International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Performance Estimation of Task Graphs based on Path Profiling

Marco Lattuada · Christian Pilato · Fabrizio Ferrandi

Received: date / Accepted: date

Abstract Correctly estimating the speed-up of a parallel embedded application is crucial to efficiently compare different parallelization techniques, task graph transformations or mapping and scheduling solutions. Unfortunately, especially in case of control-dominated applications, task correlations may heavily affect the execution time of the solutions and usually this is not properly taken into account during performance analysis. We propose a methodology that combines a single profiling of the initial sequential specification with different decisions in terms of partitioning, mapping, and scheduling in order to better estimate the actual speed-up of these solutions. We validated our approach on a multi-processor simulation platform: experimental results show that our methodology, effectively identifying the correlations among tasks, significantly outperforms existing approaches for speed-up estimation. Indeed, we obtained an absolute error less then 5% in average, even when compiling the code with different optimization levels.

M. Lattuada
Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria
Milano, Italy
E-mail: marco.lattuada@polimi.it

C. Pilato
Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria
Milano, Italy
Present address: Columbia University - Department of Computer Science
New York, NY, USA
E-mail: pilato@cs.columbia.edu

F. Ferrandi
Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria
Milano, Italy
E-mail: fabrizio.ferrandi@polimi.it

1 Introduction

Nowadays, creating Multiprocessor System-on-Chip (MPSoC) architectures is a well-established solution for the design of efficient embedded systems [1]. On one hand, these architectures can deliver significant computational power thanks to a variety of processing elements, like general purpose processors, digital signal processors and specialized hardware accelerators. On the other hand, designing the applications for these systems is challenging due to several complex and interdependent steps to be performed [2–5]. First, the application has to be decomposed into multiple tasks that can be potentially executed in parallel or accelerated by dedicated components (*partitioning*). Then, these tasks need to be assigned to the available processing elements (*mapping*) and, finally, it is necessary to determine the execution order of the tasks assigned to the same resources (*scheduling*). When exploring this large design space (either by hand or by automatic methodologies), these combined solutions demand an accurate performance estimation before taking the final decisions [4].

Different approaches have been proposed for estimating the performance of parallel applications running on the top of MPSoCs. Accurate evaluations can be obtained by running the design solutions directly on the target platforms, but in most of the cases these are not available in the early stages of the design. Alternatively, it is possible to use cycle-accurate simulators [6, 7], but they can be too slow to be adopted during the design exploration phase, when multiple solutions have to be evaluated and compared. Fast estimation techniques, based on mathematical models [8, 9], are thus usually preferred in this phase. They are indeed less accurate but much faster, allowing the possibility of exploring more solutions in less time.

Additionally, depending on the nature of the application, different representations can be used to describe the solutions and to estimate the performance. Applications running on embedded systems can be dominated either by data (e.g. audio/video/image processing, digital communications) or by control (e.g. device control, packet processing). Data-oriented applications are often represented through data-flow models and their analysis methods usually focus more on architectural aspects rather than on the application behavior, which is assumed to be highly predictable [10]. However, there is a wide range of applications that cannot be well represented by these models, mainly due to the large presence of coarse-grained parallelism and conditional constructs (e.g. data-dependent loops, branches, function calls), which can significantly vary the execution time of the single parts of the application. In this scenario, task graph models [11] are widely adopted to represent partitioned solutions and derive mapping decisions [2, 5].

Multiple techniques have been proposed for estimating the performance of a task graph and most of them model the task execution time as a constant value [12] or as a stochastic variable [9]. These task estimations are then combined to estimate the execution time of the entire application, but

without considering code correlations that may exist [13]. This can easily lead to wrong estimations that can, in turn, lead to the adoption of sub-optimal solutions. Conversely, the application behavior can be collected dynamically through code profiling [14], but this information is usually exploited only at task level, reducing the accuracy of the task graph estimation.

In this paper we present a methodology to accurately estimate the performance of control-dominated applications for heterogeneous embedded systems. To collect precise information about the control flow of the application (e.g. how many times the different sequences of branch transitions are executed), we extend the well-known *Efficient Path Profiling* [15] with a novel technique, called *Hierarchical Path Profiling*, which allows us to better correlate the profiling information with the structure of the partitioned application. Since the behavior and the correlations of the control constructs depend only on the input data, the profiling is independent from any parallel implementations or target architectures. For this reason, the profiling can be performed only once, on the sequential specification, and on a generic host machine, which is usually much faster than the target architecture. Our approach then combines these profiling data with task graph information to accurately estimate the speed-up of multiple parallel solutions with respect to the sequential version. We also integrate performance models of the different processing elements [16] and predictions of the synchronization costs [17] to have more accurate estimations of the specific target architecture. We applied our methodology to multiple embedded applications in several scenarios, which have been obtained by varying the number of processors in the target architecture and the compiler optimization levels. We then validated our estimations by comparing them with the benchmark execution on an MPSoC simulation platform. This shows that our methodology is effectively able to accurately predict the speed-up with an absolute error that is smaller than 5% in average.

The rest of this paper is organized as follows. Section 2 presents a motivating example, which clearly shows why classical techniques are inadequate for estimating the performance of control-dominated applications running on MPSoCs. Section 3 discusses previous work, while Section 4 provides preliminary definitions and discusses the applicability of the approach to different architectures. Our methodology is then detailed in Section 5 and evaluated in Section 6. Finally, Section 7 concludes the paper.

2 Motivation

Estimating the speed-up introduced by a parallel implementation of a control-dominated application is challenging since the execution times of the tasks can vary significantly and, additionally, control constructs in distinct portions of the code can be correlated. To exemplify this problem, let us consider the function `fun_0` shown in Fig. 1a. One of its parallelization is described through some annotations borrowed from the OpenMP formalism [18] and shown in Fig. 1b, while the corresponding task graph is shown in Fig. 2. Let us also

```
   int fun_0(int c1, int c2, int c3,
   int * array) {
      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 < 10) {             // 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    fun_4(array);                   // BB13
19:   return array[0] + a + b + c;    // BB13
   }
```

(a) Sequential implementation of function `fun_0`

```
   int fun_0(int c1, int c2, int c3,
   int * array) {
   int index, a, b, c;
   //Task 0
 2: index = 0;                                  // BB1
   #pragma omp parallel sections num_threads(3) {
      //Task 1
      #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
      }
      //Task 2
      #pragma omp section {
 8:      while(index < 10) {                   // BB5
 9:         array[index] = index;              // BB6
10:         if (c3)                            // BB6
11:            fun_2(array[index]);            // BB7
            else
12:            array[index]++;                 // BB8
13:         index++;                           // BB9
         }
18:      fun_4(array);                         // BB13
      }
      //Task 3
      #pragma omp section {
 7:      c = 1;                                // BB4
15:      if(c2)                                // BB10
16:         fun_3(&c);                         // BB11
         else
17:         c*=2;                              // BB12
      }
   }
   //Task 4
19: return array[0] + a + b + c;               // BB13
```

(b) Parallel implementation of function `fun_0`

Fig. 1 Implementation of the example function `fun_0`. On each line, the number on the left-hand side is the identifier associated with the statement, while the number on the right-hand side is the identifier of the basic block to which the statement belongs.

assume that the target architecture is composed of two processors (i.e. CPU_α and CPU_β), and the following information is known:

- the estimated execution time of each statement o_i (including the calls to functions `fun_1`, `fun_2` and `fun_3`) is fixed and known, as reported in Table 1 (the identifier i of o_i is reported on the left-hand side of Fig. 1a);
- the probability of condition `c1` being true is 0.5, the probability of condition `c2` being true is 0.5 and the condition `c3` is always true;
- the architecture requires 50 cycles to create the tasks and 10 cycles for either synchronizing or destructing the created tasks.

Finally, let us also assume that there exists a correlation between `c1` and `c2`, which controls the execution of `fun_1` and `fun_3`, respectively. The following situations are considered:

ⓐ `c1` and `c2` always have the same value (either true or false) during an execution of `fun_0`: `fun_1` and `fun_3` are both invoked (true) or none of them is invoked (false).
ⓑ `c1` and `c2` always have opposite values during the same execution of `fun_0`: `fun_1` and `fun_3` are called in mutual exclusion.

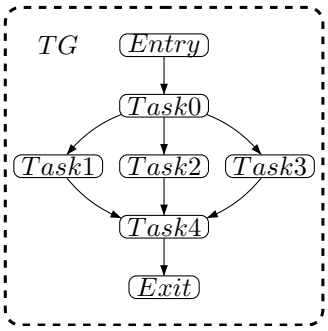


Fig. 2 Task Graph extracted from function `fun_0`.

Table 1 Estimation of clock cycles delay of each statement.

Statement	n. cycles	Statement	n. cycles	Statement	n. cycles
o_1	1	o_8	1	o_{15}	1
o_2	1	o_9	1	o_{16}	2,050
o_3	1	o_{10}	1	o_{17}	1
o_4	2,050	o_{11}	101	o_{18}	10
o_5	1	o_{12}	2	o_{19}	4
o_6	1	o_{13}	1		
o_7	1	o_{14}	1		

Table 2 Task execution times with different conditions.

Task	Conditions	`c1=true`	`c1=false`	Maximum	Average
$Task0$	(a)	1	1	1	1
	(b)	1	1	1	1
$Task1$	(a)	$2,054$	5	$2,054$	$1,029.5$
	(b)	$2,054$	5	$2,054$	$1,029.5$
$Task2$	(a)	$1,061$	$1,061$	$1,061$	$1,061$
	(b)	$1,061$	$1,061$	$1,061$	$1,061$
$Task3$	(a)	$2,052$	3	$2,052$	$1,027.5$
	(b)	3	$2,052$	$2,052$	$1,027.5$
$Task4$	(a)	4	4	4	4
	(b)	4	4	4	4

(a): the probability of condition `c1` being true is 0.5 and `c1 = c2`. (b): the probability of condition `c1` being true is 0.5 and `c1 = !c2`. In both the cases, `c3` is always true and the loop is executed 10 times.

Table 2 reports the maximum and the average execution time for all the tasks. It is important to note how the execution of `fun_1` and `fun_3` heavily impacts on the execution time of $Task1$ and $Task3$.

Now we consider the two following mapping and scheduling solutions to be evaluated:

- *SolA*: $Task1$ and $Task2$ are mapped onto CPU_α (with $Task1$ scheduled before $Task2$) and $Task3$ is mapped onto CPU_β;
- *SolB*: $Task1$ and $Task3$ are mapped onto CPU_α (with $Task1$ scheduled before $Task3$) and $Task2$ is mapped onto CPU_β.

Table 3 Estimated and real speed-ups obtained with different mapping and scheduling solutions and conditions correlations.

	Speed-up					
	Maximal Time (MT)		Average Time (AT)		Real	
Conditions	SolA	SolB	SolA	SolB	SolA	SolB
ⓐ	1.62	1.23	1.44	1.47	1.45	1.18
ⓑ	1.62	1.23	1.44	1.47	1.18	1.47

Average speed-up of `fun_0` when the probability of condition `c1` being true is 0.5, `c3` is always true and ⓐ `c1 = c2` or ⓑ `c1 = !c2`; MT is the speedup estimated when execution time of the tasks is considered constant and equal to the maximal execution time; AT is the speedup estimated when execution time of the tasks is considered constant and equal to the average execution time; $Real$ is the real speed-up.

Right part of Table 3 (*Real*) reports the different real speed-ups of `fun_0` in each of the possible cases obtained by combining the two conditions situations (ⓐ and ⓑ) with the two mapping solutions (*SolA* and *SolB*). Results show that *SolA* has a larger speed-up in situation ⓐ, while *SolB* is the best solution in situation ⓑ. Table 3 also reports the estimations that can be obtained by using traditional techniques [19] based on average (*AT*) and maximum (*MT*) execution times. The former technique averages the different execution times of each task in all the situations, while the latter adopts the maximum execution time for each of them. These techniques present the same results for the two situations ⓐ and ⓑ and these results may also lead to choose inefficient mapping and scheduling solutions. Specifically, the MT technique always suggests to choose *SolA*, which is not correct in situation ⓑ. On the contrary, the AT technique always leads to slightly prefer *SolB*, which is not the best solution in situation ⓐ.

These results show that, especially in case of control-dominated applications, the best mapping and scheduling solution can depend on the correlations that may exist among control constructs in the source code. For this reason, a methodology for a correct performance estimation of such applications has to necessarily take this aspect into account.

3 Related Work

Performance estimation is a crucial step in the design of efficient MPSoCs, where multiple design solutions have to be properly compared to determine the best decisions. Several methodologies have been proposed for evaluating the performance of parallel applications on MPSoCs. These methods can be roughly divided into three categories: direct measures, estimations by simulations, estimations by use of mathematical models.

Most of methodologies based on direct measures (e.g. [17]) are not affordable since integrating direct measurements in a design exploration framework is a long, difficult and error-prone task. Additionally, it cannot be completely automated and most of the work is thus manually performed by the designers,

limiting the number of solutions that can be effectively evaluated. Techniques based on estimations are thus usually preferred.

In simulation-based methods, the single components or the entire system are estimated with simulations at different levels of accuracy (e.g. with ARMn [20], MPARM [6], ReSP [7], gem5 [21]). For example, in [22], a complete simulation is required to evaluate each design solution. However, accurate simulators are usually quite slow, especially in case of MPSoCs where they need to simulate multiple architectural aspects. For this reason the estimation problem is usually decomposed into sub-problems, where the simulation is performed only at a higher level of abstraction. For example, in [23], the performance of the single tasks is estimated by accurately annotating the source code, while the entire application is estimated through TLM simulations.

Estimations can be also obtained by exploiting mathematical models that correlate some numerical features of a design solution, which are collected through static or dynamic analyses, with its performance. In general, they are less accurate than the ones based on simulators, but they are much faster so they allow the designer to compare much more solutions. Also in this case, these techniques adopt a two-stage approach to perform first the estimation of the single tasks and then of the entire application. For example, [12] exploits the intermediate representation of the SUIF compiler [24] to estimate the execution time of each task and, then, interval analysis to predict the execution time of the whole application. In a similar way, in [25], GCC is modified to automatically generate the workload models of the tasks, while [26] combines performance estimation of single processors to estimate the performance of JPEG encoder and decoder applications on a pipelined MPSoC. [27] considers an ILP formulation for automatically parallelizing a hierarchical task graph representation, but the cost estimation is performed by simply associating a weight with each instruction, without analyzing the correlations between the control constructs.

While all these approaches model the execution time of a task as a constant, there are performance models where the execution times of tasks and task graphs are variable. In [28], the execution time of single tasks is modeled as a function of the variations in memory accesses count and requests rate, but ignoring any other details of its internal behavior, such as conditional constructs correlations. Finally, also stochastic variables have been used in the performance models of both tasks and task graphs. For example, [8] estimates the performance of a task graph as a stochastic variable, which is based on the stochastic variables associated with the execution times of the single tasks. Similarly, in [29], stochastic variables are used to model the access time of different tasks to resources in contention, while in [19] they are used to model the execution time of the single tasks, based on multiple profiling runs executed with different data sets. The authors suggest also two possible deterministic techniques: the *worst-case estimation*, which considers the 99.9'th percentile of the execution time of each task, and the *average-case estimation*, which considers the average execution time. In [30] Distributionally Robust Monte

Carlo Simulation (DRMCS) is combined with a task-accurate performance estimation method to guarantee a robust task graph estimation. It requires to annotate each task with an interval estimation of its execution time. DRMCS is then applied to compute the worst-case execution time of the entire task graph, along with a confidence level for this estimation.

However, all these approaches are based on the assumption that the execution times of the tasks are independent and this can lead to a wrong evaluation of the design solutions, as shown in Section 2. The correlation effects among the workload of parallel tasks have been actually examined in [13], but only to correctly model the energy consumption of the analyzed solution and not for an accurate performance analysis.

To correctly model the tasks correlations induced by control constructs, we rely on *path profiling* [15]. Path profiling is a well-known technique that adopts an instrumentation of the branch constructs, followed by a series of executions of the resulting code with different data sets. This allows the designer to collect information about the dynamic behavior of the application. For this reason, several estimation methods have been based on this technique, but they are usually applied only to sequential applications. [31] discusses the performance estimation of sequential applications for real-time embedded systems. This work exploits the concept of path-based analysis to determine best and worst execution times. Similarly, [32] describes several static timing analysis techniques targeting embedded systems composed of a single processor. However, all the discussed techniques have high computational complexity, since they aim at verifying hard or soft constraints of real time systems. Moreover, these techniques are limited by the number of generated paths, since they do not exploit any techniques for path decomposition as proposed by [15]. For this reason, they cannot be applied to large applications. In [33], the path information is used instead to analyze the synchronizations among the threads: the synchronization operations are speculatively anticipated if they are on the most executed paths. The path profiling information has been thus used to optimize the communication between the threads, rather than to estimate the performances of parallelized specifications.

To the best of our knowledge, none of the existing approaches is able to estimate the performances of entire task graphs by analyzing the correlations among the task executions due to conditional constructs. In [34], we proposed a preliminary approach that is able to consider such correlations by leveraging path profiling information. However, it does not consider heterogeneous architectures nor information about mapping and scheduling. It is thus not possible to take into account the effects of executing the code on different processing elements, as well as the overhead introduced by resource contention. This paper extends this approach with the following main contributions:

- we provide the support to heterogeneous embedded systems by integrating performance models of different processing elements, along with information about mapping and scheduling decisions;

– we present a comprehensive validation of our approach by comparing the estimated speed-up with the one obtained with an open-source simulation platform for MPSoCs, and by considering different architectures and compiler optimization levels for the applications.

4 Preliminaries

This section introduces the concepts we leverage for estimating the performance of partitioned control-oriented applications. Specifically, Section 4.1 presents some basic definitions to better understand our approach, while Section 4.2 discusses its applicability to different architectural templates.

4.1 Definitions

Our methodology works on the top of the following intermediate representations, which are built for each function of the input application:

– *Control Flow Graph* (CFG) [35], a directed graph $G_{CFG} = (V, E_{CFG})$, which is an abstract representation of the paths (i.e. the sequences of branches) that might be traversed during the execution of the function; each vertex $v_i \in V$ represents a basic block BB_i; two additional vertices $Entry$ and $Exit$ are introduced to represent entry and exit points of the function execution, respectively; edges that close a loop of a path starting from the $Entry$ node are named *feedback* edges [36];
– *Control Dependence Graph* (CDG) [37], a directed graph $G_{CDG} = (V, E_{CDG})$, which represents the control dependences of the basic blocks;
– *Control Dependence Region* (CDR) [37], a partitioning of the basic blocks such that two basic blocks are in the same region if and only if they have the same set of control dependences in the CDG; the function $\gamma : C_c = \gamma(BB_i)$ returns the Control Dependence Region C_c to which the basic block BB_i belongs;
– *Loop Forest* [36], a representation of the loop hierarchy inside the CFG;
– *Hierarchical Task Graph* [38], a representation of the application decomposition induced by the partitioning specified by the designer.

Given the example of Fig. 1a, its CFG is represented in Fig. 3; the only feedback edge is the dashed edge $e_{9,5}$. The CDG and the CDRs of the same example are shown in Fig. 4, where, for example, $e_{1,2}$ represents that BB_2 is executed if and only if BB_1 has completed its execution and the value of its final condition is *true*. Conversely, BB_4 has no control dependences with BB_1, BB_2 and BB_3, so they can be executed in parallel provided that data dependences are satisfied. The example contains one loop, which has BB_5 as header and includes basic blocks BB_5, BB_6, BB_7 and BB_8. In the rest of the paper, we identify a loop with the number of its basic block header (e.g. L_5). The entire function `fun_0` is considered as a *main loop*, called L_0.

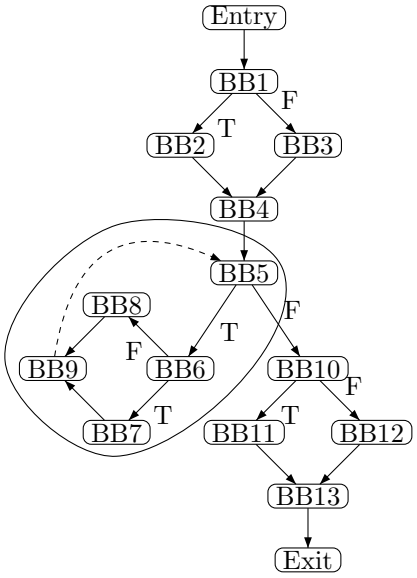


Fig. 3 The Control Flow Graph of `fun_0`.

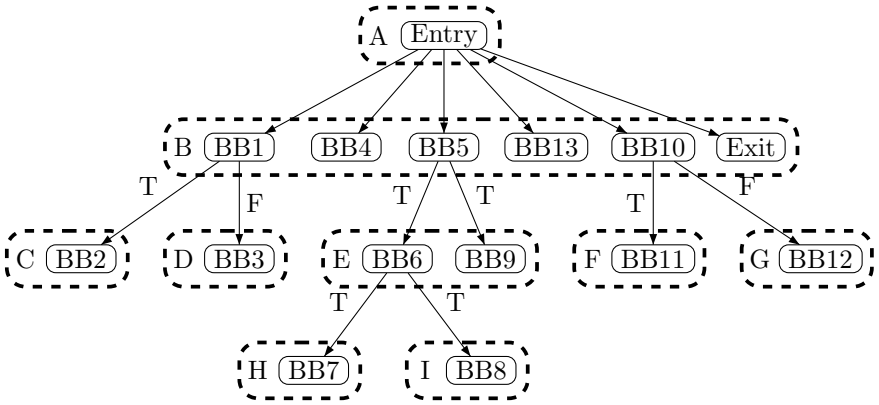
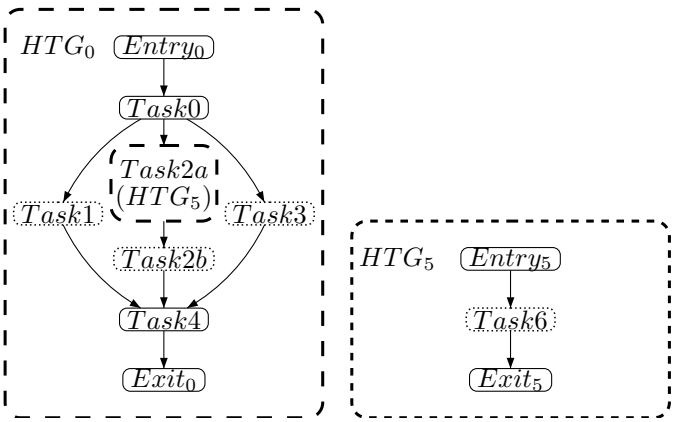


Fig. 4 The CDG of function `fun_0`; dashed boxes identify CDRs named with capital letters.

Partitioned applications are usually represented through a *task graph*, which is a directed graph whose vertices are the tasks induced by the partitioning and the edges represent precedences among them. Similarly to [38] and [39], we adopt the *Hierarchical Task Graph* (HTG) as the intermediate representation of a partitioned application. Specifically, the HTG is an acyclic 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) or *loop* (i.e. a task that represents a loop whose body is a HTG itself). In this work, to describe the parallelism, we adopt a subset of OpenMP formalism [18]. OpenMP is a C/C++/Fortran extension widely



(a) Task Graph of L_0. (b) Task Graph of L_5.

Fig. 5 Hierarchical Task Graph extracted from function `fun_0`.

adopted to describe the application partitioning directly inside the source code by means of pragmas [4]. For this reason, it is possible to activate sequential or parallel execution with simple compiler flags. It is however important to note that a complete support of OpenMP is out of the scope of this work. On the contrary, we only selected few annotations (`parallel sections` and `section`) that allow the designer to statically specify which parts of the code are meant to be executed in parallel, that is the structure of the HTG. Indeed, other OpenMP pragmas (e.g., `task`) prevent the building of the task graphs at design time. We create the HTGs by analyzing the intermediate representation produced by the compiler after the optimization phase. In such a way, we are able to take into account the effects of compiler optimizations on the code associated with each task. For example, given the annotated code shown in Fig. 1b, we create the corresponding HTG, which is shown in Fig. 5, as follows. The HTGs associated with each function are created starting from the innermost ones. For this reason, the HTG associated with `fun_0` is created after HTGs of all called functions. Then, a *simple* task is created for $Task0$ since it contains no function calls or loops. It also represents the fork of the OpenMP `parallel sections`, which is composed of three sections. The first section corresponds to a *compound* task (i.e. $Task1$) since it contains the call to function `fun_1`. The corresponding HTG is associated with the same task. The second section contains a loop, followed by a function call. For this reason, two distinct tasks are generated: $Task2a$, which is associated with HTG_5 (i.e. the HTG associated with the loop), and $Task2b$, which contains the remaining code of the section. Note that both $Task6$ (i.e. the task representing the loop body) and $Task2b$ are *compound* tasks since they contain function calls. Similarly to the first section, the third one corresponds to a *compound* task (i.e. $Task3$) due to the presence of a function call. An additional task is created to represent the join of the OpenMP `parallel sections` (i.e. $Task4$) and it contains the remaining code. Note that, in Fig.5, dotted vertices identify

compound tasks (i.e. $Task1$, $Task2b$, $Task3$, $Task6$), dashed vertices identify *loop* tasks (i.e. $Task2b$) and continue vertices identify *simple* tasks (i.e. $Task0$ and $Task4$).

Finally, mapping decisions are specified through custom code annotations, as in [39], and the information is associated with each task of graph.

4.2 Supported Target Architectures

This work targets embedded systems composed of a set of processors, which feature local memories for instructions and data [5, 39], and no operating system, but with a bare-metal synchronization, as in [39, 40]. We currently support ARM processors (with or without support for out-of-order execution) and DSPs. Supporting additional processors only requires to generate the proper model (see [16]), which can estimate the performance of the assigned tasks based on their source code. Our methodology then leverage any of these models, as described in the following sections, to estimate the performance of task graph solutions.

We support different communication infrastructures. The processing elements can be indeed interconnected through a shared bus, a network-on-chip or point-to-point links [40]. From the point of view of estimations, this simply corresponds to a different communication overhead for each data transfer based on the infrastructure adopted for its implementation.

Moreover, there is no communication between parallel tasks: communication between parallel tasks and other tasks (e.g. fork and join task) can be explicit (performed at the beginning and at the end of their execution through direct data transfers) or implicit (performed during all the execution through exploitation of shared memory). The delay for the first type of communication is well modeled by the proposed methodology since it is incorporated in task overhead cost. On the contrary, the second type of transfers can introduce approximations in the estimations since the proposed methodology does not take explicitly into account cache memories. We also assume that there is no synchronization during the execution of parallel tasks (e.g. shared variables protected by mutexes). These situations are managed only at task boundaries [17], when tasks are created or destroyed. In fact, this is a common practice to effectively allow the parallel execution of the tasks.

With these assumptions, we are able to target both commercial platforms (e.g. Atmel Diopsis 940HF [41], TI OMAP 4 [42]) and prototype architectures obtained with commercial system-level design tools (e.g. Xilinx Vivado IP Integrator [43]). These architectures are also supported by multiple MPSoC simulators, which can be adopted for virtual prototyping (e.g. [6, 7, 21, 44, 45]). These solutions can be thus easily combined to analyze the mutual effects of partitioned applications and architectural decisions (e.g. size of caches, number of processors, communication infrastructures). In this work, we adopt ReSP [7], an in-house simulation platform, to demonstrate this potentiality.

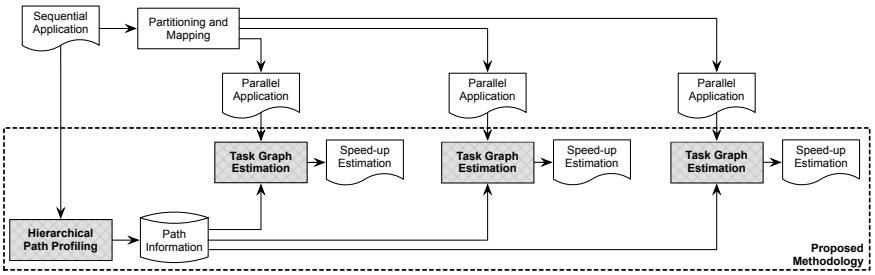


Fig. 6 Overview of the proposed methodology.

5 Proposed Methodology

Our methodology is composed of two consecutive steps, as shown in Fig. 6. First, we profile the sequential version of the application, which is obtained by ignoring any partitioning or mapping pragma annotations, in order to collect information about the behavior of the application, which is then associated with its internal representation. In this step, we adopt the *Hierarchical Path Profiling* (HPP) (i.e. our extension to the Efficient Path Profiling [15]) to collect path information in a way that is suitable to be combined with the HTG representation adopted in the subsequent task graph estimations. This part is detailed in Section 5.1. Then we estimate the speed-up introduced by any of its parallel implementations. Specifically, considering the partitioning solution to be analyzed, the methodology estimates the execution time of each path by computing the contribution of all the tasks and, then, by combining these contributions following the structure of the HTG. The final estimation is obtained by a weighted average of these estimations where the weights are the frequency of the corresponding paths. The process is repeated at each level of the hierarchy, starting from the innermost loops to the outermost ones, as detailed in Section 5.2.

5.1 Hierarchical Path Profiling

Before describing the HPP technique, we need to introduce the definition of *path*. Let $G_{CFG} = (V, E_{CFG})$ be the CFG of a function. Note that our methodology does not have any requirements about the structure of the CFG nor about the structure of its loops. The *path* P_p is defined as the sequence of basic blocks $BB_i \in V$:

$$P_p = BB_1\text{-}BB_2\text{-}\ldots\text{-}BB_n$$

where each pair of basic blocks BB_i-BB_j is connected by an edge $e_{i,j} \in E_{CFG}$. Since the CFG represents all the paths that might be traversed during a program execution, it is possible to count their occurrences and so how many times the corresponding basic blocks are executed. This technique is

usually called *path profiling*. According to this definition, the basic blocks that belong to a path are executed in sequence, without any interleaving.

However, it is worth noting that each cycle inside a *cyclic* CFG (i.e. a CFG with a feedback edge) is still a path. Then, any sequence composed of n repetitions of this path is again a path and so the number of paths may be infinite. For this reason, it is not possible to collect information about any admissible path. We thus need to select a subset of these paths, which we call *valid paths*, and collect information only about them. Our HPP considers as valid only the paths that correspond to an entire loop iteration (or function execution when considering the loop L_0). In particular, given the CFG $G_{CFG} = (V, E_{CFG})$ and the set F of its feedback edges, a path $P_p = \{BB_i\text{-}BB_{i+1}\text{-}\ldots\text{-}BB_j\}$ is considered *valid* when it satisfies one of the following conditions:

- $(BB_j, BB_i) \in F$: i.e. the last basic block BB_j is reconnected to the first basic block BB_i through a feedback edge;
- $BB_i = BB_{Entry} \wedge BB_j = BB_{Exit}$: i.e. the path starts from the initial basic block (BB_{Entry}) and terminates in the final basic block (BB_{Exit}).

Based on these conditions, the paths can be clustered in sets, called *Hierarchical Paths* (HP_i), according to the innermost loop L_i where they are completely contained. Specifically, the path $P_p = BB_i\text{-}BB_{i+1}\text{-}\ldots\text{-}BB_j$ is contained into HP_i since it refers to loop L_i, which has BB_i as header. In our example, the path BB_5-BB_6-BB_7-BB_9 is contained into HP_5 while the path BB_{Entry}-BB_1-BB_3-BB_4-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit} is contained into HP_0 since it refers to the loop L_0 (i.e. the path starts from the function entry).

However, according to this definition of *valid paths*, cyclic paths are still admitted and, for this reason, the number of paths that can be identified in a cyclic CFG is still potentially infinite. To avoid this problem, given a path $P_p \in HP_i$ that contains the execution of a nested loop L_j, we replace the sequence of basic blocks belonging to L_j with the symbol L_j^*. This represents that, during the execution of the path P_p, a certain number of iterations of L_j may be executed. Following this definition, both the paths BB_{Entry}-BB_1-BB_3-BB_4-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit} and BB_{Entry}-BB_1-BB_3-BB_4-BB_5-BB_6-BB_8-BB_9-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit} can be represented with the same path BB_{Entry}-BB_1-BB_3-BB_4-L_5^*-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit}. Indeed, they provide the same information for computing the execution time of HTG_0. Then, details about the basic blocks executed during the nested loop L_5 are used to compute the execution time of HTG_5 (i.e. the one associated with the loop).

It is worth noting that, in the EPP technique proposed in [15], the paths extracted from the execution trace are a complete partition of the trace itself; as a result, each execution of a basic block is counted as part of one and only one path. On the contrary, in the HPP, the execution of a basic block can be considered as part of multiple paths and, thus, overlapping paths are admitted. For example, the execution trace BB_{Entry}-BB_1-BB_3-BB_4-BB_5-BB_6-BB_8-BB_9-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit} contains the execution of two different and valid paths: BB_{Entry}-BB_1-BB_3-BB_4-L_5^*-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit} and BB_5-BB_6-BB_8-BB_9. Then, for example, the execution of BB_6 is

included in both the paths. Indeed, while the latter naturally contains the basic block in the loop iteration, the former implicitly contains the contribution of BB_6 through the contribution of L_5^*. As a result, including the contribution of L_5^* in the outermost path automatically includes the performance estimation of the corresponding loop. We will use this observation to hierarchically build the performance estimation of the entire application.

The HPP keeps track of the current path in the same way of the EPP. Specifically, a variable is used to store the encoded representation of the path, which is updated every time an edge of the CFG is traversed. When a valid path terminates (i.e. the execution reaches its final basic block), the corresponding counter is incremented and a new path starts. However, while in EPP only one path is alive at a time, multiple paths can be simultaneously alive in the HPP, due to the path overlapping that has been described before. In this case, when a new loop starts (i.e. the execution reaches its header), the current path becomes "idle" and a new path starts to keep track of the loop execution. The idle path then returns active only after the termination of the nested loop. More details about this aspect can be found in [34].

Once HPP has been applied and all paths have been hierarchically clustered, they are projected onto the CDRs defined in Section 4: each path can be represented as the set of executed CDRs. We call this projection *Control Region Path* (CRP). In particular, let $P_p \in HP_l$ be a path belonging to loop L_l, the CRP_p associated with path P_p is defined as:

$$CRP_p = \{CDR_i | \exists BB_j \in P_p : CDR_i = \gamma(BB_j)\} \tag{1}$$

where γ is the function that associates a basic block with its CDR. Since the function γ is surjective (i.e. more basic blocks can belong to the same CDR), the size of a control region path CRP_p results equal or smaller than the size of the corresponding path P_p, without loosing any information since the CDR represents all the basic blocks that have to be executed under the same control conditions.

By elaborating path profiling information, it is then possible to derive information about the average number of iterations for each loop. The average number N_l of iterations of a loop L_l, which is nested in L_j, can be computed as:

$$N_l = \frac{\sum_{CRP_p \in HP_l} f_p}{\sum_{CRP_q \in HP_j : \gamma(BB_l) \in CRP_q} f_q} \tag{2}$$

where f_p corresponds to the number of times that path P_p is executed. The numerator is the total number of iterations of L_l, which is computed as the sum of the number of executions of all paths contained in HP_l. The denominator corresponds to how many times the loop L_l is executed, which is computed as the sum of the number of executions of paths of L_j which enter L_l.

Note that, we compute a unique speed-up for each partitioned solution. However, for many control-dominated applications, the behavior of the application and, in turn, the results of the path profiling depend on the input data. So, in case of multiple input data sets, the path profiling information will be

obtained by averaging the results obtained on the single runs. Computing the single speed-up for each input data set is possible, but this approach has some criticalities. In fact, if we obtain that the best solution is different for each data set, multiple solutions have to be implemented at the same time in the final system, which can introduce resource problems (e.g. memory to be reserved for object code). Additionally, it would be necessary to implement a runtime mechanism to automatically determine the solution to be adopted based on the input data set and this is a challenging task.

Appendix A shows the results of applying the HPP to the example shown in Section 2.

5.2 Task Graph Estimation

This section shows how we combine the path profiling information obtained with the HPP with the HTG representation and the mapping and scheduling decisions in order to produce a performance estimation.

For doing this, given a HTG to be estimated, this is transformed into $\overline{HTG}$ to take into account the mapping and scheduling information of each task, extracted from the design solution. Specifically, an edge is added from $Task_i$ to $Task_j$ when: $Task_i$ and $Task_j$ (or the tasks contained in them) share a processing element (*mapping*) and $Task_i$ is scheduled before $Task_j$ (*scheduling*).

Our methodology analyzes all the tasks of the application, starting from the task graphs at the innermost levels of the hierarchy. First, we estimate the execution time of each path by combining the contribution of its statements. Since each path may traverse multiple tasks during its execution and these tasks may be assigned to different processing elements, the contribution of each statement is computed according to the performance model of the processing element where the corresponding task has been mapped. Then, if the path contains one or more loops, their contributions are also taken into account. In this case, the average execution time of a loop iteration is multiplied by its average number of iterations, which is equal to one in case of L_0 (i.e. the HTG associated with the entire function). Since different execution paths can be traversed during a loop iteration, the average execution time of the loop iteration is estimated by considering independently each execution path and then by performing a weighted average of their contributions according to their frequency. Finally, to compute the performance estimation HTC_0 of $\overline{HTG_0}$ (i.e. a function), $\overline{HTG_0}$ is hierarchically analyzed with the procedure described by Algorithm 1. Three main steps can be identified:

1. *Task Analysis* (lines 1-17): we compute the task contributions to the different paths;
2. *Task Graph Analysis* (lines 18-23): we analyze the vertices of $\overline{HTG_l}$ in topological order to compute start and end times of each task;
3. *Task Graph Performance Estimation* (lines 24-27): the end time of task $Exit$ is used to estimate the performance of the entire $\overline{HTG_l}$.

Algorithm 1 Pseudo-code of $HTC_l = estimate(HTG_l(V_l, E_l))$.

1: **for all** task $v_t \in V_l$ **do**
2: $BC_{i,t} = f(o_{s1}, o_{s2}, \ldots, o_{sn})$
3: **if** v_t is a loop task containing HTG_m **then**
4: $\overline{BC}_{i,t} = BC_{i,t} + HTC_m$
5: **else**
6: $\overline{BC}_{i,t} = BC_{i,t}$
7: **end if**
8: **for all** $BB_i : BC_{i,t} > 0$ **do**
9: $CC_{c,t} = CC_{c,t} + BC_{i,t}$ where $c = \gamma(BB_i)$
10: **end for**
11: **for all** $CRP_p \in HP_l$ **do**
12: **for all** $CDR_i \in CRP_p$ **do**
13: $TPC_{p,t} = TPC_{p,t} + CC_{i,t}$
14: **end for**
15: $\overline{TPC}_{p,t} = TPC_{p,t} + OC_t$
16: **end for**
17: **end for**
18: **for all** task $v_t \in V_l$ in topological order **do**
19: **for all** $CRP_p \in HP_l$ **do**
20: $START_{p,t} = max(STOP_{p,u})$
21: $STOP_{p,t} = START_{p,t} + \overline{TPC}_{p,t}$
22: **end for**
23: **end for**
24: **for all** $CRP_p \in HP_l$ **do**
25: $PC_p = STOP_{p,Exit}$
26: **end for**
27: $HTC_l = weightedSum(PC_p)$

Before analyzing a HTG, all nested HTGs have to be already analyzed since contributions of the innermost loops or of the called functions have to be taken into account. For example, the performance estimation of HTG_0 can be computed only after estimating the performance of HTG_5. Similarly, the HTGs associated with `fun_1`, `fun_2`, `fun_3` and `fun_4` must be estimated before estimating the HTG associated with `fun_0`. The contribution of a function call is estimated as fixed and not depending on the particular call site, potentially introducing an approximation in the estimated parallel solution. An alternative solution is to create a clone of the complete function HTG for each call site in order to produce better estimation results. However, this can significantly increase the complexity of the proposed methodology. Note that recursive functions are not supported by the proposed methodology.

Before estimating the performance HTC_l of the HTG_l, several intermediate estimations need to be performed to compute the contribution of each task to each path and then of each path to the entire task graph. These contributions are computed starting from the contributions of the single statements which compose the path, aggregated according to the structure of the HTG and the CFG of the specification. To estimate the contribution of the statements, different methods can be adopted, such as analytical models [16,46] or cycle-accurate simulators [20]. In this work, we adopt estimations based

on analytical models. In particular, given a statement to be characterized, we adopt as features the sequence of low-level instructions (i.e. RTL instructions produced by compiler for the target processing element) that correspond to the specific statement and to the preceding ones in the execution flow. The performance model, which is built by means of linear regression on a set of characteristic applications, takes as input the sequence of low-level instructions associated with the statement and produces as output the estimation of the corresponding execution time. Additional details can be found in [16]. After the estimation of the execution time of the single instructions, we perform the following intermediate estimations:

1. $BC_{i,t}$ (line 2) is the contribution of a basic block to the execution time of a task. It is computed as the estimated execution time of the statements of BB_i which belong to the task v_t:

$$BC_{i,t} = f(o_{s1}, o_{s2}, \ldots, o_{sn}) \tag{3}$$

   where o_{si} is a statement of BB_i which belongs to the task v_t and $f(\ldots)$ is the estimation of the execution time of the statements, which takes into account also the processing element where the task v_t has been mapped as described above.
2. $\overline{BC}_{i,t}$ (line 4 and line 6) is the contribution of a basic block to the execution of a task and includes also the contributions of nested loops. If a task is a *loop* and HTG_i is the nested HTG, the estimated loop performance HTC_i is added to the contribution of the header BB_i:

$$\overline{BC}_{i,t} \begin{cases} BC_{i,t} + HTC_i & \text{if } v_t \text{ is a loop task containing } HTG_i \quad (4a) \\ BC_{i,t} & \text{otherwise} \quad (4b) \end{cases}$$

3. $CC_{c,t}$ (line 9) is the contribution of a CDR to the execution time of a task. It is computed as the sum of the contributions of the basic blocks belonging to the CDR:

$$CC_{c,t} = \sum_{\forall BB_i : c = \gamma(BB_i)} \overline{BC}_{i,t} \tag{5}$$

4. $TPC_{p,t}$ (line 13) is the execution time of the task t when the path P_p is executed. It is computed as the sum of the contributions of all the CDRs belonging to P_p:

$$TPC_{p,t} = \sum_{\forall c : CDR_c \in CRP_p} CC_{c,t} \tag{6}$$

5. $\overline{TPC}_{p,t}$ (line 15) is the overall execution time for the task t (including the task management overhead, if any) when the path P_p is executed. It is computed as the sum of the execution time plus the overhead cost:

$$\overline{TPC}_{p,t} = TPC_{p,t} + OC_t \tag{7}$$

6. $START_{p,t}$ (line 20) is the time from the beginning of the execution of an iteration of L_l in which the task t starts the execution of the path P_p, while $STOP_{p,t}$ (line 21) is the time in which the task t ends the execution of the path P_p. PC_p (line 25), is the contribution of each path P_p to the average performance of the task graph. The start time $START_{p,t}$ is computed as:

$$START_{p,t} = max_{v_u \in pred(v_t)} STOP_{p,u} \tag{8}$$

where $pred(v_t)$ is the set of the predecessors of v_t in $\overline{HTG_l}$. Equation 8 states that the start time of a task is the maximum between end times of the tasks that precede v_t in $\overline{HTG_l}$. The end time $STOP_{p,t}$ is computed as:

$$STOP_{p,t} = START_{p,t} + \overline{TPC}_{p,t} \tag{9}$$

Equation 9 states that the end time of a task v_t during the execution of path P_p is the start time of the task plus the time required for its execution ($\overline{TPC}_{p,t}$). Finally, PC_p (i.e., the contribution of path P_p to HTC_l) is computed as:

$$PC_p = STOP_{p,Exit} \tag{10}$$

Equation 10 states that the contribution of each path is the end time of the task $Exit$.

7. HTC_l (line 27) is the overall task graph execution time. It is computed as a weighted average of the contributions given by all paths:

$$HTC_l = N_l \cdot \frac{\sum_{P_p \in HP_l} (PC_p \cdot f_p)}{\sum_{P_p \in HP_l} f_p} \tag{11}$$

where N_l is the average number of iteration of L_l ($N_0 = 1$) and f_p represents how many times the path P_p is executed.

Let S_0 be the performance of the sequential specification, the estimated speed-up μ introduced with the parallelization is then computed as:

$$\mu = \frac{S_0}{HTC_0} \tag{12}$$

5.3 Analysis of the Proposed Methodology

It is worth noting that the estimation presents some approximations because of the simplifications that have been necessarily introduced. First, the execution time of each called function is estimated to be constant and equal to its average execution time: calling contexts and inter-functions correlations are not analyzed as discussed above. In the same way, the correlations between statements belonging to different loops are not taken into account and the execution time of the nested loops is estimated to be constant (i.e. the average execution time of an iteration multiplied by the average number of iterations). However, applying the proposed methodology to the two cases presented in Section 2, we obtain $2,648.5$ and $2,122$ cycles respectively (details are shown

in Appendix A), and these results have been confirmed by the execution times obtained with simulation. This shows that, by exploiting profiling information, the proposed methodology is able to take into account the contribution of each statement when estimating the overall performance of the application.

The algorithm complexity is $O(|C| \cdot |HP_l| \cdot |V_l|)$ where C is the number of CDRs, HP_l is the set of paths for L_l and V_l is the set of tasks for HTG_l, as it results from line 13 of Algorithm 1. In the Equations 4, 5, 6, a linear additive model is adopted to combine the contributions of the different path components. $TPC_{p,t}$ is the estimation of the execution time for the task statements sequentially executed: it is possible to easily integrate more complex models for estimating the overall execution time of these sequences of statements, but this requires to compute independently all the $TPC_{p,t}$ starting from the single statements, which may increase the complexity of the approach.

6 Experimental Evaluation

We tested our methodology on several C-based benchmarks mapped on different architectures. Section 6.1 describes the experimental setup, while Section 6.2 shows the results that have been obtained.

6.1 Experimental Setup

Our methodology has been integrated in PandA [47], a hardware/software codesign framework based on GCC [48]. We tested this methodology on several benchmarks, which have been extracted from different benchmark suites for embedded systems: *MiBench* [49], *OpenMP Source Code Repository (OmpSCR)* [50] and *Splash 2* [51]. Their characteristics are reported in Table 4 and Table 5. The parallelism has been described with OpenMP: some of these benchmarks already contain such annotations, while the remaining ones have been manually partitioned. We then applied our framework to the resulting code and we exploit the intermediate representation of the GCC to build the corresponding HTG representation as described in Section 4.1.

To implement the HPP, we added the proper instrumentations and we execute the resulting code on the host machine to collect information about the executed paths. Additional details can be found in [34]. Note that this instrumentation usually introduces an execution overhead that ranges from 20% to 200% with respect to the non-instrumented execution on the same machine. However, since the profiling is performed directly on the host system, which is usually much faster than the target architecture or its cycle-accurate simulator, the actual overhead of the instrumented application with respect to the original application executed on the target architecture is significantly less. For some architectures, the instrumented execution on the host system can be even faster than non-instrumented execution on target. Additionally, the path profiling is performed only once on the sequential application and the

Table 4 Characteristics of analyzed benchmarks.

Suite	Benchmark Name	Lines	Fun.	Loops	MD	If	Par	Tasks
OmpSCR	array delay	429	8	15	2	8	2	8
	fft	666	17	36	5	17	2	7
MiBench	basicmath	764	11	39	4	16	4	16
	blowfish	2646	14	54	2	55	3	10
	dijkstra	222	6	7	2	12	1	4
	grad	944	11	39	4	16	2	8
	large corner detection	4823	19	83	4	501	4	14
	large edge detection	4823	19	83	4	501	4	14
	short math	944	11	39	4	16	1	4
	small corner detection	4823	19	83	4	501	4	14
	small edge detection	4823	19	83	4	501	4	14
Splash 2	jpeg	931	7	23	4	5	1	4
	square root	944	11	39	4	16	2	6
	string search	2821	3	8	2	17	1	4

$Lines$ is the number of source code lines; $Fun.$ is the number of functions; $Loops$ is the number of loops; MD is the maximum depth of the loop trees; If is the number of conditional constructs; Par is the number of parallel sections; $Tasks$ is the number of parallel tasks.

Table 5 Execution times of the benchmarks when executed on the uniprocessor architecture.

Suite	Benchmark Name	O0	O2
OmpSCR	array delay	$7.20 \cdot 10^8$	$6.02 \cdot 10^8$
	fft	$2.31 \cdot 10^6$	$1.63 \cdot 10^6$
MiBench	basicmath	$4.29 \cdot 10^8$	$3.79 \cdot 10^8$
	blowfish	$4.73 \cdot 10^6$	$4.62 \cdot 10^6$
	dijkstra	$3.41 \cdot 10^8$	$1.60 \cdot 10^8$
	grad	$3.82 \cdot 10^7$	$1.94 \cdot 10^7$
	large corner detection	$3.94 \cdot 10^7$	$1.71 \cdot 10^7$
	large edge detection	$1.84 \cdot 10^6$	$1.71 \cdot 10^6$
	short math	$2.96 \cdot 10^8$	$3.79 \cdot 10^8$
	small corner detection	$1.84 \cdot 10^6$	$9.12 \cdot 10^5$
	small edge detection	$8.42 \cdot 10^5$	$6.01 \cdot 10^5$
Splash 2	jpeg	$5.30 \cdot 10^6$	$3.11 \cdot 10^6$
	square root	$4.97 \cdot 10^7$	$1.04 \cdot 10^7$
	string search	$3.48 \cdot 10^6$	$2.03 \cdot 10^6$

$O0$ and $O2$ are the cycles execution times of the benchmarks, compiled with -O0 and -O2 respectively, when simulated on a single processor architecture.

evaluation of multiple parallel solutions does not require to perform multiple path profilings. For these reasons, the instrumentation overhead is acceptable. Applying the optimizations proposed in [15] (e.g. the use of registers to store intermediate results) would allow to further decrease this overhead, but this is out of the scope of this paper.

In our experiments, the target architectures are composed of ARM processors (from 1 to 4), with a shared 32 Mbyte memory connected through a shared bus. We adopted the ARM922T processors [52] with 333 Mhz clock frequency, based on ARM9TDMI core (ARMv4T architecture) with a 8KB instruction cache and a 8KB data cache. Different performance models have been created with the methodology proposed in [16] to consider the effects of different compiler optimizations sets on the application performance. In

Table 6 Average absolute estimation error of analyzed techniques.

Error on the 2-processor architecture				
	-O0		-O2	
Technique	Error (%)	Std. Dev. (%)	Error (%)	Std. Dev. (%)
Maximal Time	24.35	23.93	22.42	25.30
Average Time	14.27	23.03	15.04	25.14
Path Based	3.72	2.90	4.60	5.28

Error on the 3-processor architecture				
	-O0		-O2	
Technique	Error (%)	Std. Dev. (%)	Error (%)	Std. Dev. (%)
Maximal Time	24.62	24.49	25.20	26.84
Average Time	14.91	23.30	16.01	25.52
Path Based	3.79	2.94	3.30	3.16

Error on the 4-processor architecture				
	-O0		-O2	
Technique	Error	Std. Dev.	Error	Std. Dev.
Maximal Time	56.54	72.41	60.32	74.78
Average Time	35.33	74.83	40.06	78.03
Path Based	2.76	2.98	4.82	6.63

particular, we built performance models for applications compiled with no optimizations (`-O0`) and with a standard set of active optimizations (`-O2`). The task management costs have been obtained by applying the profiling technique proposed in [17]. Mapping decisions for these architectures have been obtained by applying the methodology proposed in [4] and then specified as source code annotations [39]. This approach automatically produces a partitioning of the resources among parallel tasks at each level of the hierarchy. The scheduling decisions are instead automatically computed by applying a topological sorting on the task graphs. Thanks to these assumptions, given a hierarchical task graph HTG_l, there are no interferences between tasks at different levels of the hierarchy and the dependences added to create $\overline{HTG_l}$ are sufficient to effectively compute the performance estimation.

To validate the speed-up estimations produced by our methodology, we adopt *ReSP* (*Re*flective *S*imulation *P*latform) [7], which is freely downloadable from [53]. ReSP is a highly configurable Virtual Platform targeted to the modeling and analysis of MPSoC systems and built on top of the SystemC and TLM libraries at different levels of abstraction. Note that, in our experiments, the cache coherence is guaranteed by a directory-based mechanism, which overhead is directly managed by the simulation platform itself.

6.2 Experimental Results

We evaluated the benefits of considering profiling information by comparing our methodology with the following traditional techniques [19]:

- *Maximal Time (MT)*: the weight of each task is the estimation of its worst-case execution time and the profiling information is used to compute the maximum number of iterations for unbounded loops;

– *Average Time (AT)*: the weight of each task is the estimation of its average execution time and the profiling information is used to compute the average number of loop iterations, along with the branch probabilities.

Note that, in both the cases, the execution time of the task graph HTG is estimated as the longest path in the transformed task graph $\overline{HTG}$.

These techniques have been applied to the benchmarks listed in Table 4, compiled with different levels of GCC optimizations (`-O0` and `-O2`). The results have been compared with the results obtained with our path-based methodology (called *PB*) under the same conditions. For each application, we created eight situations to be analyzed: the two code optimization levels combined with the four considered architectures, i.e., from 1 to 4 processors. Each of these eight situations is analyzed with the three estimation techniques (i.e. MT, AT and PB) and then simulated with ReSP for validation.

Table 6 shows the average error produced by the three techniques when estimating the speed-up for the multiprocessor architectures with respect to the uniprocessor one. The error is computed as $\frac{SU_{Est}-SU_{Real}}{SU_{Real}}$ where SU_{Est} is the estimated speed-up and SU_{Real} is the measured speed-up.

First, there are no significant differences in the accuracy of the estimations with different optimization levels for all the techniques. Indeed, applying code optimizations increases the error in estimating the performance of the single tasks, but the overall effects on the speed-up estimation are mitigated since the error is introduced in the estimations of both the sequential and the parallel versions of the applications. The results also show that our technique (i.e. PB), by properly adopting the complete path profiling information, is able to achieve better results (3.83%) than state-of-the-art techniques (i.e. MT and AT). Additionally, the AT technique produces better estimations than the MT technique (22.60% vs 35.57%) since it exploits more profiling information (e.g. the branch probabilities). The error introduced when estimating architectures with 4 processors with AT and MT techniques grows significantly, as explained in the following.

The results for each benchmark are reported in Tables 7, 8 and 9: the estimation error is reported for each combination of estimation technique, compiler optimization level and target architecture. Note that the error is positive when the technique overestimates the real speed-up, negative otherwise. For the PB technique, we report also the results obtained without taking into account the mapping information during the estimation: it is worth noting that this is equivalent to consider a target architecture composed of a number of processors equal or larger than the maximum degree of parallelism of the benchmark. In fact, in this case, there is no contention on the computational resources and the estimation computed considering mapping information corresponds to the one obtained by ignoring the mapping decisions. Results show that ignoring mapping and scheduling information introduces a large error in estimating the speed-up on the architectures with fewer processors (i.e. two or three) since, in this cases, the contention on the resources is much more relevant and ignoring this information leads to wrong estimations.

Table 7 Estimated speed-up for the architecture with two processors.

			MT		AT		PB			
							Not Mapped		Mapped	
Benchmark Name	OL	Real	SU	Err.	SU	Err.	SU	Err.	SU	Err.
array delay	0	1.132	1.665	47.1	1.144	**1.1**	1.233	8.9	1.144	**1.1**
	2	1.097	1.465	33.5	1.194	**8.8**	1.247	13.7	1.194	**8.8**
basicmath	0	1.558	1.573	**1.0**	1.605	3.0	2.564	64.6	1.605	3.0
	2	1.573	1.607	2.2	1.607	2.2	2.684	70.6	1.584	**0.7**
blowfish	0	1.339	1.717	28.3	1.564	16.8	1.702	27.1	1.369	**2.3**
	2	1.420	1.707	20.2	1.560	9.9	1.800	26.8	1.366	**-3.8**
dijkstra	0	1.000	1.891	89.2	1.891	89.2	0.993	−0.7	0.997	**-0.3**
	2	1.000	1.994	99.5	1.994	99.5	0.993	−0.7	0.997	**-0.3**
fft	0	1.223	1.591	30.1	1.254	**2.5**	1.602	31.0	1.254	**2.5**
	2	1.236	1.523	23.2	1.201	**-2.9**	1.454	17.6	1.201	**-2.9**
grad	0	1.427	1.484	**4.0**	1.484	**4.0**	1.959	37.3	1.484	**4.0**
	2	1.606	1.580	**-1.6**	1.580	**-1.6**	2.218	38.1	1.580	**-1.6**
jpeg	0	1.150	1.348	17.2	1.229	6.9	1.365	18.7	1.163	**1.1**
	2	1.075	1.251	16.3	1.132	5.3	1.212	12.7	1.064	**-1.1**
large corner detection	0	1.674	1.976	18.0	2.004	19.7	3.010	79.8	1.839	**9.8**
	2	1.509	1.892	25.3	1.904	26.1	2.560	69.6	1.704	**12.9**
large edge detection	0	1.099	1.453	32.2	1.318	19.9	1.180	7.3	1.169	**6.3**
	2	1.092	1.442	32.1	1.294	18.5	1.750	60.3	1.074	**-1.6**
short math	0	1.785	1.796	0.6	1.779	**-0.3**	2.914	63.3	1.779	**-0.3**
	2	1.790	1.795	**0.3**	1.795	**0.3**	2.923	63.3	1.774	−0.9
small corner detection	0	1.594	1.876	17.7	1.873	17.5	2.801	75.7	1.681	**5.5**
	2	1.527	1.673	9.6	1.704	11.6	2.562	67.8	1.632	**6.9**
small edge detection	0	1.030	1.453	41.1	1.218	18.3	1.114	8.2	1.069	**3.8**
	2	1.090	1.452	33.2	1.224	12.3	1.250	14.7	1.075	**-1.4**
square root	0	1.488	1.699	14.2	1.579	**6.1**	2.188	47.0	1.579	**6.1**
	2	1.384	1.604	15.9	1.579	**14.1**	2.224	60.7	1.579	**14.1**
string search	0	1.991	1.997	0.3	1.995	**0.2**	3.943	98.1	1.995	**0.2**
	2	1.979	1.997	**0.9**	1.998	1.0	3.965	100.4	1.998	1.0

OL is the optimization level; $Real$ is the speed-up measured with Resp; SU is the estimated speed-up; $Err.$ is the speed-up estimation error.

Table 8 Estimated speed-up for the architecture with three processors.

			MT		AT		PB			
							Not Mapped		Mapped	
Benchmark Name	OL	Real	SU	Err.	SU	Err.	SU	Err.	SU	Err.
array delay	0	1.130	1.664	47.2	1.143	**1.1**	1.233	9.1	1.143	**1.1**
	2	1.104	1.464	32.6	1.145	**3.7**	1.247	13.0	1.145	**3.7**
basicmath	0	1.817	1.890	4.0	1.852	**1.9**	2.564	41.1	1.910	5.1
	2	1.840	1.907	3.6	1.797	**-2.3**	2.684	45.9	1.784	−3.0
blowfish	0	1.449	2.054	41.8	1.869	29.0	1.702	17.5	1.504	**3.8**
	2	1.511	1.947	28.9	1.798	19.0	1.800	19.1	1.604	**6.2**
dijkstra	0	0.996	1.891	89.9	1.891	89.9	0.993	−0.3	0.995	**-0.1**
	2	1.000	1.994	99.5	1.994	99.5	0.993	−0.7	0.995	**-0.5**
fft	0	1.226	1.591	29.8	1.254	**2.3**	1.602	30.7	1.254	**2.3**
	2	1.241	1.523	22.7	1.202	**-3.1**	1.454	17.2	1.202	**-3.1**
grad	0	1.426	1.482	**3.9**	1.482	**3.9**	1.959	37.3	1.482	**3.9**
	2	1.606	1.577	**-1.8**	1.577	**-1.8**	2.218	38.1	1.577	**-1.8**
jpeg	0	1.226	1.660	35.4	1.339	9.2	1.365	11.3	1.297	**5.8**
	2	1.087	1.671	53.7	1.305	20.0	1.212	11.5	1.138	**4.7**
large corner detection	0	1.674	1.876	12.0	1.953	16.6	3.010	79.8	1.839	**9.8**
	2	1.509	1.889	25.2	1.901	26.0	2.560	69.6	1.701	**12.7**
large edge detection	0	1.097	1.443	31.5	1.218	11.0	1.180	7.6	1.069	**-2.6**
	2	1.089	1.439	32.2	1.288	18.3	1.750	60.7	1.071	**-1.6**
short math	0	2.272	2.242	−1.3	2.267	**-0.2**	2.914	28.2	2.267	**-0.2**
	2	2.265	2.241	**-1.1**	2.240	**-1.1**	2.923	29.0	2.240	**-1.1**
small corner detection	0	1.595	1.876	17.6	1.873	17.4	2.801	75.6	1.681	**5.4**
	2	1.526	1.576	**3.3**	1.737	13.8	2.562	67.9	1.587	4.0
small edge detection	0	1.013	1.443	42.4	1.218	20.2	1.114	9.9	1.069	**5.5**
	2	1.083	1.449	33.8	1.221	12.7	1.250	15.4	1.072	**-1.0**
square root	0	1.467	1.697	15.7	1.577	**7.5**	2.188	49.1	1.577	**7.5**
	2	1.594	1.604	**0.7**	1.579	−0.9	2.224	39.6	1.579	−0.9
string search	0	2.973	2.978	0.2	2.978	0.2	3.943	32.6	2.975	**0.1**
	2	2.921	2.974	**1.8**	2.977	1.9	3.965	35.8	2.977	1.9

OL is the optimization level; $Real$ is the speed-up measured with Resp; SU is the estimated speed-up; $Err.$ is the speed-up estimation error.

Table 9 Estimated speed-up for the architecture with four processors.

			MT		AT		PB Not Mapped		PB Mapped	
Benchmark Name	OL	Real	SU	Err.	SU	Err.	SU	Err.	SU	Err.
array delay	0	1.225	1.751	43.0	1.233	**0.7**	1.233	**0.7**	1.233	**0.7**
	2	1.187	1.659	39.8	1.247	**5.1**	1.247	**5.1**	1.247	**5.1**
basicmath	0	2.226	2.298	3.2	2.302	3.4	2.564	15.2	2.206	**-0.9**
	2	2.199	2.297	4.5	2.301	4.7	2.684	22.1	2.226	**1.2**
blowfish	0	1.658	2.655	60.1	2.154	29.9	1.702	**2.7**	1.702	**2.7**
	2	1.761	2.557	45.2	2.059	16.9	1.800	**2.2**	1.800	**2.2**
dijkstra	0	0.991	3.850	288.7	3.850	288.7	0.993	**0.2**	0.993	**0.2**
	2	0.991	3.949	298.4	3.949	298.4	0.993	**0.2**	0.993	**0.2**
fft	0	1.379	2.259	63.9	1.438	**4.3**	1.602	16.2	1.438	**4.3**
	2	1.403	2.064	47.1	1.337	−4.7	1.454	**3.6**	1.337	−4.7
grad	0	1.855	1.959	**5.6**	1.959	**5.6**	1.959	**5.6**	1.959	**5.6**
	2	2.305	2.218	**-3.8**	2.218	**-3.8**	2.218	**-3.8**	2.218	**-3.8**
jpeg	0	1.302	2.025	55.6	1.475	13.3	1.365	**4.9**	1.365	**4.9**
	2	1.120	2.040	82.1	1.475	31.7	1.212	**8.2**	1.212	**8.2**
large corner detection	0	2.482	3.340	34.6	3.733	50.4	3.010	21.3	2.750	**10.8**
	2	2.091	3.502	67.5	3.479	66.4	2.560	22.4	2.350	**12.4**
large edge detection	0	1.118	1.973	76.5	1.404	25.6	1.180	5.6	1.120	**0.2**
	2	1.138	1.937	70.3	1.440	26.6	1.750	53.8	1.126	**-1.0**
short math	0	2.882	2.983	3.5	2.914	**1.1**	2.914	**1.1**	2.914	**1.1**
	2	2.899	2.982	2.9	2.923	**0.8**	2.923	**0.8**	2.923	**0.8**
small corner detection	0	2.440	3.338	36.8	3.451	41.5	2.801	14.8	2.450	**0.4**
	2	1.868	3.259	74.5	3.245	73.7	2.562	37.2	2.325	**24.5**
small edge detection	0	1.062	1.973	85.7	1.403	32.1	1.114	4.9	1.092	**2.8**
	2	1.124	1.993	77.4	1.404	25.0	1.250	11.3	1.120	**-0.3**
square root	0	2.105	2.823	34.1	2.188	**4.0**	2.188	**4.0**	2.188	**4.0**
	2	2.287	2.982	30.4	2.224	**-2.8**	2.224	**-2.8**	2.224	**-2.8**
string search	0	3.936	3.948	0.3	3.946	0.3	3.943	**0.2**	3.943	**0.2**
	2	3.952	3.982	0.8	3.965	**0.3**	3.965	**0.3**	3.965	**0.3**

OL is the optimization level; $Real$ is the speed-up measured with Resp; SU is the estimated speed-up; $Err.$ is the speed-up estimation error.

Analyzing the results, we can identify different classes of benchmarks. In particular, benchmarks like *basicmath*, *grad* and *string search* are characterized by a substantial data parallelism (e.g. parallel execution of different iterations of the same loop), which covers most of the application execution. These applications contain few conditional constructs, without any specific correlation among the execution times of their tasks. All techniques are thus able to estimate their speed-up with a good accuracy. Profiling information can be useful to obtain good speed-up estimations also in case of data parallelism and tasks with similar execution times that are executed in parallel. In fact, for example, benchmarks like *array delay* and *blowfish* are characterized by the presence of parallel sections consisting of parallelized loop iterations. In these benchmarks, the speed-up obtained in the single parallel sections can be easily estimated as their tasks have the same execution time. However, profiling information has to be necessarily considered also in this situation due to the proportion of the tasks composing sequential and parallel parts of the application, as stated by the well-known Amdahl's Law. Since the MT technique is not able to correctly estimate this proportion, its speed-up estimation can lead to a significant error also in this case. In particular, for its intrinsic characteristics of adopting the maximum time, the MT technique systematically overestimates the execution time of the single tasks. Then, if the tasks composing the same parallel section are quite similar, as in the case of data parallel applications, all the tasks are overestimated in the same way. The MT technique thus overestimates the weight of the parallel part much more than the sequen-

tial one, overestimating the speed-up introduced by the parallelization. On the contrary, simple profiling information, such as the branch probabilities and the loop average iterations adopted by the AT technique, provides sufficient information to correctly estimate this proportion and, in turn, the overall speed-up. In these cases, the PB technique obtains almost the same results since the profiling of executed paths does not introduce any additional information to improve the estimation since no correlations are contained into the code. Conversely, when different tasks are correlated, adopting the path profiling information becomes critical. For example, in the susan benchmarks (*corner detection* and *edge detection*), there are parts of the code executed in parallel that are actually in mutual exclusion. Thus, the profiling information adopted by the AT technique is not sufficient and leads to optimistic estimations, as shown also in Section 2. Finally, consider the results about the *dijkstra* benchmark: in this case we introduced a *false parallelism* in the application since the code contained in parallel tasks is always in mutual exclusion. This situation has been artificially created to show how the proposed methodology is able to properly analyze also these situations. Indeed, our methodology correctly predicts a slow-down in the application due to the synchronization overhead of the tasks. The other techniques, instead, are not able to detect the mutual exclusion and, thus, they predict an incorrect positive speed-up.

Finally, Table 10 highlights how the estimation error changes when increaing the number of processors. In the benchmarks with substantial data parallelism (e.g. *grad*), there is no significant difference in the estimation error for all the techniques when considering more processors. Moreover, if the benchmark is characterized by parallel sections with four tasks that are equivalent from the performance point of view, there is not any benefit in increasing the number of processors from two to three. In fact, on the architecture with two processors, each processor has to execute two of the parallel tasks in sequence, while on the architecture with three processors, one of them has still to execute two tasks. For this reason, there is no difference in the speed-up. However, the additional cost required for creating more tasks induces a slow-down in the application execution, as correctly modeled by all techniques. On the contrary, if there is a correlation between the execution times of the parallel tasks, the errors in estimating the parallel version of the application and, in turn, of the speed-up increase when increasing the number of processors, as shown, for example, in the *jpeg* benchmark. When the tasks are completely correlated (e.g. they are in mutual exclusion as in *dijkstra*), these effects become very significant and can lead to large errors. On the contrary, the PB technique is able to take into account all these task correlations and, thus, the error is not significantly affected when increasing the number of processors.

7 Conclusions

In this paper, we proposed a methodology to better estimate the speed-up of a parallel code that takes into account the assignments of the tasks to the pro-

Table 10 Relationship between the number of processors and the estimation error.

Benchmark	OL	Technique	2 processors		3 processors		4 processors	
			SU	Err.	SU	Err.	SU	Err.
grad	0	Real	1.427	-	1.426	-	1.855	-
		Maximal Time	1.484	4.0	1.482	3.9	1.959	5.6
		Average Time	1.484	4.0	1.482	3.9	1.959	5.6
		Path Based	1.484	4.0	1.482	3.9	1.959	5.6
	2	Real	1.606	-	1.606	-	2.305	-
		Maximal Time	1.580	-1.6	1.577	-1.8	2.218	-3.8
		Average Time	1.580	-1.6	1.577	-1.8	2.218	-3.8
		Path Based	1.580	-1.6	1.577	-1.8	2.218	-3.8
jpeg	0	Real	1.150	-	1.226	-	1.302	-
		Maximal Time	1.348	17.2	1.660	35.4	2.025	55.6
		Average Time	1.229	6.9	1.339	9.2	1.475	13.3
		Path Based	1.163	1.1	1.297	5.8	1.365	4.9
	2	Real	1.075	-	1.087	-	1.120	-
		Maximal Time	1.251	16.3	1.674	53.7	2.040	82.1
		Average Time	1.132	5.3	1.305	20.0	1.475	31.7
		Path Based	1.064	-1.1	1.138	4.7	2.212	8.2
dijkstra	0	Real	1.000	-	0.996	-	0.991	-
		Maximal Time	1.891	89.2	1.891	89.9	3.850	288.7
		Average Time	1.891	89.2	1.891	89.9	3.850	288.7
		Path Based	0.997	-0.3	0.995	-0.1	0.993	0.2
	2	Real	1.000	-	1.000	-	0.991	-
		Maximal Time	1.991	99.5	1.994	99.5	3.949	298.4
		Average Time	1.994	99.5	1.994	99.5	3.949	298.4
		Path Based	0.997	-0.3	0.995	-0.5	0.993	0.2

OL is the optimization level; $Technique$ is the technique adopted for the estimation; SU is the estimated speed-up; $Err.$ is the speed-up estimation error.

cessing elements of the architecture and the correlation that may exist among their execution times. In particular, such estimation is computed by combining the HTG representation with a single profiling of the sequential version of the application, which is collected on a generic host machine. We applied our methodology to estimate the speed-up of a set of parallel benchmarks on different MPSoC architectures, which have been obtained by varying the number of processors, and we validated the results on a simulation platform. The results show that the proposed methodology is effectively able to produce much more accurate estimations with respect to classical approaches based on constant execution time for the tasks.

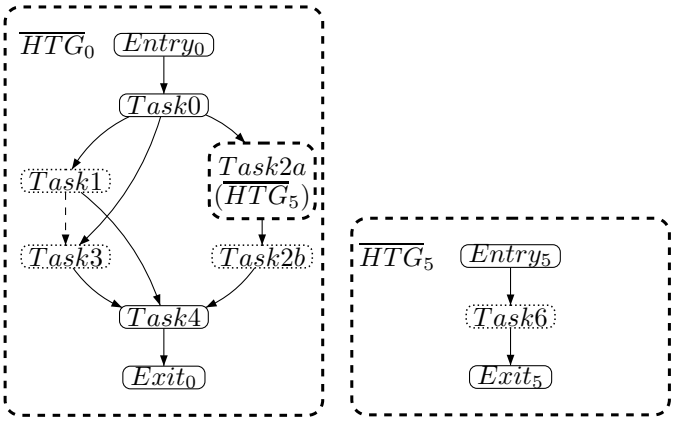
References

1. W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual Design Automation Conference, DAC '04*, pages 681–685, 2004.
2. R. Niemann and P. Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.
3. P. Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems.* Springer, 2 edition, 2010.

4. F. Ferrandi, C. Pilato, A. Tumeo, and D. Sciuto. Mapping and Scheduling of Parallel C Applications with Ant Colony Optimization onto Heterogeneous Reconfigurable MPSoCs. In *Proceedings of the 15th IEEE Asia and South Pacific Design Automation Conference, ASP-DAC '10*, pages 799–804, January 2010 2010.
5. F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. Ant colony heuristic for mapping and scheduling task and communications on heterogeneous embedded systems. *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems*, 29(6):911–924, June 2010.
6. L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *The Journal of VLSI Signal Processing*, 41(2):169–182, 2005.
7. G. Beltrame, L. Fossati, and D. Sciuto. ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1857–1869, December 2009.
8. Y. A. Li and J. K. Antonio. Estimating the execution time distribution for a task graph in a heterogeneous computing system. In *Proceedings of the 6th Heterogeneous Computing Workshop, HCW '97*, pages 172–184, 1997.
9. S. Manolache. Analysis and optimisation of real-time systems with stochastic behaviour. Technical report, Linkoping University, 2005.
10. P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, CASES '03*, pages 63–72, 2003.
11. E. G. Coffman. *Computer and Job Shop Scheduling Theory*. Wiley, New York, 1976.
12. A. Sahu, M. Balakrishnan, and P. R. Panda. A generic platform for estimation of multithreaded program performance on heterogeneous multiprocessors. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09*, pages 1018–1023, 2009.
13. S. Yaldiz, A. Demir, S. Tasiran, P. Ienne, and Y. Leblebici. Characterizing and exploiting task-load variability and correlation for energy management in multi-core systems. In *ESTImedia*, pages 135–140, 2005.
14. H. Hubert, B. Stabernack, and K.-I. Wels. Performance and memory profiling for embedded system design. In *in Proceedings of the International Symposium on Industrial Embedded Systems, SIES '07*, pages 94–101, july 2007.
15. T. Ball and J. R. Larus. Efficient path profiling. In *MICRO-29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, 1996.
16. M. Lattuada and F. Ferrandi. Performance Modeling of Embedded Applications with Zero Architectural Knowledge. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES/ISSS '10*, pages 277–286, 2010.
17. F. Ferrandi, M. Lattuada, C. Pilato, and A. Tumeo. Performance Modeling of Parallel Applications on MPSoCs. In *IEEE International Symposium on System-on-Chip, SOC '09*, pages 64–67, 2009.
18. OpenMP. Application Program Interface, version 2.5, May 2005.
19. N. R. Satish, K. Ravindran, and K. Keutzer. Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 149–158, New York, NY, USA, 2008. ACM.
20. X. Zhu and S. Malik. Using a communication architecture specification in an application-driven retargetable prototyping platform for multiprocessing. In *Proceedings of the conference on Design, automation and test in Europe, DATE '04*, pages 1244–1249, 2004.
21. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

22. A. Miele, C. Pilato, and D. Sciuto. A Simulation-Based Framework for the Exploration of Mapping Solutions on Heterogeneous MPSoCs. *International Journal of Embedded and Real-Time Communication Systems*, 4(1):22–41, 2013.
23. K.-L. Lin, C.-K. Lo, and R.-S. Tsay. Source-level timing annotation for fast and accurate tlm computation model generation. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 235–240, 2010.
24. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. T., S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. The SUIF Compiler System: a Parallelizing and Optimizing Research Compiler. Technical report, Stanford, CA, USA, 1994.
25. J. Kreku, K. Tiensyrjä, and G. Vanmeerbeeck. Automatic workload generation for system-level exploration based on modified GCC compiler. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10*, pages 369–374, 2010.
26. H. Javaid, A. Janapsatya, M. S. Haque, and S. Parameswaran. Rapid runtime estimation methods for pipelined MPSoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10*, pages 363–368, 2010.
27. D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES/ISSS '10*, pages 267–276, 2010.
28. S. Kim and S. Ha. System-level performance analysis of multiprocessor system-on-chips by combining analytical model and execution time variation. *Microprocessors and Microsystems*, 2014.
29. A. Kumar, B. Mesman, H. Corporaal, and Y. Ha. Iterative probabilistic performance prediction for multi-application multiprocessor systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(4):538–551, 2010.
30. Y. Xu, B. Wang, R. Hasholzner, R. Rosales, and J. Teich. On robust task-accurate performance estimation. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 171:1–171:6, New York, NY, USA, 2013. ACM.
31. R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design, ICCAD '97*, pages 598–604, 1997.
32. S. Malik, M. Martonosi, and Y. S. Li. Static timing analysis of embedded software. In *Proceedings of the 34th annual Design Automation Conference, DAC '97*, pages 147–152, 1997.
33. A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, ASPLOS-X*, pages 171–183, 2002.
34. F. Ferrandi, M. Lattuada, C. Pilato, and A. Tumeo. Performance estimation for task graphs combining sequential path profiling and control dependence regions. In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign, MEMOCODE '09*, pages 131–140, 2009.
35. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
36. V. C. Sreedhar, G. R. Gao, and Y. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.
37. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
38. M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.
39. K. Bertels, V. Sima, Y. Yankova, G. Kuzmanov, W. Luk, G. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti. Hartes: Hardware-software codesign for heterogeneous multicore platforms. *IEEE Micro*, 30:88–97, 2010.
40. M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming

of multimedia MP-SoCs. In *Proceedings of the IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '07*, pages 9–14, 2007.
41. Atmel Corporation. DIOPSIS 940HF. http://www.atmel.com, 2009.
42. Texas Instruments. TI OMAP 4. http://www.ti.com, 2011.
43. Xilinx. Vivado Design Suite. http://www.xilinx.com, 2013.
44. A. Gerstlauer. Host-compiled simulation of multi-core platforms. In *Proc. of the IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 1–6, June 2010.
45. Synopsys Inc. Platform Architect. http://www.synopsys.com/Systems/ArchitectureDesign, 2012.
46. M. S. Oyamada, F. Zschornack, and F. R. Wagner. Applying neural networks to performance estimation of embedded software. *Journal of Systems Architecture*, 54(1-2):224–240, 2008.
47. PandA. PandA framework, http://trac.ws.dei.polimi.it/panda.
48. GNU Compiler Collection. GCC, version 4.3, http://gcc.gnu.org/.
49. 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 *Proceedings of the IEEE International Workshop on Workload Characterization, WWC '01*, pages 3–14, 2001.
50. A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano. The OpenMP Source Code Repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP '05*, pages 244–250, 2005.
51. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture, ISCA '95*, pages 24–36, 1995.
52. ARM922T. Technical Reference Manual, http://infocenter.arm.com.
53. Politecnico di Milano. ReSP web-site, http://code.google.com/p/resp-sim/, 2010.



(a) Task Graph of L_0. (b) Task Graph of L_5.

Fig. 7 $\overline{HTG}$ created from HTG considering SolB.

Table 11 Results of applying the Hierarchical Path Profiling Technique to the example of Fig. 1.

ID	Basic blocks	CRP	Results (a)	Results (b)
(1)	BB_{Entry}-BB_1-BB_2-BB_4-L_5^*-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit}	A,B,C,F	5	0
(2)	BB_{Entry}-BB_1-BB_2-BB_4-L_5^*-BB_5-BB_{10}-BB_{12}-BB_{13}-BB_{Exit}	A,B,C,G	0	5
(3)	BB_{Entry}-BB_1-BB_3-BB_4-L_5^*-BB_5-BB_{10}-BB_{11}-BB_{13}-BB_{Exit}	A,B,D,F	0	5
(4)	BB_{Entry}-BB_1-BB_3-BB_4-L_5^*-BB_5-BB_{10}-BB_{12}-BB_{13}-BB_{Exit}	A,B,D,G	5	0
(5)	BB_5-BB_6-BB_7-BB_9	B,E,H	100	100
(6)	BB_5-BB_6-BB_8-BB_9	B,E,I	0	0

ID is the identifier of the path, *Basic blocks* is the sequence of the basic blocks composing the path, *CRP* are the corresponding Control Dependence Regions, *Results* shows how many times the sequence of basic block is executed as counted by the HPP on 10 execution of function `fun_0` when the probability of condition `c1` being true is 0.5, condition `c3` is always true and (a) `c1 = c2` or (b) `c1 = !c2`.

Appendix A. Example of Application of Task Graph Estimation Technique based on Path Profiling

This appendix shows how the proposed methodology is applied to estimate the performance of the example presented in Section 2 when SolB is considered: $Task1$ and $Task3$ are assigned to CPU_α, $Task2a$ and $Task2b$ are assigned to CPU_β. The resulting $\overline{HTG}$ is shown in Fig. 7: the edge $< Task1, Task3 >$ is added to represent the scheduling order, as discussed in Section 5.2. The estimation starts with the application of the Hierarchical Path Profiling on the host machine, which results are reported in Table 11. For the sake of readability, we report also the sequence of basic blocks which compose each path, even if this information is equivalent to the one provided by the corresponding CRP. The order of the Control Dependence Regions in a Control Region Path is not relevant since the basic blocks are interleaved during the execution. The table shows how HPP is able to profile the paths (1), (2), (3) and (4) and to collect correlations about the execution of basic blocks before and after a loop, even if it is executed, with a representation that can be easily mapped onto the HTG.

Before estimating the execution time (HTC_0) of `func_0`, HTC_5 is estimated as follows:

Table 12 Contribution $BC_{i,t}$.

BB	$Task6$
1	0
2	0
3	0
4	0
5	1
6	2
7	101
8	2
9	1
10	0
11	0
12	0
13	0

(a) Contribution $BC_{i,t}$ for HTG_5.

	Task					
BB	$Task0$	$Task1$	$Task2a$	$Task2b$	$Task3$	$Task4$
1	1	2	0	0	0	0
2	0	2,050	0	0	0	0
3	0	1	0	0	0	0
4	0	1	0	0	1	0
5	0	0	1	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	1	0	0	1	0
11	0	0	0	0	2,050	0
12	0	0	0	0	1	0
13	0	0	0	10	0	4

(b) Contribution $BC_{i,t}$ for HTG_0.

Table 13 Contribution $\overline{BC}_{i,t}$.

BB	$Task6$
1	0
2	0
3	0
4	0
5	1
6	2
7	101
8	2
9	1
10	0
11	0
12	0
13	0

(a) Contribution $\overline{BC}_{i,t}$ for HTG_5.

	Task					
BB	$Task0$	$Task1$	$Task2a$	$Task2b$	$Task3$	$Task4$
1	1	2	0	0	0	0
2	0	2,050	0	0	0	0
3	0	1	0	0	0	0
4	0	1	0	0	1	0
5	0	0	1,051	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	1	0	0	1	0
11	0	0	0	0	2,050	0
12	0	0	0	0	1	0
13	0	0	0	10	0	4

(b) Contribution $\overline{BC}_{i,t}$ for HTG_0.

1. the contribution $BC_{i,t}$ of each basic block is computed (line 2 of Algorithm 1): the results are reported in Table 13a (e.g. $BC_{9,6} = f(o_{13}) = 1$ since o_{13} is the only statement of BB_9);
2. the contribution $\overline{BC}_{i,t}$ of each basic block including nested loops is computed (lines 4 and 6 - Table 14a - e.g. $\overline{BC}_{7,6} = BC_{7,6}$ since $Task6$ is *simple*);
3. the contribution $CC_{c,t}$ is computed summing the contribution of the single basic blocks (line 9 - Table 15a - e.g. $CC_{E,6} = \overline{BC}_{6,6} + \overline{BC}_{6,9} = 3$ since CDR_E is composed of BB_6 and BB_9);
4. the contributions of the single Control Dependence Regions are summed to compute the contributions $TPC_{p,t}$ (line 13 - Table 16a - e.g. $TPC_{\textcircled{6},6} = CC_B + CC_E + CC_I = 6$ since path (6) is composed of B, E and I);

Table 14 Contribution $CC_{i,t}$.

CDR	$Task6$
A	0
B	1
C	0
D	0
E	3
F	0
G	0
H	101
I	2

(a) Contribution $CC_{i,t}$ for HTG_5.

	Task					
CDR	$Task0$	$Task1$	$Task2a$	$Task2b$	$Task3$	$Task4$
A	0	0	0	0	0	0
B	1	4	1,051	10	2	4
C	0	2,050	0	0	0	0
D	0	1	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	2,050	0
G	0	0	0	0	1	0
H	0	0	0	0	0	0
I	0	0	0	0	0	0

(b) Contribution $CC_{i,t}$ for HTG_0.

Table 15 Contribution $TPC_{p,t}$.

Id	CDRs	$Task6$
⑤	B,E,H	105
⑥	B,E,I	6

(a) Contribution $TPC_{p,t}$ for HTG_5.

Path		Task					
Id	CDRs	$Task0$	$Task1$	$Task2a$	$Task2b$	$Task3$	$Task4$
①	A,B,C,F	1	2,054	1,051	10	2,052	4
②	A,B,C,G	1	2,054	1,051	10	3	4
③	A,B,D,F	1	5	1,051	10	2,052	4
④	A,B,D,G	1	5	1,051	10	3	4

(b) Contribution $TPC_{p,t}$ for HTG_0.

Table 16 Contribution $\overline{TPC}_{p,t}$.

Id	CDRs	$Task6$
⑤	B,E,H	105
⑥	B,E,I	6

(a) Contribution $\overline{TPC}_{p,t}$ for HTG_5.

Path		Task					
Id	CDRs	$Task0$	$Task1$	$Task2a$	$Task2b$	$Task3$	$Task4$
①	A,B,C,F	1	2,104	1,101	20	2,062	4
②	A,B,C,G	1	2,104	1,101	20	13	4
③	A,B,D,F	1	55	1,101	20	2,062	4
④	A,B,D,G	1	55	1,101	20	13	4

(b) Contribution $\overline{TPC}_{p,t}$ for HTG_0.

Table 17 Starting and ending times of tasks.

Tasks		Paths ⑤	Paths ⑥
$Entry_5$	$START_{p,Entry_5}$	0	0
	$STOP_{p,Entry_5}$	0	0
$Task6$	$START_{p,6}$	0	0
	$STOP_{p,6}$	105	6
$Exit_5$	$START_{p,Exit_5}$	105	6
	$STOP_{p,Exit_5}$	105	6
	PC_p	105	6

(a) Starting and ending times of tasks of HTG_5.

Tasks		Paths ①	Paths ②	Paths ③	Paths ④
$Entry_0$	$START_{p,Entry_0}$	0	0	0	0
	$STOP_{p,Entry_0}$	0	0	0	0
$Task0$	$START_{p,0}$	0	0	0	0
	$STOP_{p,0}$	1	1	1	1
$Task1$	$START_{p,1}$	1	1	1	1
	$STOP_{p,1}$	2,105	2,105	56	56
$Task2a$	$START_{p,2a}$	1	1	1	1
	$STOP_{p,2a}$	1,102	1,102	1,102	1,102
$Task2b$	$START_{p,2b}$	1,102	1,102	1,102	1,102
	$STOP_{p,2b}$	1,122	1,122	1,122	1,122
$Task3$	$START_{p,3}$	2,105	2,105	56	56
	$STOP_{p,3}$	4,167	2,118	2,118	69
$Task4$	$START_{p,4}$	4,167	2,118	2,118	1,122
	$STOP_{p,4}$	4,171	2,122	2,122	1,126
$Exit_0$	$START_{p,Exit_0}$	4,171	2,122	2,122	1,126
	$STOP_{p,Exit_0}$	4,171	2,122	2,122	1,126
	PC_p	4,171	2,122	2,122	1.126

(b) Starting and ending times of tasks of HTG_0.

5. the overhead for the task management is added to $TPC_{p,t}$ to compute $\overline{TPC}_{p,t}$; since there is not any overhead cost in this task graph, $\overline{TPC}_{p,t} = TPC_{p,t}$ (line 15 - Table 17a - $\overline{TPC}_{⑥,6} = TPC_{⑥,6}$ since $Task6$ has not overhead cost);
6. the start and end times of each task are computed (lines 20 and 21 - Table 18a - e.g. $START_{⑥,6} = STOP_{⑥,Entry5}$ since $Entry_5$ is the only predecessor of $Task6$; $STOP_{⑥,6} = START_{⑥,6} + \overline{TPC}_{⑥,6}$); the execution times of the two paths are computed as the end time of task $Exit$ (line 25 - last line of Table 18a - e.g. $PC_{⑥} = STOP_{⑥,Exit_5}$);
7. the estimation of the whole HTG_5 can be computed (line 27):

$$HTC_5 = N_5 \cdot \frac{PC_{⑤} \cdot f_{⑤} + PC_{⑥} \cdot f_{⑥}}{f_{⑤} + f_{⑥}} = 10 \cdot \frac{105 \cdot 100 + 6 \cdot 0}{100 + 0} = 1,050 \qquad (13)$$

After HTC_5 has been estimated, HTC_0 can be estimated in the same way and Fig. 8 shows how the different contributions are combined. These contributions are:

1. the contribution of each basic block $BC_{i,t}$ (lines 2), obtained from the clock cycles of Table 1; the results are reported in Table 13b;

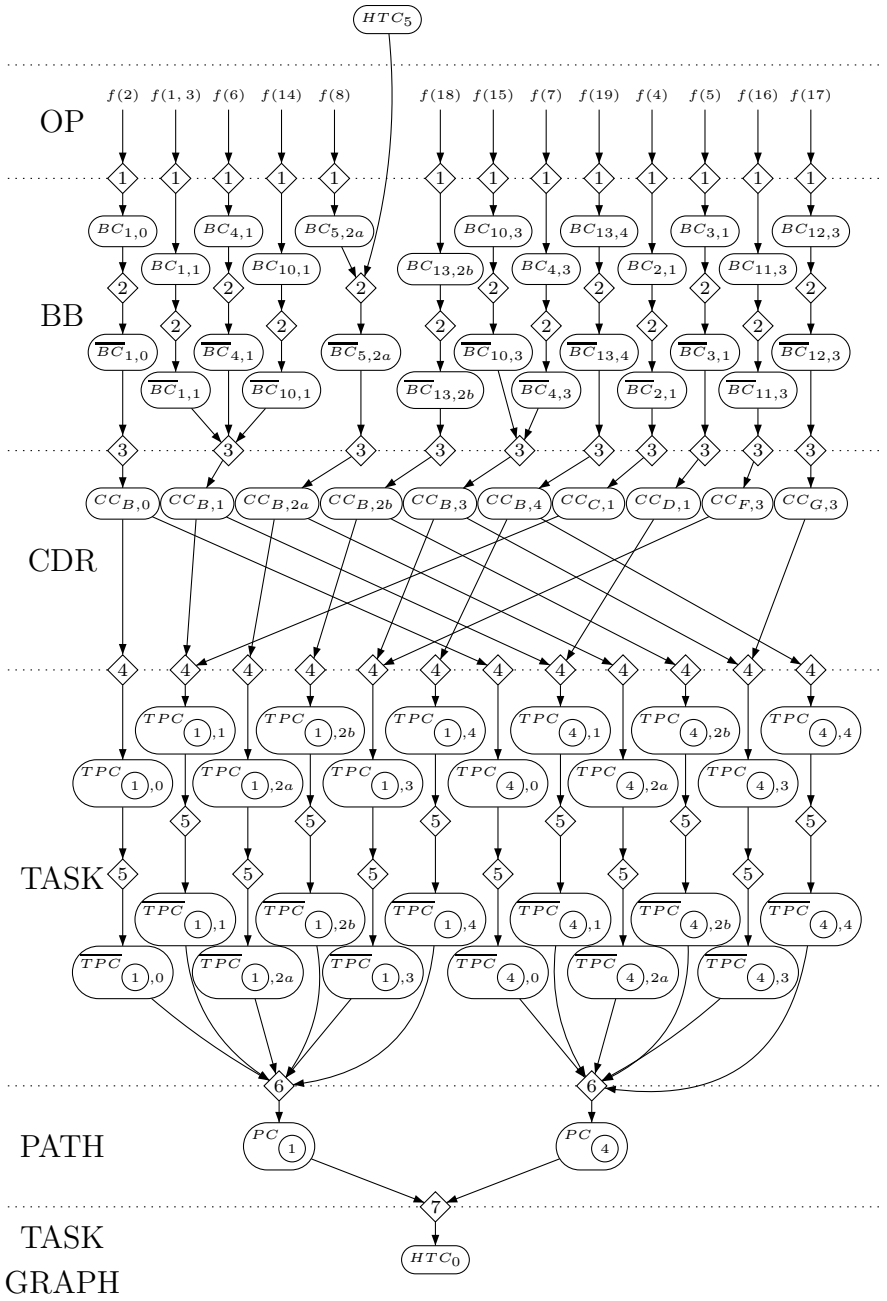


Fig. 8 Composition of contributions to produce HTC_0 when `c1` and `c2` have always the same values; contributions (rounded rectangles) are computed from top to bottom of the graph using operations described in the proposed methodology (rhombuses); the corresponding levels are reported in the left of the figure.

2. the contribution of each basic block including nested loops $\overline{BC}_{i,t}$ (lines 4 and 6); the results are reported in Table 14b; note in particular that $\overline{BC}_{5,2a} = BC_{5,2a} + HTC_5 = 1 + 1,050$;
3. the contribution of each Control Dependence Region $CC_{c,t}$ (line 9); the results are reported in Table 15b;
4. the contribution of each path to each task $TPC_{p,t}$ (line 13); the results are reported in Table 16b;
5. the contribution of each path to each task, along with the overhead cost, $\overline{TPC}_{p,t}$ (line 15); the creation cost (50) is added to $Task1$ and $Task2a$; the synchronization and destruction cost (10) is added to $Task3$ and $Task2b$; the results are reported in Table 17b;
6. $START_{p,t}$ and $STOP_{p,t}$ (lines 20 and 21); the results are reported in Table 18b, where the selected topological order is: $Entry_0$-$Task0$-$Task1$-$Task2a$-$Task2b$-$Task3$-$Task4$-$Exit_0$;
7. the contribution of each path PC_p (line 25): the results are reported in the last line of Table 18b;
8. HPC_0 in the two cases presented in Section 2:
   (a) the CRPs executed are $P_{①}$ and $P_{④}$, so the execution time estimated for the parallel version is:

$$HTC_0 = \frac{PC_{①} \cdot f_{①} + PC_{④} \cdot f_{④}}{f_{①} + f_{④}} = \frac{4,171 \cdot 5 + 1,126 \cdot 5}{5 + 5} = 2,648.5 \qquad (14)$$

   (b) the CRPs executed are $P_{②}$ and $P_{③}$, so the execution time estimated for the parallel version is:

$$HTC_0 = \frac{PC_{②} \cdot f_{②} + PC_{③} \cdot f_{③}}{f_{②} + f_{③}} = \frac{2,122 \cdot 5 + 2,122 \cdot 5}{5 + 5} = 2,122 \qquad (15)$$

Finally, the speed-up for the two situations presented in Section 2 can be computed. The execution time of the sequential specification is $3,123$ cycles in both the cases, so the estimated speed-ups are 1.18 and 1.47, respectively.