Performance Estimation for Task Graphs Combining Sequential Path Profiling and Control Dependence Regions

Fabrizio Ferrandi, Marco Lattuada, Christian Pilato, Antonino Tumeo
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Via Ponzio 34/5, Milano, Italy
{ferrandi,lattuada,pilato,tumeo}@elet.polimi.it

***Abstract*—The speed-up estimation of a parallelized code is crucial to efficiently compare different parallelization techniques or task graph transformations. Unfortunately, most of the times, during the parallelization of a specification, the information that can be extracted by profiling the corresponding sequential one (e.g. the most executed paths) are not properly taken into account. In particular, correlating sequential path profiling with the corresponding parallelized code can help in the identification of code hot spots, opening new possibilities for automatic parallelization. For this reason, starting from a well-known profiling technique, the Efficient Path Profiling, we propose a methodology that statically estimates the speed-up of a parallelized specification, just using the corresponding hierarchical task graph representation and the information coming from the dynamic profiling of the initial sequential specification. Experimental results show that the proposed solution outperforms existing approaches.**

I. Introduction

Today, Multiprocessor Systems-on-Chip (MPSoCs) are the de-facto standard for embedded system design [1]. *Performance analysis* [2] is a key step of the design process for such systems. Recording a program behavior and analyzing its performance allow new possibilities for code transformations by identifying hot spots or bottlenecks. This is an important procedure in multiprocessor systems, and becomes fundamental with embedded architectures, where program performance, memory occupation and code compactness are critical aspects. Profiling is one of the most known and studied techniques for performance analysis, used for hand-tuning of programs or for various smart compilation techniques. Common compilers implement control flow profiles, which are gathered through code instrumentation or statistical sampling of the program counter. In particular, once the application is translated in form of graph, these profiles can refer to vertices (basic blocks) or edges (branch transitions) and count how many times these elements are executed. The optimization process can thus focus on the most time consuming parts of the applications. Even if these profiles are widely adopted, they are not so accurate in the estimation of the mostly executed paths (sequences of branch transitions). Path profiling [3] is a well-known technique to obtain the frequency of the paths, improving the profiling information with a limited instrumentation overhead. Nevertheless, these profiling techniques have been scarcely investigated in parallel applications, currently represented by extending standard languages with annotations that describe parallelism, like OpenMP [4]. Normally, the programmer explicitly divides the application in tasks and then analyzes the resulting performance, through the use of proper analysis tools. Usually, profiling information is exploited only to estimate the performance of the single tasks, without considering the correlations among them. The estimations are then composed to obtain the best, average or worst performance estimation of the whole task graph. However, without considering the correlations, performance analysis tools are not able to gather important information on code hot spots and load balancing.

In this paper we propose a solution to estimate the performance of a parallelized specification by exploiting the basic blocks correlations existing among the different tasks. In particular, we adopt the path profiling technique to identify the correlations on the related sequential code and we project this information on the Control Dependence Regions [5] of each path. We also define a more efficient representation of the paths, called Control Region Path. Furthermore, we adopt the Hierarchical Task Graph (HTG) [6] as a representation for partitioned applications. In HTGs, a vertex may have associated another HTG in a hierarchical way, resulting more powerful than Direct Acyclic Graphs (DAGs), where feedback edges are not allowed. The hierarchy allows the representation of typical parallel embedded applications (e.g., loop-partitioned and real time), which are naturally described with cycles. The contributions of this paper can be summarized as follows:

- it proposes a methodology for the speed-up estimation of parallel code starting from the information gathered from the related sequential one;

- it applies this methodology to cyclic task graphs, extending the classes of applications that can be approached;

- it extends the Efficient Path Profiling (EPP) [3] with a new solution, the Hierarchical Path Profiling (HPP), which better identifies the basic blocks correlations, in particular when cycles are involved.

The remainder of this paper is organized as follows. Section II is about the related work. Section III gives some preliminary definitions while Section IV presents the motivation of this work. The proposed methodology is detailed in Section V and the experimental results are presented in Section VI. Finally, Section VII concludes the paper.

II. Related Work

One of the main purposes of profiling is the identification of the most executed paths inside a program, where the optimization algorithms will focus. There are several classes of profiling techniques which allow getting this type of information. Among them, we find edge profiling [7] and whole program profiling [8]. Edge profiling is a simple technique, but not necessarily cheap in terms of execution overheads and instrumentation code size, which only aims at recording information about how many times each branch transition occurs. From this information the path executions can be approximately estimated. On the other hand, whole program profiling usually gives very exact information about paths execution but with a bigger execution overhead cost. Path profiling, which counts the sequence of edges and basic blocks, is a trade-off between these two techniques. One of the most important work about path profiling is the Efficient Path Profiling [3] which will be detailed in Section III-B. This basic algorithm has been extended by different authors to support inter-procedural paths [9] and inter-iteration paths [10].

In [11] performance estimation for real-time embedded systems is discussed. This work considers best and worst case execution exploiting the concept of path-based analysis, but without leveraging the effectiveness of path decomposition proposed by [3]. Furthermore, it mainly considers the estimation of sequential applications. Several static timing analysis techniques, targeting the estimation of performance for embedded systems, are described in [12]. The target architecture considered is based on a single processor, and almost all the techniques discussed have high computational complexity, since they target the verification of real-time systems with hard or soft constraints. Furthermore, the presented average-case performance estimation techniques are limited by the number of paths generated, since they do not exploit any techniques for path decomposition.

Bammi *et al.* [13] propose a technique to estimate the performance of embedded applications without the need of a cycle accurate processor model. An instrumented source code, annotated with timing information, is generated by analyzing the object code, and then compiled and executed on the host to get an estimation of its performance on the target architecture. This technique allows obtaining performance estimations much faster than solutions based on Instruction Set Simulators (ISSs) [14], [15], [16], which cannot be easily exploited for the (fast) trade-off analysis required by an optimizing compiler. We adopt a similar approach to profile the code, but instead of analyzing the object code, we start from GIMPLE [17], the intermediate representation used by *GNU GCC* [18]. Being a higher intermediate representation, GIMPLE allows reducing the instrumentation overhead.

Fine grain instrumentation is also used in [19] to obtain accurate execution time and memory statistics. Similarly to [13], it is faster than ISS-based techniques, but still too slow with respect to the performance analysis tools used for exploration of parallelization. Our estimation technique, instead, statically and efficiently estimates the task graph performance through path profiling. Thus, such performance analysis tools could obtain better results by including this methodology.

In [20], synchronization operations are speculatively anticipated if they are on the most executed paths. In this case, path profiling information has been used to optimize communication between the threads, rather than performing estimation of parallelized specifications. It is straightforward to extend our methodology to also consider communication.

Profiles can also be used to estimate trip counts of loops [21]. Common loop-oriented optimization techniques may have benefits from a proper estimation of this number. Real time constraints analysis also benefits from trip counts estimations [22]. Considering paths, our solution is also able to compute this kind of information.

III. Preliminaries

In this section we introduce the basic elements to understand our estimation procedure for parallelized code. We describe the intermediate representations used by our methodology, we briefly present path profiling and, finally, we discuss the model of concurrency that has been adopted.

A. Intermediate representation

The proposed methodology works on the following intermediate representations, widely used in compilers:

- the *Control Flow Graph* (CFG) [23], a directed graph $G_{CFG} = (N, E_{CFG})$ which is an abstract representation of paths (sequences of branches) that might be traversed during the execution of a function;

- the *Control Dependence Graph* (CDG) [5], a directed graph $G_{CDG} = (N, E_{CDG})$ representing control dependences of basic blocks; that is if a basic block can control whether or not another basic block will be executed.

- the *Control Dependence Regions* (CDR) [5], a partitioning of the basic blocks in equivalence classes; two basic blocks are in the same region if they have the same set of control dependences in the CDG;

- the *Loop Forest* [24], a representation of the hierarchy of the loops contained into the CFG;

where N represents the basic blocks contained into the initial specification. The function $\gamma : C_i = \gamma(BB_j)$ returns the identifier of the Control Dependence Region C_i associated with the basic block BB_j.

Given the example of Fig. 1, its CFG is represented in Fig. 2. Its CDG and the control dependence regions which each basic block belongs to are instead shown in Fig. 3, where, for example $e_{1,2}$ represents that BB_2 is executed *iff* BB_1 has been executed and the value of the condition was *true*. On the other hand, operations in BB_4 have not any control dependences with BB_1, BB_2 and BB_3, and they can be executed in parallel if the data dependences are respected. Only one reducible loop, with BB_5 as header, is shown in the example. For the sake of simplicity, in the rest of the paper a loop will be identified with its header number (i.e. L_5). Basic blocks BB_5, BB_6, BB_7, BB_8 and BB_9 are considered to belong to L_5 and the related

```
   int fun_0(int c1, int c2, int c3,
   int * array, int size) {
      int index, a, b, c;
 1:   a = c2 + c1;                        // BB1
 2:   index = 0;                          // BB1
 3:   if(c1)                              // BB1
 4:      fun_1(&a);                       // BB2
      else
 5:      a *= 2;                          // BB3
 6:   a += b;                             // BB4
 7:   c = 1;                              // BB4
 8:   while(index < size) {               // BB5
 9:      array[index] = index;            // BB6
10:      if (c3)                          // BB6
11:         fun_2(array[index]);          // BB7
         else
12:         array[index]++;               // BB8
13:      index++;                         // BB9
      }
14:   b = a + c1;                         // BB10
15:   if(c2)                              // BB10
16:      fun_3(&c);                       // BB11
      else
17:      c*=2;                            // BB12
18:   return array[0] + a + b + c;        // BB13
   }
```

Fig. 1. Sequential implementation of the example function `fun_0`. On each line, the operation has a progressive number on the left side and the number of the basic block which the operation belongs to on the right side.

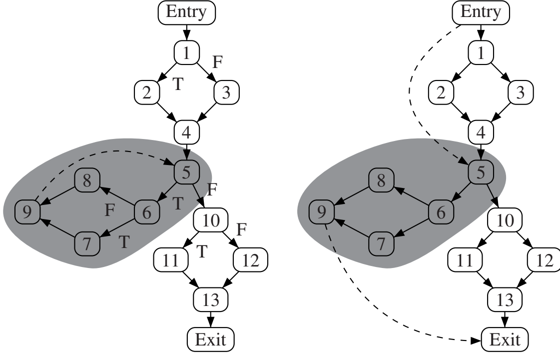

Fig. 2. The CFG of `fun_0` (on the left) and the related PG (on the right)

level of the hierarchy. The entire function itself is considered as the main loop L_0. Finally, since we interface the *GNU GCC* compiler [18], our intermediate representations are based on GIMPLE [17].

B. Path Profiling

Before defining Path Profiling, we need to introduce the concept of *path*. Let $G_{CFG} = (N, E_{CFG})$ be a CFG. The *path* P_p is defined as the sequence:

$$P_p = \{BB_1, BB_2, \ldots, BB_n\} \quad (1)$$

where $BB_i \in N$ and the pair of basic blocks $\langle BB_i, BB_{i+1} \rangle$ has the corresponding edge $e_{i,i+1} \in E_{CFG}$. Note that two basic blocks contiguous in a path are also contiguous in the execution trace which the path is extracted from. As described above, since the CFG represents all the paths that might be traversed during a program execution, it is possible to count the frequency of each path with an appropriate profiling of this representation. This technique is usually called *path profiling*.

Ball and Larus [3] proposed an algorithm to efficiently profile the execution frequency of paths in CFGs. This algorithm is known as Efficient Path Profiling (EPP) and it uses the

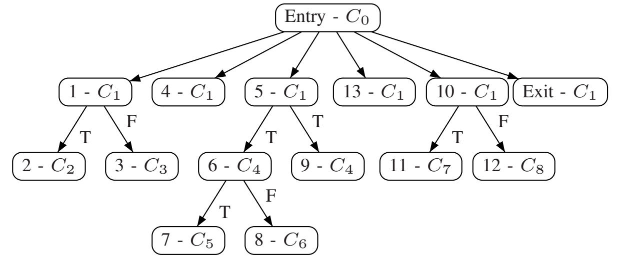

Fig. 3. The Control Dependence Graph of the function `fun_0`.

concept of *state* to model the valid paths (i.e., paths which are counted). These paths are only the ones that connect $Entry$ to $Exit$. CFGs with loops are managed by substituting each back-edge e_{jk} with two new edges connecting basic block $Entry$ with BB_k and basic block BB_j with $Exit$. The graph so obtained is named as *Path Graph (PG)*. Figure 2 shows the CFG associated to the example and the related Path Graph.

C. Model of Execution

In this work, we target embedded Multiprocessor Systems-on-Chip (MPSoCs) composed of different processing elements that communicate through a shared memory. We adopt explicit fork and join operations as model of concurrency. This programming model requires that each task spawning threads (called *fork task*) has a corresponding *join task*, which can be executed only after all the created threads have completed their execution. This concurrency model is well supported by the *OpenMP* [4] standard and the corresponding programs can run on such shared memory MPSoCs with a minimal operating system layer. Architecture properties are important for the correct performance evaluation of a specification as well as for path profiling. We take into account the architecture properties during the mapping of the GIMPLE nodes (that are language and processor independent) to the target assembler statements. Following an approach similar to the ones proposed in [13], [19], [25], we use an analytical model that, given the list of assembler statements associated with a GIMPLE node, is able to return an estimation of the number of cycles required by the target processors. We know to introduce some error in this estimation. However, we can accept this kind of approximations since the accuracy of such operation estimation is usually high, as shown in the literature, and we are focusing on a fast estimation technique to be used in task optimization.

Similarly to [6], we adopt the Hierarchical Task Graph (HTG) as the intermediate representation of a parallel program. In particular, the HTG is a directed graph whose vertices can be: *simple* (i.e. a task with no sub-tasks), *compound*, (i.e. a task that consists of other tasks in a HTG, for example higher level structures such as subroutines), *loop* (i.e. a task that represents a loop whose iteration body is a HTG itself). The hierarchical task graph can be extracted from the control flow graph of a sequential program by identifying the edges through data and control dependences analysis. This results in an acyclic graph, where the task can be classified as: *fork* (i.e.

TABLE I
CLOCK CYCLES REQUIRED FOR THE OPERATIONS IN `FUN_0`.

Op	n. cycles	Op	n. cycles	Op	n. cycles
o_1	10	o_7	10	o_{13}	10
o_2	10	o_8	10	o_{14}	10
o_3	10	o_9	10	o_{15}	10
o_4	20,500	o_{10}	10	o_{16}	20,500
o_5	10	o_{11}	1,010	o_{17}	10
o_6	10	o_{12}	20	o_{18}	40

tasks with multiple successors), *join* (i.e. tasks with multiple predecessors), *normal* (i.e. all the remaining tasks).

IV. MOTIVATION

Given the profiling of a sequential specification, it can be difficult to estimate the speed-up introduced by one of its possible parallelizations, even admitting some approximations and supposing that architectural effects (task creation/destruction/synchronization and communication) can be predicted and modeled as additive. For example, consider the function in Fig. 1, executed 10 times. The number of cycles required by each operation in the sequential specification is analyzed following an approach similar to [13], [19], [25] and an example is shown in Table I. For the sake of simplicity, in the rest of the example we assume that the execution time of the sub-functions `fun_1`, `fun_2` and `fun_3` is fixed and not data-dependent. Nevertheless, this information is not sufficient to compute a good estimation of the speed-up obtained by one of its possible parallelizations (e.g. the one shown in Fig. 4 whose corresponding task graph is shown in Fig. 5). In fact, there are several issues that should be considered when estimating how long the execution of the task graph takes. First, the number of loop iterations have to be accurately estimated, since, as in this case, it heavily affects the execution time of the task where the loops are contained (i.e., $Task2$). A more precise information about the average loop iterations number can be obtained using edge profiling techniques, but this information is not sufficient yet to produce correct estimation results. In fact, the speed-up of the function depends on how the values of conditions (i.e., `c1` and `c2`) are correlated, activating or not the execution of functions `fun_1` and `fun_3`.

In particular, let us consider the two following situations:

A) `c1` and `c2` always have opposite values; this means that the basic blocks executed in the same path are BB_2 and BB_{12} or BB_3 and BB_{11}.

B) `c1` and `c2` always have the same values (true or false); this means that the basic blocks executed in the same path are BB_2 and BB_{11} or BB_3 and BB_{12}.

Let us also assume that the probability of condition `c1` being true is 0.50 and $Task2$ has an estimated execution time of 10,520 cycles (the loop is executed 10 times and the condition `c3` is always true).

In the first situation the execution of the sequential specification requires 31,130 cycles. The execution of the parallel code requires 20,580 cycles if `c1` is true, 20,560 otherwise. The average parallel execution time is 20,570, so the real speed-up is $\mu_A = 1.5133$. In the second situation the execution of

```
   int fun_0(int c1, int c2, int c3,
   int * array, int size) {
    int index, a, b, c;
    //TASK0
    #pragma omp parallel sections num_threads(3) {
       //TASK1
       #pragma omp section {
 1:       a = c2 + c1;                          // BB1
 3:       if(c1)                                // BB1
 4:          fun_1(&a);                         // BB2
          else
 5:          a *= 2;                            // BB3
 6:       a += b;                               // BB4
14:       b = a + c1;                           // BB10
       }
       //TASK2
       #pragma omp section {
 2:       index = 0;                            // BB1
 8:       while(index < size) {                 // BB5
 9:          array[index] = index;              // BB6
10:          if (c3)                            // BB6
11:             fun_2(array[index]);            // BB7
             else
12:             array[index]++;                 // BB8
13:          index++;                           // BB9
          }
       }
       //TASK3
       #pragma omp section {
 7:       c = 1;                                // BB4
15:       if(c2)                                // BB10
16:          fun_3(&c);                         // BB11
          else
17:          c*=2;                              // BB12
       }
    }
    //TASK4
18: return array[0] + a + b + c;               // BB13
   }
```

Fig. 4. Parallel implementation of the function `fun_0`

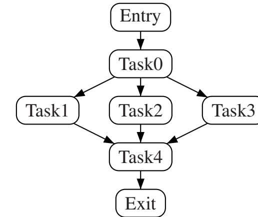

Fig. 5. Task Graph associated with the parallelization proposed in Fig. 4

the sequential specification requires in average 31,130 cycles (51,620 if `c1` is true, 10,640 otherwise). The parallel execution time is in average 15,570 cycles (20,580 if `c1` is true, 10,560 otherwise), so the real speed-up is $\mu_B = 1.9994$.

Unfortunately, in these cases the techniques like edge profiling or EPP are not able to detect this difference. In particular,

TABLE II
RESULTS OF EPP APPLIED TO THE FUNCTION `FUN_0` EXECUTED 10 TIMES.

Path	BBs	A	B
P_0	En,1,2,4,5,6,7,9,Ex	5	5
P_1	En,1,2,4,5,6,8,9,Ex	0	0
P_2	En,1,2,4,5,10,11,13,Ex	0	0
P_3	En,1,2,4,5,10,12,13,Ex	0	0
P_4	En,1,3,4,5,6,7,9,Ex	5	5
P_5	En,1,3,4,5,6,8,9,Ex	0	0
P_6	En,1,3,4,5,10,12,13,Ex	0	0
P_7	En,1,3,4,5,10,12,13,Ex	0	0
P_8	En,5,6,7,9,Ex	100	100
P_9	En,5,6,8,9,Ex	0	0
P_{10}	En,5,10,11,13,Ex	5	5
P_{11}	En,5,10,12,13,Ex	5	5

consider the information about frequencies extracted by EPP and reported in Table II. For example, in the first situation, the path $En, 1, 3, 4, 5, 10, 11, 13, Ex$ is not counted because the loop L_5 is executed for at least one iteration (i.e., BB_5 and BB_{10} are not contiguous in the execution trace). The only paths considered by EPP are $En, 1, 3, 4, 5, 6, 7, 9, Ex$ and $En, 5, 10, 11, 13, Ex$. In conclusion, when a loop iteration is executed at least once between two conditional statements, the EPP algorithm looses the correlations among the paths before and after the loop, giving the same results for both the cases. Therefore, all the methodologies that use this information would estimate the same speed-up.

V. Proposed Methodology

The proposed methodology aims at providing a static estimation of the speed-up introduced by parallelization. It can be divided into three steps. Firstly, a path profiling of the sequential specification is performed, considering the loop hierarchy. Secondly, the profiling results are organized into a more compact representation based on the control dependence regions. Thirdly, the speed-up of the parallelized code is statically estimated, efficiently combining the information obtained from this representation on the related HTG.

A. Hierarchical Path Profiling

In this section we describe the profiling technique, the Hierarchical Path Profiling (HPP), that extends the EPP and is able to maintain the correlation between what happens before and after a loop. The HPP has some analogies with Structural Path Profiling (SPP) [26], in particular regarding the loop hierarchy. However, in HPP the paths can cross loop boundaries, while in SPP they cannot.

The HPP is applied to the PG described in Section III-B, with a different definition of *valid path*. In particular, a path $P_p = \{Entry, BB_i, BB_{i+1}, \ldots, BB_j, Exit\}$ is *valid* if it starts from *Entry* and ends in *Exit*, like in EPP [3], BB_i and BB_j are connected by a back-edge (i.e., $e_{ji} \in E_{CFG}$) or they belong only to the loop L_0. According to this definition, in the proposed example, the path $En, 1, 2, 4, 5, 6, 7, 9, Ex$ is not valid anymore, since there is not the edge $e_{9,1}$ into the CFG and BB_9 belongs to L_5. The path $En, 5, 6, 7, 9, Ex$ is still valid, since there is the edge $e_{9,5}$ in the CFG.

Then, all the HPP paths are clustered with respect to the loop which they belong to. In particular, the path P_p is said to belong to the loop L_l and, thus, to the cluster HP_l, if L_l is the innermost loop which BB_i (i.e., the first basic block besides $Entry$) belongs to. For example, the path $En, 5, 6, 7, 9, Ex$ is contained into HP_5. We partially relax also the assumption that each consecutive pair of basic blocks of a path have to be contiguous in the execution trace. In this way, differently from EPP, the path $En, 1, 2, 4, 5, 10, 11, 13, Ex$ can effectively be extracted from the execution trace also when the loop has been executed for, at least, one iteration.

To apply HPP we classify PG edges into four categories:

- *Starting Path*: they connect a basic block outside a loop with a loop header; $e_{4,5}$ is the only edge in the example;

- *Ending Path*: the edges which directly connect a basic block to $Exit$ (i.e., the edges $e_{13,Ex}$ and $e_{9,Ex}$);

- *Exit Loop*: they connect a basic block inside a loop with a basic block outside a loop (i.e., the only edge $e_{5,10}$);

- *Normal*: all the other edges.

Algorithm 1 Pseudo-code of *Hierarchical Path Profiling*

```
1:  current_loop = L_0
2:  BB_last = En
3:  curr_path = {En}
4:  while ending of function execution do
5:      BB_new = get currently executed basic block
6:      if e_{BB_last,BB_new} ∈ StartingPath then
7:          current_loop = L_new
8:          append BB_new to curr_path
9:          add curr_path to idle paths
10:         curr_path = {En}
11:     else if e_{BB_last,Ex} ∈ EndingPath then
12:         append Ex to curr_path and increment its frequency
13:         a new path p = En starts
14:     else if e_{BB_last,BB_new} ∈ ExitLoop then
15:         update current_loop
16:         idle path of the current loop becomes curr_path
17:     end if
18:     BB_new is appended to curr_path
19:     BB_last = BB_new
20: end while
```

The proposed algorithm operates as described in Algorithm 1. Considering the example of Figure 2 and the situation in which `c1` and `c2` are both true, it behaves as follows. When the function execution begins, we start a new path $p_i = En$ (line **3**). The path is updated (line **18**) with the executed basic blocks until we reach BB_5. At this point the current path is $En, 1, 2, 4$. Since $e_{4,5}$ is a *Starting Path Edge* (line **6**), when the execution of BB_5 starts (i.e., $BB_{new} = BB_5$), p_i becomes idle (line **9**) and a new path p_j starts (line **10**), also including the current basic block (line **18**). At this point we have $p_j = En, 5$, which is updated until the execution of BB_9. Since $e_{9,Ex}$ is a *Ending Path edge* (line **11**), the current path $p_j = P_8 = En, 5, 6, 7, 8, 9, Ex$ is then terminated (line **12**), and its frequency incremented by one. Subsequently, a new path p_j starts (line **13**) and it behaves as described above for all the ten iterations of the loop. At the end, when the execution of BB_{10} starts (i.e., $e_{5,10} \in ExitLoop$ is traversed), p_i returns active as $p_i = En, 1, 2, 4, 5, 10$, deleting the current path (line **16**). The path is updated until we reach $e_{13,Ex}$, which is a *Ending Path edge* (line **11**), incrementing the frequency of the path $p_i = P_2 = En, 1, 2, 4, 5, 10, 11, 13, Ex$. Then, the algorithm stops, since the execution of the function is terminated.

The presence of idle paths is one the most important differences between EPP and HPP. In each instant, EPP allows only one path to be *live*. At the opposite, the number of live paths in HPP is the nesting level of the current loop. Algorithm 1, which analyzes a basic block at each iteration, is linear with the number of basic block executed into the trace. In Table III we show the results obtained by applying the HPP technique to the example of Fig. 1, for the two cases

TABLE III

RESULTS OF HPP ON THE FUNCTION `fun_0`. P_i REPRESENTS THE PROFILED PATHS. RELATIVE AND ABSOLUTE FREQUENCIES ARE REPORTED OR THE TWO PROPOSED CASES, `HP` THE HIERARCHICAL CLUSTER WHICH EACH PATH BELONGS TO, `CRP`S THE RELATED CRP, AND $PC_{i,t}$ THE CONTRIBUTIONS OF TASK `t` TO EACH PATH.

HP	P_i	Basic Blocks	CRPs	Absolute A	Absolute B	Relative A	Relative B	$PC_{i,0}$	$PC_{i,1}$	$PC_{i,2}$	$PC_{i,3}$	$PC_{i,4}$
HP_0	P_2	En,1,2,4,5,10,11,13,Ex	$HC_{0,0}, HC_{0,1}, HC_{0,2}, HC_{0,7}$	0	5	0.0	0.5	0	20,540	10,520	20,520	40
	P_3	En,1,2,4,5,10,12,13,Ex	$HC_{0,0}, HC_{0,1}, HC_{0,2}, HC_{0,8}$	5	0	0.5	0.0	0	20,540	10,520	30	40
	P_6	En,1,3,4,5,10,11,13,Ex	$HC_{0,0}, HC_{0,1}, HC_{0,3}, HC_{0,7}$	5	0	0.5	0.0	0	50	10,520	20,520	40
	P_7	En,1,3,4,5,10,12,13,Ex	$HC_{0,0}, HC_{0,1}, HC_{0,3}, HC_{0,8}$	0	5	0.0	0.5	0	50	10,520	30	40
HP_5	P_8	En,5,6,7,9,Ex	$HC_{5,1}, HC_{5,4}, HC_{5,5}$	100	100	1.0	1.0	0	0	1,050	0	0
	P_9	En,5,6,8,9,Ex	$HC_{5,1}, HC_{5,4}, HC_{5,6}$	0	0	0.0	0.0	0	0	60	0	0

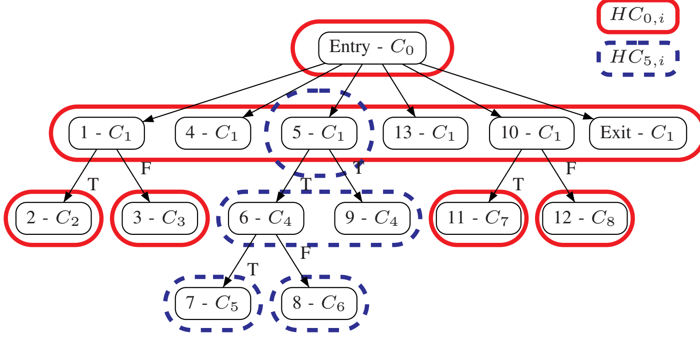

Fig. 6. Representation of the hierarchical control dependence regions $HC_{l,i}$ of the function `fun_0`, clustering the basic blocks belonging to the same control equivalent region C_i of the loop L_l.

discussed in Section IV. Comparing the HPP results with the ones obtained by EPP and shown in Table II, it can be noticed that HPP is able to maintain the correlation between what happens before and after the execution of L_5 (i.e., between the execution of basic blocks BB_2 and BB_{11}).

B. Control Region Paths

Once the HPP profiling has been performed and the paths have been hierarchically clustered, we project the control dependence regions onto the paths. In particular, let L_l be a loop, we define the *Hierarchical Control Dependence* region (HC) as:

$$HC_{l,i} = \{ \ BB_j \mid C_i = \gamma(BB_j) \wedge \exists P_p \in HP_l : BB_j \in P_p \ \}$$

For the example shown in Fig. 1, the hierarchical control dependence regions are represented in Figure 6. In particular, the dashed lines represent the regions for the loop L_5 and the filled ones for the loop L_0. Note that BB_5, being the header of L_5, is included both into $HC_{0,1}$ and $HC_{5,1}$. In fact, regardless the loop is executed or not, at the higher level of the hierarchy, the header BB_5 (i.e., the test of the condition) is always executed at least one time. Therefore, each region $HC_{l,i}$ contains all the basic blocks of L_l that are dependent on the same value of the control condition. Thus, when this value has been evaluated, the region and all the related basic blocks will be executed for sure. Thus, we can represent each path (i.e., sequence of basic blocks) of the loop L_l as the set of control regions to be executed, i.e., the *Control Region Paths*.

In particular, let $P_p \in HP_l$ be a path of loop L_l, the *Control Region Path* (CRP) CRP_p associated with path P_p is defined as:

$$CRP_p = \{HC_{l,i} | \exists BB_j \in P_p : C_i = \gamma(BB_j) \wedge P_p \in HP_l\}$$

Since the function γ is surjective for each loop L_l of the hierarchy, the size of the control region path CRP_p results equal or smaller than the size of the corresponding path P_p. This produces a more compact representation of the paths, without loosing any information. The CRPs of the example in Fig. 1 are shown in Table III.

C. Static Task Graph Execution Time Estimation

Let $G_{CFG} = (N, E_{CFG})$ be the CFG of a sequential specification and $HTG_0 = (V, E)$ be the HTG related to one of its parallelization. HTG_0 is recursively analyzed with the procedure described by Algorithm 2.

In particular, the methodology analyzes all the tasks of $HTG_l = (V_l, E_l)$ in topological order and starts by computing, for each CRP_i in HP_l, the contribution $CC_{l,i,t}$ (line **4**) given by task $v_t \in V_l$ to each region $HC_{l,i}$:

$$CC_{l,i,t} = \sum_{\forall s: o_s \in v_t \wedge o_s \in BB_h \wedge BB_h \in HC_{l,i}} c_s \qquad (2)$$

where c_s is the number of cycles required by the operation $o_s \in v_t$. Then, the contribution $PC_{p,t}$ (line **19**) given by task v_t to the path P_p is computed as:

$$PC_{p,t} = \sum_{\forall i: HC_{l,i} \in CRP_p} CC_{l,i,t} \qquad (3)$$

i.e., the sum of all the contributions of the regions that belong to P_p.

Note that if $HC_{l,i}$ contains the header of a loop L_n completely contained into the task v_t (lines **5-13**), the Equation 3 is also applied to the CRPs of the loop L_n and the execution time LC_n (line **12**) associated with the loop L_n can be estimated as:

$$LC_n = N_n * \frac{\sum_{P_q \in HP_n} PC_{q,t} * f_q}{\sum_{P_q \in HP_n} f_q} \qquad (4)$$

where f_q is the frequency associated with the path P_q and N_n is the average number of iterations for the loop L_n.

Thus, the related $CC_{l,i,t}$ is updated (line **13**) as follows:

$$CC_{l,i,t} = CC_{l,i,t} + LC_n \qquad (5)$$

In fact, when $HC_{l,i}$ contains the basic block BB_n, which represents the header of a loop L_n nested in L_l, the execution of $HC_{l,i}$ must consider the additional cycles due to the loop L_n. This process continues, as described above, until the contributions of the task have been computed for all the paths in HP_l (lines **3-20**). Defined $ACC_{p,t}$ as the execution time

needed to execute the operations of the path P_p in the tasks from $Entry$ to $v_t \in V_l$ (and assumed $ACC_{p,Entry} = 0 \ \forall P_p$), the task v_t contributes (line **21**) as follows:

$$ACC_{p,t} = \max_{v_u \in pred(v_t)} ACC_{p,u} + PC_{p,t} + c_t \quad \forall P_p \in HP_l \quad (6)$$

where $v_u \in pred(v_t)$ is a predecessor of v_t in HTG_l (i.e., $e_ut \in E_l$) and c_t is the overhead that can be associated with the creation/destruction/synchronization of task v_t. Note that Eq. 6 can also be applied to task graphs that are not compliant with the fork/join model. In fact, this model of execution only refers to the programming model supported by the target architecture and not to a limit of the methodology. The overall task graph execution time for HTG_l (line **24**) is then computed as a weighted average of the contributions given by all the paths:

$$HTC_l = N_l * \frac{\sum_{P_p \in HP_l} ACC_{p,Exit} * f_p}{\sum_{P_p \in HP_l} f_p} \quad (7)$$

Since L_l (represented by HTG_l) is nested in L_j (eventually L_0), and HTG_l is associated only to one task $v_l \in V_j$, the contribution $CC_{j,i,l}$ (line **16**) is then updated:

$$CC_{j,i,l} = CC_{j,i,l} + HTC_l \quad (8)$$

Thus, the analysis can recursively continue until the computation of HTC_0, associated to L_0 and representing the estimation of the parallelized specification, has been completed.
Let E_s be the performance of the original specification, the estimated speed-up introduced with the parallelization is then computed as:

$$\mu = \frac{E_s}{HTC_0} \quad (9)$$

Algorithm 2 Pseudo-code of *estimate*($HTG_l(V_l, E_l)$)

```
1:  for all task v_t ∈ V_l do
2:    for all P_p ∈ HP_l do
3:      for all HC_{l,i} contained into CRP_p do
4:        compute the region contribution CC_{l,i,t}
5:        if BB_n ∈ HC_{l,i} and L_n completely nested in v_t then
6:          for all HC_{n,i} associated to L_n do
7:            compute the region contribution CC_{n,i,t}
8:          end for
9:          for all P_q ∈ HP_n do
10:           compute the path contribution PC_{q,t}
11:         end for
12:         compute the loop execution cost LC_n
13:         update the CC_{l,i,t} with LC_n
14:       else
15:         if BB_n ∈ HC_{l,i} and HTG_n is associated to v_t then
16:           update CC_{l,i,t} with HTC_n =estimate(HTG_n)
17:         end if
18:       end if
19:       compute the path contribution PC_{p,t}
20:     end for
21:     update the cost of the path ACC_{p,t}
22:   end for
23: end for
24: compute the overall task graph execution time HTC_l
```

As shown by the Algorithm 2, the procedure for estimating $HTG_l(V_l, E_l)$ is composed by an outermost loop repeated $|V_l|$ times. For each task, all the paths at the current level of the hierarchy are analyzed and they can contain, in the worst case, at least one operation for each hierarchical control dependence region. Therefore, the analysis **3-20** is repeated, in the worst case, $|C|$ times (i.e., the number of control regions contained into the specification) and the analysis **2-22** is repeated $|P|$ times, where $|P|$ is the number of paths. The complexity of the estimation for task graph HTG_l is, thus, $O(|V_l| \cdot |P| \cdot |C|)$.

Applying the methodology to the example presented in Fig. 1 and Fig. 4, we obtain the results reported in the right side of Table III. First, we compute $PC_{p,t}$ for all the paths and all the tasks. The execution time required by loop L_5 has been estimated as $LC_5 = 10,500$ (the only path executed is CRP_8). Therefore, $PC_{i,2} = 10,520$ for all the $P_i \in HP_0$. The sequential execution time E_s is $31,130$. Note that it can also be computed by using the proposed methodology and considering all the operations in the same task $v_a \in V_0$. Finally, we can compute the speed-up for the two situations presented in Section IV:

A) the paths executed are P_3 and P_6, so the execution time estimated for the parallel version is:

$$E_p = HTC_0 = \frac{20,580 * 0.5 + 20,560 * 0.5}{0.5 + 0.5} = 20,570$$

and the related speed-up is:

$$\mu_A = \frac{31,130}{20,570} = 1.5133$$

B) the CRPs executed are P_2 and P_7, so the execution time estimated for the parallel version is:

$$E_p = HTC_0 = \frac{20,580 * 0.50 + 10,560 * 0.50}{0.50 + 0.50} = 15,570$$

and the related speed-up is:

$$\mu_B = \frac{31,130}{15,570} = 1.9994$$

In conclusion, the proposed methodology, differently from EPP, is able to correctly estimate the speed-up in the two situations discussed in Section IV.

VI. Experimental Results

The proposed methodology has been implemented in C++ inside PandA [27], a hardware/software co-design framework based on the *GNU GCC* compiler [18].

The considered target architecture is an embedded MPSoC composed by eight ARM920 processors with a shared memory. Each processor has 16KB of instruction cache and 8KB of data cache. The data cache is write-through and adopts a write-update coherency policy. We slightly modified the SimIt-ARM cycle-accurate simulator [14] to model such architecture. In particular, since SimIt-ARM does not support multi-core simulation, we modified it to support concurrent tasks execution on different cores with private caches. Communication costs are not addressed in this work, but the extension is straightforward (i.e., by modifying Eq. 6). In this section we compare the two profiling techniques discussed in this paper (EPP and HPP) from the point of view of the instrumentation.

TABLE IV
BENCHMARK CHARACTERISTICS. LINES IS THE NUMBER OF SOURCE CODE LINES, FUN THE NUMBER OF FUNCTIONS (EXCEPT THE SYSTEM AND MATH LIBRARY ONES), LOOP THE NUMBER OF LOOPS, FD THE MAXIMUM DEPTH OF THE LOOP FORESTS, IF THE NUMBER OF CONDITIONAL CONSTRUCTS, DATASET THE BENCHMARK DATASET AND CYCLES THE NUMBER OF CYCLES SPENT FOR THE SEQUENTIAL EXECUTION ON THE SIMIT SIMULATOR WITH DIFFERENT OPTIMIZATION LEVELS.

Benchmark	Lines	Fun	Loop	FD	If	Dataset	Cycles -O0 opt. level	Cycles -O2 opt. level
basicmath	402	3	20	6	10	large	$8.58 \cdot 10^{10}$	$6.63 \cdot 10^{10}$
blowfish	588	9	15	4	10	(test)	$6.70 \cdot 10^{11}$	$4.82 \cdot 10^{11}$
corners detection	2928	23	50	4	179	large	$2.10 \cdot 10^{9}$	$1.01 \cdot 10^{9}$
Delayline	2033	52	60	3	25	(w/o inputs)	$1.10 \cdot 10^{6}$	$9.91 \cdot 10^{5}$
dijkstra	352	12	13	2	14	large	$8.93 \cdot 10^{9}$	$5.56 \cdot 10^{9}$
fft	855	23	11	1	37	(test)	$4.22 \cdot 10^{10}$	$3.66 \cdot 10^{10}$
fft6	1133	34	32	3	37	(test)	$1.05 \cdot 10^{10}$	$1.02 \cdot 10^{10}$
fmm	4417	94	117	2	257	input.16384	$9.45 \cdot 10^{11}$	$7.67 \cdot 10^{11}$
graphsearch	2327	43	44	2	94	exampleGraph_01	$4.70 \cdot 10^{5}$	$4.67 \cdot 10^{5}$
jacobi1	892	20	14	3	30	(test)	$8.17 \cdot 10^{11}$	$5.97 \cdot 10^{11}$
JPEG encoder	1688	19	71	3	67	input_small	$5.27 \cdot 10^{8}$	$4.74 \cdot 10^{8}$
openmpbench	564	13	30	3	4	(w/o inputs)	$1.93 \cdot 10^{10}$	$1.72 \cdot 10^{10}$
smoothing	2928	23	50	4	179	input_small	$2.93 \cdot 10^{9}$	$1.20 \cdot 10^{9}$
stringsearch	3076	12	22	2	19	(w/o inputs)	$1.77 \cdot 10^{6}$	$1.59 \cdot 10^{6}$
water-nsquared	2076	16	62	5	45	input	$9.55 \cdot 10^{11}$	$7.37 \cdot 10^{11}$

Then, we compare the methodology presented in Section V against three other common speed-up estimation models and the real speed-up obtained by the execution of the source code on the simulator on a set of manually partitioned applications extracted from MiBench [28], from Splash 2 [29] and from OmpSCR [30], that are three free suites of representative benchmarks for embedded and parallel computing. The benchmarks and their characteristics are reported in Table IV.

A. Path profiling

Both the EPP and the HPP techniques have been implemented inside our framework, without any optimizations. The results related to instrumentation and paths counting are reported in Table V. The instrumented source code is generated starting from the GIMPLE code at the end of the target independent optimization flow. For this reason, different results (see $Inc.\&Init.$ and NP) are obtained when changing the optimization level of the GNU GCC compiler. This also means that the estimation takes into account the middle-end optimizations, when activated. Then, we executed the instrumented code for 100 times and averaging the resulting execution times. The profiling has been performed on a host linux machine with the Intel Xeon X5355 CPU (4 cores at 2,33 GHz with 4 MB of L2 cache per couple of cores).

Starting from the same path graph, both the techniques count a path each time they reach the end of a function or of a loop. This results in the same number of path counter writes ($write$) for the two methods. The number of paths (NP) obtained with HPP is instead lower with respect to EPP. In fact, HPP performs a different path composition when loops are involved. As described above, EPP considers the path which enters in the loop and the path which exits from it as two distinct paths, when at least one iteration is performed, while HPP fuses these two paths into a single one. Finally, the instrumentation overhead introduced in both the techniques (*oh diff*) ranges from 20% to 200%. This is not a limitation, since we profile on a host system much faster than the target architecture or its cycle-accurate simulator. However, these number cannot be directly compared to Ball and Laurus' implementation. Their performance analysis tool, in fact, instruments (SPARC) binary executables, reducing the overheads by performing data-flow analysis to exploit the architectural registers. Our tool, instead, implements the two techniques completely in software, acting on the architecture independent intermediate representation before the object code generation and producing an architecture agnostic instrumentation. Table V shows that HPP has an overhead systematically lower than EPP. In fact, since HPP uses a different definition of valid paths, we have been able to reduce the number of activated paths, reducing the data structure needed to store them.

B. Speedup estimation

In this section we compare the real speed-up obtained by simulating with SimIt the sequential and parallelized source codes to the following estimation models:

- *Case A*: the contribution of each task is based on its worst case execution time [11] and the average number of iterations for not countable loops has been set to 5, similarly to [31];
- *Case B*: the contribution of each task is based on an average execution time [11] where the branch probabilities are considered equiprobable. The number of iterations has been set as in *Case A*;
- *Case C*: the number of iterations and the branch probabilities are based on the results obtained by a dynamic profiling with EPP;
- *Case D*: this case refers to the methodology proposed in this paper and detailed in Section V.

These methods have been applied to the benchmarks in Table IV with different levels of compiler optimizations. The results are reported in Table VI. Note that there is not any methodology able to exploit all the information coming from the EPP preserving the HTG structure since EPP identifies paths which cross the boundaries of the single task graph. However, EPP results can still be used to easily estimate the branch probabilities and the number of iterations and can be used as inputs by the Algorithm 2 described in Section V.

Since the target compiler for ARM processors may perform target dependent optimizations, some inaccuracies could occur.

TABLE V

COMPARISON OF THE EPP AND HPP TECHNIQUES. INC. & INIT. COUNTS THE INCREMENTS AND INITIALIZATIONS OF VARIABLES, WRITE THE PATH COUNTER WRITES, NP THE ACTIVATED PATHS, OH DIFF IS THE DIFFERENCE BETWEEN HPP AND EPP OVERHEADS.

	-O0 opt. level						-O2 opt. level					
	EPP		HPP			oh	EPP		HPP			oh
	Inc.& Init.	NP	Inc. & Init.	NP	write	diff (%)	Inc. & Init.	NP	Inc. & Init.	NP	write	diff (%)
basicmath	$1.260 \cdot 10^7$	37	$1.255 \cdot 10^7$	23	$2.745 \cdot 10^6$	-2.5	$1.260 \cdot 10^7$	39	$1.255 \cdot 10^7$	25	$2.745 \cdot 10^6$	-2.2
blowfish	$2.115 \cdot 10^5$	57	$1.786 \cdot 10^5$	38	$6.725 \cdot 10^4$	-0.8	$2.114 \cdot 10^5$	57	$1.785 \cdot 10^5$	38	$6.705 \cdot 10^4$	-6.6
corners detection	$1.473 \cdot 10^6$	102	$1.473 \cdot 10^6$	76	$2.095 \cdot 10^5$	-4.7	$1.516 \cdot 10^6$	150	$1.517 \cdot 10^6$	124	$2.095 \cdot 10^5$	-5.9
DelayLine	$8.228 \cdot 10^3$	68	$8.749 \cdot 10^3$	41	$2.294 \cdot 10^3$	0.0	$8.221 \cdot 10^3$	66	$8.741 \cdot 10^3$	40	$2.292 \cdot 10^3$	0.0
dijkstra	$7.998 \cdot 10^7$	39	$7.998 \cdot 10^7$	28	$2.050 \cdot 10^7$	-1.9	$8.016 \cdot 10^7$	43	$8.016 \cdot 10^7$	26	$2.041 \cdot 10^7$	-1.7
fft	$4.865 \cdot 10^8$	44	$4.865 \cdot 10^8$	26	$1.509 \cdot 10^8$	-0.5	$4.942 \cdot 10^8$	43	$4.942 \cdot 10^8$	25	$1.509 \cdot 10^8$	-2.1
fft6	$7.025 \cdot 10^8$	71	$7.227 \cdot 10^8$	48	$1.963 \cdot 10^8$	-2.1	$7.025 \cdot 10^8$	72	$7.227 \cdot 10^8$	48	$1.963 \cdot 10^8$	-0.1
fmm	$4.635 \cdot 10^8$	399	$4.647 \cdot 10^8$	281	$1.075 \cdot 10^8$	-1.7	$4.635 \cdot 10^8$	399	$4.649 \cdot 10^8$	279	$1.074 \cdot 10^8$	-0.5
graphsearch	$2.325 \cdot 10^3$	63	$2.331 \cdot 10^3$	46	$4.050 \cdot 10^3$	-1.0	$2.466 \cdot 10^3$	62	$2.577 \cdot 10^3$	45	$3.500 \cdot 10^2$	-3.3
jacobi1	$3.000 \cdot 10^8$	37	$3.000 \cdot 10^8$	28	$1.000 \cdot 10^8$	-0.4	$3.000 \cdot 10^8$	37	$3.000 \cdot 10^8$	28	$1.000 \cdot 10^8$	-0.4
JPEG encoder	$5.994 \cdot 10^5$	133	$6.014 \cdot 10^5$	75	$1.509 \cdot 10^5$	-4.6	$5.995 \cdot 10^5$	131	$6.015 \cdot 10^5$	74	$1.509 \cdot 10^5$	-2.2
openmpbench	$7.032 \cdot 10^8$	64	$7.033 \cdot 10^8$	38	$2.341 \cdot 10^8$	-1.9	$7.032 \cdot 10^8$	64	$7.033 \cdot 10^8$	38	$2.341 \cdot 10^8$	-3.2
smoothing	$5.703 \cdot 10^6$	35	$5.587 \cdot 10^6$	22	$1.741 \cdot 10^6$	-8.7	$5.703 \cdot 10^6$	35	$5.587 \cdot 10^6$	22	$1.741 \cdot 10^6$	-7.1
stringsearch	$1.085 \cdot 10^6$	21	$1.088 \cdot 10^6$	13	$3.542 \cdot 10^5$	-0.5	$1.085 \cdot 10^6$	24	$1.088 \cdot 10^6$	15	$3.542 \cdot 10^5$	-0.7
water-nsquared	$1.954 \cdot 10^8$	164	$1.963 \cdot 10^8$	99	$3.814 \cdot 10^7$	0.0	$1.953 \cdot 10^8$	162	$1.963 \cdot 10^8$	98	$3.812 \cdot 10^7$	-5.0

However, we have verified that their impact is similar on both the parallelized and the sequential code. So we can confirm that the speed-up ratio estimation is not affected. Analyzing the results in Table VI with respect to the benchmark characteristics shown in Table IV, we can make the following considerations. In *basicmath* and *blowfish*, the profiling-based techniques obtain far better results when no optimizations are applied. On the other hand, the differences become negligible when the code is restructured by the optimizations. One of the reasons is the introduction of loop optimizations by the compiler. Nevertheless, the profiling techniques reach an accurate estimation in both the cases. In the *smoothing*, *fmm* and *Delayline* benchmarks, the technique based on EPP information (C) obtains very poor results compared to the other techniques. In these benchmarks there are several control constructs and loops, and EPP looses all the correlations among the paths, leading to a highly inaccurate estimation. The HPP-based technique (D), instead, is able to correctly model such cases. The *Delayline* and the *fmm* benchmarks are very interesting when analyzing the quality of the parallelization. In fact, the approaches A, B and C estimates the presence of a speed-up in the parallelized code. However, when such code is executed on the simulator, we see that the parallelization is not efficient at all, and no speed up is obtained ($\mu_{SimIt} = 1$). In both these cases, the proposed technique correctly estimates the lack of any speed-up. This means that our methodology may be highly suitable during the design space exploration, allowing the designer to obtain a fast preliminary evaluation on different parallelization approaches without requiring multiple, time-consuming simulations or executions on the target platform. In *smoothing*, the control constructs are very unbalanced (i.e., some branches have a larger probability to be taken) and the approach based on worst case (A) is able to model this situation, obtaining results that are very close to the proposed methodology. However, code restructuring due to optimizations changes the situation, and only the proposed methodology (D) is able to accurately model it. Note that, in general, when the branch probabilities are unbalanced, the approach based on the worst case (A) behaves better than the probabilistic one (B). *Graphsearch* and *JPEG* are examples for this scenario. In *jacobi1* and *openmpbench*, the number of control constructs and loops is very limited. Again the branch probabilities are very unbalanced and the approach based on the worst case obtains the best results (A). However, the error of the proposed methodology is still acceptable (less than the 13%). In *fft6*, the effect of the control is negligible and the speed up of the parallelized code is correctly predictable by all the models with a very limited error. In *dijkstra*, instead, the branch probabilities of the control constructs are almost equiprobable. Consequently, the approach based on equiprobable branches accurately models such situation. In all the remaining cases, the profiling techniques C and D systematically outperform the probabilistic ones. On these benchmarks, they obtain very similar results.

VII. CONCLUSIONS AND FUTURE WORKS

In this paper we proposed a methodology that effectively combines a new path profiling technique, the Hierarchical Path Profiling (HPP), with the information coming from the control dependence regions to obtain the static speed-up estimation of a parallel code, represented by a Hierarchical Task Graph. We applied our methodology to a set of common benchmarks for embedded and parallel computing, showing that it produces more accurate estimations than other standard approaches. Such a solution may be integrated in future auto-parallelizing compilers or performance analysis tools for MPSoCs to obtain a fast evaluation of the quality of the parallelization.

Future works will focus on the integration of all the optimizations proposed in literature to reduce the instrumentation overhead, on the analysis of the correlations among the operations and on the effects due to the target architecture (e.g. hits and misses on instruction and data caches or communication) to further improve the estimation accuracy.

ACKNOWLEDGMENT

Research partially funded by the European Community's Sixth Framework Programme, hArtes project. The authors wish to thank Satnam Singh, Microsoft Research, and the reviewers for their suggestions in improving this work.

TABLE VI
COMPARISONS AMONG THE DIFFERENT SPEED-UP ESTIMATION MODELS AND THE ONE OBTAINED BY SIMULATION. μ_{SimIt} IS THE SPEED-UP COMPUTED BY THE SIMIT SIMULATOR, μ_i ARE THE SPEED-UPS RELATED TO THE CASES UNDER DISCUSSION.

Benchmark	Opt	μ_{SimIt}	μ_A	diff (%)	μ_B	diff (%)	μ_C	diff (%)	μ_D	diff (%)
basicmath	-O0	1.599	1.987	24.3	1.987	24.3	1.639	**2.5**	1.639	**2.5**
	-O2	1.585	1.605	**1.3**	1.605	**1.3**	1.606	**1.3**	1.606	**1.3**
blowfish	-O0	1.856	1.237	33.4	1.401	24.5	1.799	**3.1**	1.799	**3.1**
	-O2	1.865	1.801	**3.4**	1.799	3.5	1.798	3.6	1.798	3.6
corners detection	-O0	1.781	1.878	5.4	1.680	5.7	1.800	**1.1**	1.800	**1.1**
	-O2	1.750	1.401	19.9	1.698	3.0	1.790	**2.3**	1.790	**2.3**
Delayline	-O0	1.000	1.691	69.1	1.510	51.0	1.291	29.1	1.000	**0.0**
	-O2	1.000	1.583	58.3	1.435	43.5	1.271	27.1	1.000	**0.0**
dijkstra	-O0	1.602	1.977	23.4	1.686	**5.2**	1.716	7.1	1.716	7.1
	-O2	1.594	1.915	20.1	1.662	**4.3**	1.705	7.0	1.705	7.0
fft	-O0	1.410	1.054	25.2	1.072	24.0	1.366	**3.1**	1.366	**3.1**
	-O2	1.450	1.037	28.5	1.072	26.1	1.366	5.8	1.366	**3.1**
fft6	-O0	1.965	1.957	0.4	1.965	**0.0**	1.966	0.1	1.966	0.1
	-O2	1.952	1.943	0.5	1.956	**0.2**	1.956	**0.2**	1.956	**0.2**
fmm	-O0	1.000	1.305	30.5	1.124	12.4	1.452	45.2	1.000	**0.0**
	-O2	1.000	1.220	22.0	1.104	10.4	1.396	39.6	1.000	**0.0**
graphsearch	-O0	1.802	1.373	23.8	1.164	35.4	1.689	**6.3**	1.689	**6.3**
	-O2	1.541	1.335	13.4	1.156	25.0	1.421	**7.8**	1.421	**7.8**
jacobi1	-O0	1.560	1.461	**6.3**	1.237	20.7	1.364	12.6	1.364	12.6
	-O2	1.494	1.415	**5.3**	1.208	19.1	1.341	10.2	1.341	10.2
JPEG encoder	-O0	2.912	2.094	28.1	1.723	40.8	2.650	**9.0**	2.650	**9.0**
	-O2	3.021	2.084	31.0	1.549	48.7	2.673	**11.5**	2.673	**11.5**
openmpbench	-O0	2.022	2.017	**0.2**	2.054	1.6	1.941	4.0	1.941	4.0
	-O2	2.021	2.013	**0.4**	2.047	1.3	1.932	4.4	1.932	4.4
smoothing	-O0	1.778	1.660	**6.6**	1.273	28.4	3.887	118.6	1.897	6.7
	-O2	1.754	1.291	26.4	1.012	42.3	3.887	121.6	1.897	**8.2**
stringsearch	-O0	1.074	1.991	85.4	1.991	85.4	1.081	**0.7**	1.081	**0.7**
	-O2	1.072	1.991	85.7	1.991	85.7	1.080	**0.7**	1.080	**0.7**
water-nsquared	-O0	1.297	1.242	4.2	1.227	5.4	1.276	**1.6**	1.276	**1.6**
	-O2	1.401	1.259	10.1	1.283	8.4	1.369	**2.3**	1.369	**2.3**
Maximum				85.7		85.7		121.6		12.6
Mean				23.1		22.9		16.3		4.1
Standard deviation				23.6		23.2		30.4		3.7

REFERENCES

[1] W. Wolf, "The future of multiprocessor systems-on-chips," in *DAC*, 2004, pp. 681–685.

[2] L. Thiele, "Performance analysis of distributed embedded systems - tutorial session," in *EMSOFT*, 2007, p. 10.

[3] T. Ball and J. R. Larus, "Efficient path profiling," in *MICRO-29*, 1996, pp. 46–57.

[4] M. Sato, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in *ISSS*, 2002, pp. 109–111.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[6] M. Girkar and C. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 166–178, Mar. 1992.

[7] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 4, pp. 1319–1360, 1994.

[8] J. R. Larus, "Whole program paths," *SIGPLAN Notice*, vol. 34, no. 5, pp. 259–269, 1999.

[9] D. Melski and T. W. Reps, "Interprocedural path profiling," in *CC*, 1999, pp. 47–62.

[10] S. Tallam, X. Zhang, and R. Gupta, "Extending path profiling across loop backedges and procedure boundaries," in *CGO*, 2004, pp. 251–262.

[11] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," in *ICCAD*, 1997, pp. 598–604.

[12] S. Malik, M. Martonosi, and Y. S. Li, "Static timing analysis of embedded software," in *DAC*, 1997, pp. 147–152.

[13] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu, "Software performance estimation strategies in a system-level design tool," in *CODES*, 2000, pp. 82–86.

[14] W. Qin and S. Malik, "Flexible and formal modeling of microprocessors with application to retargetable simulation," in *DATE*, 2003, pp. 556–561.

[15] X. Zhu and S. Malik, "Using a communication architecture specification in an application-driven retargetable prototyping platform for multiprocessing," in *DATE*, 2004, pp. 1244–1249.

[16] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *Journal of VLSI Sig. Proc. Syst.*, vol. 41, no. 2, pp. 169–182, 2005.

[17] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan, "Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations," in *5th Int'l Workshop on Languages and Compilers for Parallel Computing*, 1993, pp. 406–420.

[18] GNU Compiler Collection, "GCC, version 4.3, http://gcc.gnu.org/."

[19] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr, "A sw performance estimation framework for early system-level-design using fine-grained instrumentation," in *DATE*, 2006, pp. 468–473.

[20] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of scalar value communication between speculative threads," in *ASPLOS-X*, 2002, pp. 171–183.

[21] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[22] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. V. Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Systems*, vol. 18, no. 2-3, pp. 129–156, 2000.

[23] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[24] V. C. Sreedhar, G. R. Gao, and Y. Lee, "Identifying loops using DJ graphs," *ACM Trans. Prog. Lang. Syst.*, vol. 18, no. 6, pp. 649–658, 1996.

[25] M. Oyamada, F. Wagner, M. Bonaciu, W. Cesario, and A. Jerraya, "Software Performance Estimation in MPSoC Design," 2007, pp. 38–43.

[26] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani, "An efficient online path profiling framework for java just-in-time compilers," in *PACT*, 2003, pp. 148–158.

[27] "PandA framework, http://trac.ws.dei.polimi.it/panda."

[28] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC*, 2001, pp. 3–14.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995, pp. 24–36.

[30] A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano, "The OpenMP Source Code Repository," in *PDP*, 2005, pp. 244–250.

[31] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurate static estimators for program optimization," in *PLDI*, 1994, pp. 85–96.