Extraction of Recursion Level Parallelism for Embedded Multicore Systems

Miguel Angel Aguilar*, Rainer Leupers*, Gerd Ascheid*, Juan Fernando Eusse†
*Institute for Communication Technologies and Embedded Systems
RWTH Aachen University, Germany
{aguilar, leupers, ascheid}@ice.rwth-aachen.de
†Silexica GmbH, Germany
eusse@silexica.com

***Abstract*—Recursive programs that typically implement divide-and-conquer algorithms are well-suited for multicore systems, as they offer a high degree of parallelization potential. So far, existing parallelizing compilers have mainly focused on extracting other parallel patterns, such as data or pipeline level parallelism. In this paper, we propose a toolflow for the extraction of recursion level parallelism for embedded multicore systems. To achieve this, the toolflow verifies not only the mutual independence of recursive call-sites, but also selects an appropriate task granularity to ensure a good trade-off between load balancing and parallelization overhead. Profitable parallelization opportunities are implemented by using compiler directives from the OpenMP tasking model. Results show the effectiveness of our toolflow, as it is able to speedup sequential recursive programs between 2.5× and 3.8× on a quad-core platform.**

I. Introduction

While Moore's law is gradually coming to an end, the demand for high performance embedded systems is still continuously increasing. In response, both industry and academia have moved towards the use of Multiprocessor System-on-Chips (MPSoCs), as they provide a good trade-off between conflicting constraints, such as performance, energy and cost. However, software has to be carefully parallelized to exploit the full potential of multicore systems. This task is even more challenging considering that the current practice for software development relies on program transformation [1], where developers have to manually parallelize existing sequential software to optimize it for multicore systems. Parallelizing compilers are a promising solution to address this challenge. However, these technologies have mainly focused on loop parallelization [2]–[6], missing the optimization potential of recursive procedures. Therefore, this paper addresses the extraction of *Recursion Level Parallelism* (RLP).

Our work focuses on exploiting parallelism in programs with multiple recursion, where recursive functions contain two or more self-invocations. These programs typically implement divide-and-conquer algorithms, which recursively break problems into smaller sub-problems that are easier to solve. If the sub-problems are independent (i.e., the recursive call-sites are mutually independent), it is possible to exploit a scalable form of nested parallelism. Here for a recursive call a task is created, which then can further spawn parallel work as nested tasks in subsequent recursive calls. OpenMP [7] and Cilk [8] are examples of programming paradigms that provide facilities to parallelize functions containing mutually independent recursive call-sites.

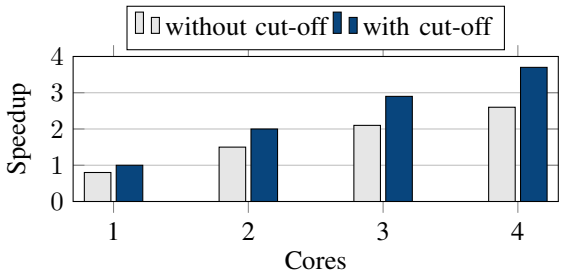


Fig. 1: Impact of the Task Granularity on NQueens

However, detecting the mutual independence of recursive call-sites is not enough to ensure a profitable parallelization. The performance of parallel recursive programs greatly depends on the *task granularity* [9]. The task granularity can be controlled by a mechanism often referred as *cut-off* [10], which defines if a dynamic call of a recursive function is executed as a parallel task or as a sequential function call. The selection of a proper cut-off value is a critical issue. Coarse-grained tasks result in load imbalance, while fine-grained tasks result in high task management overhead. Fig. 1 exemplifies the impact of the task granularity on the `NQueens` benchmark by comparing speedup results with and without cut-off mechanism (Section III describes details of the experimental environment). Clearly using the cut-off mechanism to control the task granularity provides the best results. Therefore, a key challenge here is to select an appropriate task granularity to achieve good trade-off between load balancing and overhead.

In this paper, we propose a toolflow to parallelize programs with multiple recursion for embedded multicore systems. In our approach a model of the program is built, and then analyzed by an algorithm, which verifies mutual independence of recursive call-sites and selects an appropriate cut-off value. Finally, the parallelization opportunities identified by our toolflow are implemented by automatically annotating the input program with properly parameterized OpenMP directives. In summary, this paper makes the following contributions:

- A parallelization approach for embedded multicore systems for sequential applications with multiple recursion.
- Automatic identification of a proper cut-off value to control the task granularity and thus achieve a good trade-off between load balancing and parallelization overhead.
- Evaluation of the effectiveness of the approach by parallelizing a set of multi-recursive programs on a commercial multicore platform [11].

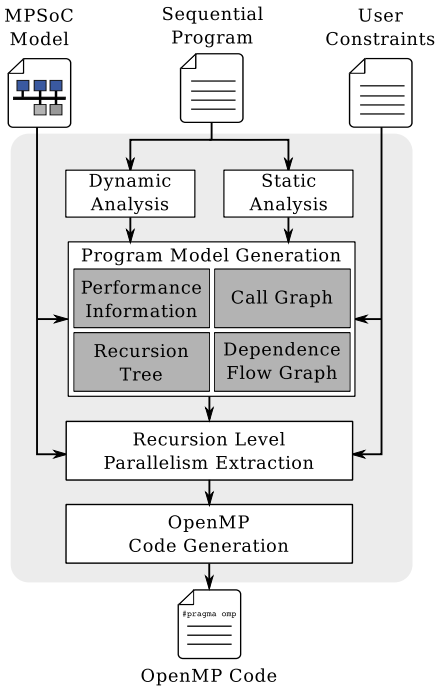


Fig. 2: Parallelization Toolflow

II. Proposed Approach

Fig. 2 shows an overview of our toolflow. It takes as inputs a sequential C/C++ program, a model of the target MPSoC and constraints provided by the developer to guide the analysis. Then it performs both a dynamic analysis based on profiling information and a static analysis based on compile time information. Afterwards, the program model is built and then analyzed by an algorithm that identifies parallelization opportunities in functions with multiple recursion. Finally, a code generator inserts OpenMP pragmas and the needed logic to control the task granularity to efficiently parallelize the program. In the following sections, the proposed toolflow is explained in detail.

A. Toolflow Inputs

1) MPSoC Model: This model enables our toolflow to perform a platform specific analysis. The MPSoC model provides a simplified view of the underlying platform in terms of processing and communication elements. We formally describe the model as follows: let pe be a processor element, which is characterized by its *clock frequency*, a *cost model* for each instruction and *OS features* (e.g., task overhead). The set of all processing elements can be defined as $PE = \{pe_1, ..., pe_{N_{PE}}\}$. Similarly, let ce_{ij} be a communication element, which is a resource (e.g., shared memory) that allows two processing elements to communicate, pe_i and pe_j. The set of all communication elements can be defined as $CE = \{ce_1, ..., ce_{N_{CE}}\}$.

Definition 1: An MPSoC model is a directed multigraph $MPSoC = (PE, CE)$, where PE is the set of processing elements, and CE the set of communication elements, with $CE \subseteq PE \times PE$.

```
1 void f(...) {
2
3   ...
4   f(...);
5
6   f(...);
7   ...
8 }
```

(a) Multiple Static Call-Sites

```
1 void f(...) {
2   ...
3   for (condition && increment)
4   {
5     f(...);
6   }
7   ...
8 }
```

(b) Single Static Call-Site within a Loop

Fig. 3: Simplified Multiple Recursion Code Examples

2) Recursive Sequential Program: Our approach focuses on exploiting parallelism in applications with multiple recursion, where recursive functions contain two or more self-invocations. This is typically the case in divide-and-conquer algorithms. The self-invocations could be due to syntactically independent static call-sites, as shown in Fig. 3a, or due to multiple executions of a single static call-site within the body of a loop, as shown in Fig. 3b.

3) User Constraints: The toolflow takes multiple constraints that allow control over the analysis performed by our approach. The constraints include thresholds to consider a recursive function as a candidate for parallelization, as well as the maximum task management overhead that is tolerated. The constraints are described in detail in Section II-D3.

B. Dynamic and Static Analyses

The first step in our toolflow is to perform dynamic and static analyses of the input program, as Fig. 2 shows. The dynamic analysis relies on a profiling run of the program that collects runtime information, such a list of executed functions, basic block execution count, and memory accesses involving pointers or dynamic allocated memory. In order to obtain the dynamic information, the *Intermediate Representation* (IR) generated by the compiler is instrumented, and a trace is generated by executing the resultant executable. During the instrumentation process function calls to a profiling library are inserted in the IR, in order to track function entry/exits, basic block execution counts, memory allocations, memory accesses and function pointers. Debugging information is kept in the IR to enable correlation with the input source code of the program. On the other hand, the static analysis relies on compile time information collected directly from the IR, such as control flow, variable declarations and memory accesses.

C. Program Model

The program model is composed of architecture specific performance information, a dynamic call graph, a recursion tree for each independent recursion and a set of dependence flow graphs with the control and data dependency information of each function in the program.

1) Performance Information: This is a key piece of information to enable a platform specific profitability analysis of the parallelization opportunities existing in the input program. We use a microarchitecture-aware cost-table model to derive a performance estimate of the program [4]. This estimation is based on the execution count of every statement provided by the dynamic information and the characteristics provided by

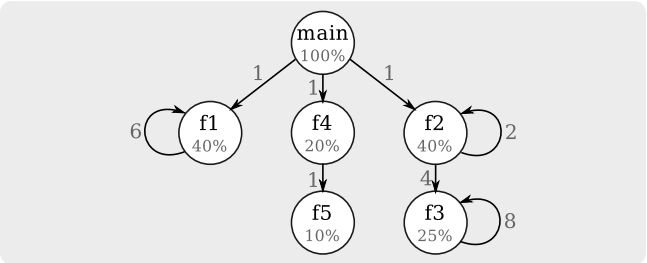


Fig. 4: Dynamic Call Graph (DCG) Example

the MPSoC model (e.g., instruction execution cost). The performance information in this approach is not influenced by the instrumentation overhead, since it is estimated instead of being measured from the platform. The function $\zeta^{pe}(S, MPSoC)$ models the performance estimation by mapping a set of statements $S = \{s_1, ..., s_{N_S}\}$ to the cycles consumed by them in the processor element pe of the MPSoC model. Here, $S \subseteq S^{\mathcal{P}}$ is an arbitrarily defined set, where $S^{\mathcal{P}}$ represents the statements of the program. Similarly, the function $\zeta^{ce}(b, MPSoC)$ models the communication overhead by returning the cycles required to communicate b bytes through the communication element ce, e.g., shared memory.

2) Call Graph: A call graph describes the calling relationships among the functions within a program. In this work we focus on a particular type of call graph, called *Dynamic Call Graph* (DGC), which contains only executed functions observed during the dynamic analysis. The DCG is formally defined as follows:

Definition 2: A *Dynamic Call Graph* (DCG) is a directed multigraph $DCG = (F, E_{cg})$, where F is the set of executed program functions and $E_{cg} \subseteq F \times F$ is a multiset that describes their calling relationships. Each edge $e^{cg}_{st} = (f_s, f_t) \in E_{cg}$ represents that the function f_s calls f_t and it includes the dynamic call count.

From the DCG it is possible to easily identify all the recursive functions within a program. Fig. 4 shows an example of a DCG, in which each node represents a function with its total workload percentage, and edges the calling relationships with the corresponding execution count. In this example `f1`, `f2` and `f3` are recursive functions.

3) Dependence Flow Graph: In our approach, the data flow analysis is enabled by the *Dependence Flow Graph* (DFG) [12]. This is an intermediate representation that combines control dependencies, data dependencies, and *Single-Entry Single Exit* (SESE) regions in a single graph. A SESE region is a part of the DFG with only one entry and one exit control points. Regions are enclosed by artificial nodes (i.e., switch and merge), as Fig. 5 shows. Data dependencies that are relevant within a region are redirected inside/outside the region through the artificial nodes. Moreover, regions represent high level language constructs, such as if-then-else blocks or loops. We use the DFG in our approach, in order to verify the mutual independence among recursive function calls. For this purpose, the *use* and *def* information of variables obtained from the static and dynamic analyses is annotated on the statements in the DFG. Then, the corresponding

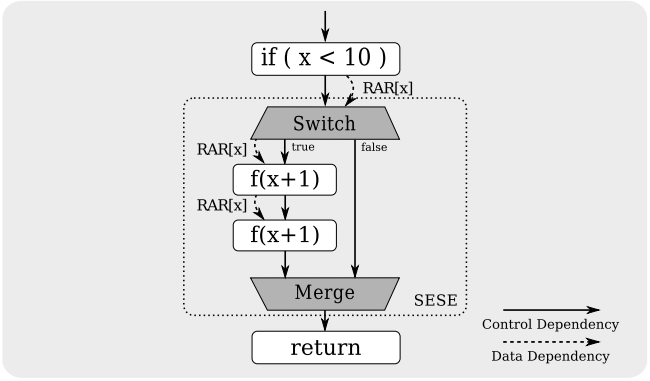
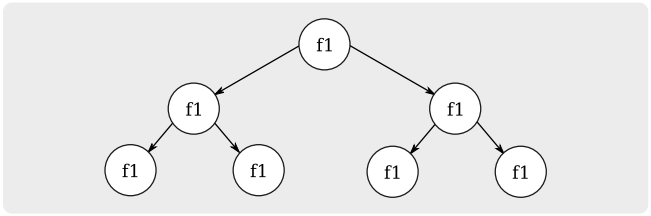


Fig. 5: Dependence Flow Graph (DFG) Example

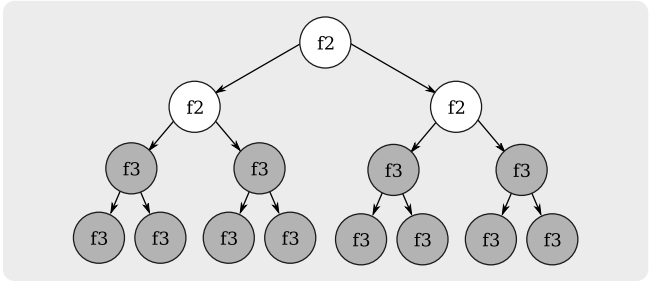
data dependency edges are created, namely: Read-After-Read (RAR), Read-After-Write (RAW), Write-After-Read (WAR) and Write-After-Write (WAW).

4) Recursion Tree: The behavior of recursive functions during a profiling run can be described in a *Recursion Tree* (RT), which is built based on the trace generated during the dynamic analysis. The RT in our approach is used to evaluate the profitability of parallelizing a given recursion and estimate an appropriate task granularity. Fig. 6 shows examples of RTs. We formally define a RT as follows.

Definition 3: A *Recursion Tree* (RT) is a directed rooted tree $RT = (I, E)$, where I is the set of nodes that represent the invocations to recursive functions, and E is the set of edges that represent their nesting relationships. The nodes are augmented with the workload of each function invocation, which is obtained from the performance information. The workload also accounts calls from the recursive functions to other intermediate non-recursive functions. Each edge $e_{xy} = (i_x, i_y) \in E$ represents that a node i_x immediately precedes a node i_y in the nesting calling hierarchy.



(a) Homogeneous Multi-Recursion



(b) Heterogeneous Multi-Recursion

Fig. 6: RTs Corresponding to the DCG from Fig. 4

As described earlier, our work focuses on multi-recursive functions, i.e., the branching factor of the root and internal nodes of the tree must be greater than one. We built one RT for every independent recursion identified in the DCG. For example, in the DCG of shown in Fig. 4 there are two independent recursions: one that only involves the function `f1` and another that involves the functions `f2` and `f3`, where `f2` is the root of the recursion. The RT for the recursion that involves the function `f1` is shown in Fig. 6a. In this work, we refer to this as *homogeneous multi-recursion*, as one function is exclusively calling itself. The RT for the recursion that involves the functions `f2` and `f3` is shown in Fig. 6b. In this work, we call this *heterogeneous multi-recursion* as it involves more than one single recursive function. Our approach supports both forms of multi-recursion.

Finally, we define the program model as the triple $\mathcal{PM} = (\mathcal{DFG}, DCG, \mathcal{RT})$, where $\mathcal{DFG}$ is the set of DFGs, DCG the dynamic call graph and $\mathcal{RT}$ the set of recursion trees.

D. Extraction of Recursion Level Parallelism

1) Work-Span Model: For a profitable parallel execution on multicore systems, programs have to exhibit enough parallelism for a given target platform. In this work, we use the *work-span* model [13] to quantify the potential parallelism existing in a given recursion. The work-span model is formulated in terms of *Directed Acyclic Graphs* (DAGs), where nodes represent tasks and the edges represent precedence relationships, i.e., a task is ready to execute when all its predecessors have finished with their work. The model also assumes that the scheduler is *greedy*, which means that no processor is left idle if there are tasks ready to be executed. The model relies on two measures: the total *work* (T_{work}) and the *span* (T_{span}). The work refers to the total time that it would take to serially execute all tasks. The span refers to the longest chain of tasks that must be executed in a sequence along any path in the DAG. This is equivalent to the *critical path* of the DAG. The work-span model defines a *speedup upper bound* for a multicore platform with P processors as follows:

$$S_{UB} = \min\left(P,\ \frac{T_{work}}{T_{span}}\right) \geq \frac{T_{work}}{T_{par}} \tag{1}$$

Fig. 7 shows a DAG example, where each box represents a task that is executed in one unit of time. Therefore, the total work (T_{work}) is 15 and the span (T_{span}) is 5, which results in a speedup upper bound of 3× (T_{work}/T_{span}), as it is also shown in the graph in Fig. 7. The work-span model can be also used to estimate a speedup lower bound by means of Brent's theorem [14]. This theorem bounds the parallel execution time T_{par} in terms of the work and the span as follows:

$$T_{par} \leq \frac{T_{work}}{P} + T_{span} \tag{2}$$

Equation 2 shows that T_{par} is composed of *imperfectly parallelizable work* that takes T_{span} independently of the number of processors P, and *perfectly parallelizable work*

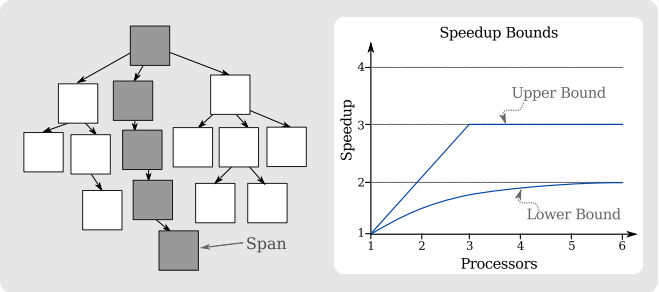


Fig. 7: Work-Span model example. Each task takes one unit of time: $T_{work} = 15$, $T_{span} = 5$

T_{work} that can be speeded up by P. Then, from Equation 2 the *speedup lower bound* can be derived as follows:

$$S_{LB} = \frac{T_{work}}{\frac{T_{work}}{P} + T_{span}} \leq \frac{T_{work}}{T_{par}} \tag{3}$$

Furthermore, Brent's theorem gives a formal motivation for *overdecomposition* derived from Equation 1:

$$\frac{T_{work}}{T_{par}} \approx P \quad \text{if} \quad \frac{T_{work}}{T_{span}} \gg P \tag{4}$$

Equation 4 says that ideally a linear speedup can be achieved by overdecomposing a problem to create more potential parallelism that can be exploited by the schedulers to achieve a better load balancing. This extra parallelism on a given platform with P processors is called *parallel slack* [13]:

$$PS = \frac{T_{work}}{P \cdot T_{span}} \tag{5}$$

2) Incorporating Overhead in the Work-Span Model: As previously described, Brent's theorem provides a lower bound to the speedup (S_{LB}) in terms of work and span. However, in practice the speedup is influenced by additional factors, such as task-creation and task-communication overhead, as Fig. 1 shows. These factors contribute proportionally to the work and the span. Therefore, here we augment the work-span model to incorporate the impact of this overhead.

The work is augmented as follows: $\widehat{T}_{work} = T_{work} + \tau \cdot N_{nt}$. Here, T_{work} is the work without the overhead, τ is the overhead required to create a single task on the target platform (a constant provided by the MPSoC model), and N_{nt} is the total number of tasks created in the DAG. Similarly, the span is augmented not only by including the task creation overhead but also the task communication overhead. A span that consider such overheads is called *burdened span* [8]. Then, the span is augmented as follows: $\widehat{T}_{span} = T_{span} + \tau \cdot N_{ts} + \zeta^{ce}(N_{ts}, MPSoC)$. Here, T_{span} is the span without the overhead, τ is the overhead to create a single task on the target platform, N_{ts} is the total tasks created along the critical path in the DAG, and $\zeta^{ce}(N_{ts}, MPSoC)$ is the overhead due to the communication among tasks along the critical path. Finally, the lower speedup bound can be redefined as follows to include the effect of the overheads:

$$\widehat{S}_{LB} = \frac{\widehat{T}_{work}}{\frac{\widehat{T}_{work}}{P} + \widehat{T}_{span}} \leq \frac{\widehat{T}_{work}}{T_{par}} \tag{6}$$

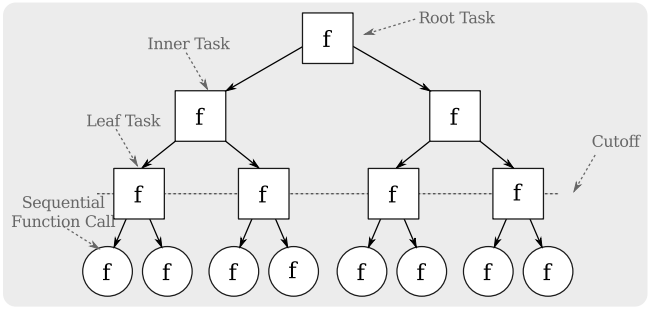


Fig. 8: Cut-off Mechanism Example

3) Parallelism Extraction: There are two fundamental constraints to achieve a profitable parallelization of multi-recursive programs, namely: *load balancing* and *overhead minimization*. On one hand, load balancing is about distributing the workload as even as possible across the available processors in the target platform. On the other hand, overhead minimization is about reducing the non-productive work related to task management. Unfortunately, these are conflicting constraints as load balancing implies having as much parallelism as possible (see Equations 4 and 5), while overhead minimization implies having as little parallelism as possible to reduce the impact of the overhead on the work and the span (see Equation 6). Therefore, the trade-off lies on providing enough decomposition to achieve a good load balancing, while keeping tasks large enough to amortize their management overhead.

The aforementioned trade-off can be achieved by selecting a proper *task granularity* [9]. In terms of the recursion tree, the task granularity defines what nodes are executed as parallel tasks and what nodes are executed as sequential function calls. The task granularity can be controlled by a mechanism called *cut-off* [10]. In this work, we use the task level cut-off mechanism, which is based on the number of parents from the root node. This approach is well-suited for tree-shaped task graphs, as is the case of the recursion tree. Fig. 8 shows an example of the maximum level cut-off mechanism applied to a recursion tree. In this example, the cut-off value is set to two, which means that all nodes with a recursion level less or equal than two are executed as tasks (boxes). Otherwise, they are executed as sequential function calls (circles). Defining a proper cut-off value is critical to control the overhead, especially in recursions with a high parallel slack, as there are potentially a large number of tasks in the recursion tree. From Fig. 8 it possible to distinguish tree types of tasks: *i)* a *root task*, which has no parent tasks but multiple child tasks, *ii) inner tasks*, which have both a parent tasks and child tasks and *iii) leaf tasks* that have no child tasks. In terms of workload, typically root and inner tasks are fine-grained, while leaf tasks are coarse-grained, then having the major impact on the load balancing.

Algorithm 1 shows our heuristic for extraction of Recursion Level Parallelism (RLP). First multiple platform definitions and user constrains are extracted (Lines 1-6). Then each independent recursion in the program is analyzed. The call to the procedure at Line 9 extracts the root function for

Algorithm 1: Extraction of Recursion Level Parallelism

Input: Program Model ($\mathcal{PM}$), $MPSoC$ Model, User Constraints (UC)
Output: OpenMP Annotated Code

```
   // Extract Platform Definitions
1  P ← GetProcessorNum(MPSoC);
2  τ ← GetTaskOverhead(MPSoC);
   // Extract User Constraints
3  η_WL ← GetWorkloadCstr(UC);
4  η_PS ← GetParallelSlackCstr(UC);
5  η_LI ← GetLoadImbalanceCstr(UC);
6  η_TO ← GetMaxTaskOverheadCstr(UC);
   // Iterate Over Indepedent Recursions
7  DCG ← GetDCG(PM);
8  PA ← ∅;
9  for f_i ∈ GetRootRecursiveFunction(DCG) do
       // Data Dependency Analysis
10     DFG^fi ← GetDFG(PM, f_i);
11     if CheckCallSitesIndependence(DFG^fi) then
           // Profitability Analysis
12         WL ← GetTotalWorkload(f_i);
13         RT^fi ← GetRT(PM, f_i);
14         (T_work, T_span) ← GetWorkAndSpan(RT^fi);
15         PS ← T_work/(P · T_span);
16         if (WL ≥ η_WL) ∨ (PS ≥ η_PS) then
               // Task Granularity Analysis
17             CO ← 1;
18             while (LI ≥ η_LI) ∨ (TO ≤ η_TO) ∨ (CO < TH)
                   do
19                 LI ← EstLoadImbalance(RT^fi, CO, P)
                   TO ← EstTaskOverhead(RT^fi, CO, τ)
                   CO ← CO + 1;
20             end
21             PA^fi ← CreatePA(DFG^fi, RT^fi, CO);
22             PA ← PA ∪ PA^fi;
23         end
24     end
25 end
   // Code Generation
26 GenSpeedupBoundsGraphs(PA, MPSoC);
27 AnnotateCode(PA);
```

every recursion. For example, in the DCG presented in Fig. 4, the root functions of each recursion are `f1` and `f2`. As the first step for every recursion, the mutual independence of the recursive call-sites is verified by analyzing the DFG. If the recursion consists of multiple independent static call-sites, it is verified that there are no data dependencies among those call-sites. If the recursion consists of a single static call-site within the body of a loop, it is verified that this loop does not present loop-carried dependencies that would prevent parallelization. The loop-carried dependencies can be directly identified from the SESE region representing the loop. Moreover, it is checked that the recursive function does not perform write accesses to global variables. If mutual independence among recursive call-sites is confirmed, then a profitability analysis is performed (Lines 12-16). In this analysis two profitability constraints are considered: *i)* the total workload of the recursion (WL) is greater than a user-defined threshold (η_{WL}) and *ii)* the parallel slack (PS) of the RT is greater than an user-defined threshold

(η_{PS}). The first constraint ensures that the recursion is a hotspot of the application, and the second constraint ensures that the RT exhibit enough parallelism for the selected target platform.

The last step is to find a proper *cut-off* value (Lines 17-22). A high cut-off value results in a good load balancing but in a high task overhead. Therefore, the aim is to select a cut-off value that provides a good trade-off between load balancing and overhead. The heuristic incrementally evaluates cut-off values between one and the height of the recursion tree (HT). For every possible cut-off value the load imbalance and task overhead are estimated. The procedure at Line 19 performs the load imbalance estimation. For this purpose we use the *Longest Processing Time* (LPT) algorithm [15], to estimate the task distribution on the processors of the target platform and then the resultant load imbalance. The procedure at Line 19 estimates the overall task overhead (TO) based on the task overhead for a single task (τ) and the number of tasks for the given cut-off value (CO). A cut-off value is selected if either the resultant load imbalance (LI) is smaller than an user-defined threshold (η_{LI}) or the resultant task overhead is higher than an user-defined threshold (η_{TO}). Once a proper cut-off value has been identified, the speedup bounds graphs (see Equations 1 and 6) and a parallel annotation (PA^{f_i}) are generated. The annotations are accumulated in a set $\mathcal{PA}$, which represents the input information for the code generation described in the next section.

E. Code Generation

The information provided by the parallel annotations is the basis for code generation. Our toolflow annotates the input code with OpenMP task directives (i.e., `#pragma omp task`) introduced in the OpenMP 3.0 specification [7]. This model is well-suited to parallelize recursive programs, as it allows to explicitly specify code regions as tasks, which can be nested within other tasks.

In the OpenMP tasking model there are two types of tasks: *tied* and *untied*. A tied task is executed by one single OpenMP thread from the beginning until the end (i.e., a suspended task cannot be stolen by other threads to continue its execution), while an untied task is more flexible, as it can be partially executed by multiple threads (i.e., a suspended task can be stolen by other threads to continue its execution). The standard specifies that tasks are by default tied, otherwise the behavior can be modified by adding the `untied` clause to the `task` construct. Previous research [16] has shown that untied tasks provide better performance due to their flexibility, therefore this type of task is used here. An example of a `task` construct with the `untied` clause is shown at Line 4 in Fig. 9a. Another key construct of the tasking model is `taskwait`. This construct specifies a synchronization point where the current task is suspended until all its children tasks are done with their work. An example of a `taskwait` construct is shown at Line 8 in Fig. 9a.

Finally, to implement the cut-off mechanism a new parameter is added to the functions involved in the recursion, to keep track of the recursion level (e.g., `int depth`), as it is shown at Line 2 in Fig. 9a. In addition, the cut-off value is defined as a global constant value, as Line 1 in Fig. 9a shows. The cut-off mechanism itself can be implemented in two ways. The first option is to use a C/C++ language `if` construct, where in the true block each recursive static call-site is annotated with an OpenMP pragma and at the end the `taskwait` construct is inserted to define the synchronization point, while in the false block the static recursive call-sites are invoked without any OpenMP pragmas. Fig. 9a shows a simplified code generation example using the language `if` construct cut-off mechanism. The other option is to use the `if` clause of the `task` construct, as it allows to dynamically serialize a task based on a boolean condition. An example of this is shown in Fig. 9b. A similar approach is performed in the case of the code generation for multi-recursions with one single static call-site within a loop (see Fig. 3b). In this case the `task` construct is inserted before the single recursive call-site and the `taskwait` construct after the loop.

```
1 const int cutoff = N;
2 void f(..., int depth) {
3   if (depth < cutoff) {
4     #pragma omp task untied
5     f(..., depth+1);
6     #pragma omp task untied
7     f(..., depth+1);
8     #pragma omp taskwait
9   } else {
10    f(..., depth+1);
11    f(..., depth+1);
12  }
13 }
```

(a) Language `if` Construct Cut-off Mechanism

```
1 const int cutoff = N;
2 void f(..., int depth) {
3   #pragma omp task untied if (depth<=cutoff)
4   f(..., depth+1);
5   #pragma omp task untied if (depth<=cutoff)
6   f(..., depth+1);
7   #pragma omp taskwait
8 }
```

(b) OpenMP `if` Clause Cut-off Mechanism

Fig. 9: OpenMP Code Generation Simplified Examples

III. Experimental Evaluation

This section describes the experimental evaluation of the proposed parallelization toolflow.

A. Experimental Environment

The toolflow was implemented on the Clang/LLVM 3.9 compiler framework, and integrated on the Eclipse GUI to enable an easy interaction with the developer. The dynamic analysis is enabled by the CoEx source level profiler [17]. The target platform considered in this work is the 66AK2H MPSoC [11]. The evaluation was performed on the quad ARM A15 MPCore running at 1.4GHz. The MPSoC model was calibrated with data obtained from the actual platform, such as the task overhead by using the EPCC OpenMP micro-benchmark suite [18].

TABLE I: Characteristics of the Benchmarks

Benchmark	Input	Computation Structure	Recursion Type	Functions in Recursion	Maximum Branching Factor	Workload (%)	Cut-off Value
FFT	16M Floats	At leaf nodes	Heterogeneous	12	1M	98	2
Fib	40	At each node	Homogeneous	1	2	100	2
Health	4 levels with 18 cities	At each node	Homogeneous	1	365	97	1
NQueens	13x13	At each node	Homogeneous	1	13	98	3
Sort	16M	At leaf nodes	Heterogeneous	2	7	90	4
Strassen	2048x2048 matrix	At each node	Heterogeneous	2	8	98	3

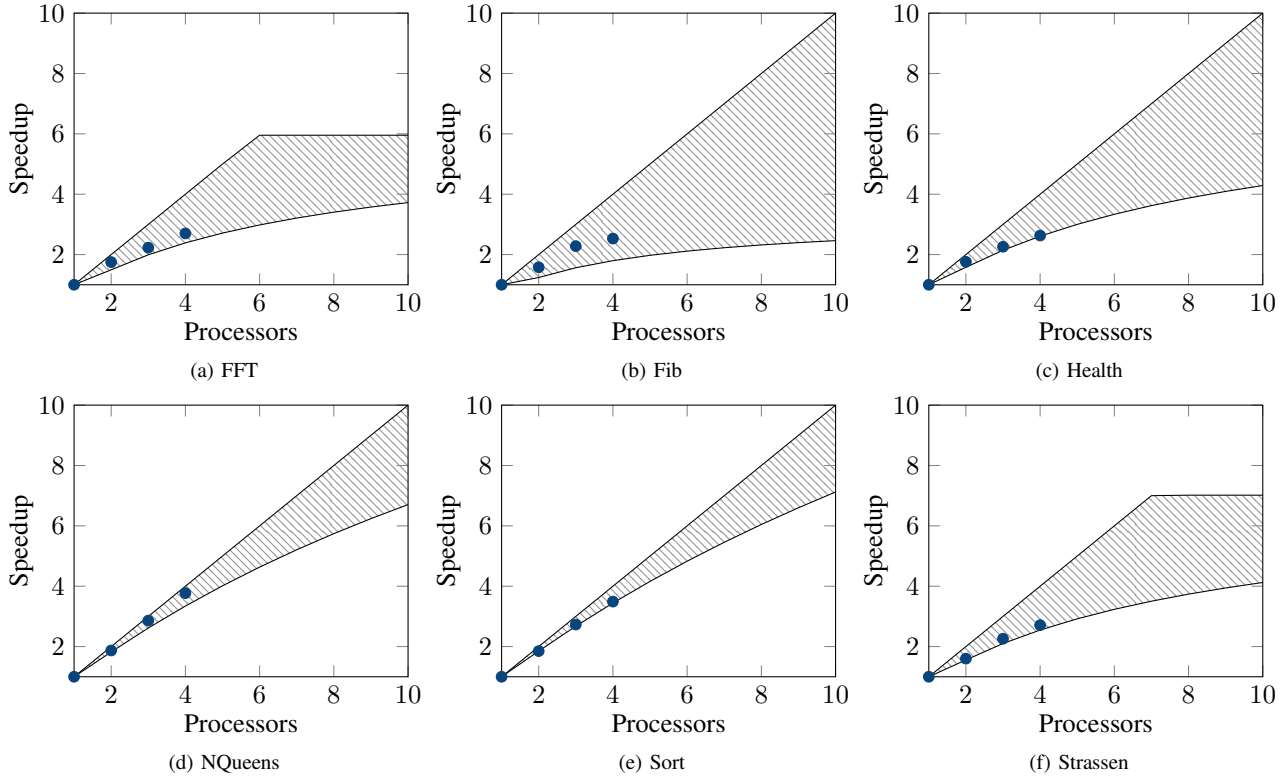


Fig. 10: Scalability Speedup Results

B. Case Studies

The case studies considered in this work are taken from the *Barcelona OpenMP Tasks Suite* (BOT) [16]. During the analysis the following user constraints values were used: *i)* 25% as the minimum workload (η_{WL}) to consider a recursion as a parallelization candidate, *ii)* 2 as the minimum parallel slack (η_{PS}), *iii)* 5% as the minimum load imbalance (η_{LI}) and *iv)* 10% as the maximum allowed task overhead (η_{TO}). After analyzing the benchmarks with our toolflow some of them were discarded due to the lack of multiple recursion (e.g., `Alignment` and `SparseLU`) or due to the lack of mutual independence among recursive call-sites (e.g., `Floorplan`). Table I summarizes the characteristics of the benchmarks that exhibit multiple recursion. The characteristics include the input used, the computation structure, the recursion type (i.e., homogeneous or heterogeneous as shown in Fig. 6), the number of recursive functions in a each independent recursion (e.g., `f1` or `f2+f3` shown in Fig. 4), the maximum branching factor in the RT, the total workload and the cut-off value selected by our toolflow considering four processors. It is worth mentioning that the `FFT` benchmark exhibits two independent recursions, However, one of them presents a low workload, therefore, it was discarded by the toolflow, as it is not a profitable parallelization candidate.

Fig. 10 shows the scalability speedup results from one to ten processors. The speedup bounds graphs were automatically generated by our framework according to Equations 1 and 6. The solid dots on the graphs represent the speedup values resulting from the experimental evaluation of the parallelized benchmarks on the target platform. The baseline used here is the execution of the sequential version of each benchmark on an ARM A15 processor, and the cut-off values were selected according to the number of processors used. Moreover, the default code generation settings were used, i.e., untied tasks and the language `if` cut-off mechanism. The first observation from these results is that the experimental speedup points fall within the speedup regions estimated by our toolflow. In addition, the speedup graphs reflects diverse results, as

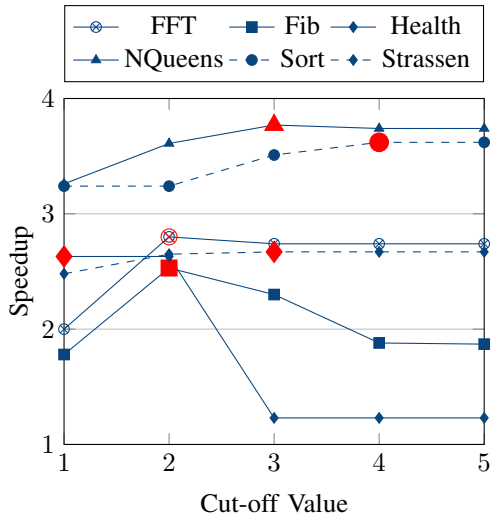


Fig. 11: Speedup Results For Multiple Cut-off Values

some benchmarks exhibit good scalability (e.g., `NQueens` and `Sort`), while others quickly saturate (e.g., `FFT`, `Fib` and `Strassen`). In contrast to benchmarks with poor scalability, scalable benchmarks present a good computation to overhead ratio and a good computation to communication ratio (high arithmetic intensity [13]).

Fig. 11 shows speedup results on four processors for different cut-off values, where the marks representing the cut-off values selected by our framework are highlighted. On one hand, for benchmarks with a large branching factor (e.g., `FFT` and `Health`) or with a poor computation to overhead ratio (e.g., `Fib`) our approach was able to correctly select small cut-off values as the most convenient ones to avoid the parallelization overhead (e.g., one or two). On the other hand, benchmarks with a small branching factor and with a good computation to overhead ratio benefit from bigger cut-off values to achieve a better load balancing, while at the same time being less sensitive to the parallelization overhead (e.g., `NQueens`, `Sort` and `Strassen`). Our toolflow was also able in these cases to select proper cut-off values.

As described in Section II-E there are two main choices during code generation: *i)* task type and *ii)* cut-off mechanism. Both aspects are evaluated here to corroborate the appropriateness of the default settings selected in the code generator phase of our toolflow. Fig. 12 shows the speedup results on four processors using tied and untied tasks. From these results, it can be observed that overall untied tasks provide slightly better speedup numbers in some cases (e.g., `Fib`, `NQueens` and `Sort`). This can be attributed to flexibility of untied tasks, which allows to resume suspended tasks in any idle thread and thus resulting in a better load balancing. This confirms that the use of untied tasks is an appropriate default setting.

Regarding the cut-off mechanism, we evaluated the OpenMP `if` clause and the language `if` construct cut-off mechanisms introduced in Fig. 9. The results presented in Fig. 13 show that overall the language `if` construct approach provides the best speedup results, in particular for benchmarks that are more sensitive to the parallelization overhead (e.g., `Fib`). The reason for this being that even when the `if` clause evaluates to `false` (i.e., for tasks with a recursion level higher than the cut-off value) there is still an associated overhead, which can be significant for benchmarks with a low computation to overhead ratio. These results support our choice of the language `if` construct as the default cut-off mechanism.

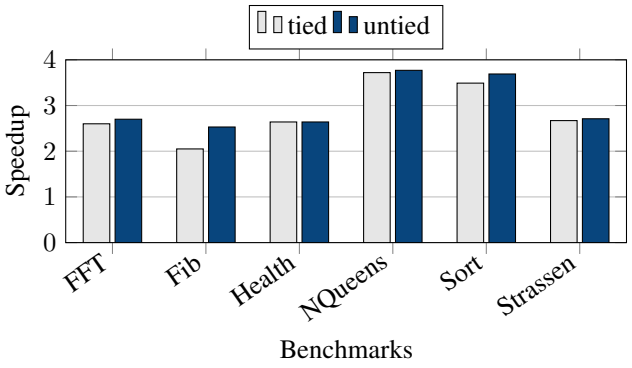


Fig. 12: Tied vs Untied Tasks Speedup Results

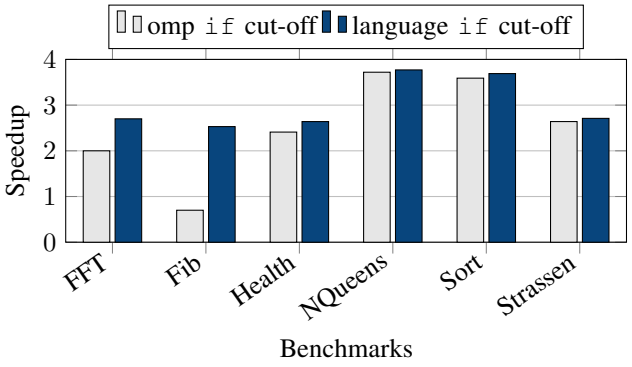


Fig. 13: Cut-off Mechanisms Speedup Results

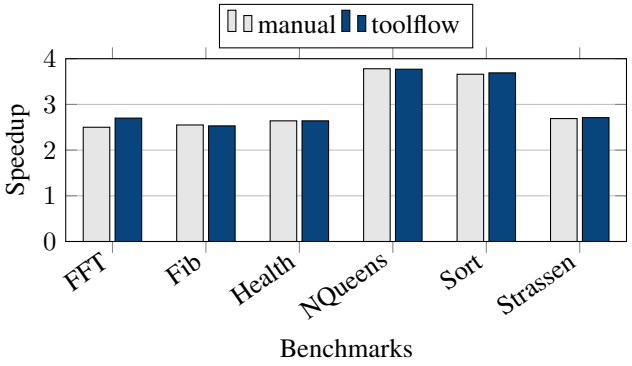


Fig. 14: Manual vs Toolflow Speedup Results

Finally, to further assess the effectiveness of our toolflow, we compared the speedup results achieved by our framework with the ones achieved by manual parallelization. This evaluation is possible since the BOT suite provides handwritten OpenMP versions of the benchmarks by expert programmers. For a proper comparison we selected the handwritten version of each benchmark, in which `untied` tasks and language `if` construct cut-off mechanism are used. Fig. 14 shows the speedup results obtained from this evaluation on four proces-

sors. As it can be observed, our framework overall achieves similar speedup results as with manual parallelization. This is an expected result, provided that in general our framework was able to generate functionally equivalent OpenMP pragmas to the handwritten ones. However, in the particular case of `FFT` our toolflow achieved a better speedup ($2.5\times$ vs $2.7\times$). The reason being that in the manual version of its heterogeneous recursion, the cut-off value is only applied to the root recursive function. Therefore, the granularity of the tasks due to the other recursive functions within the whole recursion is not controlled, which results in a significant amount of fine-grained tasks that increases the parallelization overhead. This result shows the effectiveness of our approach while parallelizing heterogeneous recursions, as it analyzes them as a whole.

IV. Related Works

Existing parallelizing compiler technologies have mainly focused on the extraction of loop level patterns, such as *Data Level Parallelism* (DLP) and *Pipeline Level Parallelism* (PLP), as well as on more irregular patterns such general *Task Level Parallelism* (TLP). Polly [2] is a state-of-the-art parallelization framework integrated in the LLVM compiler, which relies on the polyhedral model to identify DLP, which can be exploited as coarse-grained or fined-grained parallelism. Similar to our approach, Tournavitis [19] proposed a profile-driven parallelization toolflow to overcome the traditional limitations of static analysis. His work focused on the extraction of DLP and PLP. In the embedded domain, Cordes [3] proposed an approach based on a hierarchical model of the program, which is analyzed by *Integer Linear Programming* (ILP) and *Genetic Algorithms* (GAs) to extract TLP and PLP. In [6], we presented a toolflow for the extraction of DLP, PLP and TLP targeting multicore Android devices. In contrast to our approach, none of the previous works have addressed the extraction of *Recursion Level Parallelism* (RLP).

There have been also research efforts that address the parallelization of recursive programs. The compiler proposed by Rugina and Rinard [20] is an early work on the parallelization of divide-and-conquer algorithms in which its subproblems access disjoint array regions. This compiler relies on pointer and symbolic analyses to statically reason about the mutual independence of recursive call-sites. Gupta et al. [21] proposed a similar approach, where compile time analysis is complemented by a runtime system to perform speculative parallelization. However, the authors do not discuss the overhead introduced by the runtime speculation that could limit the effectiveness of the approach. Frameworks that heavily require user intervention have been also proposed. For example, in REAPAR [22] the independence of the recursive call-sites is assumed and not verified, leaving the responsibility of the data dependency analysis to the developers. Similarly, Ariadne [23] is a framework in which the developer has to insert directives to instruct the compiler where and how to parallelize recursive programs. In contrast to the previous works, our approach besides verifying mutual independence of recursive call-sites, it also performs a profitability analysis and controls the task granularity, all of this with a minimal user intervention.

V. Conclusions and Future Work

In this paper, we presented a toolflow that addresses the extraction of recursion level parallelism for multicore embedded systems. The approach relies on a program model that is built based on a combination of static and dynamic analyses. The program model is analyzed by an algorithm that first identifies recursions that could result in a profitable parallelization. Then, it verifies the mutual independence of the recursive call-sites. Afterwards, the task granularity is carefully selected to achieve a proper trade-off between load balancing and parallelization overhead. Finally, the input program is automatically annotated with OpenMP pragmas according to the parallelization opportunities identified by our toolflow. Results show that our toolflow is able to achieve similar speedups as manual parallelization by expert programmers. In future work, we plan to extend the toolflow for heterogeneous platforms by means of the OpenMP Accelerator Model.

References

[1] R. E. Johnson, "Software development is program transformation," in *Proc. of FoSER'10*. New York, NY, USA: ACM, 2010, pp. 177–180.

[2] T. Grosse, A. Groesselinger, and C. Lengauer, "Polly performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[3] D. A. Cordes, "Automatic parallelization for embedded multi-core systems using high-level cost models," Ph.D. dissertation, TU Dortmund, 2013.

[4] J. Castrillon and R. Leupers, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Springer, 2014.

[5] M. A. Aguilar, R. Leupers, G. Ascheid, and L. G. Murillo, "Automatic parallelization and accelerator offloading for embedded applications on heterogeneous MPSoCs," in *Proc. of the 53rd DAC*. New York, NY, USA: ACM, 2016, pp. 49:1–49:6.

[6] M. A. Aguilar, J. F. Eusse, P. Ray, R. Leupers *et al.*, "Towards parallelism extraction for heterogeneous multicore android devices," *IJPP*, pp. 1–33, 2016.

[7] OpenMP Review Board, "OpenMP api. version 3.0," [Online] Available www.openmp.org/wp-content/uploads/spec30.pdf (accessed 02/2017).

[8] Y. He, C. E. Leiserson, and W. M. Leiserson, "The Cilkview scalability analyzer," in *Proc. of the SPAA*. NY, USA: ACM, 2010, pp. 145–156.

[9] U. A. Acar, A. Charguéraud, and M. Rainey, "Oracle scheduling: Controlling granularity in implicitly parallel languages," in *Proc. of the OOPSLA*. NY, USA: ACM, 2011, pp. 499–518.

[10] A. Fonseca and B. Cabral, "Evaluation of runtime cut-off approaches for parallel programs," in *Proc. of the 12th VECPAR*. Springer, 2016.

[11] Texas Instruments, "Multicore DSP+ARM Keystone II System-On-Chip (SoC)," [Online] http://www.ti.com/lit/gpn/66ak2h14 (accessed 2/2017).

[12] R. C. Johnson, "Efficient program analysis using dependence flow graphs," Ph.D. dissertation, Cornell University, August 1994.

[13] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[14] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. ACM*, vol. 21, no. 2, pp. 201–206, Apr. 1974.

[15] T. Gonzalez, O. H. Ibarra, and S. Sahni, "Bounds for lpt schedules on uniform processors," *SIAM*, vol. 6, no. 1, pp. 155–166, 1977.

[16] A. Duran, X. Teruel, R. Ferrer, X. Martorell *et al.*, "Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Proc. of ICPP*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

[17] J. F. Eusse, C. Williams, and R. Leupers, "CoEx: A novel profiling-based algorithm/architecture co-exploration for ASIP design," *ACM Transactions on Reconfigurable Technology and Systems*, May 2014.

[18] J. M. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for OpenMP tasks," in *Proc. of IWOMP*. Springer, 2012, pp. 271–274.

[19] G. Tournavitis, "Profile-driven parallelization of sequential programs," Ph.D. dissertation, University of Edinburgh, 2011.

[20] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," in *Proc. of PPoPP*. NY, USA: ACM, 1999, pp. 72–83.

[21] M. Gupta, S. Mikhopadhyay, and N. Sinha, "Automatic parallelization of recursive procedures," in *Proc. of PACT*, 1999, pp. 139–148.

[22] L. Prechelt and S. U. Hanssgen, "Efficient parallel execution of irregular recursive programs," *IEEE TPDS*, vol. 13, no. 2, pp. 167–178, Feb 2002.

[23] A. Mastoras and G. Manis, "Ariadne - Directive-based parallelism extraction from recursive functions," *J. Par. Distr. Comput.*, vol. 86, pp. 16–28, 2015.