DRT: A Lightweight Runtime for Developing Benchmarks for a Dataflow Execution Model

Roberto Giorgi$^{1[0000-0003-0384-8229]}$ and Marco Procaccini$^{1[0000-0002-9719-2672]}$, and Amin Sahebi$^{2,1[0000-0002-6929-4671]}$

[1] University of Siena, Italy
[2] University of Florence, Italy
giorgi@dii.unisi.it
procaccini@diism.unisi.it
amin.sahebi@unifi.it

Abstract. Future computers may take advantage of a dataflow program execution model (PXM) for both performance and energy advantages. One key element to provide a compilation tool-chain for such machines is a framework for developing initial benchmarks. DRT (Dataflow Run-Time) is a tool that enables the fast prototyping of those benchmarks for the Dataflow Threads (DF-Threads) PXM. In this work, we show how to use DRT to develop dataflow based examples to be targeted by a future compiler for the dataflow PXM.
DRT has been written in portable C code (tested with the GNU C compiler), and it is open-source, therefore, it can be used on real machines based on architectures like x86, AArch, RISC-V ISA.
Here, we discuss some didactic examples, and we show how to study and debug the data exchange, which is flowing through frames that are detached from the data stack. We compare DRT against similar dataflow runtime libraries such as DARTS and OCR. Even though our environment is not yet optimized, we found that DRT outperforms the above runtime frameworks in terms of execution time. We also give an evaluation of the time and complexity to develop DF-Threads examples in DRT compared to the approach of using a full system simulator and FPGAs for more accurate modeling.

Keywords: Dataflow threads, Low-level API, Execution Model

1 Introduction and Motivation

Dataflow architectures and their program execution models (PXMs) have been studied since the '70s [8,7,3,4,34,38]. One of the most well-known features of dataflow execution models is that they can achieve a high level of parallelism, which leads to better power consumption and better hardware efficiency [44,30]. Dataflow architectures can significantly exploit the implicit parallelism of the applications and overcome synchronization and consistency overheads generated by von Neumann machines [20,21]. Since then, some researchers have shown the possibility of supporting a dataflow execution model for parallel threads on conventional machines [23,37,19,45]. In this work, we consider the DF-Threads execution model [19]. Other works have shown the potential of dataflow models in terms of power efficiency [6,22] and also as accelerators for High-Performance Computing and machine learning applications [28,29,9]. As can be seen in Fig. 1, the DF-Threads execution model can exhibit an important speedup compared to OpenMPI when running the Matrix Multiplication benchmark over a cluster. More details about the scalability and efficiency of DF-Threads are described in [18].

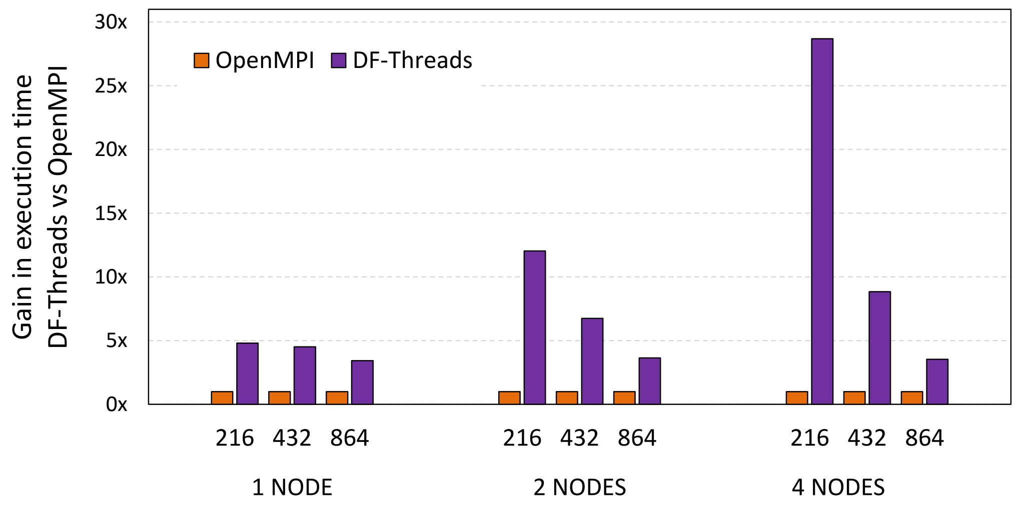


Fig. 1: Gain in execution time of DF-Threads compared to OpenMPI. The benchmark is Matrix Multiplication for different sizes of the matrices (216, 432, 864 - the block size is 8 elements) and 1, 2, 4 nodes. Data is derived from [14].

Even though dataflow models have shown many great features, conventional programming languages do not support them very well [43]. This limitation, together with the possible high performance gains, motivated us to introduce a tool, which could reduce the gap between conventional languages and dataflow execution models.

In this paper, we present the Dataflow Run-Time tool (DRT) to quickly develop and test the execution of dataflow codes based on DF-Threads API. Our contributions in this paper are:

- Introducing a dataflow runtime (DRT), which is presented first in this paper.
- Illustrating how the DRT tool can be used for debugging and studying the movement of data frames (a feature that is not available in standard debuggers).
- Comparing the execution time speedup of DRT against similar dataflow runtime.

The rest of this paper is structured as follows: in Section 2, we describe the background. Then in Section 3, we introduce the DRT tool, illustrate how to carry out an experiment, write and debug dataflow codes with DRT. Then in Section 4, we show the DRT runtime evaluation, and in Section 5, we present related works. Finally, in Section 6, we conclude and briefly introduce future works.

2 Background

2.1 DF-Threads

In order to demonstrate the dataflow execution model, we use DF-Threads as described in [12]. In Fig. 2, we show a simplified high-level overview of DF-Threads execution (right) and a classical (von Neumann style) execution (left). In the classical execution, the parallel threads can read/write from/to any location of the memory. Therefore, a high synchronization and coherency overhead may be generated. As mentioned in detail in [12], each of these DF-Threads has a different behavior according to the memory access pattern. Consequently, it may need different execution and hardware support. It is worth recalling that using standard libraries like *Pthreads* is not required. Here, we briefly recall the specification of the DF-Threads API:

- DF-Threads follow the dataflow semantics: a thread is ready when its input is fully available; it starts executing when the scheduler decides to assign it to a physical resource (e.g., a core).
- The management of a DF-Thread lifetime happens through the following functions, which are described in Table 1: ***df_schedule***, ***df_ldframe***, ***df_write***, ***df_destroy***.
- DF-Threads are isolated in terms of memory accesses, and their execution can be repeated in the case of faults since their inputs are retained [40].

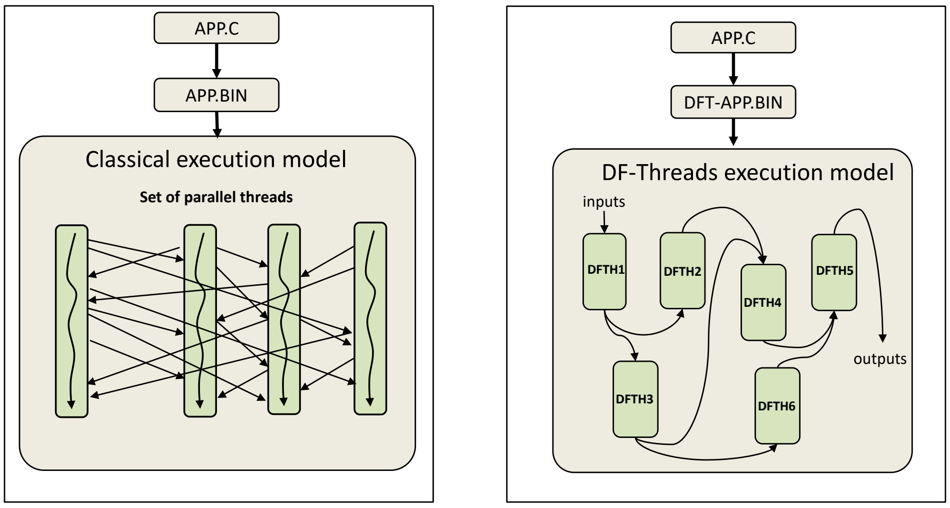


Fig. 2: Simplified representation of the DF-Threads execution model. On the left, we represent the irregular read and write of generic threads. On the right, the exchange of data among threads happens in a more regular fashion [22].

Table 1: DF-Threads function definitions [16]

DF-Threads API function	Description
uint64_t df_schedule(void* ip, uint64_t sc)	Create the DF-Thread and its associated frame; *sc* is the synchronization count, which represents the number of inputs that the DF-Thread will receive.
uint64_t df_ldframe()	Retrieve the frame pointer associated with the current DF-Thread.
uint64_t df_write(void* fp, uint64_t val)	The value *val* will be stored in a location pointed by *fp*, and for each write, the *sc* (which is specified by scheduler before) will be decremented.
uint64_t df_destroy()	Terminate the current DF-Thread and deallocate its input frame.

2.2 Writing dataflow codes with the DF-Threads API

This Section shows the workflow to map the desired application into a dataflow code (here DF-Threads). While this translation could be done by a compiler, we do not have such a compiler at the moment (the compiler could be future work).

We use fine-grain algorithms to show the potentiality of our tool in mapping several DF-Threads on real architectures. We choose the Recursive Fibonacci (RFIB) as a "simple yet complex enough" example to illustrate the development methodology for DF-Threads programs. The RFIB algorithm is a well-known example used to create many threads and stress the runtime and the scheduling management.

In Fig. 3, we describe the original C code and its mapping into DF-Threads, together with the dynamic behavior of the dataflow code. In this case, two DF-Threads are created: RFIB and "adder".

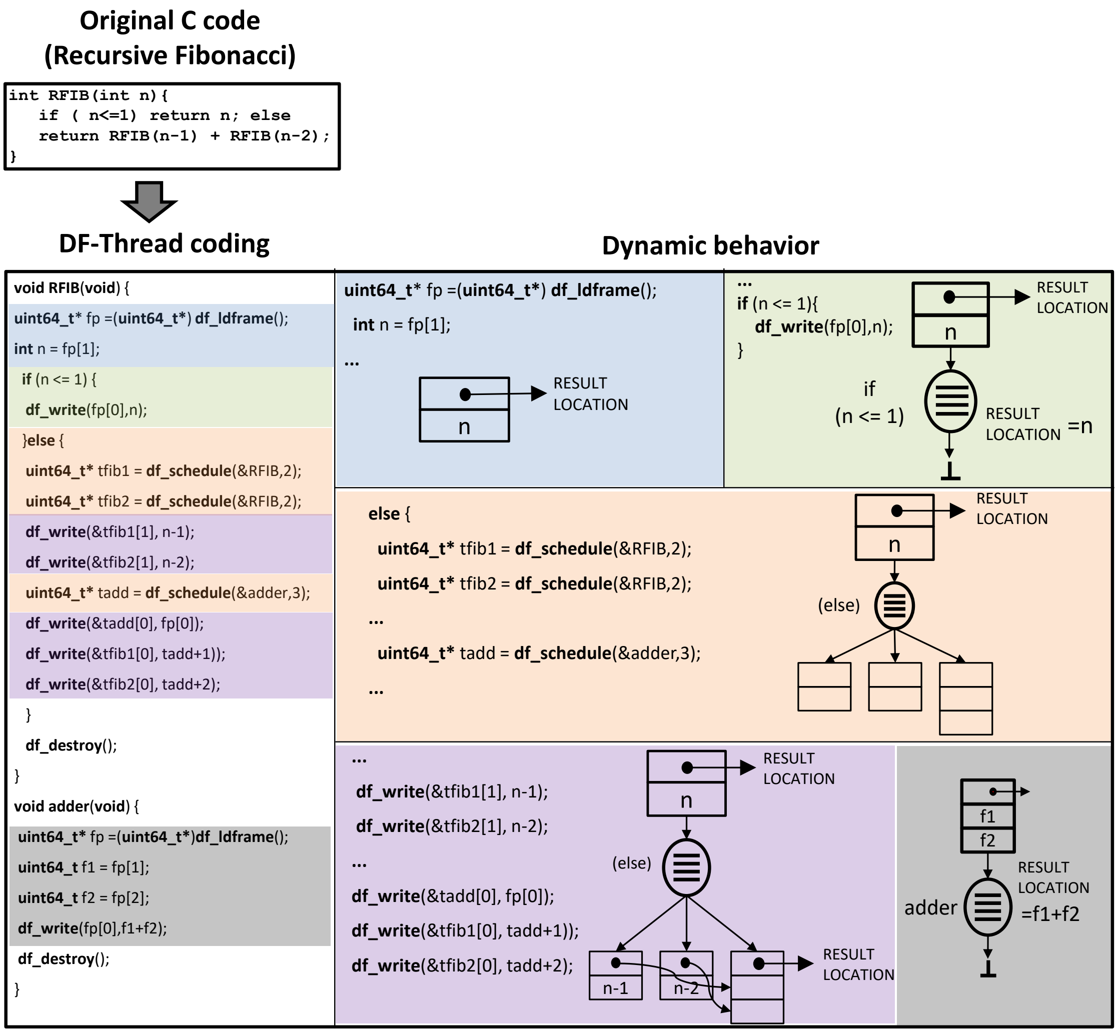
The key operation is the **df_schedule**, which creates a DF-Thread, whose code is specified by the parameter *ip* (the instruction pointer or the name of the corresponding function). With the same operation, a portion of memory (*frame*) is allocated and associated with the same DF-Thread. The size of the frame is determined by the number of inputs of the DF-Thread that is specified by the *sc* value of the **df_schedule**. The **df_schedule** returns the address (frame pointer) to the allocated memory space (the *frame*). The next step is to write the DF-Thread input and the output locations. This can be done by using the **df_write**. Once the frame pointer (*fp*) has been retrieved by the **df_ldframe**, the **df_write** will store the data (here `n-1`, `n-2`) in the location of *fp*[1] and *fp*[2], respectively. Please note that *fp*[0] has been reserved as the output location, into which the DF-Thread will write the result. For each write into the frame, the *sc* value will be decremented by 1 (this is implied by **df_write** and it is part of the implementation of the **df_write** itself). In the end, **df_destroy** will terminate the current DF-Thread [17].

3 Introducing DRT

Developing a novel architecture may require considerable time when using an architectural simulator [2,14]. To reduce this development-cycle time, in the case of the dataflow execution model, we designed a tool that we call "Dataflow Run-Time" (DRT). The aim of the DRT is to make it easier for the software community to use a dataflow program execution model (here DF-Threads): by studying the simple examples that we propose, or building new examples, the compiler experts could derive an appropriate compilation path, which could target the DF-Threads PXM. This tool is compatible with real machines like x86, AArch, RISC-V. DRT only requires the installation of the GCC compiler for compiling and running DF-Threads programs.

DRT enables the fast development and debugging of the DF-Threads' API and its data exchange mechanism, which is based on *frames* (see Fig. 3).

According to an initial test done in DRT, we can reduce the development-cycle time from minutes/hours to seconds (see Section 3). As shown in Fig. 4, we currently need to map manually ('manual coding into DFT syntax') high level programs ('.c code') to the DF-Threads API. Then, the DRT enables a standard compiler (GCC in our case) to generate a binary that can run on standard architectures. The availability of DRT provides a basis for direct writing dataflow codes but also enables compiler experts to further build on this workflow and integrate it in a compiler (lower part of Fig. 4, which is not in the scope of this paper).



```
int RFIB(int n){
   if ( n<=1) return n; else
   return RFIB(n-1) + RFIB(n-2);
}
```

```
void RFIB(void) {
uint64_t* fp =(uint64_t*) df_ldframe();
int n = fp[1];
 if (n <= 1) {
  df_write(fp[0],n);
 }else {
  uint64_t* tfib1 = df_schedule(&RFIB,2);
  uint64_t* tfib2 = df_schedule(&RFIB,2);
  df_write(&tfib1[1], n-1);
  df_write(&tfib2[1], n-2);
  uint64_t* tadd = df_schedule(&adder,3);
  df_write(&tadd[0], fp[0]);
  df_write(&tfib1[0], tadd+1));
  df_write(&tfib2[0], tadd+2);
  }
  df_destroy();
}
void adder(void) {
 uint64_t* fp =(uint64_t*)df_ldframe();
 uint64_t f1 = fp[1];
 uint64_t f2 = fp[2];
 df_write(fp[0],f1+f2);
 df_destroy();
}
```

```
uint64_t* fp =(uint64_t*) df_ldframe();
 int n = fp[1];
...
```

```
...
if (n <= 1){
  df_write(fp[0],n);
}
```

```
else {
  uint64_t* tfib1 = df_schedule(&RFIB,2);
  uint64_t* tfib2 = df_schedule(&RFIB,2);
...
  uint64_t* tadd = df_schedule(&adder,3);
...
```

```
...
 df_write(&tfib1[1], n-1);
 df_write(&tfib2[1], n-2);
...
df_write(&tadd[0], fp[0]);
df_write(&tfib1[0], tadd+1));
df_write(&tfib2[0], tadd+2);
```

Fig. 3: Illustrating the operations of the basic DRT API functions with a simple Recursive Fibonacci (RFIB) example. On the left, there is the representation of the RFIB function and its coding in DF-Thread style. On the right, we detail the specific dynamic behavior. Example rearranged from [27].

Similar efforts exist like the Delaware Adaptive Runtime System (DARTS) [25] and the Open Community Runtime (OCR) [27], so we compare them with DRT in Section 3. DRT is available as open-source at http://drt.sf.net [3].

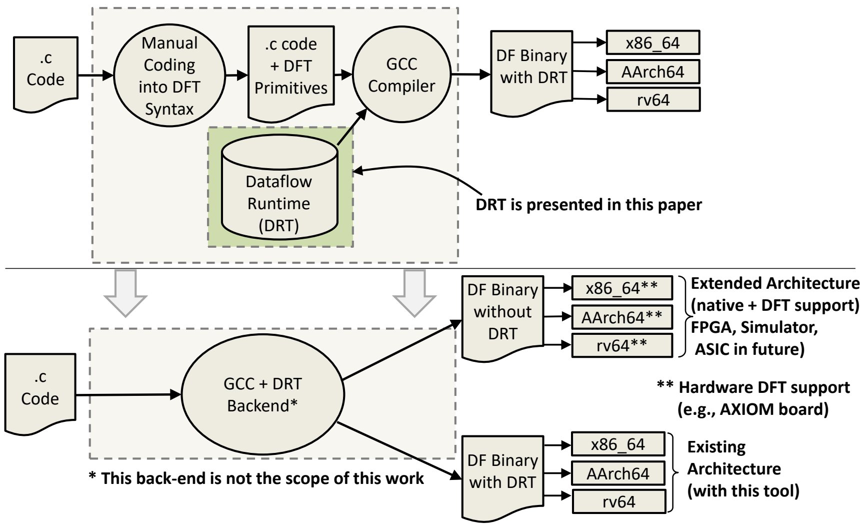

.c Code
Manual Coding into DFT Syntax
.c code + DFT Primitives
GCC Compiler
DF Binary with DRT
x86_64
AArch64
rv64
Dataflow Runtime (DRT)
DRT is presented in this paper
DF Binary without DRT
x86_64**
AArch64**
rv64**
Extended Architecture (native + DFT support)
FPGA, Simulator, ASIC in future)
** Hardware DFT support (e.g., AXIOM board)
.c Code
GCC + DRT Backend*
* This back-end is not the scope of this work
DF Binary with DRT
x86_64
AArch64
rv64
Existing Architecture (with this tool)

Fig. 4: The role of DRT in developing applications based on the DataFlow Threads (DFT) execution model. In the top part, we show the current setup of DRT. In the bottom part, we show the production framework that we envision. The idea is that DRT could help develop a future DRT backend of a standard compiler.

3.1 Debugging with DRT

In order to check the correctness of the dataflow execution, it is crucial to monitor the movement of the data and the status of the DF-Threads. In fact, whenever new DF-Threads are created, the DRT is responsible for providing a memory block called *frame* to store data, meta-information settings, and keeping track of the DF-Thread status. When a producer DF-Thread wants to write its outputs into the consumer DF-Thread, the DRT performs the write operations and decreases the synchronization count (*sc*) of the consumer DF-Thread. When the *sc* reaches zero, the DF-Thread becomes ready to be executed, and it is moved into a ready queue by the DRT. Once a DF-Thread terminates its execution, the DRT deallocates the associated *frame* from memory, dequeues the next DF-Thread from the ready queue, and assigns it to an available core. Finally, DRT will report if the program's output is successfully calculated, e.g., making a checksum with the reference program's result. In the following, we illustrate a debugging session in detail for the RFIB benchmark (Fig. 5, Fig. 6).

DRT offers the possibility to customize the development environment through the command-line for exploring up to four levels of debugging. The debug level one displays the used frame pointers only, while the second and third level print the executed dataflow instructions and the content of the frames, respectively. The fourth level gives us the statistical information about

[3] Checkout the DRT repository by this command:
svn co https://svn.code.sf.net/p/drt/code/

the DF-Threads, the memory, and the queues. The user can specify the level of debugging by the environment variable `DRT_DEBUG`.

In Fig. 5, as an illustrative example for analyzing the benchmark behavior, we show the output of DRT when the debug level is set to three for the RFIB benchmark and its input is n=4. We also illustrate the corresponding dataflow graph in Fig. 6. The first line describes the command line for executing a dataflow code with DRT. In the third line, the DRT initializes the environment and allocates the memory space for storing the frames based on the application requirements. Lines 4 and 5 show the creation of the scheduled function (the RFIB function, see Table 2) and the `report` function to collect the results. In lines 6 and 7, the *df_write* writes the value (*val*) in the output frame and decrements the associated synchronization count (*sc*). Lines 10 to 19 describes the recursive calls of the RFIB functions. Finally, the current DF-Thread will be terminated, and its input frame will be deallocated (line 20).

```
 1  ~/drt-code $ DRT_DEBUG=3 ./RFIB 4
 2  computing Recursive Fibonacci(4)
 3  -DRT: FRAME-MEM allocation+initialization done.
 4  TS: fi=0  ip=0x403a46  fp=0x609f60  sc=1/1
 5  TS: fi=1  ip=0x401795  fp=0x609fc0  sc=2/2
 6  TW: fi=1  ip=0x401795  fp=0x609fc0 val=0x609f6000  sc=1/2
 7  TW: fi=1  ip=0x401795  fp=0x609fc0 val=0x4  sc=0/2
 8  ++main
 9  -DRT: Starting Dataflow launcher.
10  TE: fi=1 ipnew=0x401795  fpnew=0x609fc0
11  TS: fi=2  ip=0x401795  fp=0x60a020  sc=2/2
12  TS: fi=3  ip=0x401795  fp=0x60a080  sc=2/2
13  TW: fi=2  ip=0x401795  fp=0x60a020 val=0x3  sc=1/2
14  TW: fi=3  ip=0x401795  fp=0x60a080 val=0x2  sc=1/2
15  TS: fi=4  ip=0x400d81  fp=0x60a0e0  sc=3/3
16  TW: fi=4  ip=0x400d81  fp=0x60a0e0 val=0x609f6000  sc=2/3
17  TW: fi=2  ip=0x401795  fp=0x60a020 val=0x60a0e001  sc=0/2
18  TW: fi=3  ip=0x401795  fp=0x60a080 val=0x60a0e002  sc=0/2
19  TD: fi=1  ip=0x401795  fp=0x609fc0  sc=2
20  TE: fi=2 ipnew=0x401795  fpnew=0x60a020
21  ++report
22  DF-Thread RFIB = 3
23  *** SUCCESS ***
```

Fig. 5: DRT sample output. DRT_DEBUG is an environment variable for specifying the debug level. The DF-Threads functions are mapped to internal operations where TS stands for thread scheduling, TE stands for thread-end, TD stands for thread drop, TW stands for thread write, *ip* stands for instruction pointer, and *fp* stands for frame pointer. Other debugging information is *fi* for frame index, *sc* stands for synchronization count, *ipnew/fpnew* are the *ip/fp* just freed.

The list of *ip* and *fp* addresses that are shown in Fig. 5 correspond to the same addresses that can be retrieved through standard disassembler tools (e.g., objdump). However, the usage of such tools gives us only a static view, while DRT enables a dynamic analysis showing the entire sequence of executed instructions with additional information about the DF-Threads, memory, and queue status. For example, the *ip*=0x401795 corresponds to the address of the code of the RFIB function (see Table 2). All the corresponding functions and their *fp* addresses generated in the function RFIB are shown in Table 2.

Table 2: The function name and its corresponding frame pointer address that are shown in Fig. 5 (same as in objdump tool).

Frame pointer address	Corresponding function
0x401795	RFIB
0x400d81	adder
0x403a46	report

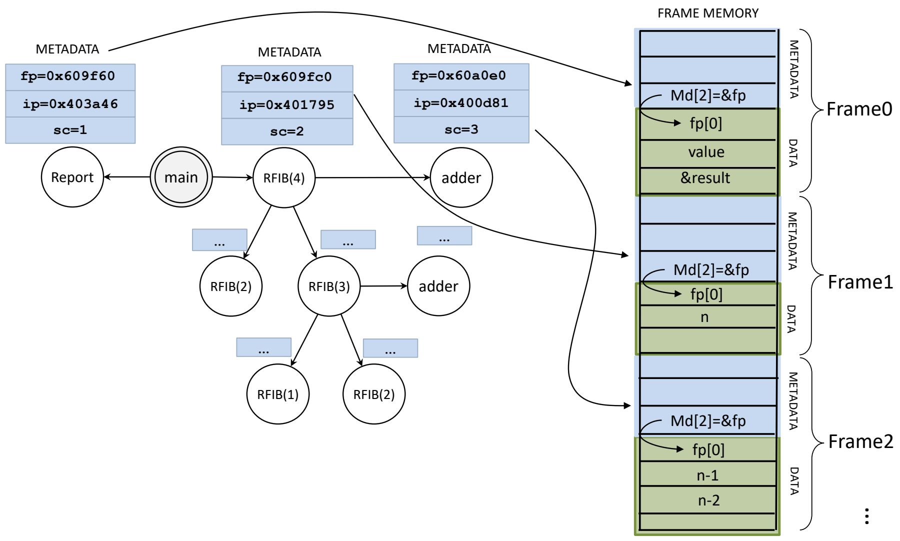


Fig. 6: An example of the RFIB(4) function of the Recursive Fibonacci (RFIB) example for illustrating the organization of data frames and their metadata. The metadata includes *fp* (frame pointer), *ip* (instruction pointer) and *sc* (synchronization count). The data illustrated here is extracted from the output of the DRT tool.

```
void df_write(uint64_t *fp, uint64_t val)
{
    *fp=val;            //write the value
    uint64_t *md=METADATA(fp); //retrieve metadata
    md[MDSC]--;         //decrement synchronization count
    if (md[MDSC] == 0) //move the frame to READY QUEUE
       TSETREADY(md[MDQSTATUS]);
}
```

Fig. 7: An example of a modeled function in the DRT implementation, where METADATA extracts the metadata pointer from the frame, MDSC is the offset of the synchronization count, and MDQSTATUS is the offset of the status bits that indicate whether the frame is in ready or waiting status.

In order to show the effectiveness of the internal modeling of the DRT function, we consider the implementation of the *df_write* function (see Fig. 7). The *df_write* needs two arguments, the pointer to the output *frame* (*fp*) and the value to write in such *frame.* Internally, the *df_write* extracts the metadata pointer from the given *frame* and, based on the *sc* information, *df_write* decides whether the DF-Thread is in ready or waiting status. Other useful debugging information, not shown in this simple example for the sake of simplicity, are the status of queues, the total number of allocated frames, the total number of writes, total number of frames that are in ready or waiting status.

4 Evaluation

In this Section, we compare the performance of DRT against other similar environments, namely OCR [27] and DARTS [25]. OCR and DARTS use a dataflow model to manage threads, similarly to DRT: the common main idea is to decouple the higher layers of the software stack from the underlying hardware by using a possibly universal interface. For details about OCR, DARTS, and other related environments, see Section 5. As explained in Section 3, we wrote some initial benchmarks manually due to the lack of a compiler. Therefore, at this stage, we cannot afford to make more extensive tests with large benchmarks.

To demonstrate the capabilities of the DRT, we selected two simple benchmarks:

- Recursive Fibonacci (RFIB) in order to generate a high number of threads easily.
- Blocked Matrix Multiplication (BMM) as it is a very commonly used kernel in many applications (especially in Artificial Intelligence, Deep Neural Networks, etc.), and it moves much data around.

The two benchmarks are using the same exact algorithm for all three frameworks. The output of the benchmarks is validated against the output produced with other independent tools executing the same benchmarks.

For the sake of simplicity, we analyze the sensitivity with the input set by using n=10, 15, 20, 25 for RFIB and s=128, 256, 512, 1024 for BMM, where n is the index of the corresponding Fibonacci number and s is the size of the square matrices that are multiplied. For the block size of the matrices, we used b=8, where b is the number of the elements inside a block. The purpose of DRT is to explore the correctness of the dataflow execution, not to scale the performance across cores. Nevertheless, to make a fair comparison against other environments, we restricted our evaluation to a single core execution.

For each of the three runtime frameworks (DRT, DARTS, and OCR), we measure the time spent in the Region Of Interest (ROI) of each benchmark, and we repeat at least ten times the experiments to obtain statistically valid measurements. We report the execution time speedup by using DARTS as the baseline. As we can observe from Fig. 8 and Fig. 9, DRT can outperform by one order of magnitude DARTS for smaller inputs. DRT outperforms OCR by a factor of about 13x for n=25. While the OCR and DARTS are well optimized, DRT can still be improved. However, as stated before the main goal of DRT is just to provide a tool for developing DF-Thread benchmarks and a future compiler; more performance could be achieved by using DF-Thread native support as shown in Fig. 4.

4.1 DRT versus other architectural exploration tools

DRT is also serving to explore new architectures based on dataflow concepts. While designing such a non-yet existing architecture, different approaches can be used:

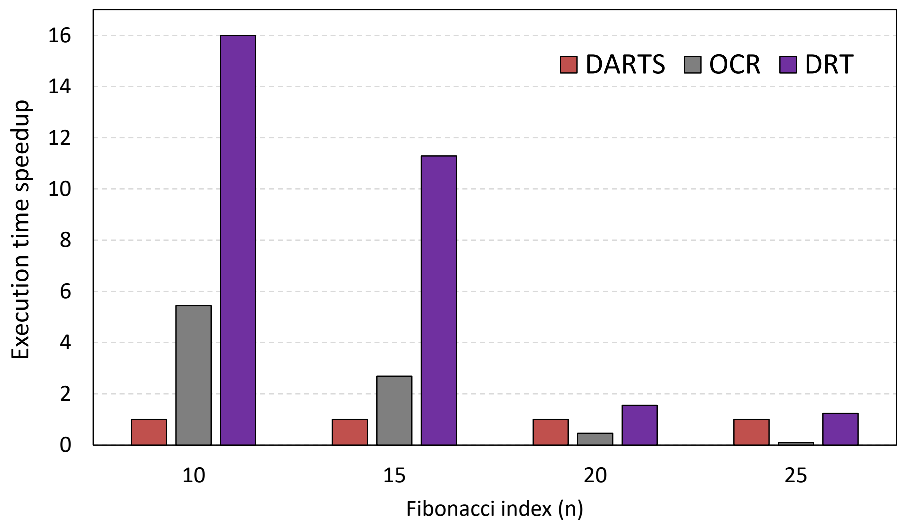


Fig. 8: RFIB execution time speedup comparison between DRT, DARTS and OCR runtime. Here OCR is the baseline. DRT reaches better performance due to a simplified management of the dataflow execution.

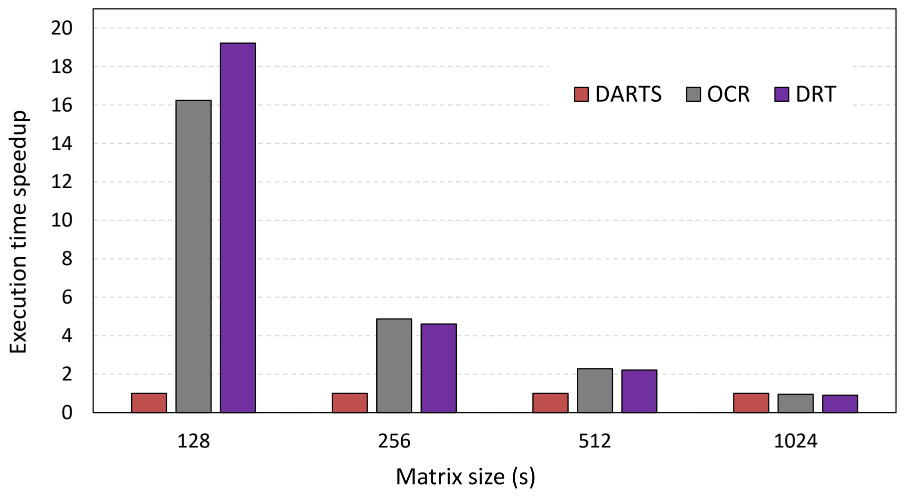


Fig. 9: Blocked Matrix Multiplication execution time speedup comparison between DRT and DARTS and OCR, with the DARTS as baseline. While for larger Matrix sizes the execution time tends to be the same for three tools, it is important to note that during the development-cycle, we typically use smaller inputs. So, the shorter execution time of DRT during tests helps focus on the development.

- An architectural simulator (e.g., the COTSon [2]).
- A hardware prototype. In our work, we use the AXIOM board (provided by SECO [35]), which includes four 64-bit ARM cores, an FPGA, and a GPU [13,10].
- A runtime tool like DRT; in the following, we discuss how this tool can be used to understand the data exchange among the dataflow threads.

While it is possible to develop DF-Threads codes on a simulator or on an FPGA prototype, we found that it is more productive to use a tool such as the DRT, a minimalistic API written in around 300 lines of C code, through which it is possible to test and debug the implementation of a specific feature in seconds, while doing that on an FPGA may require days [15] (see Table 3). In Fig. 10, we show the simulation time of the COTSon simulator compared to the DRT. As we can see, we can obtain up to four orders of magnitude speedup while executing a benchmark RFIB. The speedup in simulation time of a simulator is lower compared to an FPGA, but the development-cycle time can be much higher; this is discussed below.

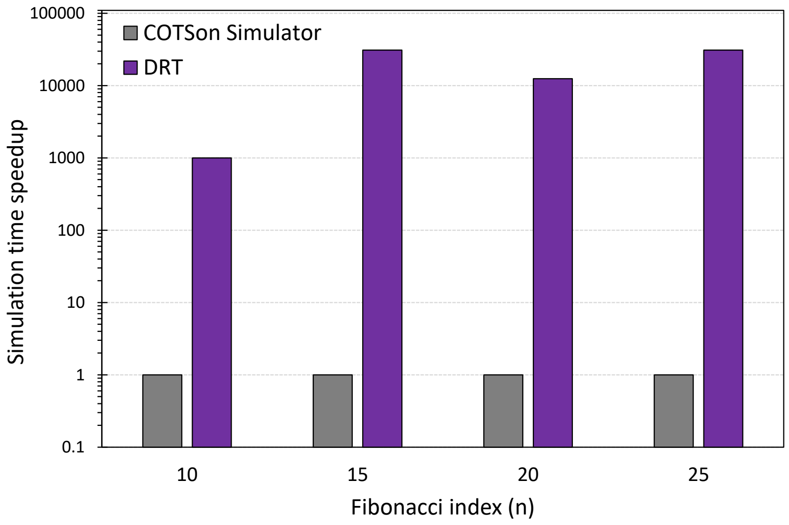


Fig. 10: Simulation time speedup comparison between DRT and the COTSon simulator by using the RFIB example. DRT significantly decreases the development-cycle time to develop a dataflow program.

Table 3: Comparing lightweight DRT with other different tools for developing dataflow codes and the related architectures. As we can see DRT, is using only 300 lines of C code.

	DRT	Simulator[24]	FPGA[33]
SLOC of the framework	∼ 300	∼ 112,000	∼ 1,000,000
Openness the development framework	High (open-source)	Medium (partly open-source)	Limited (proprietary tools)
Complexity of the development-cycle	Low (seconds)	High (minutes)	Very high (hours)

In terms of evaluating the DRT in relation to other approaches for developing the initial codes that use the dataflow execution model, we compare other tools for modeling new architectures like the simulator and the FPGA prototype in Table 3. The usage of these tools is necessary when exploring hardware support for the dataflow execution [14,15]. We considered the following metrics:

- SLOC[4]: these are the source lines of code of the corresponding framework; these numbers are all publicly available; for the simulator, we referred to the COTSon simulator [24], and for the FPGA, we referred to the software stack of the AXIOM board [33].
- Openness of the development framework: while DRT can be downloaded and installed in seconds, COTSon requires at least some hours to complete the setup and some days to become familiar with the modeling of the components; moreover, some parts of the code (AMD SimNow) are not open-source; regarding the FPGA-board, the software stack is open-source, but the tools are typically proprietary and may require licensing and complex setup procedures.
- The complexity of the development-cycle: while it is rather simple to make modifications, test, and debug a program through the DRT tool, it may require minutes to complete a full simulation in the COTSon simulator, and it may require hours to modify and re-generate a full design in the FPGA framework [14,15].

5 Related Work

In recent years, there have been some works regarding dataflow architectures and their execution models that we summarized below. In the following, we highlight some works that are related to ours, and we point out the differences.

BMDFM [32] is a hybrid dataflow runtime environment that provides a dataflow execution model with its extended instruction set. BMDFM has been implemented on conventional multi-core platforms to show a complete parallelization environment.

FREDDO [26] uses the distribution of Data-Driven Threads (DDT) over conventional multi-core processors. FREDDO is written in C++ and focuses mainly on Object-Oriented programs.

Sucuri [36] is a Python dataflow library to execute Dataflow Graphs (DFGs) over a multi-core distributed system. Sucuri is based on a centralized and local scheduler in each node that can execute the ready tasks in their local queues. The compiler partitions the DFG, then, during the runtime, each related DFG part will be distributed among the associated node.

Swift/T [42] is a new implementation of swift language [41] that provides high-level programmability for implicit dataflow programming. It addresses some optimization for the Swift parallel scripting language, along with Turbine compiler, which C/C++/Fortran programmers can develop their software based on this platform.

Trebuchet [1] presents the implementation of dynamic dataflow architecture. Trebuchet presents the execution of code blocks based on a multi-thread dataflow model.

In XKaapi [11], the authors show a dataflow task acceleration on multi-core CPUs, and GPUs. XKaapi has been written in C++ language, and a work-stealing method has been presented for scheduling ready tasks via a runtime system.

These six works - BMDFM, Freddo, Sucuri, Swift/T, Trebuchet, and Xkaapi - use dataflow approaches to improve the execution time. In contrast, DRT ambition is to provide a tool for testing and debugging dataflow benchmarks, while the performance is obtained by deploying one DF-Thread implementation [12,17,39]. In particular, DRT represents a key element to develop a toolchain to support a dataflow execution model, which could be targeted by a compiler. While there are many similarities between DRT and the above works, we choose a more detailed comparison with the Codelet program execution model [46,5] and Open Community Runtime (OCR) [27,31].

[4] Source lines of code

In the Codelet execution model concept [46], Codelet is a fine grain event-driven unit of computation, smaller than a thread, aims to exploit the parallelism of Exascale platforms. The runtime environment DARTS[25] has been presented in such a way that a high-level program will turn into Codelet Graph with the API interface, and the runtime executes the Codelets based programs to exploit the maximum parallelism and power efficiency of the underlying hardware. DARTS uses a double level hierarchy to structure programs: threaded procedures (TP) and Codelets; TP includes several Codelets. In contrast, DF-Threads leaves more freedom to the programmer by using a flat hierarchy of threads.

The Open Community Runtime [27] is based on event driven tasks. OCR is a runtime that is influenced by the Codelet execution model and is inspired by the Asynchronous Many Task (AMT) models. A high level program written in OCR runtime is organized with Directed Acyclic Graph, which is structured with relocatable data-blocks, events, or tasks. These elements are called nodes connected to each other by edges, which represent the dependencies between nodes. DARTS and OCR trigger threads by using both data and events. In DF-Threads, we do not need this distinction: events can be treated as data.

DF-Threads [12] introduces a low-level API, which enables a high-level code into a hybrid dataflow model that can benefit from the high parallelism while parallel computations are the potential to distribute over nodes and cores.

6 Conclusion and future work

In this paper, we present Dataflow Run-Time (DRT), a tool for fast prototyping of benchmarks written for a dataflow program execution model (PXM). While running such benchmarks on an architecture that provides dataflow support could provide a large speedup (e.g., up to 28x) compared to OpenMPI counterparts, the compiler technology is not yet developed enough. Therefore, the contribution of this paper helps bridge the gap between future compilers and a dataflow PXM. In fact, DRT is a step-forward to have a general compiler for executing in a dataflow style. Through DRT, we can develop dataflow codes/benchmarks, test and debug them with a better development-cycle time than other modeling tools like architectural simulators or FPGAs. We also describe how to perform the debugging of the data flow among DF-Threads' frames, and we compare the efficiency of DRT against other similar environments such as DARTS and OCR.

We illustrate and evaluate more in detail two simple benchmarks (RFIB, BMM); however, we have provided other programs in our repository (http://drt.sf.net).

DF-threads is a general approach to execute parallel programs in a more efficient way than with the von Neumann paradigm. We hope that in the future, with the help of the fully fledged compiler, we could extend our work to more general applications.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments and suggestions. The work of this paper is partly funded by the European Commission on AXIOM H2020 (id. 645496), TERAFLUX (id. 249013), HiPEAC (id. 871174).

References

1. Alves, T.A.O., Marzulo, L.A.J., Franca, F.M.G., Costa, V.S.: Trebuchet: Exploring tlp with dataflow virtualisation. Int. J. High Perform. Syst. Archit. **3**(2/3), 137 – 148 (May 2011)

2. Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., Ortega, D.: COTSon: infrastructure for full system simulation. SIGOPS Oper. Syst. Rev. **43**(1), 52–61 (2009)
3. Arvind, Culler, D.E.: Dataflow architectures. Annual Review of Computer Science (1986)
4. Arvind, K., Nikhil, R.S.: Executing a program on the mit tagged-token dataflow architecture. IEEE Trans. Comput. **39**(3), 300–318 (1990). https://doi.org/10.1109/12.48862
5. CAPSL: The codelet execution model. `https://www.capsl.udel.edu/codelets.shtml`
6. Chen, Y., Emer, J., Sze, V.: Using dataflow to optimize energy efficiency of deep neural network accelerators. IEEE Micro **37**(3), 12–21 (2017)
7. Dennis, J.B.: Data flow computation. In: Broy, M. (ed.) Control Flow and Data Flow: Concepts of Distributed Programming. pp. 345–398. Springer Berlin Heidelberg, Berlin, Heidelberg (1986)
8. Dennis, J.B., Misunas, D.P.: A preliminary architecture for a basic data-flow processor (1974)
9. Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., LeCun, Y.: Neuflow: A runtime reconfigurable dataflow processor for vision. In: CVPR 2011 WORKSHOPS. pp. 109–116 (2011)
10. Filgueras, A., Vidal, M., Mateu, M., Jimnez-Gonzlez, D., lvarez, C., Martorell, X., Ayguad, E., Theodoropoulos, D., Pnevmatikatos, D., Gai, P., Garzarella, S., Oro, D., Hernando, J., Bettin, N., Pomella, A., Procaccini, M., Giorgi, R.: The axiom project: Iot on heterogeneous embedded platforms. IEEE Design and Test (nov 2019)
11. Gautier, T., Lima, J.V.F., Maillard, N., Raffin, B.: Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 1299–1308 (2013)
12. Giorgi, R., Faraboschi, P.: An introduction to DF-Threads and their execution model. In: IEEE MPP. pp. 60–65. Paris, France (Oct 2014)
13. Giorgi, R., Khalili, F., Procaccini, M.: AXIOM: A Scalable, Efficient and Reconfigurable Embedded Platform. In: IEEE Proc.DATEi. pp. 1–6 (Mar 2019)
14. Giorgi, R., Khalili, F., Procaccini, M.: A design space exploration tool set for future 1k-core high-performance computers. In: ACM RAPIDO Workshop. pp. 1–6 (2019)
15. Giorgi, R., Khalili, F., Procaccini, M.: Translating timing into an architecture: The synergy of cotson and hls (domain expertise – designing a computer architecture via hls). Hindawi - International Journal of Reconfigurable Computing (Dec 2019)
16. Giorgi, R., Procaccini, M.: Bridging a data-flow execution model to a lightweight programming model. 2019 International Conference on HPCS (2019)
17. Giorgi, R., Scionti, A.: A scalable thread scheduling co-processor based on data-flow principles. ELSEVIER Future Generation Computer Systems **53**, 100–108 (Dec 2015)
18. Giorgi, R.: Scalable embedded computing through reconfigurable hardware: comparing df-threads, cilk, OpenMPI and jump. ELSEVIER Microprocessors and Microsystems **63**, 66–74 (aug 2018)
19. Giorgi, R., et al.: TERAFLUX: Harnessing dataflow in next generation teradevices. ELSEVIER Microprocessors and Microsystems **38**(8, Part B), 976–990 (2014)
20. Hum, H.H., Maquelin, O., Theobald, K.B., Tian, X., Gao, G.R., Hendren, L.J.: A study of the earth-manna multithreaded system. International Journal of Parallel Programming (1996)
21. Kabrick, R., Perdomo, D.A.R., Raskar, S., Diaz, J.M.M., Fox, D., Gao, G.R.: Codir: Towards an mlir codelet model dialect. In: 2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM). pp. 33–40. IEEE (2020)
22. Kavi, K., Arul, J., Giorgi, R.: Performance evaluation of a non-blocking multithreaded architecture for embedded, real-time and dsp applications. In: 14th Int.l Conf. on Parallel and Distributed Computing Systems (ISCA-PDCS-01). pp. 365–371. Richardson, TX, USA (Aug 2001)
23. Kavi, K.M., Giorgi, R., Arul, J.: Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. IEEE Trans. Computers **50**(8), 834–846 (Aug 2001)
24. Labs, H.: COTSon: Infrastructure for full system simulation. `https://sourceforge.net/projects/cotson/files/`
25. Labs, H.: Darts: An asynchonous fine-grained runtime based on the codelet model. `https://github.com/szuckerm/DARTS`, accessed: 2021.01
26. Matheou, G., Evripidou, P.: Freddo: an efficient framework for runtime execution of data-driven objects. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) (01 2016)

27. Mattson, T.G., Cledat, R., Cav, V., Sarkar, V., Budimli, Z., Chatterjee, S., Fryman, J., Ganev, I., Knauerhase, R., Min Lee, Meister, B., Nickerson, B., Pepperling, N., Seshasayee, B., Tasirlar, S., Teller, J., Vrvilo, N.: The open community runtime: A runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (2016)
28. Najjar, W.A., Lee, E.A., Gao, G.R.: Advances in the dataflow computational model. Parallel Comput. **25**(13–14), 1907–1929 (Dec 1999)
29. Nowatzki, T., Gangadhar, V., Ardalani, N., Sankaralingam, K.: Stream-dataflow acceleration. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. pp. 416–429. ISCA '17, Association for Computing Machinery, New York, NY, USA (2017)
30. Nowatzki, T., Gangadhar, V., Sankaralingam, K.: Heterogeneous von neumann/dataflow microprocessors. Commun. ACM **62**(6), 83–91 (2019)
31. OCR: Open community runtime v1.0. `https://xstack.exascale-tech.com/git/public/ocr.git`, accessed: 2021.01
32. Pochayevets, O.: BMDFM: A hybrid dataflow runtime parallelization environment for shared memory multiprocessors. In: MS thesis in Computer Engineering (2006)
33. Project, E.: Agile, extensible, fast i/o module for the cyber-physical era. `https://git.axiom-project.eu/`, accessed: 2021.01
34. Sarkar, V., Hennessy, J.: Partitioning parallel programs for macro-dataflow. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. pp. 202–211. LFP '86, Association for Computing Machinery, New York, NY, USA (1986)
35. SECO, s.r.l.: http://www.seco.com, `http://www.seco.com`
36. Silva, R.J.N., Goldstein, B., Santiago, L., Sena, A.C., Marzulo, L.A.J., Alves, T.A.O., Frana, F.M.G.: Task scheduling in sucuri dataflow library. In: 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW). pp. 37–42 (2016)
37. Stavrou, K., et al.: Programming abstractions and toolchain for dataflow multithreading architectures. In: Proc. 8th Int.l Symp. on Parallel and Distributed Computing (ISPDC 2009). pp. 107–114. IEEE (July 2009)
38. Swanson, S., Michelson, K., Schwerin, A., Oskin, M.: Wavescalar. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. p. 291. MICRO 36, IEEE Computer Society, USA (2003)
39. Verdoscia, L., Giorgi, R.: A data-flow soft-core processor for accelerating scientific calculation on FPGAs. Mathematical Problems in Engineering **2016**(1), 1–21 (Apr 2016), article ID 3190234
40. Weis, S., Garbade, A., Fechner, B., Mendelson, A., Giorgi, R., Ungerer, T.: Architectural support for fault tolerance in a teradevice dataflow system. Springer Int.l Journal of Parallel Programming **44**(2), 208–232 (Apr 2016)
41. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: A language for distributed parallel scripting. Parallel Computing **37**(9), 633 – 652 (2011), emerging Programming Paradigms for Large-Scale Scientific Computing
42. Wozniak, J.M., Armstrong, T.G., Wilde, M., Katz, D.S., Lusk, E., Foster, I.T.: Swift/t: Large-scale application composition via distributed-memory dataflow processing. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. pp. 95–102 (2013)
43. Yazdanpanah, F., Alvarez-Martinez, C., Jimenez-Gonzalez, D., Etsion, Y.: Hybrid dataflow/von-neumann architectures. IEEE Transactions on Parallel and Distributed Systems (2014)
44. Yijun Liu, Furber, S.: A low power embedded dataflow coprocessor. In: IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05). pp. 246–247 (2005)
45. Zuckerman, S., Landwehr, A., Livingston, K., Gao, G.: Toward a self-aware codelet execution model. In: 2014 Fourth Workshop on DFM. pp. 26–29 (2014)
46. Zuckerman, S., Suetterlein, J., Knauerhase, R., Gao, G.R.: Using a "codelet" program execution model for exascale machines: Position paper. In: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. pp. 64–69. EXADAPT '11, Association for Computing Machinery, New York, NY, USA (2011)