EFL:
Implementing and Testing an Embedded Language
Which Provides Safe and Efficient Parallel Execution

David Dayan, Moshe Goldstein, Shimon Mizrahi, Max Rabin, Devora Berlovitz, Or Berlovitz, Elad Bussani Levy, Moshe Naaman, Mor Nagar, Ditsa Soudry, Raphael B. Yehezkael
Flexible Computation Research Laboratory
Jerusalem College of Technology
Moshe Goldstein e-mail: goldmosh@jct.ac.il

Miroslav Popovic
Faculty of Technical Sciences, University of Novi Sad,
Trg D. Obradovica 6, 21000 Novi Sad, Serbia
e-mail: miroslav.popovic@rt-rk.com

***Abstract*—As a deterministic parallel programming language that guarantees safety of parallel executions, EFL was designed to allow the embedding of parallel code blocks into a sequential host language program. An EFL pre-compiler, which is described here, has been implemented that translates EFL blocks into the host language. The EFL pre-compiler and runtime supports parallel building blocks such as parallel assignments, parallel for loops, etc. EFL was successfully tested on a task tree architecture, using the work-span method to estimate parallelism and speedup, followed by experiments to measure actual execution times and calculate real speedups on a specific platform. Results show that speedup increases almost linearly with the size of a task tree.**

Keywords- parallel programming; task trees; flexible computation; parallel design patterns; order-independent execution; Python multiprocessing module; MPI

I. INTRODUCTION

EFL, an Embedded Flexible Language, supports well-defined order-independent execution [1]. The programming and execution models of EFL are described in [2]. EFL was designed to allow the embedding of parallel code blocks into a sequential host language program. A pre-compiler is needed to translate EFL embedded blocks of code into native parallel code in the host language. Such a pre-compiler can be implemented for any host language, allowing programmers to learn EFL only once, using it for their parallel programming needs in different languages. A first EFL pre-compiler for Python as host language has been implemented [3].

In this paper we discuss our first EFL pre-compiler which has been implemented using the parallelization capabilities provided by Python's Pools available in the multiprocessing module [4]. A second version of the EFL pre-compiler, which is being implemented upon the mpi4py-based module called DTM [5], will be briefly presented. Next, we validate EFL usability by a case study that addresses an important kind of parallel programs, which are referred to as *task trees* [6] [7] [8] [9] [10] [11], and we will use the *work* and *span* method [12] to estimate parallelism and speedup for general task trees, as well as for binary task trees. Then, we construct a family of binary task trees in EFL, measure their *top-down* and *bottom-up execution* times, calculate corresponding speedups, and compare them to estimated values. As we will see, results show that EFL task trees are efficient parallel architectures, because both the speedup of *top-down* and *bottom-up parallel execution*, over equivalent sequential Python code, increases almost linearly with the size of a task tree.

The remainder of the paper is organized as follows: section 2 presents EFL implementation, section 3 contains a case study – the EFL task trees efficiency analysis referred above, section 4 make reference to the EFL implementation of the well-known *pipeline* parallel pattern [13] [14], section 5 covers related work, section 6 provides concluding remarks and further work to be done.

II. EFL IMPLEMENTATION

EFL was designed as an embedded language for parallel programming and make it easier. Our implementation of EFL uses Python 3.x [15] as its host language, but it supports Python 2.x too. There were several factors taken into consideration when deciding which host language to use:

(a) We wanted a host language that is easy to learn, and Python was a good candidate in this respect. This would place EFL as an ideal language to be taught to new programmers as explained in [16].

(b) Another factor considered was cross platform compatibility. It was important to choose a language that anybody can use on any platform so that nobody is precluded from using EFL because of the operating system they run.

(c) Another factor considered was the parallel capabilities of the candidate host language. The functional nature of Python, along with the robust and easy to use multiprocessing module [4], provides a platform for creating parallel code [15] [17]. We will discuss below our choice of Python's multiprocessing module instead of Python's Threading module.

The EFL Pre-compiler

In order to get EFL implemented, we developed an EFL pre-compiler which leaves all the sequential parts of the program written in the Host language without any change, and translates the code inside EFL-blocks into parallelized code written using parallel programming capabilities of the Host language.

To build the EFL pre-compiler we used JavaCC [18], an open source parser and lexical analyzer generator. JavaCC generates pure Java code which we compile into a JAR file for distribution. We chose to use JavaCC both for its ease of use and so that the pre-compiler would be compatible with any Java-supported platform.

The EFL BNF grammar and all the material concerning the EFL pre-compiler, may be found elsewhere [3].

Implementing the EFL Execution Model

There are two modules built into the standard Python core that allow parallelism - threading and multiprocessing. We use Python's multiprocessing module to implement EFL parallelism because of the limitations imposed by the GIL (Global Interpreter Lock) on the parallelism capabilities of the threading module. The multiprocessing module uses sub-processes instead of threads to achieve parallelism. Although only one thread can execute Python code due to the GIL, multiple processes with one thread each, may run concurrently on multiple cores [15][25]. Within the multiprocessing module there are different methods for creating and working with processes. One way is using the Process class to create a Process object that is set to execute a function, and calling its start() method to begin the execution. In order to pass arbitrary variables to the function using the Process class, we would have had to use Queues to pass parameters to the function. The other option is to create a Pool of processes. A Pool is a collection of a fixed number of processes. The number of processes in a Pool are defined when the Pool is created. The number of sub-processes defaults to the number of cores in the computer. Another benefit of the Pools is that parameters can be passed without using a Queue. The Pool class also includes map functionality allowing simple implementation of EFL's *maploop*. One of the biggest deficiencies of Pools is that the sub-processes of the pool are 'daemonic' processes. Daemon processes are created by a parent non-daemonic process and are able to continue to run even if the parent process ends. However, daemonic processes cannot spawn sub-processes. That means that the functions executed by a Pool cannot themselves spawn child processes. Processes created with the Process class can be created either as daemons or not daemons. This difference ends up having a very significant impact on the abilities of EFL. If we use Pools as the mechanism of parallelism in our implementation of EFL, then EFL blocks cannot contain function calls that have EFL blocks within them. That is because the functions called from within EFL-blocks are executed by a Pool and therefore, are run as daemon processes. This means that they cannot contain another EFL block which would be trying to create a new Pool of processes. This would limit EFL to one level of child-processes and no more. However, if we were to use the Process class, we would be able to run functions called by EFL in a non daemonic process and they would in turn be able to contain EFL blocks that also create child processes. This would allow nesting of EFL within EFL that executes all of the processes and sub-processes in parallel. This would be ideal in order to allow the programmer the freedom to embed EFL in any level of a program with no regard to how any particular function is called by EFL or by the host language. This freedom of Processes would be the ideal way to implement EFL. However, in reality, there is a limit on the number of sub-processes that an operating system can handle. Although an operating system is responsible for managing all of the processes running on a computer, there are limits on the number of concurrent processes that an operating system can manage and there are performance considerations that must be considered. The performance of the applications worsens considerably when the number of processes exceeds the total number of processes that the system can handle and the number of processors. This will cause frequent context switching and its associated inefficiencies. Starvation may occur with Pools, but only occur when a worker process loops indefinitely. This is because the Pool lets each task assigned to it finish before executing the next task assigned to it.

In order to decide whether to use Processes or Pools, we did a simple test that executes a task in parallel using Processes, and using Pools, and we compared the running time of the two. We ran an algorithm on an n * n array where an operation (finding whether the number is prime) is executed for each element of the n * n array in parallel. This is the program:

```
import multiprocessing, collections, time
def testPool(bigNum):
   pool = multiprocessing.Pool()
   processes = collections.deque()
   bigArr = [[0] * bigNum] * bigNum
   for i in range(bigNum):
      for j in range(bigNum):
        processes.append(pool.apply_async(f,
              args = [i * bigNum + j]))
   for i in range(bigNum):
      for j in range(bigNum):
        processes.popleft().get()
def testProcesses(bigNum):
   processes = collections.deque()
   for i in range(bigNum):
     for j in range(bigNum):
       p = multiprocessing.Process(target=f,
             args = (i * bigNum + j,))
       processes.append(p)
       p.start()
   for i in range(bigNum):
      for j in range(bigNum):
        processes.popleft().join()
def main():
   bigNum = 100
   start_time = time.time()
   testPool(bigNum)
   elapsed_time = time.time() - start_time
   start_time = time.time()
   testProcesses(bigNum)
   elapsed_time = time.time() - start_time
if __name__ == '__main__':
   main()
```

TABLE I EXECUTION TIMES

Input Size	Process (secs.)	Pool (secs.)
2 * 2	0.269999980927	0.15700006485
10 * 10	10.75	0.153000116348
15 * 15	22.4300000668	0.174999952316
100 * 100	Crashed	3.25500011444

The execution times are shown in Table I. Even with a 15 by 15 array, using Process is drastically slower, and on a simple 100 by 100 array, the operating system isn't even capable of handling this many processes. Although we would have preferred to use Processes because of their flexibility, in view of the above, Pools have a distinct performance advantage. To get the best of both Pools and Processes, we implemented our own custom Pool class, based on the Pool class that ships with Python. Our modified Pool creates child processes that are not daemonic processes. That way, child processes of the Pool can themselves create Pools, allowing in principle an unlimited depth of parallelism, which is ideal for the implementation of the EFL language. It also has the safety that Pools afford, in that it only creates a fixed, limited number of child processes when the Pool is created, and each asynchronous job that is added to the Pool is queued until a new child process is able to execute it. This was an achievement that even gives EFL an advantage over a Python programmer who manages manually the processes without the help of EFL. If, however, the programmer creates too many levels of child processes, the number of processes that wait to be executed, can still grow exponentially like they did when we tried to use the Process module. However, this exponential growth is limited by the number of Processes per Pool, thus the growth is slowed and most programs should suffice with this implementation.

The Python code generated by the EFL pre-compiler consists of two phases. The first phase is the spawning of parallel tasks and the second waits for the tasks to end and collects the results. This mode of operation is like the well-known *fork-join* (or *master-workers*) parallel pattern and it provides the framework for code with any depth of parallelism.

For example, the following code shows how a *for*-block containing an *assignment*-block gets translated into two *while* loops in Python:

```
EFL{
for (arrix = 0;  arrix < len(inArray);  arrix = arrix + 1) {
  outArray[arrix] =  cpuIntensiveFunc(inArray[arrix]);
}
}EFL
```

It would translate into Python in a manner similar to this simplified pseudo-code:

```
//spawn tasks to be executed in parallel
arrix = 0
while arrix < len(inArray):
  parallelHandle[arrix] =  \
    spawnParallelTask(cpuIntensiveFunc, inArray[arrix])
   arrix = arrix + 1
//collect results
arrix = 0
while arrix < len(inArray):
 outArray[arrix] =
   getParallelResult(parallelHandle[arrix])
 arrix = arrix + 1
```

In this way, it was possible to translate EFL code into parallelized code.

It is important to note that the nature of the multiprocessing module is to delegate a function to a process. It was decided that only certain statements within EFL would actually be parallelized with processes. For example, two assignments

```
EFL{
a = 5 + 5;
b = 6 + 6;
}EFL
```

would not be worth parallelizing. The overhead incurred on each parallel call is too great and in order to optimize EFL, only function calls are parallelized. Therefore, in order to maximize EFL's effectiveness and efficiency, it is best used for function calls to CPU intensive operations. Simple, less CPU intensive functions should be avoided in EFL and are best called from Python directly. Only parallelizing function calls also solves many issues of scoping, because EFL code is translated into the host language code where it is written. That way, any variables or functions that exist within the lexical scope where the EFL-block is written are also present where the EFL-block is translated. In order to take statements that are not function calls and parallelize them, the pre-compiler would have to create a function that can be called by a process. However, this leads to a problem of scoping because the statements may rely on certain variables to exist within its scope that will not be carried over into the newly created function. By limiting parallelization to function calls, and only calling functions written in Python (or Python-wrapped), there is no problem of missing variables. We found this to be a simple and elegant solution. Regardless, however, it is still incumbent upon the programmer to assume full parallelism and adhere to the programming model of EFL [1], such as once-only assignment, to create deterministic parallelized code, even if the code will not actually be parallelized. The programmer should not rely on EFL code to be executed in any particular order.

Detecting violations of once-only assignments (debug mode)

By once-only assignment we mean that it is an EFL programming error that a variable will be assigned values more than once. Violations of once-only assignment, which is one of the limitation rules enforced by the EFL programming model, cannot be checked at compile time. To demonstrate this point, consider the code example below:

```
EFL{
if f(x) == 0 {
   x = value1;
} else {
   y = value1;
}
x = value2;
}EFL
```

Let's suppose that the function f(x) is written using sequential execution only. Therefore there are no once-only assignment violations in the execution of f(x). Thus, a violation of once-only assignment occurs iff f(x) is sometimes zero. So, if we could detect violations of once-only assignment at compile time, we could apply it to the example above, and thereby determine whether or not f(x) is sometimes zero. This is impossible by Rice's theorem [19].

Therefore, a checking mechanism was implemented in the EFL pre-compiler to produce code in the host language that checks for violations of once-only assignments at runtime. Although, a violation of once-only assignment is considered invalid according to the definition of the EFL programming model, the checking mechanism can be excluded from the compiled code with a special flag passed to the pre-compiler. This allows for more experienced programmers to produce efficient code and use the once-only assignment checker as a development tool.

In the long term we would like to find a complete and efficient implementation of our flexible computation approach to parallel programming, by using suitable hardware support. Towards this end, we have developed a prototype OR-memory implementing an atomic or lock free or= composite assignment using an FPGA [20]. The or= composite assignment also ensures well defined results and is more efficient to implement than once-only assignment, particularly with execution occurring in parallel. We intend to expand on this matter in a future work.

EFL pre-compiler – MPI version

MPI (Message-Passing Interface) [13][21] is a widely used parallel programming interface for message-passing-based parallel computing. It is characterized by the absence of shared memory among processes. An MPI version of the EFL pre-compiler has been developed upon the DTM (Distributed Task Manager) module, which is part of DEAP (Distributed Evolutionary Algorithms in Python) package [5]. DTM is a Python module written using the mpi4py module. In contrast to MPI, which requires explicit parallel programming, DTM allows implicit parallel programming implementation in a similar level of abstraction as the multiprocessing Python module. This new version of the EFL pre-compiler generates DTM-Python parallel code. An example of the EFL-code of the multiplication of a 2-dimensional matrix by a vector and its translation to DTM-Python parallel code can be found elsewhere [3]. The DTM-based version of the EFL pre-compiler is a research project in progress. A more detailed discussion of it will be published in the near future.

III. VALIDATION AND TESTING

Task trees and task graphs [11][22] appear in many real-world applications ranging from safety critical infrastructures, such as gas, water, and electricity distribution systems [7], across embedded systems and consumer electronics appliances, to computer games [23].

The method for Statistical usage testing and operational reliability estimation based on operational profiles and test suite quality indicators [6][7][8][9][10] may help programmers to achieve freedom from violations of once-only assignment with high reliability. Although this method was developed primarily for parallel programs based on task trees, it could be extended and applied to other kinds of parallel programs. To do so, programmers need to define adequate test suite quality indicators and to develop the corresponding test case generators. The high level task trees [6][7][8][9][10] are suitable for representing EFL blocks embedded in code written in a sequential programming language. In the future we would like to provide such a framework for EFL.

First of all, we introduce task trees formally, and we continue by introducing the work and span method that we used to analyze task trees implemented in EFL, which we refer to more briefly as EFL task trees. Next we analyze general EFL task trees, as well as EFL binary task trees to derive analytical estimates for their parallelism and speedup. Subsequently, we present and analyze the experimental results – measured execution times and calculated real speedups. We finalize this section by stating the threats to validity of experimental results.

A. Task trees

In this subsection we briefly introduce and extend some definitions from [8], which will help us to treat task trees more formally.

Definition 1. A *task* τ is a callback function that executes as a process, task, or thread.

Definition 2. 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 $TK = \{\tau_1, \tau_2, \ldots, \tau_k\}$, and a set of (k-1) links $L = \{l_1, l_2, \ldots, l_{(k-1)}\}$.

Definition 3. A *root rt* is a predecessor of all the nodes in a task tree.

Definition 4. For any two directly connected nodes in a task tree, the node closer to the root is a *predecessor* of the other node, and the other node is its *successor*.

Definition 5. *A task tree leaf* is a task that has no successors.

A parallel task tree execution may be formally described as a composition of two operators, namely P() and S(), where the former represent parallel execution of its parameters, whereas the latter corresponds to sequential execution. A task tree can be executed in parallel *top-down* or *bottom-up*.

Definition 6. A *task tree execution path*, a.k.a. a path in a task tree or a trace, is a sequence of terminations of ndividual tasks $\tau_1\tau_2 \ldots \tau_k$ during the task tree execution. The length of this sequence is k.

In order to illustrate these definitions we are introducing an example task tree in Fig. 1.

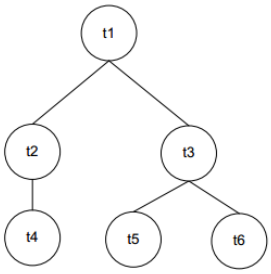

Fig. 1. An example of a task tree.

This task tree comprises six tasks, namely t_1, t_2, t_3, t_4, t_5, and t_6. The task t_1 is the task tree root, whereas tasks t_4, t_5, and t_6 are task tree leafs. The remaining tasks are having both predecessors and successors. The trace "$t_1t_2t_3t_4t_5t_6$" is an example of a valid top-down trace, whereas the trace "$t_4t_5t_6t_2t_3t_1$" is an example of a valid bottom-up trace.

Next we introduce the work and span method we used to analyze task trees.

Definition 7. A *binary task tree* is a task tree where a node has at most two successors.

Definition 8. A *complete binary tree* (also known as a *perfect binary tree*) is a binary tree in which all leaves have the same depth or same level.

Note that for brevity we sometimes omit the term "complete" when that attribute is obvious from the shape of a task tree at hand. Next, let n be the inclusive number of hierarchical levels starting from the root and going downwards towards leafs then the number of task in a complete binary task tree is equal to $2^n - 1$.

A complete binary task tree example is shown in Fig. 2. This binary task tree has n = 3 levels of hierarchy, so it comprises $2^3 - 1 = 7$ tasks, namely t_1, t_2, t_3, t_4, t_5, t_6 and t_7. The task t_1 is the task tree root, whereas tasks t_4, t_5, t_6 and t_7 are task tree leafs. The trace "$t_1t_2t_3t_4t_5t_6t_7$" is an example of a valid top-down trace, whereas the trace "$t_4t_5t_6t_7t_2t_3t_1$" is an example of a valid bottom-up trace.

Next we introduce the *work* and *span* method we used to analyze task trees.

B. Work and span method

Task trees introduced in a previous section only specify the predecessor-successor relations among individual tasks within a program; they do not contain any timing related information. In order to analyze timing of programs, based on task trees, we use *high-level* DAGs. We start this subsection by briefly introducing *high-level* DAGs from [12], which we later use to analyze EFL programs. We continue by introducing two fundamental measures of DAGs from [24], namely *work* and *span*, as well as two derived measures, namely *parallelism* and *speedup*. Finally we outline the steps of the *work* and *span* method that we use to analyze EFL programs.

Definition 9. A DAG is a graph $G = (V, E)$, whose vertices in V are tasks (performing some data processing), whereas the edges in E represent control flow dependencies, such that the edge (u, v) in E specifies that u must execute before v.

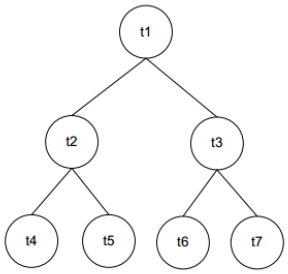

Fig. 2. An example of a binary task tree.

Definition 10. A *high-level* DAG is a DAG that is illustrated in a diagram such that only its nodes are shown (edges are intentionally hidden to simplify the diagram and further analysis) according to the following convention. The tasks are shown as oval graphical symbols whose width represents time. If the task v is executed in parallel with the task u, P(u, v), the task v is drawn bellow the task u. If the task v is executed after the task u, S(u, v), the task v is drawn to the right (in the direction of increasing time).

Next we introduce *work*, *span*, *parallelism*, and *speedup*.

Definition 11. *Work*, T_1, is the sum of execution times of all the tasks in the DAG.

Definition 12. A *path* in a DAG is an inclusive sequence of tasks from the root task (the left most task) to a leaf task (a task with no task following it).

Definition 13. *Critical path* in a DAG is the path has the largest execution time.

Definition 14. *Span*, T_∞, is the execution time of the critical path.

Definition 14. *Parallelism*, Ψ, is *work* divided by *span*, i.e. $\Psi = T_1 / T_\infty$.

Definition 15. *Speedup*, S_P, is given as ratio $S_P = T_1 / T_P$, where T_P is the DAG's parallel execution time on P processors.

From [24] we know that T_P is bounded as follows:

$$T_P \leq (T_1 - T_\infty) / P + T_\infty$$

Based on this bound on T_P we easily provide the corresponding theoretical *minimal speedup*, $S_{min}(P)$:

$$S_P \geq S_{min}(P) = P / [1 + (P\text{-}1) / \Psi]$$

Finally, after introducing previous definitions, we outline the *work* and *span* method. The method comprises the following steps:

(1) For a given EFL program construct the corresponding *high-level* DAG.
(2) Determine *work* and *span*.
(3) Calculate *parallelism*, and *minimal speedup*.

C. Analyzing EFL task trees

In this section we apply the work and span method to EFL task trees. Given any particular EFL task tree we may analyze it simply by conducting the three steps listed in the previous section. Constructing high-level DAGs for top-down parallel execution is rather easy. We place the root task to the leftmost position in the diagram, we continue by placing the tasks form the second level of task tree hierarchy to the right from the root, such that we place them one beneath the other. We continue in this manner until we reach

the task tree leafs, which end up in the rightmost positions in the diagram. Constructing DAGs for the bottom-up execution is symmetrical – it just goes in the opposite direction. Once we have DAGs available, determining work and span, and then calculating parallelism and minimal speedup is even easier.

In the text that follows we illustrate this process. We do this for the complete binary task tree shown in Fig. 2.

Step 1: Construct high-level DAGs. Fig. 3 and Fig. 4 show *high-level* DAGs for the top-down and the bottom-up parallel execution of the binary task tree in Fig. 2, respectively.

In both figures we assume that all the tasks have the same duration, e.g. the unit time. In Fig. 3 the root task t_1 starts first. When t_1 finishes, t_2 and t_3 execute in parallel, and after they finish, t_3, t_4, t_5, and t_6 execute in parallel. In Fig. 4 execution goes in reverse, t_3, t_4, t_5, and t_6 execute in parallel first, then t_2 and t_3 execute in parallel, and after them t_1 executes solo.

Step 2: Determine *work* and *span*. Based on the definitions given in the previous section and DAGs in Fig. 3 and Fig. 4 we easily determine that work and span for both DAGs are the same. Work is:

$T_1 = 7$

Span is:

$T_\infty = 3$

Step 3: Calculating *parallelism* and *minimal speedup*. Based on the definitions given in the previous section and work and span from the previous step 2 we calculate parallelism and minimal speedup. *Parallelism* is:

$\Psi = 7 / 3 = 2.33$

Minimal speedup as a function of the number of available processors P:

$S_{min}(P) = P / [1 + (P\text{-}1) / 2.33]$

For example, $S_{min}(1) = 1$, $S_{min}(2) = 1.39$, but when P increase to infinity $S_{min}(P)$ converges to $S_{min}(\infty) = 2.33$.

Next we derive *work*, *span*, *parallelism*, and *minimal speedup* for a complete binary task tree with n levels of hierarchy of unit tasks (tasks lasting one unit of time).

Proposition 1. *Work*, *span*, *parallelism*, and *minimal speedup* for a complete binary task tree with n levels of hierarchy of unit tasks are given as follows:

$T_1 = 2^n - 1$

$T_\infty = n$

$\Psi = (2^n - 1) / n$

$S_{min}(P) = P / [1 + n(P\text{-}1) / (2^n - 1)]$

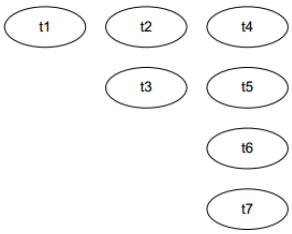

Fig. 3. DAG for *top-down* parallel execution of the task tree shown in Fig. 2.

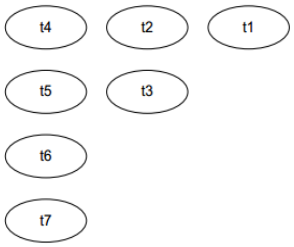

Fig. 4. DAG for bottom-up parallel execution of the task tree shown in Fig. 2.

Proof. It follows directly from properties of complete binary trees and definitions introduced in the previous section

Remark. *Parallelism* and *minimal speedup* given in Proposition 1 may also be used for complete binary task trees that comprise tasks that all perform the same amount of work, such that they all have the same time duration. This time duration may be any value, e.g. 400 milliseconds, or 10 seconds, so not necessarily exactly 1 second. Table III shows values for theoretical parallelism and minimal speedup for this family of EFL task trees. The first column contains values for n, whereas the second and third columns contain the corresponding values for Ψ and Smin = Smin(2), respectively.

D. Experiments: measuring EFL task trees speedup

In this subsection we present the experiments we made within the case study. For the experiments we used the dual-core symmetric multiprocessor, Intel® Core(TM) i5 CPU M 520 @ 2.4 GHz, 4 GB RAM, with Windows7 Professional® 64-bit OS. We made a family of EFL programs representing a family of complete binary task trees with n = 2, 3, 4, 5, and 6 number of hierarchical levels. All the tasks performed the same processing that was taking approximately 400 milliseconds. We measured both *bottom-up* and *top-down* parallel execution times three times each for each task tree within this family of task trees. We then calculated the average execution times and calculated the corresponding real speedups. Finally, we calculated theoretical parallelism and minimal speedup for this family of EFL task trees, and compared real speedup with the minimal speedup.

Resulting data is organized and presented here in two tables. Table II contains measured average execution times and calculated real speedups, where "s" stands for "sequential", "p" for "parallel", "bu" for "bottom-up", and "td" for "top-down", so:

T_{sbu} is the sequential *bottom-up* execution time,
T_{pbu} is the parallel *bottom-up* execution time,
T_{std} is the sequential *top-down* execution time,
T_{ptd} is the parallel *top-down* execution time,
S_{bu} is the *bottom-up* speedup, and
S_{td} is the *top-down* speedup.

Fig. 5 shows measured *bottom-up* speedup, S_{bu}, and *top-down* speedup, S_{td}, as well as theoretical *minimal speedup*, S_{min}, as functions of the number of hierarchical values, n.

TABLE II EXECUTION TIMES AND SPEEDUPS FOR THE EFL TASK TREE FAMILY

n	T_{sbu}	T_{pbu}	T_{std}	T_{ptd}	S_{bu}	S_{td}
2	1.246	1.0461	1.193	1.055	1.19	1.13
3	2.761	1.819	2.765	1.814	1.52	1.52
4	5.880	3.315	5.904	3.321	1.77	1.78
5	12.136	6.343	12.140	6.444	1.91	1.88
6	24.706	12.439	24.857	12.504	1.99	1.99

By having a look at Fig. 5, two facts become obvious. The first fact is that both real speedups, S_{bu} and S_{td}, increase almost linearly with the size of the task tree measured by the number of its hierarchical levels, *n,* proving EFL efficiency. The second fact is that both real speedups are above the theoretical *minimal speedup*, and all the three speedups have the same trend, as well as similar values, justifying the use of the *work* and *span* method for EFL task trees.

E. Threats to validity

This section discusses the threats to validity of the results gained from the presented experiments.

Threats to internal validity are influences that may affect the dependent variables without the researcher's knowledge. Our concern is that internal operation of the underlying operating system MS Windows 7 and Python 3.3 runtime library may bias our results. In particular, some background processes may be inactive during the execution of some of the task trees and later they may be started during the execution of some other task trees. Thus, task trees in the former case would be executed faster than in the latter case. One should keep this in mind while interpreting data in Table II.

Threats to external validity are conditions that limit researcher's ability to generalize the results of the experiments. All the experiments were made on a dual-core symmetric multiprocessor, Intel® Core(TM) i5 CPU M 520 @ 2.4 GHz, 4 GB RAM, with Windows7 Professional® 64-bit OS. Running the experiments on a different platform would very likely yield different quantitative results than those shown in Table II, because task tree execution times depends on the target platform.

Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, the measure of task tree execution time is measured by the local operating system. Normally, the task tree execution time varies from one execution to another. In order to minimize the effect of these variations we repeated each task tree execution (both *bottom-up* and *top-down*) and we calculated the average execution times.

TABLE III THE THEORETICAL PARALLELISM AND MINIMAL SPEEDUP FOR THE EFL TASK TREE FAMILY.

n	Ψ	$Smin$
2	1.5	1.2
3	2.33	1.4
4	3.75	1.58
5	6.2	1.72
6	10.5	1.83

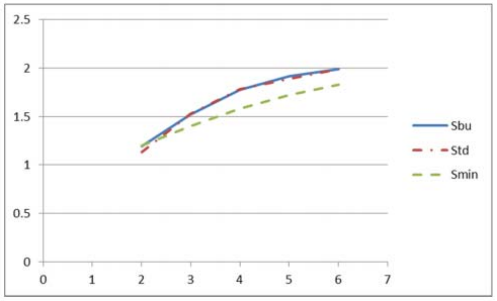

Fig. 5. Speedups as functions of task tree size (*n*).

The results of the presented experiments should be interpreted by keeping in mind above threats to validity.

IV. PARALLEL PROGRAMMING PATTERNS IN EFL

It is well-known that a Design Pattern describes a proven approach for dealing with a common situation in design. It suggests what to do to obtain an elegant, modifiable, extensible, flexible and reusable solution for the problem at hand. As a proof-of-concept of EFL as a parallel programming tool, a project is in progress to implement all the parallel design patterns [14] in EFL, the results of which will be published in the future. The "pipeline" parallel design pattern has been implemented in EFL (the code is available elsewhere [3]). Its run time was compared against that of a sequential version. Let M be the number of items to be processed, and N the number of stages. The run time of the sequential version is O(M*N), because only after one item has been processed by all the stages of the pipeline, the next one is permitted to be processed. On the other hand, in the EFL version, the run time is O(M+N) because when an item is being processed by some stage, the next item can be processed by the previous stage.

V. RELATED WORK

Task trees and task graphs appear in many real-world applications [6][7]. The resulting task trees are executed bottom-up and top-down in parallel by the new Task Tree Executor (TTE) [6] runtime library. Of course, the legacy code had to be refactored in a form of call-back functions, which are called by TTE for individual tasks as TTE traverses the task tree bottom-up and top-down [7].

Since task trees are parallel programs, the conventional method traditionally used for sequential programs could not be used, and the new method of statistical usage testing and operational reliability estimation was developed [8].

Following emergence of new Parallel Programming Frameworks (PPFs), TTE [6] evolved over time into two better versions [9][10]. Both of these TTE versions introduce fine-grained parallelism based on lightweight task and provide significantly better performance than the initial TTE version [6]. Another PPF that enables making task graphs is Intel Threading Building Blocks (TBB) [12].

We used the *work* and *span* method in our case study to estimate parallelism and speedup of EFL task trees. A short account on assumptions and previous work related to this method seems appropriate at this place. One may use the

original *work* and *span* method [24] if the computation Directed Acyclic Graph (DAG), i.e. the task graph, is executed by a greedy scheduler, such as a work-stealing scheduler [11]. In the case study we assumed that the underlying OS Windows 7 scheduler together with the Python 3.3 runtime may be approximated by a greedy scheduler. The results of the case study justify introducing this assumption.

On the other hand, the original work and span method [24] was primarily developed for analyzing programs written in the Cilk programming language, and the instructions one should use to construct a DAG for a parallel program at hand are hardly applicable to other programming languages. Therefore the work and span method that we used in our case study in this paper is based on the high-level DAGs, which were introduced [12] for the same reason.

VI. Conclusions and Further Work

We validated efficiency of execution of EFL task trees. In the case study we used both the work and span method to estimate parallelism and minimal speedup of a family of EFL task trees, and experiments to measure the execution times and calculate the real speedups for bottom-up and top-down parallel executions, where the baseline was the equivalent family of task trees implemented as sequential Python code. Experimental results show that real speedups increase almost linearly with the size of an EFL task tree, and therefore prove EFL task tree efficiency in exploiting available processing power of a target execution platform. Curves for real speedups are above the curve for the theoretical minimal speedup, but these curves have the same trend, justifying the use of the work and span method for EFL task trees.

We see a need for work in the following areas: (a) EFL pre-compiler implementation for additional host languages, (b) EFL serial execution may aid for debugging EFL code, (c) a new Software Transactional Memory (STM) [25] prototype of the EFL pre-compiler is being designed for Python.

Acknowledgment

This work has been partly supported by the Serbian Ministry of Education and Science, through grants No. TR 32031 and III 44009, both for time period 2011-2015.

We wish to thank the Research Authority of the Jerusalem College of Technology for supporting the Flexible Computation Research Lab where this research was carried out. We also wish to thank J. J. Florentin for his invaluable assistance and encouragement.

References

[1] R. B. Yehezkael, M. Goldstein, D. Dayan, "Flexible Algorithms: Enabling Well-defined Order-Independent Execution with an Imperative Programming Style", submitted to this conference, 2015.

[2] M. Goldstein, D. Dayan, , M. Popovic, Sh.Mizrahi, R. B. Yehezkael, D. Berlovitz, O. Berlovitz M. Rabin, "EFL: An Embedded Language Enabling Well Defined Order-Independent Execution", 2015. Available at http://flexcomp.jct.ac.il/TechnicalReports/EFLprinciples.pdf.

[3] http://www.flexcomp.jct.ac.il/EFLimp.

[4] http://docs.python.org/py3k/library/multiprocessing.html.

[5] F-M De Rainville et al., "DEAP: A Python Framework for Evolutionary Algorithms", in Proc. GECCO'12, ACM, 2012.

[6] M. Popovic, I. Basicevic and V. Vrtunski, "A Task Tree Executor: New Runtime for Parallelized Legacy Software," in Proc. 16th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems, 2009.

[7] I. Basicevic, S. Jovanovic, B. Drapsin, M. Popovic and V. Vrtunski, "An Approach to Parallelization of Legacy Software," in Proc. 1st IEEE Eastern European Regional Conference on Engineering of Computer Based Systems, 2009.

[8] M. Popovic and I. Basicevic, "Test case generation for the task tree type of architecture," Information and Software Technology, vol. 52, no. 6, pp. 697-706, 2010.

[9] M. Popovic and I. Basicevic, "An Intel Cilk Plus Based Task Tree Executor Architecture," in Proc. 11th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, 2012.

[10] M. Popovic, M. Djukic, V. Marinkovic and N. Vranic, "On Task Tree Executor Architectures Based on Intel Parallel Building Blocks," Computer Science and Information Systems (ComSIS) , vol. 10, no. 1, pp. 369-392, 2013.

[11] K. Agrawal, C. E. Leiserson and J. Sukha, "Executing Task Graphs Using Work-Stealing," in Proc. 24th IEEE International Parallel and Distributed Processing Symposium, 2010.

[12] M. Popovic, M. Basicevic, M. Djukic and N. Cetic, "Estimating Parallelism of Transactional Memory Programs," in Proc. 3rd IEEE International Conference on Information Science and Technology, 2013.

[13] M. Sottile, T. Mattson and C. Rasmussen, Introduction to Concurrency in Programming languages, Chapman & Hall/CRC, 2010.

[14] T. Timothy Mattson, B. Beverly Sanders and B. Massingill, Patterns for parallel programming, Addison-Wesley Professional, 2004.

[15] M. Lutz, Programming Python 4th Edition, Sebastopol, CA: O'Reilly & Associates, Inc, 2011.

[16] R. Yehezkael, "Flexible Algorithms: Overview of a Beginners' Course," IEEE Distributed Systems Online, vol. 7, no. 11, pp. art. no. 0611-oy002, 2006.

[17] K. Hinsen, "Parallel Scripting with Python," IEEE Computing in Science and Engineering, vol. 9, no. 6, pp. 82-89, 2007.

[18] https://javacc.java.net.

[19] H. Rice, "Classes of recursively rnumerable sets and their decision problems," Trans. AMS, vol. 74, no. 2, pp. 358-366, 1953.

[20] D. Bacon, R. Rabbah and S. Shukla, "FPGA Programming for the Masses," CACM, vol. 56, no. 4, pp. 56-63, 2013.

[21] Message Passing Interface Forum, "MPI: A Message-Passing Inte4rface Standard, Version 3.0", High-Performance Computing Center Stuttgart, Stuttgart, Germany, 2012.

[22] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism, O'Reilly Media, Inc, 2007.

[23] A. Tagliasacchi, R. Dickie, A. Couture-Beil, M. J. M.J. Best, A. Fedorova and A. Brownsword, "Cascade: A Parallel Programming Framework for Video Game Engines," in Proc. Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA), in conjunction with ISCA, 2008.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009.

[25] M. Herlihy and N. Shavit, The Art of Multiprocessing Programming, Burlington, MA: Morgan Kaufmann Publs., 2008.