On Task Tree Executor Architectures Based on Intel Parallel Building Blocks

Miroslav Popovic[1], Miodrag Djukic[1], Vladimir Marinkovic[1], and Nikola Vranic[2]

[1] Faculty of Technical Sciences, Trg D. Obradovića 6,
21000 Novi Sad, Serbia
{miroslav.popovic, miodrag.djukic, vladimir.marinkovic}@rt-rk.com
[2] RT-RK Computer Based Systems LLC, 27 Narodnog fronta 23a,
21000 Novi Sad, Serbia
nikola.vranic@rt-rk.com

Abstract. Our aim was to optimize a SOA control system by evolving the architecture of the service component that transforms system models into task trees, which are then executed by the runtime library called the Task Tree Executor, TTE. In the paper we present the two novel TTE architectures that evolved from the previous TTE architecture and introduced finer grained parallelism. The novel architectures execute TTE tasks as more lightweight TBB tasks and Cilk strands rather than the OS threads, which was the case for the previous TTE architecture. The experimental evaluation based on time needed for TTE reliability estimation, by statistical usage tests, shows that these novel TTE architectures are providing the average relative speedup, RS, from 8x to 11x, over the original TTE, on a dual-core machine. Additional experiments made on eight-core machine showed that RS provided by TTE based on TBB scales perfectly, and goes up to 77x.

Keywords: service oriented architecture, architecture evolution, task trees, parallel programming, parallel building blocks.

1. Introduction

Providing proper parallel data processing is one of the greatest challenges to be dealt with when designing software solutions for management of critical infrastructures, such as oil, gas and electricity distribution systems. The main task of these systems is to provide continuous system supervision and control, based on data acquisition and processing, while fulfilling high availability, reliability, and security standards. Additionally, numerous economic, serviceability, and maintainability aspects regarding different operational activities must be addressed as well. Nowadays all these requirements are typically satisfied by a Service Oriented Architecture (SOA)

based system comprising a complex suite of service components, thus guiding the system designer to the set of necessary data and functionalities that need to be simultaneously served [1-2].

One of the most complicated components for design is a service component that provides various calculations using various models of the system, which commonly take a form of a graph or a tree. The examples of such calculations for the electricity distribution system are network topology analysis, load flow calculation, network state estimation, performance indices, etc. The main factors that are complicating the design of this kind of service components are that these nontrivial calculations have to be performed on large-scale graph models and near to real-time. Designers are also frequently facing the additional economic limitation that they have to somehow reuse legacy software, because of its enormous size – typically millions of lines of FORTRAN code that were developed over a couple of decades.

In our previous work we have used two approaches to design and develop such service components for the electricity distribution systems. The first approach [3-4] is based on: (i) transforming network models into task trees that are managed by Task Tree Executor (TTE) and (ii) refactoring legacy code by introducing callback functions that are executed as TTE tasks. The second approach [5] is based on: (i) repackaging legacy code as DLLs (Dynamic Linkable Libraries) and (ii) executing them as parallel applications by Calculation Engine (CE). The advantage of the second approach is that it requires less development effort and that it is more robust, but the advantage of the first approach is that it provides more parallelism, because it is finer grained than the second approach. TTE runs TTE tasks as separate threads, whereas CE launches the application DLLs within separate processes.

The goal of the work presented in this paper was to evolve the TTE architecture based on threads into the two new TTE architectures based on Intel Parallel Building Blocks (PBB) in order to provide even finer grained parallelism. The first novel TTE architecture is based on Intel Treading Building Blocks (TBB), whereas the second novel TTE architecture is based on Intel Cilk Plus (Cilk). Essentially, the TTE tasks that were executed as threads by the previous TTE [3-4] are now executed as more lightweight TBB tasks by the TTE based on TBB, or as Cilk strands by the TTE based on Cilk.

The advantages of this TTE architecture evolution are threefold. Firstly, both TBB and Cilk are known of being able to provide better multicore CPU utilization than the local OS, such as MS Windows or Linux (i.e. TBB and Cilk can better parallelize their tasks/strands than OS can parallelize its threads). Secondly, both TBB and Cilk provide almost infinite number of tasks, whereas the local OS provides rather limited number of threads within a process (the order of couple of thousands of threads at maximum). Thirdly, the explicit and rather suboptimal CPU load control within the previous TTE architecture is now delegated to the excellent load balancing functionality of the TBB and Cilk runtime libraries within the two novel TTE architectures.

The content of this paper is organized as follows. The related work and the description of target class of software systems that are addressed by the proposed solutions are presented in Subsections 1.1. and 1.2, respectively.

The three TTE architectures are described in Section 2, which is divided into the three subsections. The previous TTE architecture based on OS threads is described in Subsection 2.1, the novel TTE architecture based on TBB is described is Subsection 2.2, and the novel TTE architecture based on Cilk is described is Subsection 2.3. The statistical usage testing method and the results of the experimental evaluation of novel TTE architectures are presented in Sections 3 and 4, respectively. The latter Section 4 is partitioned into the five subsections, which are covering the baseline performance, the performance of the TTE architecture based on TBB, the performance of the TTE architecture based on Cilk, the scalability check for the TTE architecture based on TBB, and the threats to validity of experimental results, respectively. Final conclusions are given in Section 5.

1.1. Related Work

The next two subsections discuss work related to the TTE architectures based on Intel TBB and Intel Cilk Plus, respectively.

Work Related to the TTE Architecture Based on Intel TBB

TBB uses templates for common parallel iteration patterns, enabling programmers to attain increased speed from multiple processor cores without having to be experts in synchronization, load balancing, and cache optimization (see [6]). Generally, TBB provides more comfort to programmers and better results in terms of program speedup when compared to the practice of using raw threads, which was also the case in our particular work presented in this paper.

Two features of TBB that provide the foundation for its robust performance are the TBB work-stealing scheduler and the TBB scalable memory allocator. To prove that, Kukanov and Voss [7] used experiments on several benchmarks to demonstrate the potential scalability of TBB based applications and to show that the TBB allocator is competitive with other allocators.

One of the key advantages of a logical task is that it is much lighter than a thread, e.g. starting and terminating a task on Linux is around 18 times faster than starting and terminating a thread, whereas on Windows, this ratio is more than a 100 (see [8]). Additionally, TBB manages these light units of work very efficiently. Bhattacharjee et al. [9] used real hardware and simulations to detail various scheduler and synchronization overheads in order to assess these overheads on TBB and OpenMP. They found that these can amount to 47% of TBB benchmark runtime and 80% of OpenMP benchmark runtime, i.e. TBB is almost as twice as better when compared to OpenMP in respect to scheduling and synchronization overheads.

Because of all of its features mentioned above, TBB is finding successful applications in various soft real-time systems, sometimes also called near to

real-time systems. One type of such systems is the system managing critical infrastructure, which we are primarily interested in. Another type of such systems is modern video games. Although it might come surprisingly, these two types of systems have much in common. They both use physical models and AI components, they both have real-time requirements, and they both may be classified as complex system.

A significant effort has been made to demonstrate TBB's applicability in modern video games industry. For example, Werth [10] provided useful instructions and hands-on examples on optimizing games architectures with TBB, in his talk on the recent game developer's conference in San Francisco. Several other groups joined this R&D track by trying to create adequate parallel programming frameworks (PPFs) for video games engines.

One notable example in that direction is Cascade, a PPF for video games engine, developed by Tagliasacchi et al. [11]. In Cascade, Cascade tasks are linked by dependencies in a task dependency graph, which is traversed at runtime by the Cascade Job Manager (CJM) that assigns tasks to threads for execution. CJM does this rather efficiently, for example the Cascade implementation of Sequence Alignment algorithm completes 1.5 times quicker than the OpenMP implementation.

As pointed out by the creators of Cascade, TBB is closest in spirit to their system, when compared with other PPFs. However, their claim that TBB does not support explicit construction of task graphs and that graphs are constructed only recursively via spawn call is actually not true. On the other hand Cascade is also very similar to our TTE architecture. The main difference is that Cascade supports acyclic task graphs, whereas TTE supports task trees.

Work Related to the TTE Architecture Based on Intel Cilk Plus

Original Cilk programming language appeared as an extension of C providing language constructs for parallel control and synchronization. This extension was made to be very efficient in terms of runtime overheads, for example the typical cost of spawning an OS thread is 2-6 times the cost of the C function call on a variety of modern processors (see [12]). Once spawned, these parallel threads are scheduled very efficiently on a shared memory multiprocessor (SMP), by the Cilk scheduler that is based on the *work stealing* scheduling method. Blumofe and Leiserson [13] showed that the expected time to execute a well-structured computation on P processors using their work-stealing scheduler is $T_1/P + O(T_\infty)$, where T_1 is the minimum serial execution time of the multithreaded computation and T_∞ is the minimum execution time with an infinite number of processors.

Original Cilk language has been developed, as an ANSI C extension, since 1994 at the MIT. A commercial version of Cilk, called Cilk++, that supports both C and C++, was developed by Cilk Arts, Inc. In 2009, Intel Corporation acquired Cilk Arts, the Cilk++ technology and the trademark. In 2010, Intel released a commercial implementation in its compilers under the name Intel

Cilk Plus. In this paper we use Intel Cilk Plus and refer to it later in the text briefly as Cilk.

Other authors have been successfully using Intel Cilk Plus before us. For example, Kirkegaard and Aleen [14] studed the potential of individual optimizing techniques in terms of speedup. They applied 5 techniques on the Google's AOBench benchmark, to achieve the overall 16.47x speedup.

Similarly, Luk et al. [15] used Intel Cilk Plus to demonstrate their synergetic approach to throughput computing. The experimental results they collected on a dual-socket quad-core Nehalem show that their approach achieves an average speedup of almost 20x over the best serial cases for an important set of computational kernels.

Finally, Agrawal et al. [16] developed the Nabbit, a work-stealing library for execution of task graphs with arbitrary dependencies. They evaluated the performance of Nabbit using a dynamic program representing the Smith-Waterman algorithm. Their results indicate that when task-graph nodes are mapped to reasonably sized blocks, Nabbit exhibits low overhead and scales as well as or better than other scheduling strategies. Interestingly, Nabbit is rather similar both to Cascade and to the TTE architectures presented in this paper. The main difference among them is that Nabbit and Cascade support acyclic task graphs, whereas TTE supports task trees.

1.2. Target Environment

Infrastructure of an industrial control system (ICS) consists of domain specific equipment and smart devices that are connected to Remote Terminal Units (RTUs), which are used for monitoring and control of an industrial process. A modern large scale industrial system typically uses Supervisory Control and Data Acquisition (SCADA) system as a front-end for communication with a network of RTUs.

A separate and independent environment may be laid on top of any SCADA system. This layer, called Intelligent Control System (ICS), encompasses necessary control logic and process related intelligence, which can be very complex. The structure of ICS is shown in Fig. 1. Seen from the SOA standpoint, system exposes the Master Data Service (MDS) that stores logically consistent dataset describing existing elements of the system infrastructure. Besides MDS, other services used for enterprise integrations may be exposed, depending on ICS's role and purpose.

Each of the service components (SCs) in ICS is managing exactly one aspect of the overall system functionality, such as dynamic data management, performing necessary calculations, providing graphical representation of infrastructure elements, etc. The SC providing TTE driven parallel calculations on a given Operational Model (OM) is in the focus of this paper (it is labeled as C-TTE in Fig. 1). OM is a dataset that stores the model of the system. This dataset must be correct, e.g. if ICS manages an electricity distribution system, at least it has to satisfy the first and the second Kirchhoff law.

Interaction between the system user and the services is provided through a thin client application. Usually, there is no demanding data processing inside the client application itself, its only responsibility is to obtain data from particular service in the system periodically, or on user demand.

When the system evolvement aspects are taken into the consideration, one of the most important design goals is to provide the plug-and-play like integration capabilities. Therefore, the main internal communication backbone is designed in accordance with the publisher/subscriber paradigm, which provides loose coupling between service components and services.

For synchronous, point-to-point calls (represented with dashed arrows in Fig. 1), interfaces are provided to allow data access by other services and by external UI clients. However, the communication within the system is predominantly asynchronous, based on publishing and subscribing to different message topics (represented with full arrows in Fig. 1). An important aspect regarding the communication in the system is the fact that all the datasets describe the current infrastructural state of the system, which changes over time.
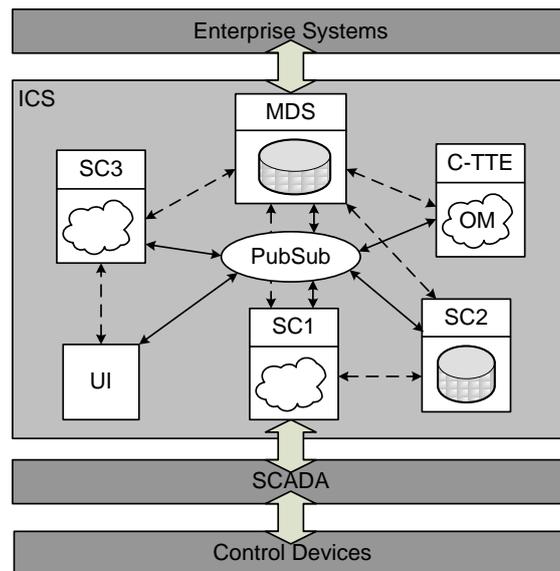
Fig. 1. Target Environment

2. The Three TTE Architectures

The next three subsections present the original TTE architecture based on OS threads, the TTE architecture based on Intel TBB [17], and the TTE architecture based on Intel Cilk Plus [18].

2.1. TTE Architecture Based on Threads

As shown in Fig. 2.a, an application that was refactored from legacy software to operate on slices of system model, which correspond to individual TTE tasks, does that by making use of the TTE application programming interface (API). The TTE API provides the following functions [3]:

1. TS_CreateTaskGraph
2. TS_AddTask
3. TS_DeleteTask
4. TS_SetBottomUpProcFun
5. TS_SetTopDownProcFun
6. TS_ExecuteBottomUp
7. TS_ExecuteTopDown
8. TS_DestroyTaskGraph
9. TS_ExecuteBottomUpSequentially
10. TS_ExecuteTopDownSequentially

Fig. 2. The three TTE Architectures: (a) Thread-based, (b) TBB-based, (c) Cilk-based

The API function *TS_CreateTaskGraph* creates the task graph; its parameters are the identification (ID) of the root task, the pointer to the bottom-up processing function, the pointer to the top-down processing function, and the maximal number of local OS threads (pthreads) that will be used to execute the task graph in parallel. The bottom-up processing function and top-down processing function are the callback functions, which have the task ID as their parameter. The API function *TS_AddTask* adds a new task to the task graph, given the ID of the predecessor task and the ID of the new

task. The API function *TS_DeleteTask* deletes the given task and all of its successors from the task graph.

The API function *TS_SetBottomUpProcFun* redefines the pointer to the bottom-up processing function, whereas the API function *TS_SetTopDownProcFun* redefines the pointer to the top-down processing function. The API function *TS_ExecuteBottomUp* executes the task graph bottom-up in parallel, whereas the API function *TS_ExecuteTopDown* executes the task graph top-down in parallel. Finally, the API function *TS_DestroyTaskGraph* deletes the task graph.

The last two API functions are used only for the debugging and benchmarking purposes. The first one of them executes the task graph bottom-up sequentially, whereas the second one executes the task graph top-down sequentially. The most frequently used functions are the functions no. 1, 2, 6, 7, and 8. The function no. 3 is used to change the existing task graph, while the functions no. 4 and 5 are used only to redefine the callback functions.

The TTE architecture based on threads comprises two main components, namely the C module *TaskScheduler* (shown as a class in Fig. 2 and Fig.3 for the sake of standard UML representation) and the class *Task*. The module *TaskScheduler* provides the TTE API by exporting its public functions, as listed and discussed above. Internally, this module hides the pointer to the task tree root and the pointers to the callback functions as its private (static) data. As a reaction to external application calls to the functions *TS_CreateTaskGraph*, *TS_AddTask*, *TS_DeleteTask*, and *TS_DestroyTaskGraph*, the module *TaskScheduler* builds and maintains the task tree by adding and deleting instances of the class *Task*.

The class *Task* provides two field members that enable building task trees. These are the pointer to the predecessor task and the list of the successor tasks in the task tree. The function *TS_AddTask* adds a new task by (i) locating its predecessor task, (ii) setting the new task's predecessor field to the address of the predecessor task, and (iii) adding the address of the newly created task to the list of the successor tasks in the corresponding field member of the predecessor task. Deleting a task from the task tree is more complex because deleting a given task means deleting itself and all its successors, and the successors of the successors, i.e. it means deleting the complete sub-tree from the given task and below it.

When it comes to parallel task tree execution, the API function *TS_ExecuteTopDown* starts task tree top-down execution by calling the class *Task* member function *executeTDinParallel* on the root task, which in turn recursively traverses the task tree from its top, i.e. root task, downwards across all the successors, until it reaches all the task tree leafs. In each recursion, this function first calls the top-down callback function and then it starts new local OS threads for each of the current task's successors by calling the *PthreadsLibrary* function *CreateThread* (using the pthreads API). The simplified pseudo code of the class *Task* member function *executeTDinParallel* is the following:

```
executeTDinParallel(task) =
    callback tdCallback(task.id)
    for each successor in task.successors
        CreateThread(executeTDinParallel, successor)
    WaitForAllChildTherads()
```

Similarly, the API function *TS_ExecuteBottomUp* starts task tree bottom-up execution by calling the class *Task* member function *executeBUinParallel* on the root task, which in turn recursively traverses the task tree from its top, i.e. root task, downwards across all the successors until it reaches all the task tree leafs. In each recursion, this function first starts new local OS threads for each of the current task's successors by calling the *PthreadsLibrary* function *CreateThread* (over the pthreads API) and then it calls the bottom-up callback function. The simplified pseudo code of the class *Task* member function *executeBUinParallel* is the following:

```
executeBUinParallel(task) =
    for each successor in task.successors
        CreateThread(executeBUinParallel, successor)
    WaitForAllChildTherads()
    callback buCallback(task.id)
```

2.2.   TTE Architecture Based on Intel TBB

The simplified architecture of the complete system that is based on Intel TBB is shown in Fig. 2.b. As shown in Fig. 2.b, the novel TTE architecture based on Intel TBB is almost the same as the previous TTE architecture based on threads. The main difference between these two architectures at the high-level architectural view used in Fig. 2 is that the novel TTE architecture makes use of the Intel TBB runtime library rather than using the local OS (MS Windows or Linux) pthreads library, as was the case in the previous architecture. This evolutionary step was essentially made by modifying the class *Task* such that the member functions *executeTDinParallel* and *executeBUinParallel*, which were responsible for the parallel task tree execution, in the novel architecture delegate parallel top-down and bottom-up task tree execution to new member functions *executeTDbyTBB* and *executeBUbyTBB*, respectively.

This modification was completely transparent to the module *TaskScheduler*, thus the way it builds and maintains the task tree remained unchanged, as well as the way it starts parallel top-down and bottom-up task tree execution. Moreover, and even more importantly, this modification within the TTE architecture was completely transparent to the legacy applications. This was of utmost importance, because legacy applications are so huge in size, they may literally comprise millions of lines of code.

Although, at the high-level of abstraction, simplified pseudo code for the member functions *executeTDinParallel* and *executeBUinParallel* from the previous architecture remains valid for new member functions *executeTDbyTBB* and *executeBUbyTBB*, respectively, implementing them in C++ naturally required using TBB design patterns. More precisely, since TBB tasks are created as instances of C++ classes extending the TBB library class *task*, two auxiliary classes were introduced, namely the class *TbbTaskTD* and the class *TbbTaskBU*. The former is used by the member function *executeTDbyTBB*, whereas the latter is used by the member function *executeBUbyTBB*.

Both of these auxiliary classes are rather simple. Since each TTE task within a TTE task tree is assigned a TBB task, both of these auxiliary classes have a field member that stores the corresponding TTE task. These field members are normally set by the class constructors. The *execute* methods of both auxiliary classes simply call the corresponding new *Task* member function, in particular the *TbbTaskTD* member function calls the *Task* member function *executeTDbyTBB*, whereas the *TbbTaskBU* member function calls the *Task* member function *executeBUbyTBB*.

Once these auxiliary classes were introduced, synthesizing new *Task* member functions was rather straightforward. Concretely, the simplified pseudo code of the function *executeTDbyTBB* is the following:

```
executeTDbyTBB(task) =
   callback tdCallback(task.id)
   if task.successors == Ø  return
   et = TbbTaskTD(null)
   et.set_ref_count(1)
   for each successor in task.successors
      et.increment_ref_count()
      et.spawn( new TbbTaskTD (successor) )
   et.wait_for_all()
   task::destroy(et)
```

In the pseudo code above the name *et* stands for the *empty task*. Similarly, the simplified pseudo code of the function *executeBUbyTBB* is completely symmetrical:

```
executeBUbyTBB(task) =
   if task.successors != Ø
      et = TbbTaskBU(null)
      et.set_ref_count(1)
      for each successor in task.successors
         et.increment_ref_count()
         et.spawn( new TbbTaskBU (successor) )
      et.wait_for_all()
      task::destroy(et)
   callback buCallback(task.id)
```

2.3. TTE Architecture Based on Intel Cilk Plus

The simplified architecture of the system based on Intel Cilk Plus is shown in Fig. 2.c. As shown in Fig. 2.c, the TTE architecture based on Intel Cilk Plus is very similar to the previous two TTE architectures, which were presented in the previous two subsections. The main difference between the TTE architecture based on Intel Cilk Plus and the original TTE architecture based on OS threads is that the former uses the Intel Cilk Plus runtime library, whereas the latter uses the local OS pthreads library.

This evolutionary step is like in Subsection 2.2 made by modifying the class *Task* such that the member functions *executeTDinParallel* and *executeBUinParallel*, which were originally responsible for the parallel task tree execution, now simply delegate parallel top-down and bottom-up task tree execution to new member functions *executeTDbyCilk* and *executeBUbyCilk*, respectively. As such, this modification is again transparent to the module *TaskScheduler*, as well as to all the legacy applications.

Thanks to Cilk's expressiveness, the simplified pseudo code for the member functions *executeTDinParallel* and *executeBUinParallel* from the previous architecture, almost directly map to the pseudo code for new member functions *executeTDbyCilk* and *executeBUbyCilk*, respectively. Essentially, the call to the function *CreateThread* is replaced with the keyword **cilk_for** and the call to the function *WaitForAllChildTherads* is replaced with the keyword **cilk_sync**.

Once these mappings were introduced, synthesizing new *Task* member functions was rather straightforward. Consequently, the pseudo code of the function *executeTDbyCilk* is the following:

```
executeTDbyCilk(task) =
    callback tdCallback(task.id)
    for each scsr in task.successors
        cilk_spawn scsr.executeTDbyCilk(scsr)
    cilk_sync
```

In the pseudo code above the name *scsr* stands for the *successor task*. Similarly, the pseudo code of the function *executeBUbyCilk* is completely symmetrical:

```
executeBUbyCilk(task) =
    for each scsr in task.successors
        cilk_spawn scsr.executeBUbyCilk(scsr)
    cilk_sync
    callback buCallback(task.id)
```

3. Statistical Usage Testing

We used the method published in [19] for statistical usage testing and operational reliability estimation of all three TTE architectures. The method is mostly based on the approach created by D.M. Woit [20-23] and on the following work of several authors [24-27] that modernized that approach and adapted it to a form of the model-based testing.

For the sake of completeness of this paper, we provide a brief overview of the method [19] in this section. We start with some definitions, then provide formulas for the number of test cases N and for the confidence level M, and finally outline the method in a form of a series of steps.

A *task* τ is a callback function that executes as a local OS thread. A *task tree* is an undirected radial (i.e. acyclic) graph of tasks *TG* whose nodes are tasks interconnected with links indicating predecessor-successor relations. A task tree comprises a set of k tasks $TK = \{\tau_1, \tau_2, …, \tau_k\}$, and a set of ($k$-1) links $L = \{l_1, l_2, …, l_{(k-1)}\}$.

A *task tree execution path*, a.k.a. a *path* in a task tree or a *trace*, is a sequence of terminations of individual tasks $\tau_1\tau_2…\tau_k$ during the task tree execution. The length of this sequence is always equal to k. A *task forest* is a series of task trees of the same complexity (the same number of nodes) that is generated as a test suite. A *test case* is a single task tree execution described by the corresponding path.

Let r_t be a software product *tree-reliability* and r_p be a *path-reliability*. Then it may be easily shown that the product reliability r is obtained by multiplying the two:

$$r = r_t r_p . \tag{1}$$

If we further assume that $r_t = r_p$, then:

$$r_t = r_p = r^{1/2} . \tag{2}$$

Similarly, let M_t be a *tree-confidence-level* and M_p be a *path-confidence level*. Then it may be easily shown that the total confidence level M is the sum of the two:

$$M = M_t + M_p . \tag{3}$$

If we further assume $M_t = M_p$, then:

$$M_t = M_p = M/2 . \tag{4}$$

Therefore, when given r and M we calculate the requested number of trees N_t and number of paths N_p for each tree as:

$$N_t = N_p = \log_{r^{1/2}} (M/2) . \tag{5}$$

Finally, the total number of test cases N is obtained as a simple product of N_t and N_p:

$$N = N_t N_p = (\log_{r^{1/2}} (M/2))^2 . \tag{6}$$

This means that we simply have to generate N_t task trees and execute them N_p times each. Thus the method of statistical testing and reliability estimation for applications based on task trees consists of the following steps:

1. Given the desired level of product reliability, calculate N_t and N_p.
2. Generate N_t task trees.
3. Execute each task tree N_p times.
4. Check the coverage metrics report.
5. If the report shows poor coverage, return to step 2.
6. Report any unexpected behavior to the design and implementation team.

4. Experimental Evaluation

Firstly, Statistical Usage Testing (SUT) and reliability estimation method described in the previous section was used to test all the TTE architectures. Secondly, SUT was used to evaluate the performance of the two new TTE architectures based on TBB and Cilk (see subsections 4.2 and 4.3) with respect to the original TTE architecture based on OS threads, which served as a baseline (see section 4.1).

The measure of the performance that was used in the experiments was the time in seconds that was needed to execute all the N test cases from the given test suite. For the sake of completeness of the paper we provide the execution time measurements data for individual test suits for both TTE architectures, and for the sake of easier performance comparison between the two architectures we provide the relative speedup (RS) calculation results. The relative speedup RS is defined as the ratio:

$$RS = T_p/T_n . \tag{7}$$

where T_p is the test suite execution time for the TTE architecture based on threads and T_n is the test suite execution time for the TTE architecture based on TBB (in subsection 4.2) or on Cilk (subsections 4.3).

All the SUT based measurements were conducted on the dual-core symmetric multiprocessor, Intel® Core(TM) i5 CPU M 520 @ 2.4 GHz, 4 GB RAM, with Windows7 Professional® 64-bit OS.

After conducting SUT based measurements, we made an additional scalability check for the TTE architecture based on TBB on the Intel Server Board SE8501HW4 with 4 Xeon MP Dual Core CPU, facilitating the total of 8 cores operating on 2.4 GHz, with 12 GB of main memory. Software used in the experiments is OS CentOS 5.4 and open Intel TBB (see subsection 4.4). Unfortunately, open Intel Cilk Plus was still not mature enough and its port on CentOS 5.4 was not available at the time of this writings, so we were not able to make the same check for TTE architecture based on Cilk.

At the end of this section we discuss various threats to validity of our experimental results (see subsection 4.5).

4.1. Baseline: Performance of the TTE Based on OS Threads

Table 1 provides T_p values and test verdicts. The columns of Table 1 are organized as follows. The column "No Tasks" contains the number of TTE tasks used to construct task trees, the column "No Trees" shows the number of tasks that may be constructed by the given number of TTE tasks, the next three columns within the common column "Duration [s]" show test suites execution time in seconds for the three distinctive values of desired reliability r (r=0.9, r=0.95, and r=0.99), and the last column "Verdict" contains the test verdict.

As could be seen from Table 1, test suite execution time increases with the number of TTE tasks and with the value of desired reliability r. As the last column indicates, TTE based on threads successfully passed all the tests.

Table 1. Measurements for TTE Based on Threads

No Tasks	No Trees	Duration [s]			Verdict
		r=0.9	r=0.95	r=0.99	
1	1	0	1	11	Pass
2	1	2	7	196	Pass
3	2	2	10	273	Pass
4	6	4	14	298	Pass
5	24	4	15	365	Pass
6	120	5	17	426	Pass
7	720	5	20	485	Pass
8	5040	18	35	558	Pass

4.2. Performance of the TTE Based on Intel TBB

The measured data and the calculated results are given in the following two tables below. Table 2 provides T_n values and test verdicts, whereas Table 3 provides calculated RS values.

The columns of Table 2 are organized in the same way as the columns of Table 1. Similarly, as in Table 1, test suite execution time increases with both number of tasks and the value of given reliability. The latter, again, causes faster growth of the test suite execution time than the former. The last column of Table 2 shows that TTE based on TBB also passed all the tests successfully. All of this seems very similar, but the measured values of test suites execution times are drastically different. Obviously, it took much less time for TTE based on TBB to complete all the tests than it did for the TTE based on threads. This is even more evident from Table 3.

At this point, it seems appropriate to mention that we were not able to calculate some of the values of relative speedup RS from the raw data in

Tables 1 and 2, because some of the values of test suite execution times were 0. Therefore, the corresponding values of *RS* were undefined (dividing 0 with 0 is undefined, and dividing the nonzero number with 0 converges towards infinity, which does not reflect reality in terms of realistic speedup that could be achieved). On the other hand, the test suites execution times are realistically always greater than zero – zero value is only a consequence of imprecise measurements. Finally, since test suites execution times were going up to several hundreds of seconds for the TTE architecture based on threads, we rounded all the 0 second measurements, in Tables 1 and 2, to the 1 second values. By doing so, we introduced a small error, which may be neglected, but we were able to provide *RS* values presented in Table 3.

Table 2. Measurements for TTE Based on TBB

No Tasks	No Trees	Duration [s]			Verdict
		r=0.9	*r*=0.95	*r*=0.99	
1	1	0	0	7	Pass
2	1	1	1	20	Pass
3	2	0	1	23	Pass
4	6	0	1	27	Pass
5	24	0	1	30	Pass
6	120	1	2	34	Pass
7	720	0	1	37	Pass
8	5040	8	8	49	Pass

Table 3. Calculated Values of Relative Speedup *RS* for TTE Based on TBB

No Tasks	No Trees	Relative Speedup RS			Average over forests
		r=0.9	*r*=0.95	*r*=0.99	
1	1	1.00	1.00	1.57	1.19
2	1	2.00	7.00	9.80	6.27
3	2	2.00	10.00	11.87	7.96
4	6	4.00	14.00	11.04	9.68
5	24	4.00	15.00	12.17	10.39
6	120	5.00	8.50	12.53	8.68
7	720	5.00	20.00	13.11	12.70
8	5040	2.25	4.38	11.39	6.00
Average over *r*		3.16	9.98	10.43	7.86

Table 3 shows the values of relative speedup *RS* of test suite execution on new and previous TTE architectures, for various numbers of tasks and desired operational reliability *r* figures. The columns of Table 3 are organized similarly as the columns of Tables 1 and 2. The additional row shows the average *RS* calculated over different values of desired operational reliability *r*, whereas the last column shows the average *RS* evaluated over a different

number of tasks (rather than the test suit verdict like in Tables 1 and 2). The bottom-right cell of Table 3 shows an overall *RS* average when evaluated over all *RS* values.

As expected, the relative speedup *RS* increased both with the number of tasks for a given operational reliability *r*, and with the desired operational reliability *r* for a given number of tasks. Obviously, RS grows much faster with the desired *r* than with the number of tasks, which appears quite natural, because the needed testing effort increases much more with operational reliability *r* than with the number of tasks. As a consequence of these trends, both the average *RS*, calculated per task, increases with the number of tasks, and the average *RS*, calculated per given operational reliability *r*, increases with the value of *r*.

The overall average relative speedup is 7.86 (bottom-right cell in Table 3), which is quite a good result for the dual-core target machine we used in the experiments. Of course, it would be interesting to see how this average speedup of around 8 changes with the number of available cores in the target platform. In Subsection 4.4 we conduct more experiments in that direction in order to check the scalability of the proposed solution.
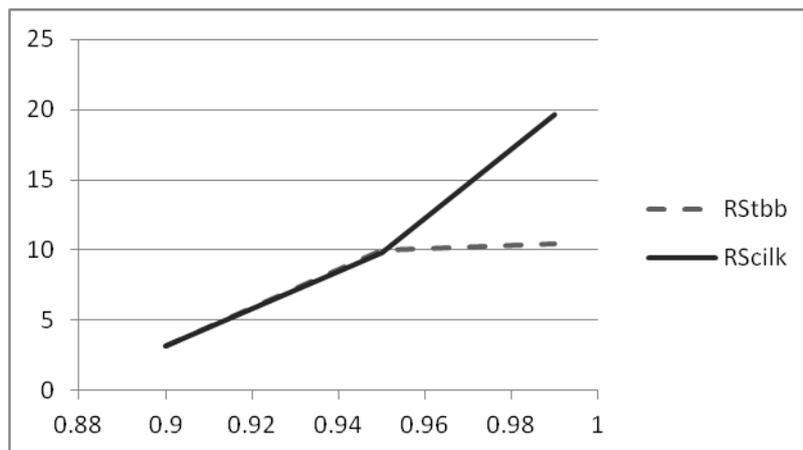
Fig. 3. The average relative speedup *RS* as a function of reliability *r*. The dashed curve shows the average RS values for the TTE based on TBB, whereas the full curve shows the average RS values for the TTE based on Cilk

Another important fact that may be seen by looking at the values of average *RS* in the last row of Table 3, is that average *RS* is around 3 only for the value *r*=0.9. For the values of *r* that are greater than 0.9, average *RS* is around 10, so the test suite execution on the novel TTE architecture is an order of magnitude faster than on the previous architectures, for the greater values of *r* (0.95 and 0.99 in Table 3). This fact becomes even more obvious by observing Fig. 3, which shows the average relative speedup *RS* as a function of a given operational reliability *r* (see the curve *RStbb* in Fig. 3).

4.3. Performance of the TTE Based on Intel Cilk Plus

The measured data and the calculated results are given in the following two tables below. Table 4 provides T_n values and test verdicts, whereas Table 5 provides calculated *RS* values.

Table 4. Measurements for TTE Based on Cilk

No Tasks	No Trees	Duration [s]			Verdict
		r=0.9	*r*=0.95	*r*=0.99	
1	1	1	1	9	Pass
2	1	1	1	10	Pass
3	2	1	1	12	Pass
4	6	1	1	13	Pass
5	24	1	1	16	Pass
6	120	1	1	17	Pass
7	720	1	2	20	Pass
8	5040	7	8	30	Pass

Table 5. Calculated Values of Relative Speedup *RS* for TTE Based on Cilk

No Tasks	No Trees	Relative Speedup RS			Average over forests
		r=0.9	*r*=0.95	*r*=0.99	
1	1	1.00	1.00	1.22	1.07
2	1	2.00	7.00	19.60	9.53
3	2	2.00	10.00	22.75	11.58
4	6	4.00	14.00	22.92	13.64
5	24	4.00	15.00	22.81	13.94
6	120	5.00	17.00	25.06	15.69
7	720	5.00	10.00	24.25	13.08
8	5040	2.57	4.38	18.60	8.51
Average over *r*		3.20	9.80	19.65	10.88

The columns of Table 4 are organized in the same way as the columns of Table 2. Similarly, as in Table 2, test suite execution time increases with both number of tasks and the value of given reliability *r*. Again, the latter causes faster growth of the test suite execution time than the former. The last column of Table 4 shows that TTE based on Cilk successfully passed all the tests. The measured values of test suites execution times for TTE based on Cilk are even smaller than the corresponding times for the TTE based on OS threads. This fact becomes more evident by observing Table 5.

Table 5 shows the values of relative speedup *RS* of test suite execution on the TTE based on Intel Cilk and on the TTE based on OS threads, for various numbers of tasks and desired operational reliability *r* figures. Table 5 is organized in the same way as Table 3.

The results of the qualitative analysis of data given in Table 5 are practically the same as the previous qualitative analysis of data given in Table 3. Again, both the average *RS*, calculated per task, increases with the number of tasks, and the average *RS*, calculated per given operational reliability *r*, increases with the value of *r*. And again *RS* grows much faster with the desired *r* than with the number of tasks.

The overall average relative speedup is 10.88 (bottom-right cell in Table 5), which is a good result for the dual-core target machine we used in the experiments. Of course, it would be interesting to see how this average speedup of around 11x changes with the number of available cores in the target platform, and we have a plan to conduct more experiments in that direction in the future.

But, even more important fact that may be seen by observing the values of average *RS* in the last row of Table 5, is that overall average *RS* of 11x is actually much limited by the *RS* value of around 3x for *r*=0.9. For the values of *r* greater than 0.9, average *RS* goes up to 20x (for *r*=0.99). So after analyzing this data, one becomes aware that the novel TTE architecture provides scalable performance relative to given operational reliability *r*. This fact becomes even more obvious by observing Fig. 3, which illustrates the average relative speedup *RS* as a function of a given operational reliability *r* (see the curve *RScilk* in Fig. 3).

Finally, Fig. 3 makes it possible to compare the two TTE solutions that are based on Intel Parallel Building Blocks. We see from Fig. 3 that the *RS* has greater values for the TTE base on Cilk than for the TTE based on TBB. The values for the former go up to 10x, whereas the values for the latter go up to 20x.

4.4. Scalability Check for the TTE Based on Intel TBB

The results presented in the previous subsections show that performance of newly developed TTE architectures scale rather well with respect to the operational reliability *r*. But, all the previously described experiments were conducted on the dual-core machine and on small task trees consisting of up to 8 tasks. In this subsection we check performance scalability of the TTE based on TBB with respect to the number of processor cores and with respect to the number of tasks in randomly generated large task trees.

For this purpose we conducted the three series of experiments for the three particular numbers of tasks (k) that were used to randomly construct tasks trees, namely k=600, k=800, and k=1000 tasks, respectively. The task trees were randomly generated by the previously developed component *TreeGrower*, which is described in [19]. In each series of experiments we indirectly measured the *RS* of TTE based on TBB in respect to the original TTE based on OS threads for various numbers of processor cores N_c, from N_c=2 to N_c=8 with the step 2 (i.e. N_c=2,4,6,8).

The *RS* was indirect measured as follows. We directly measured the execution times of SUT tests targeting r=0.9 for both TTE based on OS threads and TTE based on TBB, three times each, then we calculated the mean values of execution times, and finally we calculated the corresponding *RS* values. The final results are given in Table 6 and they are illustrated in Fig. 4.

Table 6. Relative Speedup *RS* for various numbers of cores and tasks. RS600 is the *RS* for k=600, RS800 is the *RS* for k=800, and RS1000 is the *RS* for k=1000 tasks

No Cores	RS600	RS800	RS1000
2	29.45	46.17	52.75
4	36.14	57.34	64.49
6	41.45	65.46	72.99
8	43.98	69.39	77.17

Table 6 is organized as follows. The column "No Cores" indicates the number of processor cores that were utilized by TTE based on TBB. The columns "RS600", "RS800", and "RS1000" show the RS for k=600, k=800, and k=1000 tasks, respectively.
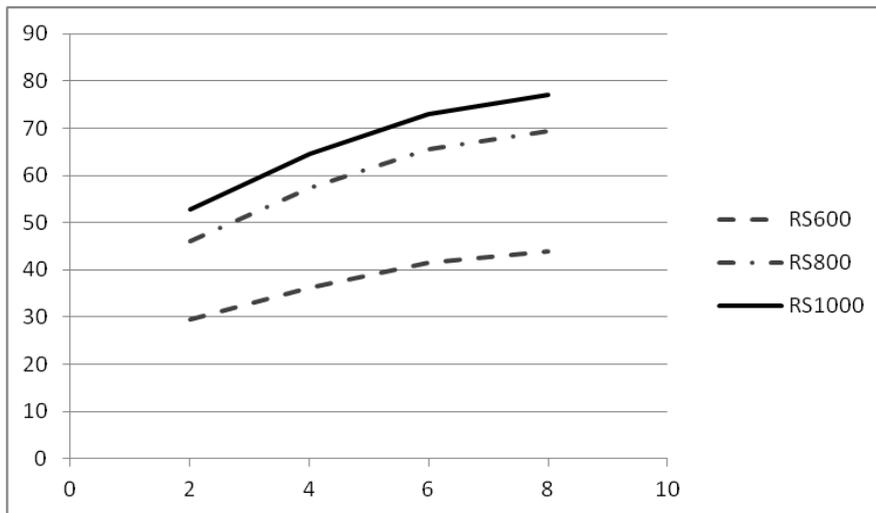
Fig. 4. The average relative speedup *RS* as a function of the number of cores N_c and the number of tasks N_t. The dashed curve shows the *RS* for k=600, the dotted-dashed curve shows the RS for k=800, and the full curve shows the *RS* for k=1000

As indicated by Fig. 4, the performance, in terms of relative speedup *RS* of the TTE based on TBB in respect to TTE based on OS threads, scales perfectly with both the number of processor cores and the number of tasks within a task tree. The *RS* increase linearly with the number of cores and

logarithmically with the number of tasks, and it goes up to 77x for N_c=8 cores and k=1000 tasks.

4.5.   Threats to validity of experimental results

At the end of this section we briefly address the threats to validity of the presented results. From all the kinds of threats, the *threats to the external validity* are the most serious threats for the presented results, because repeating the experiments on a different platform (machine and operating system) would very likely yield different results than those shown in Tables 1-6. The only way to address this issue is to repeat the experiments on several different platforms, and that remains to be done in our future work.

The other two kinds of threats, namely the *threats to internal validity* and the *threats to construct validity* do exist, but can be neglected. We minimized the former threats by disconnecting the target machine from the Internet and by closing all the other applications. The latter threats reduce here to imprecision of measuring time, which obviously can be neglected.

5.   Conclusions

Application of parallel programming techniques to design of software solutions is a promising trend. In this paper we have shown an approach to apply parallel programming techniques based on Intel Parallel Building Blocks to a class of service components within SOA based industrial systems. Moreover, we have shown an approach to introduce either the Intel TBB library, or Intel Cilk Plus library, instead of the conventional OS threads library through a corresponding evolutionary step with minimal adaptations of the legacy TTE architecture. Such evolutionary approaches to architecting new system versions are necessary because legacy software may be of extreme size, typically measured in millions of lines of code.

The results of the approach are two novel TTE architectures. The first one is based on Intel TBB that executes TTE tasks as TBB tasks, whereas the second one is based on Intel Cilk Plus that executes TTE tasks as Cilk strands. Essentially, novel TTE architectures use finer grained parallelism, which yields better multicore CPU utilization. The first novel TTE architecture based on TBB exhibited the average relative speedup *RS* of around 8x, and the maximal *RS* of 10x, over the original TTE architecture based on pthreads. Similarly, and even better, the second novel TTE architecture based on Cilk achieved the average *RS* of around 11x, and the maximal *RS* of 20x, over the original TTE architecture based on pthreads.

Additional scalability check that was made for the first novel TTE architecture based on TBB showed that its performance in terms of relative speedup *RS* scales perfectly with both the number of processor cores and the number of tasks within a task tree. The *RS* increase linearly with the number

of cores and logarithmically with the number of tasks, and it goes up to 77x for N_c=8 cores and k=1000 tasks.

In our future work we plan (i) to make the scalability check for TTE architecture based on Cilk if and when open Intel Cilk Plus port for CentOS 5.4 becomes available, (ii) to explore other algorithms for parallel task tree execution and their implementations, (iii) to evolve TTE architecture in order to support also other non Intel multicores, as well as heterogeneous multicores, and (iv) to develop a distributed TTE architecture for a system with many heterogeneous multicores.

References

1. Komoda, N., Service Oriented Architecture (SOA) in Industrial Systems. In the Proceedings of IEEE International Conference on Industrial Informatics. IEEE CPS, Los Alamitos, CA, USA, pp. 1-5. (2006)
2. Popovic, I., Vrtunski, V., Popovic, M.: Formal Verification of Distributed Transaction Management in a SOA Based Control System. In Proceedings of the 18th IEEE International Conference and Workshops on Engineering of Computer Based System. IEEE CPS, Los Alamitos, CA, USA, 206-215. (2011)
3. Popovic, M., Basicevic, I., Vrtunski, V.: A Task Tree Executor: New Runtime for Parallelized Legacy Software. In Proceedings of the 16th IEEE International Conference and Workshops on Engineering of Computer Based System. IEEE CPS, Los Alamitos, CA, USA, 41-47. (2009)
4. Basicevic, I., Jovanovic, S., Drapsin, B., Popovic, M., Vrtunski, V.: An Approach to Parallelization of Legacy Software. In Proceedings of the 1st Eastern European Regional Conference on the Engineering of Computer Based Systems. IEEE CPS, Los Alamitos, CA, USA, 42-48. (2009)
5. Trivunovic, B., Popovic, M., Vrtunski, V.: An Application Level Parallelization of Complex Real-Time Software. In Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems. IEEE CPS, Los Alamitos, CA, USA, 253-257. (2010)
6. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelisim. O'Reilly Media, Inc., 1005 Gravestein Highway North, Sebastopol, CA, USA. (2007)
7. Kukanov, A., Voss, M. J.: The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. Intel Technology Journal, Vol. 11, No. 4, 309-322. (2007)
8. Popovici, N., Willhalm, T.: Putting Intel® Threading Building Blocks to Work. In Proceedings of the 1st International Workshop on Multicore Software Engineering, ACM Press, New York, NY, USA, 3-4. (2008)
9. Bhattacharjee, A., Contreras, G., and Martonosi, M.: Parallelization Libraries: Characterizing and Reducing Overheads. ACM Transactions on Architecture and Code Optimization, Vol. 8, No. 1, Article 5, 1-29. (2011)
10. Werth, B.: Optimizing Game Architectures with Intel® Threading Building Blocks. Intel Software Network (2009). [Online]. Available: http://software.intel.com/en-

us/articles/optimizing-game-architectures-with-intel-threading-building-blocks/ (current May 2012)

11. Tagliasacchi, A., Dickie, R., Couture-Beil, A., Best, M.J., Fedorova, A., Brownsword, A.: Cascade: A Parallel Programming Framework for Video Game Engines. In Proceedings of the Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures. Institute of Computing Technology, Chinese Academy of Sciences, 47-54. (2008)

12. Randall, K.H.: Cilk: Efficient multithreaded computing. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA. (1998)

13. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM, Vol. 46, No. 5, 720 –748. (1999)

14. Kirkegaard, K., Aleen, F.: Using Intel® Cilk™ Plus to Achieve Data and Thread Parallelism: A Case Study for Visual Computing, 2011. [Online]. Available: http://software.intel.com/en-us/articles/data-and-thread-parallelism/ (current May 2012)

15. Luk, C. K., Newton, R., Hasenplaugh, W., Hampton, M., Lowney, G.: A Synergetic Approach to Throughput Computing on x86-Based Multicore Desktops. IEEE Software, Vol. 28, No. 1, 39-50. (2011)

16. Agrawal, K., Leiserson, C.E., Sukha, J.: Executing Task Graphs Using Work-Stealing. In the Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium. IEEE CPS, Los Alamitos, CA, USA, 1-12. (2010)

17. Popovic, M., Djukic, M., Marinkovic, V., Vranic, N.: A Task Tree Executor Architecture Based on Intel Threading Building Blocks. In Proceedings of the 19th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems. IEEE CPS, Los Alamitos, CA, USA, 201-209. (2012)

18. Popovic, M., Basicevic, I.: An Intel Cilk Plus Based Task Tree Executor Architecture. In Proceedings of the11th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems. WSEAS Press, 30-35 (2012)

19. Popovic, M., Basicevic, I.: Test case generation for the task tree type of architecture. Information and Software Technology, Vol. 52, No. 6, 697–706. (2010)

20. Woit, D.M.: Specifying Operational Profiles for Modules. In Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis. ACM Press, New York, NY, USA, 2-10. (1993)

21. Woit, D.M.: Estimating Software Reliability with Hypothesis Testing. Technical Report CRL-263, McMaster University. (1993)

22. Woit, D.M.: Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules. Ph.D. Thesis, Queen's University Kingstone, Ontario, Canada. (1994)

23. Woit, D.M.: A Framework for Reliability Estimation. In Proceedings of the 5th IEEE International Symposium on Software Reliability Engineering. IEEE CPS, Los Alamitos, CA, USA, 18-24 (1994)

24. Popovic, M., Velikic, I.: A Generic Model-Based Test Case Generator. In Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. IEEE CPS, Los Alamitos, CA, USA, 221-228. (2005)

25. Popovic, M., Basicevic, I., Velikic, I., Tatic, J.: A Model-Based Statistical Usage Testing of Communication Protocols. In Proceedings of the 13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems. IEEE CPS, Los Alamitos, CA, USA, 377-386. (2006)

26. Popovic, M. Kovacevic, J.: A Statistical Approach to Model-Based Robustness Testing. In Proceedings of the 14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems. IEEE CPS, Los Alamitos, CA, USA, 485-494. (2007)
27. Popovic, M.: Communication Protocol Engineering. CRC Press, Boca Raton, FL, USA. (2006)

Prof. Miroslav Popovic received his M.Sc. and Ph.D. degrees in electrical and computer engineering from the Faculty of Technical Sciences at the University of Novi Sad, Novi Sad, Serbia, in 1984 and 1990, respectively. He started his career as an assistant professor at the Faculty of technical sciences, where he remained working to the present day. He was promoted to a tenured professor in 2002. He is currently the head of the Chair of computer engineering. He wrote the book Communication Protocol Engineering (Boca Raton, Florida, USA: CRC Press, 2006) and about 150 papers published in international and domestic journals and conference proceedings. His current research interests are in the areas of parallel programming, model-based development, testing, and verification. Prof. Popovic is the member of the program committee of the IEEE Annual Conference on Engineering of Computer Based Systems (ECBS), and also the member of IEEE, IEEE Computer Society, IEEE TC on ECBS, and ACM.

Miodrag Djukic graduated from the Faculty of Technical Sciences, University of Novi Sad, in 2007, received M.Sc. one year later from the same university. His research interest is mostly focused on compilers and software tools in general, applications of artificial intelligence, and computer graphics. He is a teaching assistant at the Faculty of Technical Sciences and works on several projects for RT-RK, Research and Development Institute for Computer Based Systems. He is the member of IEEE, IEEE Computer Society, and IEEE TC on ECBS.

Vladimir Marinkovic graduated and received M.Sc. degree in electrical and computer engineering from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 2009 and 2010 respectively. He is currently pursuing Ph.D. degree from the same university. His research interests are focused on both parallelization of programs for execution on multiprocessors and multi-core processors, and compilers. In the year of 2011, he was elected to the position of teaching assistant at RT-RK, Research and Development Institute for Computer Based Systems. He is scholar of the Ministry of Science and Technology from the school year 2010/2011.

Miroslav Popovic, Miodrag Djukic, Vladimir Marinkovic, and Nikola Vranic

Nikola Vranic received his B.Sc. and M.Sc. degrees in computer engineering and computer communications at the Faculty of technical sciences, University of Novi Sad. He is currently on Ph.D. studies at the same University. He has been working on security of optical communication lines, developing compiler modules for parallelization, digital television etc. His research interests include multicore systems, code parallelization, cryptography, android, Google TV, etc. He is author and coauthor of a dozen of scientific papers in country and aboard. He is currently employed like Google TV software engineering in the company RT-RK LLC and working as assistant at the Faculty of Technical Sciences.