STR2RTS: Refactored StreamIT Benchmarks into Statically Analyzable Parallel Benchmarks for WCET Estimation & Real-Time Scheduling

Benjamin Rouxel[1] **and Isabelle Puaut**[2]

1 **University of Rennes 1, Rennes, France**
`benjamin.rouxel@irisa.fr`
2 **University of Rennes 1, Rennes, France**
`isabelle.puaut@irisa.fr`

Abstract

We all had quite a time to find non-proprietary architecture-independent exploitable parallel benchmarks for Worst-Case Execution Time (WCET) estimation and real-time scheduling. However, there is no consensus on a parallel benchmark suite, when compared to the single-core era and the Mälardalen benchmark suite [11]. This document bridges part of this gap, by presenting a collection of benchmarks with the following good properties: (i) easily analyzable by static WCET estimation tools (written in structured C language, in particular neither *goto* nor dynamic memory allocation, containing flow information such as loop bounds); (ii) independent from any particular run-time system (MPI, OpenMP) or real-time operating system. Each benchmark is composed of the C source code of its tasks, and an XML description describing the structure of the application (tasks and amount of data exchanged between them when applicable). Each benchmark can be integrated in a full end-to-end empirical method validation protocol on multi-core architecture. This proposed collection of benchmarks is derived from the well known StreamIT [21] benchmark suite and will be integrated in the TACleBench suite [10] in a near future. All these benchmarks are available at `https://gitlab.inria.fr/brouxel/STR2RTS`.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems

Keywords and phrases Parallel benchmarks, Tasks scheduling, Worst-Case Execution Time estimation

Digital Object Identifier 10.4230/OASIcs.WCET.2017.1

1 Motivations

In the past, the benchmark suite provided by the Mälardalen institute [11] has been widely accepted by the community studying the Worst-Case Execution Time (WCET) of real-time applications on single-core architectures. While multi-cores tend to replace mono-core architectures, no consensus emerged on a parallel benchmark suite when studying the Worst-Case Response Time (WCRT) of a parallel application or its global WCET. The main unsatisfied requirement of such a benchmark suite lies on the identification of parallel tasks to benefit from the multiplicity of available cores.

Current research papers on real-time multi-core mapping and scheduling already face this issue and already use representative application codes for validation. However, it is a common practice to use proprietary applications from the automotive or avionic world [1, 18], unfortunately preventing other researchers to replay tests or to compare results.

Several non proprietary parallel benchmarks already exist for the experimental validation of real time systems. However, they have some limitations. Some consist of periodical independent task sets [4, 6] only, with no synchronization/communication between tasks. Some others lack information to perform WCET estimation or scheduling, e.g.: source code [8, 9, 20], dependency representation [16, 13, 2, 19, 12], or are hardware or run-time system dependent. Other studies prefer task set generators for validation, but cannot be used for a full end-to-end experimental validation as they lack source code.

This document aims at providing a collection of parallel benchmarks for experimental evaluation of real-time systems on multi-/many-core architectures. The targeted audience is the real-time system research community at large, including researchers on WCET estimation and real-time scheduling. This document can be of benefit to experts in multi-core scheduling to experiment their techniques for task mapping and scheduling. It can be of benefit to researchers on worst-case execution time estimation, both on single-core architectures, by analyzing each task of the parallel application, and on multi-core architectures through an analysis of the entire parallel application, including for instance analyses of contentions at the shared resources such as bus, cache, etc.

To ease the creation of a collection of benchmarks with all the required information for WCET estimation and scheduling, we started from the StreamIT benchmark suite [21], which consists of a set of Digital Signal Processing (DSP) applications. Such applications consume incoming data and produce outgoing data at a specific rate, which is representative of many real-time applications.

The provided information for each application is an XML file and a C source file. The XML file describes the structure of the application through a directed acyclic graph (identification of tasks and dependencies between them, volume of data to be transmitted between tasks, WCET of tasks on a particular architecture if the benchmark is to be used for real-time scheduling only). The C code contains the source code of each task. The source code is statically analyzable and self-contained, to allow static WCET estimation techniques on any specific architecture, (but obviously other estimation techniques such as probabilistic or measurement-based are not left aside). In particular, the C code contains pragmas expressing loop bounds in the format used in the TACleBench benchmark suite [10]. We plan to integrate them in the TACleBench benchmark suite as it aims to be the reference benchmark suite for WCET estimation at code level for both single-core and multi-/many-core architecture.

The rest of this document is organized as follows. First, Section 2 compares our work with existing benchmark suites. Section 3 presents background knowledge about the StreamIT benchmark suite, which is used as the basis for STR2RTS. Section 4 provides an overview of the provided material, and finally Section 5 gives some qualitative and quantitative information on the provided benchmarks, before concluding in Section 6.

2 Related Work

The usefulness of benchmarks for the validation of systems no longer has to be demonstrated. They have been vastly used in the past to experiment new algorithms, new software, or new pieces of hardware. In computer science, there exists hundreds of different benchmark suites with different purposes and different sizes: SPEC CPU 2006, PolyBench [17], ParMiBench [12], UTDSP [13], Parsec [2], JemBench [19], ParaSuite [16], and many more. However very few of them have been engineered for *multi-core real-time* systems. This kind of systems requires more information in the benchmark suite than just the code, typical input data and a description, which is the general provided material. Indeed, to be largely accepted by the real-time community a benchmark suite must include a source code that is statically analyzable, to allow experiments with both static and non-static WCET estimation methods.

A non-exhaustive list of the requirements for benchmarks targeting real-time embedded systems would include (i) structured self-contained source code – i.e.: no goto, no dynamic memory allocation, no call to external libraries, (ii) statically computable loop bounds or flow facts for loop bounds, (iii) deadlines and periods of tasks. Adding the multi-core constraints to the system would also add new requirements to the benchmark suite, such as the amount of data exchanged between communicating tasks, and a representation of dependencies between tasks if applicable. The benchmark suite should also remain independent from any specific run-time environment (i.e.: OpenMP, MPI, etc.) to be used as easily as possible.

Starting from the single-core era, the Mälardalen benchmark suite [11] has been accepted by the WCET estimation community. It consists of small pieces of key code representing some well-known code structures found in embedded real-time software. Although representative of embedded software, this benchmark suite contains sequential codes only, and the large majority of provided codes are very small.

A common practice to evaluate scheduling strategies is to use task graph generators. They have the benefit to be architecture independent and generate a vast amount of different topologies. Task Graph For Free (TGFF) [9], Synchronous Dataflow 3 (SDF3) [20] can generate task graphs with dependent tasks in a deterministic way, allowing anybody to replay an experiment as long as the configuration parameters are known. UUniFast [4] is an algorithm generating task sets with uniform distribution in a given space. Task graph generators are very useful when the goal is to empirically validate a method on a large variety of task graph topologies. However we all need *concrete* representative applications with code for further empirical validation which is what we aim at providing here.

Three real parallel applications targeting real-time systems are often used as benchmarks – i.e.: Debie1 [7], Papabench [14] and Rosace [15]. All are control applications respectively for a satellite, a drone and a plane. But three concrete applications are not enough. Our objective with the benchmark suite we provide is to enrich the set of applications that can be used to validate multi-core real-time systems, and enlarge the scope of applications to include signal processing applications with dependencies between tasks.

De Bock *et al.* [6] proposed a benchmark generator targeting multi-core platforms. The generator input is sequential code for each task. All tasks are independent. The benchmark generator output is a task-set fitting some requirements. In comparison, the benchmarks we provide include *dependent* tasks and a representation of these dependencies as well as the amount of data exchanged between dependent tasks.

To the best of our knowledge the benchmark suite closest to this work is the StreamIT benchmark suite [21] that we use as a baseline. In the original version of StreamIT, the authors provide a representation of task's dependencies – a task graph – with communication (exchanged tokens) and source code. However, the provided C source code is only sequential, and the generated code is not WCET-friendly as some benchmarks are impossible to statically analyze, i.e. statically extracting loop bounds might not be possible with available tools. In addition, there is nearly no cache reuse, since tasks performing the same function are systematically duplicated in the generated code. Moreover, dynamic memory allocation is used for allocating messages used for inter-task communication. The C version we provide respects the task graph extracted from the original StreamIT tool, with the benefit of allowing static analyses on each function in isolation.

Finally the new TACleBench suite [10] aims at becoming the *de facto* standard benchmark suite for timing analysis. This work will be integrated in TACleBench in order to strengthen the multi-/many-core dimension of this suite.

```
void->void pipeline FFT4 {
  add OneSource(); add FFTKernel(2);
  add FloatPrinter();
}
float->float pipeline FFTKernel (int N) {
  for (int i=1; i<N; i*=2) {
    add Butterfly(i, N);
  }
}
float->float pipeline Butterfly (int N, int
      W){
  add splitjoin {
    split roundrobin(N, N);
    add Identity<float>(); add Multiply();
    join roundrobin();
  };
  add splitjoin {
    split duplicate;
    add Add(); add Subtract();
    join roundrobin(N, N);
  };
}
void->float filter OneSource {...}
float->float filter Multiply {...}
float->float filter Add {
  init {}
  work push 1 pop 2 {
    push(pop() + pop());
  }
}
float->float filter Subtract {...}
float->void filter FloatPrinter {...}
```

Listing 1 FFT4 stream program's structure.

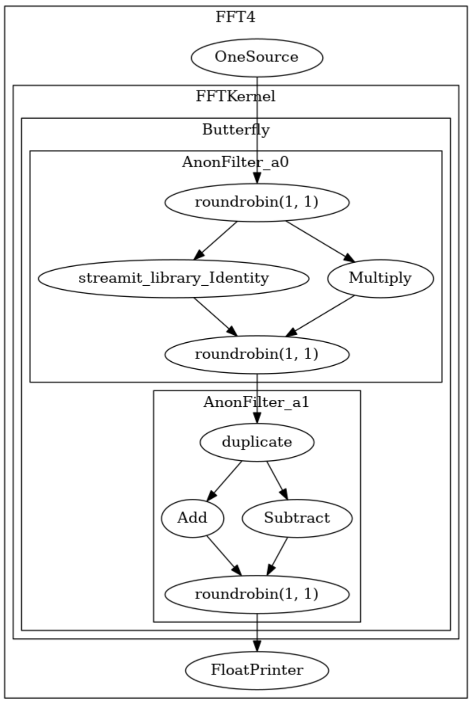

Figure 1 FFT4 SDF graph.

3 Background on StreamIT

StreamIT [21] is a high-level language for developing *streaming* applications (applications acting on flows of data) modeled as Synchronous Dataflow Graphs (SDF). The StreamIT language has a portable run-time environment and is architecture-independent. The main difference of StreamIT as compared with other streaming languages lies on a required well-defined structure on the streams, that are not an arbitrary network of nodes. One of the major properties of the StreamIT benchmarks lies on the data rate which is imposed to be fixed, thus known at compile time.

All graphs in the StreamIT languages consist of a hierarchical composition of nodes structured in *pipeline*, *split-join* and *feedbackloop* constructs. Streaming applications can then be represented as a Cyclo Static DataFlow graph (CSDF) [3]. They are constructed over the execution of two phases : the *initialization* and the *steady state*, where the latter is considered indefinitely repeating, whereas the former is performed only once and aims at registering the tasks of the steady-state in the StreamIT scheduler.

A streaming application can be seen as a flow of computational units producing and consuming data, the data stream. The basic computational unit of StreamIT applications is the *filter*. Each filter is a task that produces and consumes tokens. Communicating filters are organized in a stream in order to create a *pipeline* (chain) of *filters*. More complex stream structures can be realized with *split-join* and *feedbackloop* constructs. The former splits the data stream in parallel streams before joining them again, whereas the latter re-injects upstream data produced downstream. Conditional control-flow is not allowed at the application level (there is no concept of conditional execution of filters). In contrast, there may be control flow inside filter code. The data stream is propagated through the

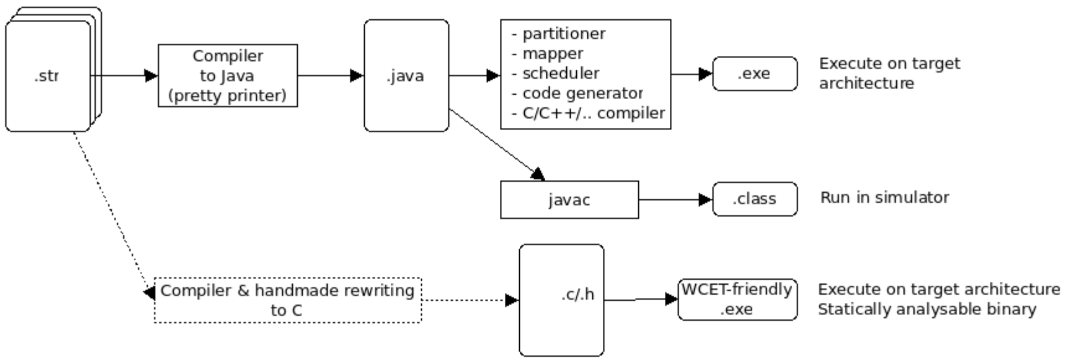
.str
Compiler to Java (pretty printer)
.java
- partitioner
- mapper
- scheduler
- code generator
- C/C++/.. compiler
.exe
Execute on target architecture
javac
.class
Run in simulator
Compiler & handmade rewriting to C
.c/.h
WCET-friendly .exe
Execute on target architecture
Statically analysable binary

Figure 2 StreamIT Tool-chain.

filters in the graph at a constant rate known, at compile time. This allows to statically know the amount of data exchanged between filters. Such data are transmitted through dataflow channels implemented as FIFO (First In First Out) queues.

The StreamIT language is illustrated on one of the smallest application from the StreamIT benchmark suite: the radix-2 case of a Fast Fournier Transform (FFT4.str). The application source code in StreamIT language is presented in Listing 1. Lines 1–21 specify the structure of the streaming applications, while lines 22–31 give the source code of the filters (for conciseness only the StreamIT code for filter *Add* is given).

The first element (line 1: FFT4) is the top-level envelope (equivalent to the *main* function in C code); it registers three other elements which are added to the global structure (a *pipeline*, i.e. chain of elements for FFT4). Elements are added directly or recursively explored depending of their type. For instance, *OneSource*, line 22 is added directly because it is a simple filter, whereas *Butterfly*, line 10 is explored because it is a composition of elements (here, a pipeline). The code of a very simple filter (*Add*) is given in lines 24–29. It is decomposed into two functions : the *initialization* part (line 25) and the *work* function for the steady-state (lines 26–28). Due to the simplistic nature of this example the initialization part is empty, but one could easily imagine some constant initialization for the steady-state. The *work* function corresponds to the C-like code that will be executed at each iteration of the steady-state. This function calls, at line 27, two functions *pop*/*push* to respectively fetch and store data from/to the FIFO channels connected to the previous/next dependent tasks.

The program structure extracted from the StreamIT application from Listing 1 is presented in Figure 1 and it illustrates the steady-state of the application.

The StreamIT benchmark suite comes with an end-to-end compilation tool chain illustrated in Figure 2. It first parses the StreamIT language and generates a Java version of the streaming application. This Java version is then converted to an intermediate representation used by internal tools to analyze the application. Following is a short summary of those tools:

- **Partitioning:** determining the number of fissions and fusions, used to determine where to insert/remove split-join nodes in the generated code;
- **Mapping:** determining on which core each job implementing a filter will run;
- **Scheduling:** determining in which order jobs will be executed;
- **Code generation:** generating code for the targeted architecture (generally C/C++) through the provision of several back-ends.

The last step generates a code which can be compiled in order to run the application on the targeted architecture (RAW processor, Tilera, RStream and so on). The Java version can

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<appl>
  <tasks>
  <task id="fft4_split2_duplicate" WCET="2286">
    <prev id="fft4_join1_round_robin" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_join2_weighted_round_robin" WCET="1380">
    <prev id="fft4_add" data-sent="1" data-type="float" />
    <prev id="fft4_subtract" data-sent="1" data-type="float" />
  </task>
  <task id="fft4_add" WCET="1600">
    <prev id="fft4_split2_duplicate" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_subtract" WCET="1600">
    <prev id="fft4_split2_duplicate" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_float_printer" WCET="1614">
    <prev id="fft4_join2_weighted_round_robin" data-sent="2" data-type...
  </task>
  <task id="fft4_one_source" WCET="1198"></task>
  <task id="fft4_split1_weighted_round_robin" WCET="1380">
    <prev id="fft4_one_source" data-sent="2" data-type="float" />
  </task>
  <task id="fft4_join1_round_robin" WCET="1198">
    <prev id="fft4_identity" data-sent="1" data-type="float" />
    <prev id="fft4_multiply" data-sent="1" data-type="float" />
  </task>
  <task id="fft4_identity" WCET="1054">
    <prev id="fft4_split1_weighted_round_robin" data-sent="1" data-type...
  </task>
  <task id="fft4_multiply" WCET="1070">
    <prev id="fft4_split1_weighted_round_robin" data-sent="1" data-type...
  </task>
</tasks>
</appl>
```

Listing 2 XML representation of the FFT4 application.

also be executed using a simulation library included in the StreamIT project. This simulator runs a sequential version of the streaming application.

Despite the work done on the StreamIT toolchain, none of the provided back-ends generate code ready to be analyzed in the context of real-time systems. The *simpleC* back-end generates only one big main function containing all the code, leading to a sequential version not suitable for multi-core analysis/execution. The *newSimple* back-end generates hard to read and to analyze source code, where it is not possible anymore to identify tasks. The *cluster* back-end generates code that may not be analyzable by loop bound extractors, and includes libraries provided by StreamIT with C++ classes and dynamic memory allocation, thus not suitable for static WCET analysis. In addition, when the same filter is used several times, its code is duplicated, thus degrading the WCET of tasks when considering architectures with caches.

As no back-end fulfills all the requirements implied by real-time systems and corresponding analyses, we modified the StreamIT benchmarks code, as detailed in Section 4.2, to fit the needs of the real-time system community. Among the tools coming with the StreamIT tool chain, we only used the simulation library to ensure that our modifications to the StreamIT codes are functionally equivalent to the original code.

4 Benchmarks overview

This section presents an example of the provided information: (i) an XML description, (ii) a C source code ; and how to use it. Then, it presents how this information was extracted from the StreamIT benchmark suite [21].

4.1 Provided information

Each benchmark is divided in 4 files, an XML file, a DOT file, a C source file and its corresponding header file. The DOT file is a graphical representation of the tasks and their dependencies using the *graphviz* software[1], and will not be presented here, as well as the header file. Following is an example of an XML description with its corresponding C source code.

An XML file summarizing all the provided information is presented by Listing 2 and corresponds to our previous example from Figure 1. This file basically describes the structure of the application as a Directed Acyclic Graph (DAG), with tasks as nodes and channels as edges. It can be used by mapping/scheduling tools as input to experimentally evaluate new mapping/scheduling strategies involving either a single application or multiple applications both modeled as DAGs. Another usage, once tasks have been assigned to cores in a multi-core platform, is to use the XML file together with the code of tasks to perform WCET estimation on the application, in particular integrating contentions to access shared resources in the WCET of the application. For each task, the XML file contains the set of predecessors of the task with the amount of data received by each of them, as well as the task WCET. The XML tag *prev* represents task's dependencies as precedences, e.g. line 15 where *Split2DUPLICATE* is a predecessor of *Subtract*. The associated attribute *data-sent* specifies the amount of data needed for one execution of the task, and the attribute *data-type* specified the type of data (e.g. : int, double, float, etc.).

The attribute *WCET* is provided as information for people aiming at performing experiments on mapping/scheduling techniques and do not wishing to perform an initial WCET analysis step. Provided WCETs were estimated by our static WCET analysis tool Heptane [5] for a the MIPS instruction set, for an architecture without caches or pipelines (roughly the provided WCET corresponds to the worst-case number of instructions executed by each task).

Listing 3 introduces the structure of the provided C source code. Each task from the aforementioned graph appears as a C function in the source file. The code of filter/task *Add* is given as an example in lines 8–14. Depending on the value of $GLOBAL\_N$ (constant evaluated by the C pre-processor), this filter reads two float items from the input channel (pop_float), then sums them before writing the result into the output channel ($push\_float$). The loop is annotated with a *pragma* specifying the loop bound, according to the TACleBench syntax for flow-facts annotations. The value of $GLOBAL\_N$ has an impact on the number of added tasks (number of added *Butterfly* from the Listing 1). In the C source code, we fix such parameter in the header file to have the C source code consistent with the XML description. In this example the value of $GLOBAL\_N$ is set to 2.

Lines 22–41 point to the *sequential_main* function that corresponds to an execution of all tasks on a single-core architecture. Function *sequential_main* first calls the initialization function of all tasks having a non-empty initialization phase (line 23). This initialization step sets up every C structure, buffers or pre-computed data required by filters for the steady-state run. The *sequential_main* then calls each filter function in a loop of $MAX\_ITERATION$ iterations (lines 28–39) for the steady-state execution. Functions are called in an order that respects dependencies between tasks. This function is provided for users interested in single-core WCET estimation. It was also used to check the correctness of code modifications applied to the StreamIT benchmark, by comparing the results to those produced by the StreamIT Java simulator.

[1] `http://www.graphviz.org/`

```
#include "FFT4.h"
// GLOBAL_N is defined in the header file and its value is 2

void fft4_one_source() { ... }
void fft4_identity() { ... }
void fft4_multiply() { ... }
void fft4_add() {
_Pragma("loopbound min "GLOBAL_N/2" max "GLOBAL_N/2)
  for(int i=0 ; i < GLOBAL_N/2 ; i++) {
    float v1 = pop_float(&AddBuf.buffer_in);
    float v2 = pop_float(&AddBuf.buffer_in);
    push_float(&AddBuf.buffer_out, v1+v2);
  }
}
void fft4_subtract() { ... }
void fft4_float_printer() { ... }
void fft4_init() { ... }
void fft4_split1_weighted_round_robin( uint32_t nb ) { ... }
void fft4_join1_round_robin() { ... }
void fft4_split2_duplicate() { ... }
void fft4_join2_weighted_round_robin( uint32_t nb ) { ... }
int sequential_main( int argv, char **argc ) {
  fft4_init();
_Pragma("loopbound min "MAX_ITERATION" max "MAX_ITERATION)
  for( int i=0 ; i < MAX_ITERATION ; i++ ) {
    fft4_OneSource();
_Pragma("loopbound min "(GLOBAL_N/2-1)" max "(GLOBAL_N/2-1))
    for( int j = 1 ; j < GLOBAL_N ; j *= 2 ) {
      fft4_split1_weighted_round_robinv(j);
        fft4_identity();
        fft4_multiply();
      fft4_join1_round_robin();
      fft4_split2_duplicate();
        fft4_add();
        fft4_subtract();
      fft4_join2_weighted_round_robin(j);
    }
    fft4_float_printer();
  }
  return EXIT_SUCCESS;
}
```

Listing 3 C version of the FFT4 stream program.

Regarding communications between tasks, a C file implementing the push/pop communication functions has to be provided and linked with the code of each application. Since the implementation of communications is architecture and system dependent, this file has to be provided for every (architecture, system) pair. As a start point we provide a simple implementation of push/pop operations that implement communications through shared memory, using statically allocated FIFO buffers. This simple implementation can be used on single-core architectures and multi-core architectures with shared memory.

4.2 Benchmark construction process

In order to extract the above information for each benchmark, we relied on the StreamIT compilation tools as much as possible and we then adapted their output to fit our needs. As presented by the dashed line in Figure 2, we modified the Java pretty-printer to generate a preliminary C version of the streaming application that later needs to be modified manually to match the analysis requirements. When finalizing the C source code through handmade modifications, we stayed independent from any specific run-time library and inter-core communication mechanism. Despite the error proneness of this method, this hand-made step is necessary to guaranty easy read/analyze/understand code with all required annotations. To validate the functional correctness of the final C source version, we performed non-regression tests considering the Java simulator output as the baseline.

Table 1 Description of provided benchmarks.

Name	#tasks	#split-join	Description
802.11a data rate 6/9/12/18/24/36	[119;132]	[17;18]	802.11a wireless LAN protocol transmitter with different configurations
Audiobeam	20	1	Real-time beam-forming on a microphone input array
Beamformer	56	2	Application to perform beam-forming on a set of inputs
CFAR	4	0	Constant False Alarm Rate detection
Complex-FIR	3	0	FIR filter with complex data types
DCT2	40	2	Discrete Cosine Transforms from Asplos'06 paper super-set
DES	423	80	DES encryption algorithm
FFT2	26	1	Fast Fourier Transform, blocked, coarse-grained version
FFT4	42	10	Fast Fourier Transform, more fine-grained
FilterBankNew	52	1	Creates a filter bank to perform multi-rate signal processing
FMRadio	43	7	FM radio with multi-band equalizer

To create the XML description, we needed the WCET of each task, the amount of data exchanged between task and the topology of the application's graph. For the first information, we relied on our tool Heptane [5] that gives us the WCET of each task in isolation. The amount of data exchanged and the topology of the graph are extracted manually from files generated by the Java simulator.

5 Provided benchmarks

Table 1 summarizes the benchmarks that are ready to use at the time of writing. The first column presents the name of the benchmark (identical to the name in the original StreamIT benchmark suite), followed by the number of tasks, the number of *split-join* nodes and a quick description (also extracted from the original StreamIT benchmark suite). For application 802.11a coming in multiple versions (to be explained later), we provide the minimum and maximum of provided values among all versions.

Table 2 shows the complexity of each benchmark. After the name of the benchmark, the second column shows the width of the graph (the maximum number of tasks at the same topological rank) which gives an idea of the amount of concurrency in the application. Following are information about task's WCET and amount of data exchanged between tasks. Both fields are described with an average and standard deviation.

Table 3 indicates which benchmarks need a *mathematic* library to compile, and use input and/or output file. Nonetheless to ensure self-containment, we provide a dummy implementation (empty shell) for the needed functions.

We found some benchmarks with multiple usages of the same task with different input parameters at different points in the application. We thus generated two versions of each benchmark: one with shared code to allow cache reuse, and another one with duplicated code. The difference between both versions lies on the ability to exploit cache reuse or not, and also accuracy of flow-facts annotations (which are more precise with duplicated code).

Table 2 Statistics for provided benchmarks.

Name	Width	WCET (cycles)	Data (tokens)	#Basic blocks (avg #instr.)	#Cond. br.	#mem. instr.
		<avg, standard deviation>				
802.11a	[7;18]	2.39e5, 6.35e5	596, 2238	1584 (6)	222	3519
6/9/12/18/24/36						
Audiobeam	15	273, 1094	3, 5	386 (7)	72	893
Beamformer	12	1.25e4, 9.65e4	4.6, 10	459 (10)	87	1188
CFAR	1	1.58e4, 1.55e4	288, 425	375 (6)	70	821
Complex-FIR	1	501, 655	1.3, 0	336 (6)	60	804
DCT2	16	1.69e4, 3958	57.6, 91	437 (6)	86	894
DES	8	2045, 1417	35.7, 29	621 (6)	111	1079
FFT2	2	2.94e5, 3.03e5	137.8, 49	477 (6)	81	1003
FFT4	2	1337, 450	23.6, 8	566 (5)	85	860
FilterBankNew	6	6315, 8306	8.9, 11	446 (6)	84	913
FMRadio	12	1632, 2234	1.6, 1	486 (6)	91	1037

Table 3 Properties of provided benchmarks.

Name	Use math library	Use I/O file	Two versions / code reuse
802.11a	yes	no	yes
Audiobeam	yes	yes	no
Beamformer	yes	no	no
CFAR	yes	no	no
Complex-FIR	no	yes	no
DCT2	yes	yes	no
DES	no	no	yes
FFT2	yes	no	no
FFT4	no	no	no
FilterBankNew	no	no	no
FMRadio	yes	no	no

The last column of Table 3 indicates whether we created multiple versions of the benchmark with code reuse or not.

Finally, some benchmarks are customizable by modifying the value of some parameters inside the StreamIT source code, e.g. the data rate of the *802.11a* application. As modifying such values has an impact on the application's structure, we generated multiple versions of the same benchmark for the different configurations.

We successfully compiled the list of benchmarks presented in Table 1 for x86_64 architecture and validated their behavior by comparing their results with the one from the Java simulator provided by StreamIT. All these benchmarks are available at

`https://gitlab.inria.fr/brouxel/STR2RTS`.

6 Conclusion

This document has presented a collection of benchmarks written in analyzable C language and based on the StreamIT benchmarks suite [21]. The purpose of the refactoring of StreamIT applications we have performed is to create self-contained analyzable architecture-independent parallel C applications to allow any kind of experiments on WCET analysis and real-time scheduling on multi-core architectures. To largely spread our work, we will integrate this collection of benchmarks into the TACLeBench project.

Due to the required handmade refactoring, we will be continuously adding new test cases over time while there are still some StreamIT applications to refactor. We foresee the end of refactoring in a couple of year for the 60% remaining benchmarks.

References

1 Matthias Becker, Dakshina Dasari, Borislav Nikolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *ECRTS*, 2016.

2 Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

3 Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258. IEEE, 1995.

4 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

5 Hardy Damien, Rouxel Benjamin, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, 2017. `doi:10.4230/OASIcs.WCET.2017.8`.

6 Yorick De Bock, Sebastian Altmeyer, Jan Broeckhove, and Peter Hellinckx. Task-set generator for schedulability analysis using the taclebench benchmark suite. In *Proceedings of the Embedded Operating Systems Workshop : EWiLi 2016*, pages 1–6. CEUR Workshop proceedings, October 2016.

7 Debie1. URL: `https://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:debie1`.

8 Robert Dick. Embedded system synthesis benchmarks suite (E3S), 2010. URL: `http://ziyang.eecs.umich.edu/~dickrp/e3s/`.

9 Robert P. Dick, David L. Rhodes, and Wayne Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

10 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET'16)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/OASIcs.WCET.2016.2`.

11 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Bj"orn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Brussels, Belgium, July 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/OASIcs.WCET.2010.136`.

12 Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. Parmibench-an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters*, 9(2):45–48, 2010.

13 C. G. Lee. UTDSP Benchmark Suite, July 2011. URL: `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`.

14 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006. `doi:10.4230/OASIcs.WCET.2006.678`.

15 Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROS-ACE Case Study: From Simulink Specification to Multi/Many-Core Execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 309–318, 2014.

16 Parasuite. URL: `http://parasuite.inria.fr/`.

17 Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite, 2012. URL: `http://www.cs.ucla.edu/pouchet/software/polybench`.

18 Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. Off-line mapping of multi-rate dependent task sets to many-core platforms. *Real-Time Systems*, 51(5):526–565, 2015.

19 Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded java benchmark suite jembench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES'10, pages 120–127, New York, NY, USA, 2010. ACM. `doi:10.1145/1850771.1850789`.

20 S. Stuijk, M. C. W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006. URL: `http://www.es.ele.tue.nl/sdf3`, `doi:10.1109/ACSD.2006.23`.

21 William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.