 5 is a data-dominated loop (guidance rule 2c).

Checking the execution time variance of the thread node candidates (guidance rule 3), we observed that TN 5 have execution time variance exceeding the threshold variance of N = 3. The variance in TN 5 is caused by two conditions inside the loop, `difind_check` and `dnew_check`, as shown on Fig. 2. They control execution of a code block B with approximately constant execution time ($\sigma < 3$). Each condition can vary the execution time of the node more than 3 times. Note that the borders of TN 5 is indicated with TN 5 start and TN 5 end in Fig. 2.

Condition `difind_check` has the biggest impact on the execution time of TN 5 since it enables/disables the biggest portion of the thread node code (in terms of execution time) among all conditional statements in the node. This condition is calculated from the control variables `ind` and `i`. Function `f` and values of the control variables enabling `difind_check` condition are given in Fig. 3.

Condition `dnew_check` has the second biggest impact. It is calculated from the `Z` array and thus depends on the input data `X` of the application. The remaining code contains another condition `th_check` (not shown), but its impact on the execution time of the thread node (σ) is less than 3 times.

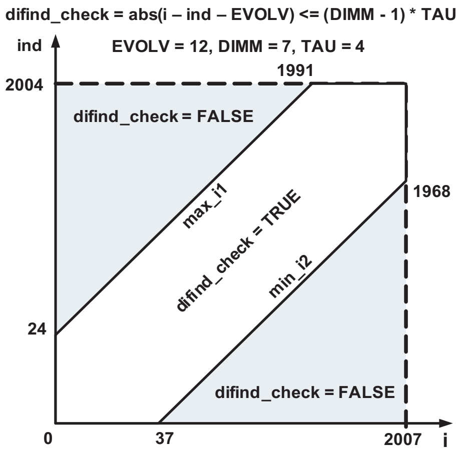

Fig. 3. Control-dependent condition.

According to the guidance rule 3, TN 5 should be split and parallelized, and the data dependent condition `dnew_check` should be visualized for further scenario analysis. Sec. IV describes our approach for performing splitting of TN 5 and extracting its data-dependent condition using code transformations.

IV. Removal of execution time variance by node splitting

In this section we use a new technique to split the thread nodes in such a way that they are compliant with the granularity range indicated in Sec. II-B. This makes it possible for the dynamic scheduler to meet the stringent low energy and execution time constraints of the medical appliance performing the epileptic seizure prediction.

To reduce the execution variance of TN 5 we move the two conditions `difind_check` and `dnew_check`, being the main causes of the variance, outside the node.

We first consider the control-dependent condition `difind_check`. From Fig. 3 it can be seen that `difind_check` condition is false in two shaded disjoint regions of the (`ind`, `i`) space. In these regions the TN 5 code following the `difind_check` condition is enabled. In the white region the `difind_check` condition is true and the only code running in TN 5 is calculation of `difind_check`.

We move the `difind_check` calculation outside the `i` loop where we replace it with a calculation of the two region borders, max_i1 and min_i2. The `i` loop is then split into two separate loops iterating over the disjoint regions as illustrated in Fig. 4. The bodies of the new loops are now placed in separate thread nodes, TN 5_1 and TN 5_2, while the loops around them are moved to the level visible in the gray box model.

The code for calculation of the `difind_check` condition is now reduced to finding the borders `max_i1` and `min_i2` of the regions where the `difind_check` condition is false

```
for(ind = 0; ind < NPT−1; ind = ind + EVOLV){
    ...
        {max_i1 , min_i2} = h(ind);

        for(i1 = 0; i1 < max_i1; i1++){
            ......... TN 5_1 start .........
            dnew_check = g(Z,i1);
            if (dnew_check) continue;
            B;
            ......... TN 5_1 end  ..........
        }

        for(i2 = min_i2 + 1; i2 < NPT; i2++){
            ......... TN 5_2 start .........
            dnew_check = g(Z,i2);
            if (dnew_check) continue;
            B;
            ......... TN 5_2 end  ..........
        }
    ...
}
```

Fig. 4. Splitting of TN 5 node based on the control-dependent condition `difind_check`.

(see Fig. 3). Since the execution time of this calculation is small, it can be merged with TN 4.

The new thread nodes TN 5_1 and TN 5_2 (Fig. 4) have now execution times that are independent of the `difind_check` condition. On the other hand, the nodes will run a variable number of times defined by the values of `max_i1` and `min_i2`, representing the `difind_check` condition. However, these values are known before the execution of TN 5_1 and TN 5_2 and available at the gray box model level. This variance can then be handled by the design-time scheduler, which will schedule the thread nodes the appropriate number of times. Also, for this application the new loops around TN 5_1 and TN 5_2 can be easily parallelized by conventional techniques [1].

The remaining source of variance in TN 5_1 and TN 5_2 is the `dnew_check` condition. For this condition on/off regions depend on the application's input data and are not known in advance. As outlined in the beginning of Sec. II, the application is analyzed for all possible inputs and similar behaviors of the data-dependent code in terms of execution time and energy consumption are identified. These similar behaviors are clustered in a set of scenarios and scheduling is done for each scenario separately. At run-time the active scenario is detected and the appropriate schedule is selected. The scenario identification is an ongoing research and it requires the data-dependent conditions to be visible in the gray box model.

To make the `dnew_check` condition visible outside node TN 5_1, the thread node borders can be moved inside the condition as shown on Fig. 5, so it becomes visible in the model. The code for calculation of the condition and other uncovered code after the border movement is placed in a new thread node TN 4_51. The same procedure can be done on TN 5_2, and the outcome is also readily parallelisable by conventional techniques [1].

```
for(ind = 0; ind < NPT−1; ind = ind + EVOLV){
    ...
        {max_i1 , min_i2} = h(ind);

        for(i1 = 0; i < max_i1; i++){
                ....... TN 4_51 start  ......
                dnew_check = g(Z,i1);
                ........TN 4_51 end ..........
                if (dnew_check) continue;
                ....... TN 5_1 start .........
               B;
                ....... TN 5_1 end ...........
                }
        }
    ...
}
```

Fig. 5. Movement of data-dependent condition `dnew_check` outside TN 5_1 node.

The final gray box model of Lyapunov Exponent calculator with split thread node TN 5 is presented in Fig. 6. It can be observed that TN 5 is now presented by four nodes: TN 4_51, TN 5_1, TN 4_52 and TN 5_2 and that both conditions that caused execution time variance larger than N=3 are now visible in the gray box model and thus can be handled in an ordered way by the design-time and run-time schedulers. The `dnew_check` condition steers the transitions between TN 4_51 and TN 5_1, and between TN 4_52 and TN 5_2. The `difind_check` condition controls the bounds `max_i1 and min_i2` of the loops around these nodes. The execution time variances of the new nodes are below the threshold N = 3.

V. Verification of the model

After splitting of TN 5 the gray box model of Fig. 1 have no large variance nodes, i.e. they all confirm the requirement of $\sigma(T) < 3$ but some of the nodes have execution times outside the specified granularity region [10μs, 100μs] (Sec. II-B): TN 2, TN 3 and the the four new nodes that appeared after the split of TN 5.

We split TN 2 and TN 3 nodes by dividing the iteration region of the loop they contain in parts. Each new node contains the same loop body as the original node, but iterates over a part of the original iteration region, such that its execution time satisfies the granularity region. The resulting nodes are shown in the final gray box model in Fig. 6. TN 2 is split in two nodes TN 2_1 and TN 2_2, while TN 3 is split in four nodes: TN 3_1, TN 3_2, TN 3_3 and TN 3_4. All of them have now execution times within the granularity region.

The execution times of nodes TN 4_51, TN 5_1, TN 4_52 and TN 5_2 are too small. However this is not a problem: they will be repeateadly executed in scenarios. In the scenario analysis the execution pattern of the loops around the nodes will be studied for sequences of the data-dependent `dnew_check` values. Each scenario will thus represent a sequence of repeated executions of the nodes and the length of the sequence will be selected to give an acceptable execution time.

VI. Conclusions

The final gray box model in Fig. 6 satisfies all requirements of the gray box model in TCM mapping methodology and can be scheduled on a platform with multiple VLIW processors running at speeds from 100 MHz to 200 MHz. We have verified that the Task Concurrency Methodology can be used on a biomedical application and presented a new technique for splitting of thread nodes in order to reduce their execution time variance.

Acknowledgements

The Lyapunov Exponent calculator was developed at the Department of Bioengineering, the Arizona State University, Tempe, AZ, USA. The authors would like to thank Prof. Leon D. Iasemidis and Dr. Shivkumar Sabesan for providing us the possibility to use their application code.

References

[1] MPSoC Summer School, http://www.mpsoc-forum.org

[2] Z.Ma, P.Marchal, D.P.Scarpazza, P.Yang, C.Wong, J.I.Gomez, S.Himpe, C.Ykman-Couvreur, F.Catthoor, *Systematic Methodology for Real-Time Cost Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogeneous Platforms,* Springer, 2007.

[3] C.Wong, *Design-Time Sub-Task Scheduling For Embedded Multimedia And Telecom Systems,* PhD thesis, Katholieke University, Leuven, Belgium, 2003.

[4] S.Himpe, *Platform Independent Source Code Transformations For Task Concurrency Management,* PhD Thesis, Katholieke University, Leuven, Belgium, 2006.

[5] Z.Ma, *Interleaved Subtask Scheduling On Multiprocessor SOC,* PhD Thesis, Katholieke University, Leuven, Belgium, 2006.

[6] V.Gheorghita, M.Palkovic, J.Hamers, A.Vandecappelle, S.Mamagkakis, T.Basten, L.Eeckhout, H.Corporaal, F.Catthoor, F.Vandeputte, K.De Bosschere, "System scenario based design of dynamic embedded systems", *ACM Trans. on Design Automation for Embedded Systems (TODAES)*, Vol.14, No.1, article 3, Jan. 2009.

[7] L.D.Iasemidis et al., "Long-term prospective on-line real-time seizure prediction," *Clinical Neurophysiology*, vol. 116, pp. 532–544, 2005.

[8] P.Marchal et al., "SDRAM-energy-aware data allocation for dynamic multi-media applications on multiprocessor platforms," *DATE*, Proceedings, pp. 516–521, 2003

[9] R.G.Andrzejak, K.Lehnertz, F.Mormann, C.Rieke, P.David and C.E.Elger, "Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state," *Phys. Rev. E*, vol. 64, pp. 061907-(1–8), 2001.

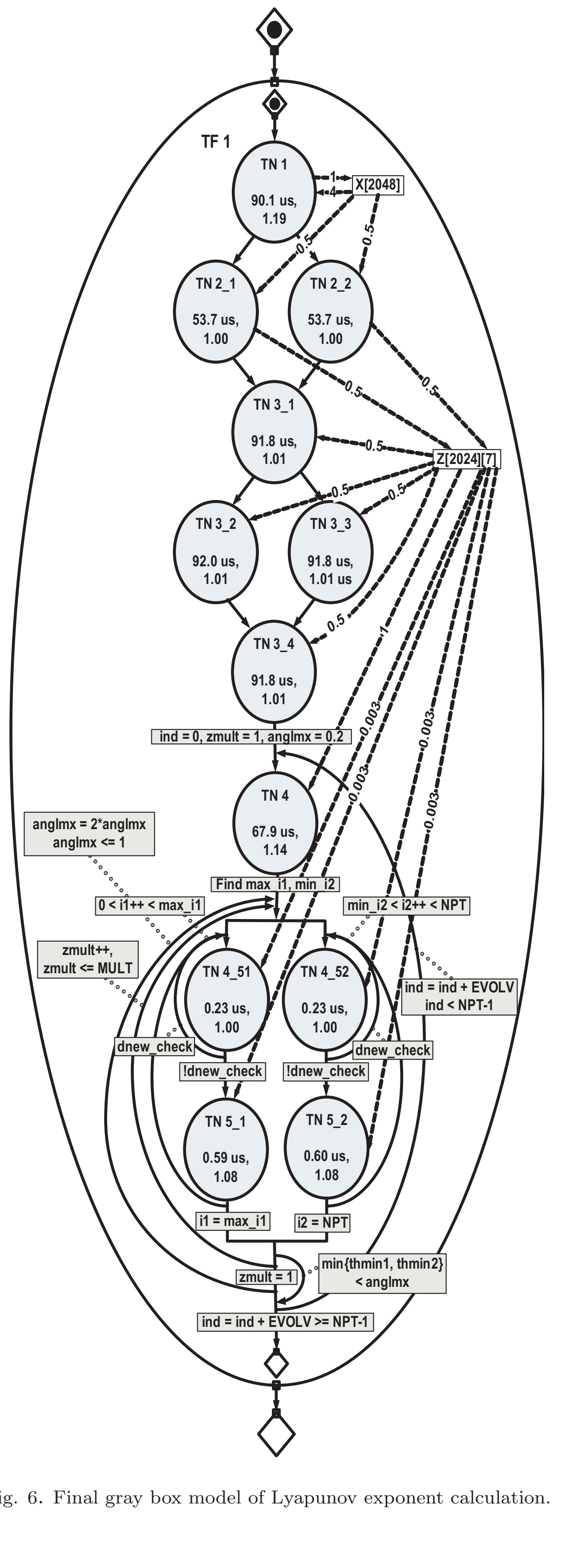

Fig. 6. Final gray box model of Lyapunov exponent calculation.