UNIVERSITÄT AUGSBURG

User Manual for the Optimization and WCET Analysis of Software with *Timing Analyzable Algorithmic Skeletons*

R. Jahr, A. Stegmeier, R. Kiefhaber, M. Frieb, and T. Ungerer

Report 2014-05 September 2014

INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Contents

1 Introduction

Timing analysis is necessary for hard real-time software to determine worst-case execution times (WCETs). One option (see [11] for an overview) is static analysis as implemented by the OTAWA tool-set [1].

Timing analysis of parallel code (i.e., multi-threaded code) is even more complex than the analysis of single-threaded code. To enable it, synchronization points of threads must be described similar to conventional flow-facts for OTAWA manually before running the analysis. Placing IDs in source code is one part; the second is to describe the relationships between these IDs in an XML file. With unstructured parallelism, this can be a big burden; with structured parallelism implemented with *Parallel Design Patterns (PDPs)* the effort is reduced and better WCET results can be achieved [8].

PDPs describe best-practice solutions for situations of parallelism. We extended the meta-pattern to describe PDPs by Mattson et al. in [4, 8] and collected PDPs from four industrial applications in the *parMERASA Pattern Catalogue* for *Timing Predictable Parallel Design Patterns* [4]. However, PDPs are abstract concepts with textual description and do not come with source code for the implementation. As visible in Figure 1.1, the respective concept on source code level is *Algorithmic Skeletons*.

So far, we are not aware of any skeleton implementation for hard-real time embedded systems based on programming language C with POSIX (or comparable) synchronization primitives. The *Timing Analyzable Algorithmic Skeletons (TAS)* close this gap. With them, (a) the implementation effort for PDPs can be reduced because tested code is used, (b) software maintainability is improved because of a separation of concerns between application logic and parallelism control, and (c) timing analysis is simplified: Instead of describing each synchronization point, e.g., a barrier or lock, a skeleton instance is described in an intermediate XML format. With an XML transformer, this intermediate TAS specific format is then translated into the XML format for OTAWA.

For the parallelization of sequential pieces of code of legacy single-core applications we developed the *pattern-supported parallelization approach* [7], which is also applicable for applications with hard real-time requirements [6]. The core idea of the approach (see Figure 1.2) is to express parallelism in a model built from the existing single-core source code. This model can then be optimized manually or automatically [5] (Chapter 2). After this, to implement the optimized model of the software, the TAS can be applied.

The structure of this technical report is as follows: Chapter 2 very briefly describes our software for automatic optimization of a software model with PDPs. The focus of Chapter 3 is on how to make use of the TAS. Support for timing analysis of software with TAS is shown in Chapter 4.

Acknowledgments: The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA).

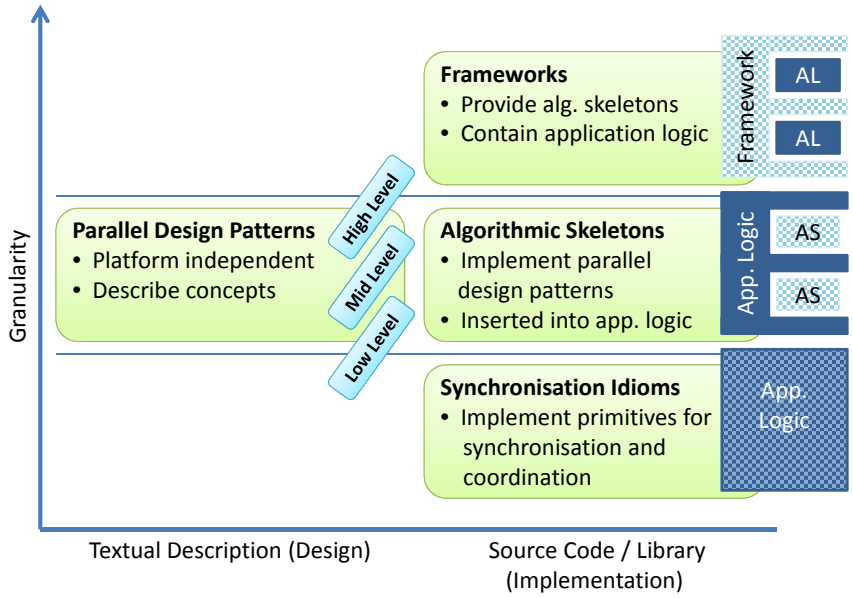

Figure 1.1: Overview of the concepts for parallelization with Parallel Design Patterns (PDPs). The illustrations on the right show the cooperation with application logic (custom code) and the source-code parallelization concepts; checkered elements contain synchronization elements.

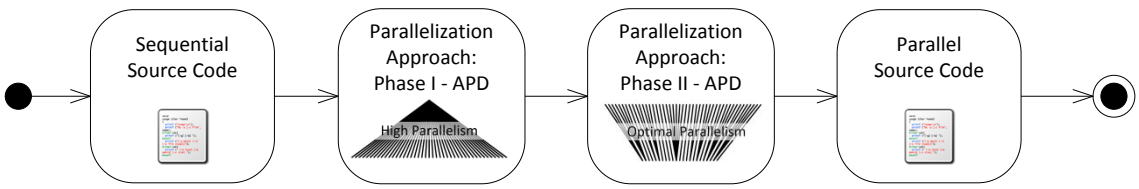

Figure 1.2: Overview of the pattern-supported parallelization approach [7]

2 Multi-objective Optimization of an Activity and Pattern Diagram (APD)

Here, very briefly the tool for model-based optimization is described. The results of paper [5] were gained with an earlier version of it. In the pattern-supported parallelization approach [7] (see Figure 1.2) the optimization tool is applied to reduce the possible parallelism towards a platform-specific optimum.

The optimization tool is implemented in Java. It consists of several Netbeans[1] projects. The optimization is done with SMPSO [9]; for this the jMetal framework [3] is used. The source code of our optimization tool is available under the GNU LGPL v3 license:

`https://github.com/parmerasa-uau/parallelism-optimization/`

The input model of the software with high degree of parallelism is described as XML file. The different PDP instances can be nested. Sequential execution order is expressed by the order in which the PDPs are arranged in non-parallel tags. The following code shows a simplified and obfuscated example for a control code.

```
<activity_pattern_diagram name="APD">
    <task_parallelism name="TP1">
        <activity name="B8F621C170007E89E6BEAD55B9FD" weight="114882" />
        <task_parallelism name="TP2">
            <activity name="7B6CE4223224129CEDAFAF251357" weight="20147" />
            <activity name="8F195E8876AC94965E8CCE779EB6" weight="39429" />
            <activity name="34E37CD625D2D525AD9428B84B57" weight="21253" />
        </task_parallelism>
        <task_parallelism name="TP3">
            <task_parallelism name="TP4">
                <activity name="A76393250B76433E65846AE6403D" weight="19338" />
                <activity name="C4180E198902A4531D6F6376B80E" weight="19870" />
            </task_parallelism>
            <activity name="0C39C15205FEA1575B1B4D880EAD" weight="65992" />
            <task_parallelism name="TP7">
                <activity name="38DB925FA49CF01772DD14E312C5" weight="68700" />
                <activity name="3921B9CDDCAC9D7D32F0088A37C3" weight="107919" />
            </task_parallelism>
        </task_parallelism>
    </task_parallelism>
    <task_parallelism name="TP11">
        <activity name="DCB0712655F3F6DBA1483BAEE56C" weight="23088" />
        <activity name="56530AB823899A834E05765083BC" weight="19660" />
        <activity name="1E0658902D48ABFB48E188A3A635" weight="21454" />
    </task_parallelism>
</activity_pattern_diagram>
```

In addition, a file with all accesses to shared variable for each activity is needed. Typically the name of an activity—see code section above—maps to one function in the code. If the function

[1] Website: `http://www.netbeans.org`

calls other functions, then their accessed variables are added to the set of variables of the calling function. An obfuscated file for the crawler crane control code can be found on GitHub.

The basic optimization idea is to find a number of threads for every PDP instance. Hence, a variable is set up for every PDP instance. If the number of threads for a skeleton is lower than the code fragments it can execute, then these code fragments are put in descending order by the weight and grouped into as many partitions as threads are available (greedy approximation algorithm). Multiple objectives are available and several of them can be chosen for a multi-objective optimization:

Objective.OBJ_CORES Number of cores needed to execute the optimized model.

Objective.OBJ_PATTERNS Number of parallel design patterns that are executed with more than one thread.

Objective.OBJ_DURATION Estimated duration of the execution time of the software based on the weight attributes in the input XML. The performance model is influenced by latencies defined in class `Platform` for the execution of a skeleton and the estimated number of accesses to shared global variables via mutator functions, which has an additional delay also defined in class `Platform`.

Objective.OBJ_GLOBALS Number of shared global variables which are accessed by more than one thread and hence need to be secured with locks.

Objective.OBJ_GLOBALS_ACCESSES Number of approximated accesses to shared global variables.

The complete configuration is done in class `parallelismanalysis.ParallelismAnalysis`. Running this class does the optimization. At the end, a list of near-optimal configurations is printed:

```
6131818.0 1.0 for 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
3510570.0 2.0 for 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
2479180.0 3.0 for 2.0 1.0 1.0 2.0 1.0 1.0 2.0 1.0 2.0 1.0 1.0
2145180.0 4.0 for 2.0 2.0 1.0 2.0 1.0 1.0 2.0 1.0 4.0 2.0 1.0
1235848.0 9.0 for 3.0 1.0 2.0 3.0 1.0 1.0 2.0 1.0 1.0 3.0 1.0
1308060.0 8.0 for 3.0 1.0 2.0 3.0 1.0 1.0 2.0 1.0 1.0 2.0 1.0
1390649.0 7.0 for 3.0 1.0 2.0 2.0 1.0 1.0 2.0 1.0 1.0 2.0 1.0
1697254.0 6.0 for 3.0 1.0 2.0 2.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0
1190640.0 11.0 for 3.0 1.0 2.0 3.0 2.0 1.0 2.0 2.0 1.0 3.0 1.0
1902501.0 5.0 for 3.0 1.0 2.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1129445.0 12.0 for 4.0 1.0 2.0 3.0 2.0 1.0 2.0 2.0 1.0 3.0 1.0
1115641.0 13.0 for 4.0 1.0 2.0 3.0 2.0 2.0 2.0 2.0 1.0 3.0 1.0
```

The list shows only Pareto-optimal configurations. In the above example, the first two columns show the estimated execution time and the number of threads working in parallel. The other columns show the number of threads per skeleton instance (of the full crawler crane model). A fitting configuration can then be selected and implemented with the TAS.

3 Usage of the Timing Analyzable Algorithmic Skeletons (TAS)

The *Timing Analyzable Algorithmic Skeletons* (*TAS*) are an implementation of the *Timing Analyzable Parallel Design Patterns (PDPs)* introduced in [4]. They are intended to facilitate the parallelization of sequential application code written in C language. The *TAS* are applicable both for the parMERASA simulator for the predictable multi-core parMERASA architecture [10] as well as for other platforms with POSIX threads.

The remainder of this section is structured as follows: Section 3.1 describes the basic knowledge and possible configurations of the *TAS*. In Section 3.2, we reveal details about applying the skeletons. Therefore, we first respond to necessary preparations before the invocation of the *TAS* is presented afterward. In order to provide a better understanding, the described process is additionally shown at an example.

3.1 Basics of Applying TAS

The source code of the skeletons can be downloaded and is available under the GNU LGPL v3 license[1]:

`https://github.com/parmerasa-uau/tas/tree/master/first-fit/`

The root directory contains a folder `tas` and several other files:

- Folder `tas` contains the TAS implementation.
- `main.c` contains a simple example for TAS with POSIX threads. The actual algorithms executed by this file are located in folder `tas` (starting example: matrix multiplication). On the parMERASA architecture this is more complex and `main_parmerasa.c` can be used.
- `Makefile` is the Makefile to build the demo application with TAS and POSIX threads. `Makefile_parmerasa` is the counterpart for the parMERASA simulator.

If TAS should be used on the parMERASA architecture then further files are necessary, which are always the same for all applications on the simulator and not shipped with TAS: `segmentation.h` and `kernel_lib` [2]. In TAS, the files `tas/tas.h` and `platform.h` must then

[1] On GitHub, also a second version is available, which is suitable only for the parMERASA platform. Here it can be defined explicitly which worker shall be used by a skeleton; hence, the mapping can be fixed because it is clear on this hardware platform which thread is executed on which core.
The URL is: `https://github.com/parmerasa-uau/tas/tree/master/static/`

be changed according to the number of available cores, also the PARMERASA flag must be set and TAS_POSIX must be unset.

On both supported platforms, within *main.c*, the user has to modify only the preprocessor directive for defining the macro *MAIN_CORE_0*. The value of this macro represents the *main* function of the application code to parallelize. Therefore, the programmer indicates the root function of the application here and if necessary includes the corresponding header file. Thereby, it is crucial that the added *#include* directive is located below the already existing *#include* statements. The following listing shows an example of defining the function application_main() in file application_main.c as root function for application code.

```
#include  application_main.h
#define   MAIN_CORE_0 application_main()
```

Within the function linked to *MAIN_CORE_0*, code is written in a sequential way and skeletons are invoked for executing segments of code in parallel. There are several different skeletons available, each implementing one specific *PDP*. The summery of all *TAS* and the implemented *Parallel Design Patterns* is listed below:

- tas_taskparallel → *Task Parallelism*
- tas_dataparallel → *Data Parallelism*
- tas_pipeline → *Pipelining*

3.2 Details for Applying TAS

In this section we take a closer look on applying skeletons. Therefore, we describe all needed actions to get from sequential to parallel code. This is the necessary work to be done in the last phase of the pattern-supported parallelization approach [7], which is our preferred way for parallelization.

We give an example for each step in order to provide a better understanding. To do this, the sequential code shown below is being parallelized for executing on the parMERASA simulator.

```
// Variables

// Execution
void func() {
  // sequential part
  int max = 4;

  int a = 1;
  int b = 2;
  if (a < b) {
    b = a;
  }
  else {
    a = b;
  }

  // parallel part 1
  int i;
  for (i=0; i<max; i++) {
    a++;
  }
```

```
  // parallel part 2
  int j;
  for (j=0; j<5; j++) {
    b++;
  }
}
```

For this, we assume that the first part of function `func` (marked with comment: `sequential part`) shall be executed sequentially. The rest of the code (marked with comments: `parallel part 1` and `parallel part 2`) is a parallel execution of two threads. Skeleton *tas_taskparallel* is a suitable solution for this parallelization problem and will be applied here for that reason.

In the remainder, Section 3.2.1 explains necessary preparations. Afterward, Section 3.2.2 describes the invocation of a skeleton. Finally, a full example is given in Section 3.2.3 to summarize all required actions.

3.2.1 Preparations for Invoking Skeletons

Different information is needed for utilizing a particular skeleton. This includes the knowledge about the type of the skeleton and the definition of code segments which shall be executed in parallel. Furthermore, a possibility of accessing variables has to be provided. Therefore, various preparations have to be done before execution of a particular skeleton is possible.

Skeletons can only execute functions with a specific prototype description. In general, they are structured as follows[2]:

```
void *function_name(void *args);
```

It corresponds to the data type *tas_runnable_t* which is the type for functions executed by the *TAS*. The name of the function (here: `function_name`) can be defined customly. The other characteristics, namely the type of the return value and the type of the parameter, are fixed. Therefore, all parallelly executed code sections must be encapsulated in functions according to this prototype. In the example we call these functions `par_part_1` for the code marked with the comment `parallel part 1` and `par_part_2` for the other parallel part of the code. The code for function `par_part_2` is shown in the listing below:

```
void par_part_2(void *args) {
  // parallel part 2
  int j;
  for (j=0; j<5; j++) {
    b++;
  }
}
```

These functions are summarized in an array. This is done by holding function pointers in this array. Therefore, each pointer refers to one particular parallel function. Each element of this array is cast to type `tas_runnable_t`. In the example, this array is called `par_code_parts` and its initialization takes place directly at declaration time[3]:

[2] This is similar to the argument of POSIX function *pthread_create*.

[3] On the parMERASA platform, `SHARED_VARIABLE(membase_uncached0) volatile` is the way to declare a common shared variable in an uncached memory region—cache consistency problems are avoided by this.

```
SHARED_VARIABLE(membase_uncached0) volatile tas_runnable_t par_code_parts[] = {
    (tas_runnable_t) par_part_1,
    (tas_runnable_t) par_part_2
};
```

A closer investigation of the code shows that in the parallel sections two variables are accessed by writing (namely `a` and `b`) and one variable is accessed by reading (namely `max`). For this reason, a possibility for accessing variables must be provided to the created functions. This can be done utilizing shared variables or by passing the variables as parameters.

All parameters to pass to a particular skeleton have to be encapsulated in a structure. The exact type of this structure is defined customly for each applied skeleton. In doing so, it is not allowed to define pointers within this structure. The reason for this is the privacy of local variables. They are located in the private stack of a thread. If pointers to such variables are passed to other threads, one thread is able to access (read or write) local data of another thread. This violates the idea of a thread's private stack. Since shared variables are not stored in this stack, these variables can solve this problem.

Shared variables are defined as global variables. When applying parMERASA simulator as target platform, shared variables are declared as follows:

```
SHARED_VARIABLE(membase_uncached0) volatile int var;
```

Here a variable `var` of type integer is declared as shared variable. In contrary, the declaration of this variable is quite simple with POSIX threads:

```
int var;
```

It is to mention that only variables applied for reading access should be passed as parameters, otherwise they should be defined as shared variables. This is because the passed data structures are copies of original variables and, therefore, the parameters are called by value. Since there are no pointers allowed within these structures, it is not possible to call values by reference.

Therefore, the variables `a` and `b` must be defined as shared variables, while `max` can be passed as parameter. The declaration of `a` and `b` is displayed below:

```
SHARED_VARIABLE(membase_uncached0) volatile int a;
SHARED_VARIABLE(membase_uncached0) volatile int b;
```

Simultaneous access to variables by different threads has to be prevented when employing shared variables. If the implementation should be lock-free, this has to take place by avoiding simultaneous write accesses by different threads. In concrete it means, if one thread writes a particular shared variable, all other parallel executed threads are not allowed to access (read or write) this variable.

If locks are available, shared variables need to be locked for each access. With respect to WCET analysis, we propose the usage of synchronization idioms introduced in [4].

A particular data structure is defined for passing parameters to parallel code segments. In our example this structure contains an element `max` and is named `process_args_t`. The following listing shows the described data structure:

```
typedef struct process_args {
  int max;
} process_args_t;
```

A further array has to be initialized to pass the parameters to all parallel functions. This array contains one instance of the defined data structure for each parallel function. Thereby, it must be ensured that the functions in the function pointer array and the corresponding parameter structures in the parameter array can be referenced by the same index. In our example, we call the array for parameters `par_part_arguments`. Furthermore, an array of pointers to type `void` (in the example: `par_args`) is declared which is used for describing the elements of the parameter array. While defining these arrays, their length has to be taken care of. The number of elements in each array is exactly the number of functions executed in parallel by the corresponding skeleton. The implementation of both arrays is displayed in the following listing:

```
SHARED_VARIABLE(membase_uncached0) volatile process_args_t * par_part_arguments[2];
SHARED_VARIABLE(membase_uncached0) volatile void * par_args[2];
```

While executing a skeleton, a pointer referring to the correct parameter structure is passed to each parallelly executed function. Thus, the parameters have to be extracted before applying them. To do this, a new variable (in the example: `my_data`) is declared within the called function. This variable has the same type as the parameter structure (in the example: the type `process_args_t`). It is set to the passed argument which is previously cast to the type of the particular data structure. The cast is needed because of the function prototype's parameter type, which is a pointer to type `void`. Afterwards, the parameters can be extracted by accessing the variable's elements. As example the function `par_part_1` is shown below:

```
void par_part_1(void *args) {
  // extracting parameters
  struct process_args *my_data;
  my_data = (struct process_args *) args;
  int max = my_data->max;

  // parallel part 1
  int i;
  for (i=0; i<max; i++) {
    a++;
  }
}
```

Finally, the content of the skeleton is brought together. Therefore, a data structure of a specific type is applied for representing the entire skeleton. The type differs according to the given skeleton. For instance, `tas_taskparallel_t` is the type for *Task Parallelism*. It is initialized with the number of functions to execute in parallel and the defined arrays of function pointers and parameter structures. In doing so, it is crucial that the number of functions matches the length of these arrays. The declaration and initialization of this data structure are shown here:

```
SHARED_VARIABLE(membase_uncached0) volatile tas_taskparallel_t par_code = {
    par_code_parts, par_args, 2
};
```

3.2.2 Invocation of Skeletons

The preparations explained in Section 3.2.1 have to be performed before it is possible to invoke a particular skeleton. For execution of a skeleton the previously defined data structures need to be set to the correct values and afterwards, the skeleton has to be applied.

Setting the data structures to correct values largely means setting the elements of the parameter structures. In the example this is done as follows:

```
par_part1_arguments[0].max = max;

par_args[0] = &(par_part_arguments);
par_args[1] = NULL;
```

Next, the skeleton is applied by the invocation of three functions. Each function call is responsible for one specific part of the skeleton's execution. A pointer to the structure representing the skeleton is passed to all of those functions. The listing below shows the example of how to invoke the skeleton *tas_taskparallel*.

```
tas_taskparallel_init(&par_code, 2);
tas_taskparallel_execute(&par_code);
tas_taskparallel_finalize(&par_code);
```

Init In this phase, some initialization actions are done and the needed threads are assigned to the skeleton. The number of threads to assign is passed as parameter to the invoked *Init* function.

Execution The skeleton's workload is assigned to its available threads and the execution takes place. If the number of parallel code sections is higher than the number of assigned threads, the workload of the skeleton cannot be executed in one round. In this case, after execution of already assigned workload, further workload is assigned to the threads. Assignment of workloads is applied iteratively until all work is done.

Finalize Here, some actions for completion are executed and the assigned threads are released.

Worker threads can be assigned to a skeleton either by a so-called static or dynamic selection. In the dynamic selection the initialization (phase *Init*) and the finalization (phase *Finalize*) of the skeleton is done directly before and after execution. In contrary, the initialization and finalization takes place before and after the execution of application code for static selection.

In both versions of selection there are advantages and shortcomings. When selecting workers dynamically, it is possible to use the threads not needed at the moment for additional workload like the execution of other functions. For static selection this is not possible. However, less overhead is caused by fixed assignment during execution compared to the dynamic version.

3.2.3 Complete Example

In order to provide a better understanding, we show the running example of the above sections as a continuous implementation. Thereby, we utilize the parMERASA simulator as target platform and apply a dynamic worker selection.

After declaration is done, the invocation of the skeleton can be implemented. As shown in the listing, the sequential parts of the code stay at their position in function `func`. The parallel code is removed because it is now located in the functions for parallel execution. Instead, the initialization of the parameter structures and the invocation of the skeleton is done in `func`. The presented sequential code implemented in a parallel way applying *TAS* is displayed in the listing below:

```
void par_part_1(void *args);
void par_part_2(void *args);

// Variables
typedef struct process_args {
  int max;
} process_args_t;

SHARED_VARIABLE(membase_uncached0) volatile int a;
SHARED_VARIABLE(membase_uncached0) volatile int b;

SHARED_VARIABLE(membase_uncached0) volatile process_args_t * par_part_arguments[2];
SHARED_VARIABLE(membase_uncached0) volatile void * par_args[2];
SHARED_VARIABLE(membase_uncached0) volatile tas_runnable_t par_code_parts[] = {
    (tas_runnable_t) par_part_1,
    (tas_runnable_t) par_part_2
};
SHARED_VARIABLE(membase_uncached0) volatile tas_taskparallel_t par_code = {
    par_code_parts, par_args, 2
};

// Execution
void par_part_1(void *args) {
  // extracting parameters
  struct process_args *my_data;
  my_data = (struct process_args *) args;
  int max = my_data->max;

  // parallel part 1
  int i;
  for (i=0; i<max; i++) {
    a++;
  }
}

void par_part_2(void *args) {
  // parallel part 2
  int j;
  for (j=0; j<5; j++) {
    b++;
  }
}

void func() {
  // sequential part
  a = 1;
  b = 2;
  if (a < b) {
    b = a;
  } else {
    a = b;
  }

  // parallel execution (runs in main thread)
  par_part1_arguments[0].max = max;

  par_args[0] = &(par_part_arguments);
  par_args[1] = NULL;

  tas_taskparallel_init(&par_code, 2);
  tas_taskparallel_execute(&par_code);
  tas_taskparallel_finalize(&par_code);
}
```

4 Analysis Support for OTAWA

To get a WCET estimate of a parallel program a suitable tool is necessary. We focus on the OTAWA toolset[1]. OTAWA applies static analysis for WCET calculation. For this, it needs the source code of the application and also the binary file of the compiled program (see Figure 4.1). In addition, it needs IDs as annotations in source code for parallel software and an XML file describing the relationships between these IDs.[2]

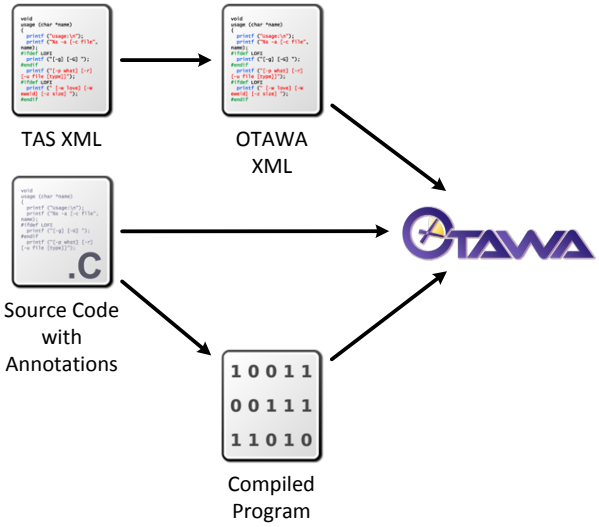

Figure 4.1: Information flow for WCET analysis with OTAWA of code with Timing Analyzable Algorithmic Skeletons (TAS); additional conventional flow facts are not shown here.

Because the notation of this OTAWA XML syntax is complex, we present a simplified XML schema not describing synchronization idioms but TAS skeleton instances. This format is described in detail in the next Section 4.1. In addition, the process of generating the OTAWA XML from it is described (Section 4.2) and tests to check both formats (Section 4.3). Because the notation of this OTAWA XML syntax is complex, we present a simplified XML schema not describing synchronization idioms but TAS skeleton instances. This format is described in detail in the next Section 4.1. In addition, the process of generating the OTAWA XML from it is described (Section 4.2) and tests to check both formats (Section 4.3).

OTAWA can perform the WCET analysis only for a small set of processors. This selection is even more restricted for multi-core processors. Hence, the remainder is mainly focused on the parMERASA architecture, which can be analyzed with OTAWA.

[1] Website: `http://www.otawa.fr/`

[2] In other words, the additional XML file describes “parallel flow-facts”.

4.1 Structure of the TAS XML Format

An overview over the XML format for the *Timing analyzable Algorithmic Skeleton (TAS)* XML can be seen in the following listing:

```
<?xml version="1.0"?>
<program xsi:schemaLocation="http://www.w3schools.com tas.xsd"
xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <threads>
    <thread>
    </thread>
    ...
  </threads>

  <tas_taskparallelisms>
    <tas_taskparallelism>
    </tas_taskparallelism>
    ...
  </tas_taskparallelisms>

  <tas_dataparallelisms>
    <tas_dataparallelism>
    </tas_dataparallelism>
    ...
  </tas_dataparallelisms>

  <tas_pipelineparallelisms>
    <tas_pipelineparallelism>
    </tas_pipelineparallelism>
    ...
  </tas_pipelineparallelisms>

  <csections>
    <csection>
    </csection>
  </csections>
  ...
</program>
```

Each TAS XML file contains a list of all participating *threads*, the *task parallelism*, *data parallel* and *parallel pipeline* skeleton instances that are used in the C program, as well as an additional *csection* tags for critical sections (OTAWA XML syntax). All parts are described in the following in more detail.

4.1.1 Participating Threads

The tag <**threads**> hosts all threads which are participating in the system and are used to process a task or pipeline step. The following listing shows an example of this:

```
  <threads>
    <thread cluster="0" core="0" id="0" routine="main"/>
    <thread cluster="0" core="1" id="1" routine="tas_thread"/>
    <thread cluster="1" core="0" id="2" routine="tas_thread"/>
    <thread cluster="1" core="1" id="3" routine="tas_thread"/>
  </threads>
```

Here, four threads are employed in the program. To enable analyzability, the mapping of the threads to the corresponding cores has to be defined. This is done by the **cluster** and **core**

attributes. Each thread also needs a unique ID that can be referenced later with the **id** attribute. Lastly, the routine of the thread has to be defined. For the main thread this is the name of the function where the first skeleton is executed, i.e., the function chosen as root for the WCET analysis. All other threads that are spawned by the main thread have the routine set to **tas_thread**.

4.1.2 Task Parallelism Instances

A task parallelism skeleton in C consists of two parts. First, the functions that are executed by the threads have to be defined. The following example defines three tasks (`vfahrwerk_part0`, `vfahrwerk_part1`, and `vfahrwerk_part2`):

```
SHARED_VARIABLE(membase_uncached0) volatile void * vfahrwerk_task_parallelism_args[3];
SHARED_VARIABLE(membase_uncached0) volatile
  tas_runnable_t vfahrwerk_task_parallelism_runnables[] = {
    (tas_runnable_t) vfahrwerk_part0,
    (tas_runnable_t) vfahrwerk_part1
    (tas_runnable_t) vfahrwerk_part2
};
SHARED_VARIABLE(membase_uncached0) volatile tas_taskparallel_t
  vfahrwerk_task_parallelism = {
    vfahrwerk_task_parallelism_runnables, vfahrwerk_task_parallelism_args, 3};
```

After defining the runnables, the skeleton is initialized, executed and finalized, like in the following C code fragment:

```
tas_taskparallel_init(&vfahrwerk_task_parallelism, 3);
tas_taskparallel_execute(&vfahrwerk_task_parallelism); // ID=tas_tp_fahrwerk
tas_taskparallel_finalize(&vfahrwerk_task_parallelism);
```

The call to `tas_taskparallel_execute` has to be annotated with a unique ID. The XML description for the skeleton can be defined with a <**tas_taskparallelism**> tag after preparing the C code. The ID set in <`tas_taskparallelism`> **has to be identical** to the annotation in the C code.

```
<tas_taskparallelisms>
  <tas_taskparallelism id="tas_tp_fahrwerk" description="Task Parallelism in Fahrwerk"
    main_as_worker="1">
    <threads>
      <thread ref="0" main="1"/>
      <thread ref="1"/>
      <thread ref="2"/>
    </threads>
    <tasks>
      <task function="vfahrwerk_part0" thread="0"/>
      <task function="vfahrwerk_part1" thread="1"/>
      <task function="vfahrwerk_part2" thread="2"/>
    </tasks>
  </tas_taskparallelism>
</tas_taskparallelisms>
```

In addition to the ID, a free description can be provided by the **description** attribute. This description will be added to the comments in the generated OTAWA XML to improve comprehensibility. Lastly the main thread can be defined to be also a worker for the tasks, indicated by setting the attribute **main_as_worker** to 1[3].

[3]However, the case that the main thread is not a worker is not supported. For running a skeleton with a single thread this is no limitation; in this case no XML code for OTAWA is generated at all.

Beside the attributes, the participating threads and the executed tasks have to be defined. The threads which will be executing the tasks are defined under the <threads> tag. Each thread listed here has to reference a thread defined under the global <threads> tag under <program>. The **ref** attribute of each thread therefore is a reference to the ID of a thread. In addition, the thread that starts all other threads has to be marked as such, setting the **main** attribute to 1.

The <tasks> tag holds a list of all functions that are executed by the task parallelism skeleton. The number of tasks and the number of threads executing the tasks should to match[4]. For each task the C function that is executed by this task is specified by the **function** attribute. In addition, the thread that is executing the task is referenced by the **thread** attribute. Logically, all threads defined for the skeletons under <threads> have to be referenced by the <task> elements.

4.1.3 Data Parallelism Instances

Data Parallelism is used when the same function is called several times with different input parameters. An example is a genetic algorithm, where several individuals (representing different parameter sets) have to be evaluated using the same fitness function. Let's assume a function `init_and_evaluate`, which does exactly that. First, the parameters for the skeleton have to be defined:

```
// TOTAL_CORE_NUM is the number used cores, i.e., number of workers plus one
SHARED_VARIABLE(membase_uncached0) volatile ga_args_t my_ga_args_data[TOTAL_CORE_NUM];
SHARED_VARIABLE(membase_uncached0) volatile void * my_ga_args[TOTAL_CORE_NUM];
SHARED_VARIABLE(membase_uncached0) volatile tas_dataparallel_t dp_ga =
{(tas_runnable_t) init_and_evaluate, my_ga_args, TOTAL_CORE_NUM};
```

Then, the skeleton can be executed. In case of the genetic algorithm, the skeleton is called several times within a loop, where each iteration corresponds to a generation. Since all individuals of a generation are evaluated in parallel by the skeleton, no further loop is required. Otherwise a nested loop would execute a bulk of evaluations with as many iterations as needed to process all individuals of the generation:

```
void demo_5_main_core_0() {
  ...
  tas_dataparallel_init(&dp_ga, TOTAL_CORE_NUM);
  for (iteration = 0; iteration < GA_GENERATIONS; iteration++) {
    ...
    tas_dataparallel_execute(&dp_ga); // ID=genetic_dp_execute
    ...
  }
  tas_dataparallel_finalize(&dp_ga);
  ...
}
```

The XML description for the above function looks then as follows:

```
<tas_dataparallelisms>
  <tas_dataparallelism id="genetic_dp_execute" description="Genetic Algorithm"
  main_as_worker="1" nr_args="8">
```

[4] If not, then (a) multiple round of execution are performed assigning functions to threads in a round-robin manner or (b) some threads do not execute functions.

```
    <threads>
       <thread ref="0" main="1"/>
       <thread ref="1"/>
       <thread ref="2"/>
       <thread ref="3"/>
       <thread ref="4"/>
       <thread ref="5"/>
       <thread ref="6"/>
       <thread ref="7"/>
    </threads>
    <task function="init_and_evaluate"/>
  </tas_dataparallelism>
</tas_dataparallelisms>
```

The tag <**tas_dataparallelism**> contains an **id**, which must exactly match the ID annotation of the source code, a **description**, which is used to add comments in the generated OTAWA XML, and an attribute **main_as_worker** that marks the main thread as worker. In addition, the attribute **nr_args** defines the total number of executions of the singular task, which may be a multiple of the threads used by the skeleton. In the above example, if a generation contains only 8 individuals, then the 8 applied threads are enough to completely process the task, therefore the nr_args attribute is set to 8. If, e.g., 8 threads would still be used to process the individuals, but the total number of individuals per generations are 40, then nr_args would be set to 40.

Similar to the task parallelism skeleton, the <`threads`> tag contains all threads used by the data parallelism skeleton, with one thread marked as **main**, being the initial main thread. In contrast to the task parallelism skeleton, only a single <`task`> tag is defined, which describes the function called by the skeleton using the <`function`> attribute. In the genetic algorithm example, this would be the `init_and_evaluate` function.

4.1.4 Pipeline Parallelism Instances

For a pipeline example, let's assume a number of matrices, that are processed in the following steps:

1. Generate a number of random matrices (create_fft_input)
2. Apply a fast Fourier transform on all matrices (a_to_A)
3. Multiply the matrices with another set of random matrices (AB_to_C)
4. Add the matrices together to a single matrix (C_to_D)
5. Apply an inverse fast Fourier transform on the matrix (D_to_d)

Each step corresponds to a pipeline stage, which can be executed in parallel. For the above example, the parameters for the skeleton would look as follows (`&io` here represents a struct that holds the input and output matrices):

```
SHARED_VARIABLE(membase_uncached0) volatile tas_pipeline_inout_t io =
  {& cache[0], & cache[1]};
SHARED_VARIABLE(membase_uncached0) volatile
  void * task_parallelism_args[STAGES] = {
  (void *) &io, (void *) &io, (void *) &io, (void *) &io, (void *) &io
};
SHARED_VARIABLE(membase_uncached0) volatile
```

```
tas_runnable_t task_parallelism_runnables[STAGES] = {
  (tas_runnable_t) &create_fft_input,
  (tas_runnable_t) &a_to_A, (tas_runnable_t) &AB_to_C,
  (tas_runnable_t) &C_to_D, (tas_runnable_t) &D_to_d
};
```

Then, the call to the pipeline skeleton would look as follows:

```
void demo_fft_init() {
...
tas_taskparallel_init(&task_parallelism, 5);
tas_pipeline_execute(&task_parallelism, ITERATIONS, &io); //ID=tas_pipeline_fft
tas_taskparallel_finalize(&task_parallelism);
...
}
```

To describe the skeleton in XML, a <**tas_pipelineparallelism**> tag is used:

```
<tas_pipelineparallelisms>
    <tas_pipelineparallelism id="tas_pipeline_fft"
    description="Parallel FFT calculation with pipeline"
    main_as_worker="1" iterations="25">
        <threads>
            <thread ref="0" main="1"/>
            <thread ref="1"/>
            <thread ref="2"/>
            <thread ref="3"/>
            <thread ref="4"/>
        </threads>
        <tasks>
            <task function="create_fft_input"/>
            <task function="a_to_A"/>
            <task function="AB_to_C"/>
            <task function="C_to_D"/>
            <task function="D_to_d"/>
        </tasks>
    </tas_pipelineparallelism>
</tas_pipelineparallelisms>
```

As attributes it has an **id**, which needs to be the exact same as the ID annotation in the source code, a **description** used for comments in the generated OTAWA XML, **main_as_worker** to mark the main thread as a worker, and **iterations** to define the number of times the pipeline is to be executed. This number has to be the same as the ITERATIONS constant in the C code above. The skeleton makes sure that the prolog and epilog of the pipeline is handled correctly.

As with the other skeletons, all participating threads are defined under the <`threads`> tag where one thread is marked as **main**. Typically the number of threads is equal to the number of pipeline stages. Under the <`tasks`> tag, all pipeline stages are defined with the <`task`> tag that contains the attribute **function**, which lists the name of the function that executes the pipeline stage.

4.1.5 Common Attributes for all Skeleton Instances

Every skeleton has additional attributes that can be set in the corresponding tags, which are `tas_taskparallelism`, `task_dataparallelism`, and `tas_pipelineparallelism`:

- **executed_after** references the skeleton that came sequentially before the currently described skeleton instance; see Figure 4.2 for an example. If this attribute is omitted it is set to BEGIN.

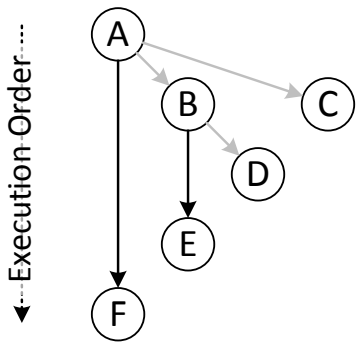

Figure 4.2: Example for execution of skeleton instances after each other (black arrows) or by call (gray arrows) in the execution order.

Example: The main thread executes two Task Parallelism skeletons (A, F) after each other. A would reference BEGIN, while F would reference the ID of A.

- **called_by** references the skeleton that called this skeleton, i.e., the skeleton is nested; see Figure 4.2 for an example.

  Example: A Pipeline Parallelism skeleton B executes a Data Parallelism skeleton D to parallelize one of its pipeline steps and therefore shorten its run time to match its other pipeline steps. D would reference the ID of B.

- **init_id** references the ID of the skeleton initialization statement. The initialization is necessary for binding threads to the skeleton during run-time. If this attribute is present, then the necessary XML code for OTAWA is generated; also the attribute `finalize_id` must be present.

- **finalize_id** references the ID of the skeleton finalization statement. The finalization is necessary for releasing threads from the skeleton after its execution and still during run-time. If this attribute is present, then the necessary XML code for OTAWA is generated; also the attribute `init_id` must be present.

4.1.6 Critical Sections

In addition to the descriptions of the TAS instances, a list of <**csection**> tags can be defined, listed under the <csections> tag. These will be copied to the generated OTAWA XML file. Csections are necessary for critical sections, i.e., threads have to make sure that only one of them executes6 a code part at a point of time.

Single Use of a Lock Variable

As an example consider a function that calculates a random number. In this function a critical section is defined for access on the globally defined seed. In C, an excerpt of the function can look like this:

```
double rando() {
  ...
```

```
  ticket_lock(&lock_random); // ID=rando_lock
  ...
  ticket_unlock(&lock_random); // ID=rando_lock
  ...
}
```

Here, a ticket lock is used to secure thread safety. Both calls to lock and unlock have to be annotated with an ID, in this case the ID is named `rando_lock`. Now a <`csection`> tag for this critical section has to be defined for all threads that can access the critical section.

```
<csections>
  <!-- additional csection for random number generator -->
  <csection id="rando_lock" description="Random Number Generator">
    <thread id="0,1,2,3"/>
  </csection>
</csections>
```

The above example defines such a lock, here for four threads with the IDs 0, 1, 2 and 3. The **description** attribute is optional and is used to set a comment in the generated OTAWA XML. Therefore, the generated XML file for the above example would be:

```

<!-- Random Number Generator -->
<csection id="rando_lock">
  <thread id="0,1,2,3"/>
</csection>
```

Repeated Use of a Lock Variable

If the lock would also be used in another code part (typically to secure accesses to the same data structure), then the lock/unlock pair would be annotated with a new ID:

```
void rando_init() {
  ...
  ticket_lock(&lock_random); // ID=rando_init_lock
  ...
  ticket_unlock(&lock_random); // ID=rando_init_lock
  ...
}
```

The intermediate XML code must then be extended with the second ID:

```
<csections>
  <!-- additional csection for random number generator -->
  <csection id="rando_lock rando_init_lock" description="Random Number Generator">
    <thread id="0,1,2,3"/>
  </csection>
</csections>
```

The <`thread /`> tag should contain only the threads which actually try to acquire the described lock if possible. However, under no circumstances a thread may be missed here.

CSection Tags for TAS

Beside the defined <`csection`> one more <`csection`> is defined in the generated OTAWA XML file, if at least one TAs instance description exists:

```
<!-- additional csection for skeletons -->
<csection id="tas_worker_get_worker_available_lock
  tas_get_workers_available_worker_available_lock
  tas_worker_release_worker_available_lock
  tas_thread_worker_available_lock">
  <thread id="0,1,2,3"/>
</csection>
```

This <csection> is created automatically and appended as last <csection> referencing all threads defined under the global <threads> tag. The listed ID list is generated in one line separated by a whitespace in the XML file but is written here as one per line to enhance readability.

4.2 Running the Converter

The converter translates from TAS XML to OTAWA XML. Its source code is available under the GNU LGPL v3 license:

`https://github.com/parmerasa-uau/tas2otawa/`

The implementation was done with Java in Eclipse. Simple XML[5] is used for parsing and writing XML files. For a translation, we recommend adding a jUnit test extending the class with name `eu.parmerasa.sik.xmlconverter.converter.AbstractTasTest`. The conversion can then be performed with method `runTest(String filename)`. This test also checks both input and output XML files (see Section 4.3).

As alternative, a zip containing the required jar files as well as a shell script for Linux and a batch script for Windows to run the converter can be built with Eclipse. After unpacking the zip, the converter can be called directly by running the scripts (**tas2otawa.bat** for Windows and **tas2otawa.sh** for Linux).

Two parameters are expected by the script, i.e., the names of the files with the input XML and the output XML, e.g.:

```
tas2otawa.sh input.xml output.xml
```

The scripts expect to be called in the directory they are in, because a relative classpath entry is defined within the scripts. If needed, the classpath entry can be changed to absolute paths. If the XML files are not in the local directory, the full path to the files has to be set as well, e.g., `c:\temp\input.xml` or `/home/input.xml`. Both Linux and Windows paths are supported.

4.3 Validating the XML Files

Both the TAS XML and the OTAWA XML files can be validated through provided XML Schema (.xsd) files. These XML files are included in the zip of the converter and can be found in the source code folders: **tas.xsd** for the intermediate XML files and **annotations.xsd** for the OTAWA XML files. To link the TAS XML to the schema file, a reference with correct namespace has to be set in the <program> tag:

[5] Website: `http://simple.sourceforge.net/`

```
<program xsi:schemaLocation="http://www.w3schools.com tas.xsd"
xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  ...
</program>
```

Again, the namespaces and attributes should be written in a single line but have line breaks here to enhance readability. The converter writes similar attributes in the created OTAWA XML file to reference annotations.xsd. In addition, DTD files are available for both XML formats. Also several checks are run in Java just before the conversion is run.

To validate the XML files, a tool of choice can be used. An example is the Eclipse Web Tool Platform (WTP). It adds a *Validate* command to the context menu of XML files.

5 Bibliography

[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer Berlin Heidelberg, 2011.

[2] C. Bradatsch and F. Kluge. parMERASA multi-core RTOS kernel. Technical Report 2013-02, Fakultät für Angewandte Informatik der Universität Augsburg, 2013.

[3] J. J. Durillo and A. J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.

[4] M. Gerdes, R. Jahr, and T. Ungerer. parMERASA pattern catalogue: Timing predictable parallel design patterns. Technical Report 2013-11, Fakultät für Angewandte Informatik der Universität Augsburg, 2013.

[5] R. Jahr, M. Frieb, M. Gerdes, and T. Ungerer. Model-based parallelization and optimization of an industrial control code. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme X, Schloss Dagstuhl, Germany, 2014, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 63–72, Schloss Dagstuhl, April 2014. fortiss GmbH, München.

[6] R. Jahr, M. Gerdes, and T. Ungerer. On efficient and effective model-based parallelization of hard real-time applications. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany, 2013, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 50–59, Schloss Dagstuhl, April 2013. fortiss GmbH, München.

[7] R. Jahr, M. Gerdes, and T. Ungerer. A pattern-supported parallelization approach. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 53–62, New York, NY, USA, 2013. ACM.

[8] R. Jahr, M. Gerdes, T. Ungerer, H. Ozaktas, C. Rochange, and P. G. Zaykov. Effects of structured parallelism by parallel design patterns on embedded hard real-time systems. In *20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014*, August 2014.

[9] A. Nebro, J. Durillo, J. Garcia-Nieto, C. Coello Coello, F. Luna, and E. Alba. SMPSO: A new PSO-based metaheuristic for multi-objective optimization. In *IEEE Symposium on Computational Intelligence in Multi-criteria Decision-making 2009, MCDM'09*, pages 66–73, March 2009.

[10] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. Zaykov, Z. Petrov, B. Boddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Casse,

A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quinones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. parMERASA – multi-core execution of parallelised hard real-time applications supporting analysability. In *2013 Euromicro Conference on Digital System Design (DSD)*, pages 363–370, 2013.

[11] R. Wilhelm, J. Engblom, E. Aandreas, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.