An Introduction to DF-Threads and their Execution Model

Roberto Giorgi
Dept. Information Engineering and Mathematics
University of Siena
Siena, Italy

Paolo Faraboschi
Intelligent Infrastructure Lab
Hewlett Packard
Barcelona, Spain

***Abstract*—Current computing systems are mostly focused on achieving performance, programmability, energy efficiency, resiliency by essentially trying to replicate the uni-core execution model n-times in parallel on a multi/many-core system. This choice has heavily conditioned the way both software and hardware are designed nowadays. However, as old as computer architecture is the concept of dataflow, that is "initiating an activity in presence of the data it needs to perform its function" [J. Dennis]. Dataflow had been historically initially explored at instruction level and has led to the major result of the realization of current superscalar processors, which implement a form of "restricted dataflow" at instruction level.**

In this paper, we illustrate the idea of using the dataflow concept to define novel thread types that we call Data-Flow-Threads or DF-Threads. The advantages we are aiming at regard several aspects, not fully explored yet: i) isolating the computations so that communication patterns can be more efficiently managed by a not-so-complex architecture; ii) possibility to repeat the execution of a thread in case of detected faults affecting the thread resources; iii) providing a minimalistic low-level API for allowing compilers and programmers to map their parallel codes and architects to implement more efficient and scalable systems. The semantics of DF-Threads is also tightly connected to their execution model, hereby illustrated.

Several other efforts have been done with similar purposes since the introduction of macro-dataflow through the more recent DF-Codelets and the OCR project. In our case, we aim at a more complete model with the above advantages and in particular including the way of managing the mutable shared state by relying on the transactional memory semantics.

Our initial experiments show how to map some simple kernel and the scalability potential on a futuristic 1k-core many-core.

Keywords—dataflow; many-core; multi-core; multi-processor programmability; execution model; scalability; energy efficiency

I. INTRODUCTION

The main challenges of future Exascale computing systems have been identified in several public roadmaps [1] and large research projects such as TERAFLUX [2], X-STACK [3], UHPC [4] – to name a few – as programmability, simplicity of architecture, resiliency, scalability. In order to reach the roadmap goals, systems should be able to manage hundreds of millions or more threads of a possibly fine grain size in a dataflow fashion. Therefore, managing those threads appropriately is becoming relevant for designing such systems.

Since the proposal of macro dataflow [5], many had proposed to use some clustering of instructions in order to create threads, or units of computations larger than a single instruction in order to efficiently distribute the workload across multiple cores: ETS, Multithreaded Monsoon, TAM, HEP, Tera, EARTH, SDF, DDM, DTA, just to name a few [6][7].

More recently, research outlines the importance of interfaces to decouple the software stack from the underlying machine: some noticeable efforts had been presented in the TERAFLUX project [8], the Data-Flow Codelet model [9], and the Open Community Run-time (OCR) project [13]. In these cases, a dataflow inspired interface to start computation on the availability of data or events has been proposed.

In particular, the dataflow paradigm provides an elegant way to encapsulate computations to obtain several advantages:

i) in order to avoid "spaghetti-like" accesses to data-memory, DF-Threads obey a specific memory model, which provides isolation of accesses, and therefore a greater locality and the possibility to reduce the energy associated to data transfers: the associated memory model is described in Sec. IV;

ii) an important property of DF-Threads is their capability of being restarted in presence of detected faults affecting the thread resources such as cores, therefore providing a much better resilience of the system, without the need of using full-system classical checkpointing/restart technique. This property of DF-Thread has been thoroughly discussed in a recent paper [14] and hence here we discuss the rest of DF-Thread properties. Experiments [14] had shown that the system is capable of continuing to execute a complex computation by relying on the capability of restarting a single DF-Thread that had encountered faults, e.g., on cores or interconnects.

iii) a minimalistic API (Sec. III) and its architectural support is also described here (Sec. V): the aim is to provide simple yet powerful interface that can be targeted by compilers and programmers to map high level languages; previous experiments [2] had shown that is possible to enable OpenMP like programming models like StarSs/OmpSs [15], OpenAcc [16] or OpenStream [19] so that they can be mapped on DF-Threads architectural support such as the T* Instruction Set Extension [8][26]. In addition to this, the DF-Thread API and its public availability on the COTSon [24] simulator aims at a simpler experimentation and adoption. Also, the DF-Thread API may allow for different implementations ranging from a purely software one to a high specialized one on e.g. FPGAs.

We propose that DF-Threads could be specialized into several types (see Sec. II). One of the main reasons to have different thread types is that each of these threads has a different behavior in relation to their memory accesses and may need a different hardware support for their execution. Similar concepts are used in Staged Execution, which aims at dividing a program into segments and executing each segment at the core having the data and/or functionality to best run that segment. Some works using these concepts include Accelerated Critical Sections [10], Apple's Grand Central Dispatch [11], producer-consumer pipelines and Computation

Spreading [12] and Cilk [13]. We extend this discussion after presenting more details on the proposed architecture (Sec. V).

The main contributions of this paper are:

i) Precise description of a possible API for a dataflow based execution model (DF-Threads);

ii) Related discussion of a possible chip architecture and the architectural support;

iii) Presenting initial results on the scalability of the proposed DF-Threads and their execution model.

The rest of the paper is organized as follows: we introduce the DF-Threads types (Sec. II) with the aim to define a Data-Flow based Execution Model that in turn relies on the proper definition of Data-Flow API (Sec. III) and its semantics; the latter also includes the definition of Memory Model (Sec. IV), its related architectural support (Sec. V) and a description of a possible chip architecture (Sec. VI). Finally, we discuss results (Sec. VII), related work (Sec. VIII) and we conclude the paper.

II. DF-THREADS TYPES

Since our proposal aims at a holistic bookkeeping of the thread resource usage, in order to avoid unforeseen conflicts on using resources like cores, memory, I/O we first differentiate what is a DF-Thread and what is not.

In particular, we assume that for providing system services and I/O there exist S-Threads (System Threads) and in case we need to execute some legacy code that we do not want to re-compile or we cannot re-compile into DF-Threads we will encapsulate such thread in a L-Thread (Legacy Thread). L-type and S-type threads can respectively execute on legacy-cores or on system-cores, i.e., on "bigger-cores" with more resources like caches, functional units, potential for speculation to accelerate single threads or to perform I/O operations and implement the full set of OS services.

DF-Threads on the other side aim at isolating the access to data and code through different level of "grading": in particular we will consider DF1-Threads and DF2-Threads and some specialization of them in the following.

A. DF1-Thread

The DF1 thread is defined for ideal reference: if the compilation is able to produce only DF1 threads we foresee a greater potential for acceleration and saving power (e.g., by disabling branch prediction and branching units). A DF1-Thread is defined as follows:

- Communication among threads preserves Single Assignment Semantics.
  - A consumer thread can only be activated when all its input data are ready (data flow principle).
  - As DF1 threads only operate on values; the only memory consistency requirement is that when a thread starts to execute, it must observe the correct value of variables written to its associated memory (DF-frame memory) by the producer threads.
- The execution of the DF1-Thread code is done in a control-flow manner.
- Each thread has a single entry and a single exit point.
  - No jumps between DF1-Threads are allowed
  - Internal Jumps must also be avoided
- Thread granularity is compiler controlled, aiming at the coarser grain possible that results in efficient execution
- We allow the frame to define and use Thread Local Storage (see Memory Model Section – sec. IV).
- Each thread and data object could be tagged with a combination of compiler and runtime support [15][16].

B. DF1b-Thread

A DF1b thread is a DF1-Thread with the addition of internal control flow (jumps, branches, loops). While DF1 threads enable more efficient data flow, sometimes it may be more efficient or necessary to take advantage of code-locality.

It has to noted that in both cases (DF1, DF1b) the DF-threads are not allowed to jump anywhere: this is the basis for a side-effect free, repeatable execution [14][29]. It has also to be noted that it's not impossible to generate DF1 threads (we provide public examples: http://cotson.sf.net).

C. DF2-Thread

DF2-threads support flexible data structures and have an efficient support for streaming [19]. A DF2-Thread is a DF1 (or DF1b) thread that includes references to objects (arrays and dynamically created structures) which obey pure functional semantics (i.e. they are single assignment). DF2 threads can also make limited "service calls", for example to allocate memory dynamically. Such "service calls" should be extreme lightweight i.e. in the order of a few cycles or can be supported by the hardware co-processor that we call Distributed Thread Scheduler (DTS hereby) (see V.B and the DF_TALLOC in Sec. III) through some special instruction extension as outlined later in this paper (see Sec. V.A).

D. DF2tm Thread

This is a DF2 thread but with the addition that it can contain references to objects in Transactional Memory. When accessing shared memory, the DF-threads can be marked as fully transactional or can be split in DF2tm threads so that its management can activate transactional bookkeeping too such as TCC [20] or Intel TSX [21].

E. L-Thread

An L-Thread is a legacy thread that will contain any code that is not possible to fit in the previous categories (DF1, DF2), and in particular that may require side-effects or where the control flow is particularly DF "unfriendly". They can be used for legacy code (e.g. to run code on cores with legacy features).

F. S-Thread

An S-Thread is a thread taking care of more complex system calls and I/O. This type of thread is potentially blocking therefore can compromise the predictability of its duration. It contains certain system calls or doing I/O and therefore will be scheduled on cores that support specific I/O services, not available on other cores.

The DTS will be informed of L- and S- thread (not only DF-thread) scheduling requests (and can possibly take over the scheduling to the physical cores) in order to properly book-keep scheduling requests to the available cores.

It has to be noted that classical signaling like in p-threads is not required: either threads work locally, in producer-consumer fashion or use shared data through transactions (sec. IV).

G. Discussion on the several DF-Thread Types

As detailed in the previous subsection, DF-threads can have less or higher "dataflow purity": the highest level of dataflow purity will be assigned to behavior that gets values as inputs in a memory that we call *DF-frame* and produces values as output to one or more DF-frames (DF1-threads). DF2-threads still communicate in a dataflow fashion once collision on a data happens and it is resolved through transactions. DF1 and DF2 threads may need to interact with a memory manager (in these cases we aim at fast operations as outlined in previous work [22]). Both types of DF threads can be abandoned and restarted in the event of an error; if unwanted locally allocated storage is left this would produce wasted memory. In more modern languages this can be handled by automatic garbage collection, but in languages which rely on programmer invoked reclamation, this may introduce overheads.

The DF-threads can be abstracted as having a set of (limited number) of inputs, a set of (limited number) of outputs and may use local variables (Thread Local Storage or TLS) such as local arrays or more complex data structures. The output of one DF-thread can be forwarded to the input of other DF-thread(s) through DF-frames. From a programmer point of view the data also have different types of consistency based on the Memory Model that we adopt. DF1-threads for example, will use DF-frames allocated in a Frame Memory or data in the Thread Local Storage. More details are given in Section IV.

Please note that this model guarantees the consistency of the data by construction since it does not allow two threads that may change or read the same address to run simultaneously. When we use the Transactional Memory through the DF2tm-threads, we may relax this condition, allowing DF2tm-threads that share same data elements in their write set to be executed in parallel, assuming that the TM mechanism will resolve the conflicts if they occur. In the Memory Model Section (sec. IV), we detail also this and other type of memory consistency.

The entire management of the system needs to be done in a hierarchical manner and with a combination of new HW/SW interfaces. At the top level, the system (not the programmer) needs to decide where each DF/L/S-thread will be executed and how the "output variables" (write list of a DF-thread) know the address of the input variable of the DF-thread which uses it. Early Dataflow models supported data structuring via mechanisms which proved to be inefficient [23] but we intend to build on later work [2][22]]27][8], which has shown how these mechanisms can be introduced to provide efficient support for a full range of language features. The major departure from the "classical DF" architecture is the introduction of Transactional Memory and its integration with the dynamic frame management and hardware thread scheduling. Transactional Memory allows DF-Threads to manipulate shared global state but, because of the isolation property of transactions, allows the thread to proceed unhindered (except for possible transaction restart on conflict) and without explicit synchronization with other threads.

Regarding the support in the Operating System, the only need is for a driver for passing the higher level scheduling policies to the lower level DTS, the hierarchical distributed scheduler.

III. DF-THREAD API

The following prototypes define the DF-Thread API in a C-like syntax, where uint_8, uint64_t are fixed width types as defined, e.g., in C++11, for exemplification here but can be overloaded with any base type of a, e.g., 64-bit machine.

`void *DF_TSCHEDULE(bool cnd, void *ip, uint64_t sc);`
This function allocates the resources (a DF-frame of size 'sc' words and a corresponding entry in the Distributed Thread Scheduler – or DTS) for a new DF-thread and returns its Frame Pointer (fp); 'ip' specifies the Instruction Pointer of the first instruction of the code of this DF-thread; the above actions are actually performed based on the predicate 'cnd' (condition). The allocated DF-thread is not executed until its 'sc' reaches 0.

`void DF_DESTROY();`
The thread that invokes DF_DESTROY finishes and its DF-frame is freed, (the corresponding entry in the Distributed Thread Scheduler is also freed).

`uint64_t DF_TREAD(uint64_t offset);`
Loads the data indexed by 'offset' from the self (current thread) DF-frame. Assumption: the DF-thread has an implicit pointer to its DF-frame

`void DF_TWRITE(uint64_t val, void *fp,uint64_t off);`
The data 'val' is stored into the DF-frame pointed to by 'fp' at the specified offset 'off'. Note: we assume that writes are snooped by the architecture (in particular by the DTS) so that for every 64-bit word that is written the 'sc' of the DF-Thread - to which the fp belongs – is decremented.

`void *DF_TALLOC(uint64_t size, uint_8 type);`
Allocates a block of memory of 'size' words and returns the pointer (or null); 'type' specifies the special purpose memory type (see section IV). The DTS tracks the allocated memory.

`void DF_TFREE(void *p);`
Frees memory pointed to by 'p'. The DTS tracks the deallocated memory.

This API is thought as a target for compiler or programmers that are building run-time libraries. It can be mapped on any instruction set: in previous work we analyzed the possibility of implementing this API as an Instruction Set Extension called T-Star (T*) for x86_64 [2][8]: in this case this API provides more flexibility for implementations. The programmer gets access to memory regions with the specific semantics by allocating explicitly the needed memory type. DF_TREAD/DF-TWRITE operations can be easily mapped on the underlying load/store operations and their behavior will change when accessing a DF-Frame of a certain type (e.g., no 'sc' decrement for Thread Local Storage or TLS, or activating speculative store buffering for Transactional Memory).

The DTS keeps track – locally and in a distributed fashion-- of entries that we call 'continuations' which store the self-frame pointer (fp) the instruction pointer (ip), the synchronization count (sc) and other information related to the associated core where the thread runs (as outlined in Figure 1). Other memory type pointers can be stored in the DF-frame associated to the DF-Thread.

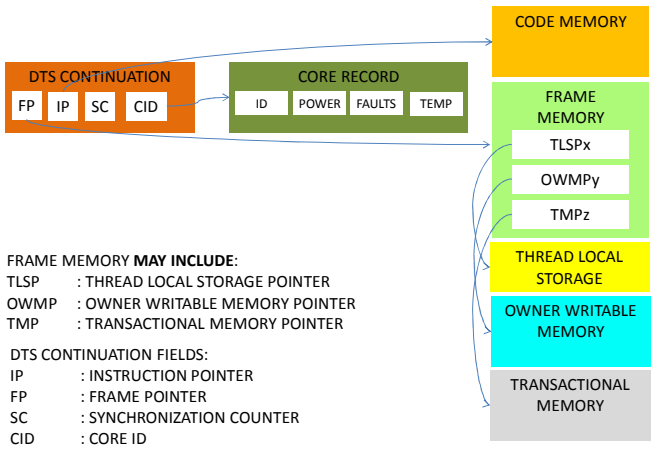

Figure 1. High level Memory Model Implementation.

IV. DF-THREAD MEMORY MODEL

The DF-Thread Memory Model (DFTMM for short) relies on the fact that in shared-memory systems (even in the non-coherent case) the memory is used to implement the communication among threads. Therefore, we can identify the following thread communication patterns:

a) **"N-to-1"**: we have N threads producing data that will be consumed later on by a single thread; this is a classical producer-consumer pattern; in order to implement this, we associate a "frame" of memory taken from a logical region that we call "Frame Memory" or FM;

b) **"1-to-1"**: we use this to indicate 'self-communication', i.e., the same thread is consuming a large portion of dynamically allocated private memory; we call the memory from this region "Private Memory" or PM (this is also known in the literature as Thread-Local Storage or TLS);

c) **"N-to-N"**: in the case when a mutable shared state is necessary for the computation, we rely on the compiler capability to identify such code and use as basic mechanism the atomic transactions provided by the Transactional Memory [17][20][21] or TM. Also note that this kind of state based computation could be theoretically treated by FM, but TM is currently more widely available [21].

d) **"1-to-N"**: communication is managed through Frame Memory; a common case is the in-place-update when a single writer wishes to make available, e.g., an array element to several consumers shortly: we suggest to manage this case through the distribution of a pointer to the element (which resides in a certain frame that could be garbage collected later on (cf. Figure 1); as of a similar definition introduced by prof. Ian Watson, we call this Owner Writable Memory (OWM).

We believe that there is a very good potential of supporting transactions through the basic mechanisms provided by our DTS, however such contribution is outside the scope of this paper, hence will not discussed further here.

One important implication of this memory model is that we are not necessarily implying hardware coherency, but the system is assumed to be consistent under a correct program (protection mechanisms may be activated through classical page-protection bits [18]).

V. ARCHITECTURAL SUPPORT TO DF-THREADS

The proposed DF-thread API (Sec. III) is also easily supported by the architecture either by a specific Instruction Set Extension (as outlined in Section V.A) or by mapping the functions to existing ISA instructions, therefore providing purely software-based implementation. The latter case will not be discussed in this paper as it involves a detailed discussion regarding the mechanisms to assure an atomic update of the synchronization count associated with each DF-Thread. In the following we investigate a hardware based support through specialized instructions (the T* instructions), the Distributed Thread Scheduler which can be thought as a co-processor that is driven by the T* instructions and a possible architecture.

A. *THE T* INSTRUCTION SET EXTENSION*

The six functions DF_TSCHEDULE, DF_TDESTROY, DF_TREAD, DF_TWRITE, DF_TALLOC and DF_TFREE introduced above can be easily mapped on the respective T* instructions TSCHEDULE, TDESTROY, TREAD, TWRITE, TALLOC, TFREE as defined in [8]. In our initial experiment we found possible to map the DF-Thread API on such instructions by using the GCC in-line assembly and properly mapping the language data types into the fixed size registers. This provides a path to map the T* instructions into higher level languages such as C in order to build the "DF-Thread library". Since the mapping is straightforward from the already provided T* specification [8], we do not detail it further. However, we now provide a prototype implementation on the http://cotson.sf.net website (see for example the TSU4 branch) and a preliminary evaluation in section VII.

B. *THE DISTRIBUTED THREAD SCHEDULER (DTS)*

In order to efficiently provide architectural support for threads scheduling to the OS, we can adopt dedicated hardware that can act as a co-processor (similarly, e.g., to a mathematical co-processor) that implements the behavior of the T* instructions. We call this co-processor Distributed Thread Scheduler or DTS. The actual on-chip implementation of the DTS is realized in a distributed and hierarchical fashion: each core in a node is equipped with a dedicated unit, called Local Thread Scheduling Unit or LTSU, which is responsible for the main scheduling activities, while a node-level unit, the Distributed Thread Scheduling Unit or DTSU, coordinates L-TSUs while keeping a global view of the node and communicates with D-TSUs of the other nodes (cf. Figure 2).

It should be noted that the distributed design (1 LTSU per core and 1 DTSU per node) avoids the creation of bottlenecks.

Every LTSU maintains the list of all threads that are locally ready for execution: this is done by assigning to threads a continuation, as defined in Section III. A continuation is a tuple of information needed to handle the thread execution during its lifetime. The LTSU may issue a request to the node's DTSU, which holds information about whether or not a new thread can be scheduled locally to the node. In this case, the request is forwarded to the LTSU of a node-local; otherwise the request is forwarded to remote DTSUs for scheduling on a core residing in another node. When an LTSU receives a request for the new thread scheduling from the DTSU, it allocates the resources needed by such thread for execution and notifies the requesting LTSU.

VI. CHIP ARCHITECTURE

In order to make implementations more attractive, a possible architecture can be based on well-known blocks (except the DTS and the Fault Detection Units DFDU, LDFU [14]). As shown, at chip level we have a hierarchical structure where a Network on Chip (NoC) connects the Nodes and the I/O hub, while memory and devices are external to the chip. Each Node consists of given number of Cores (eventually asymmetric, i.e., a mix of bigger cores and smaller cores), memory controllers, and the last level(s) of the cache hierarchy. Each core in turn consists of the main processing unit, the upper level(s) of the cache hierarchy (e.g., L1 and L2 caches) and other core level standard hardware (cf. Figure 2).

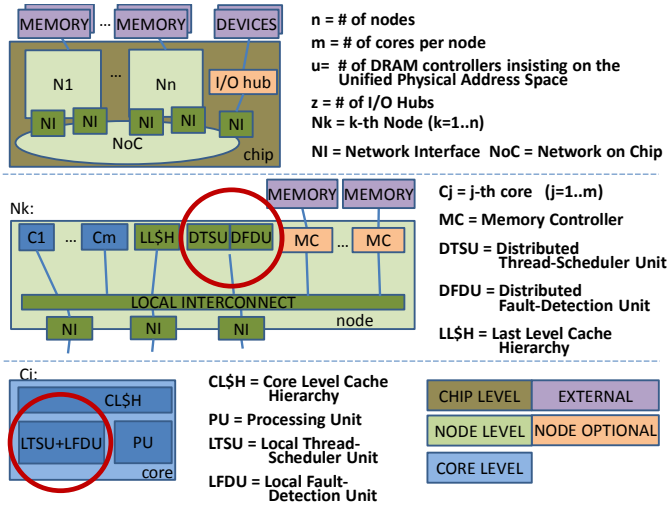

Figure 2. Chip Architecture encompassing the DTSU and LTSU in the two (red) circles. The set of DTSUs and LTSUs form the Distributed Thread Scheduler (DTS).

VII. INITIAL EXPERIMENTAL RESULTS

Our initial experiments were carried out by mapping the DF-Threads onto the described architecture as provided by an extension of the COTSon simulator [24]. In particular the tool allowed us to implement the T* extension and the DTS. Moreover, the simulator provides the capabilities of simulating cores up to 32-cores and several nodes (up to 32 in our tests) therefore achieving the modeling of a 1024-core chip.

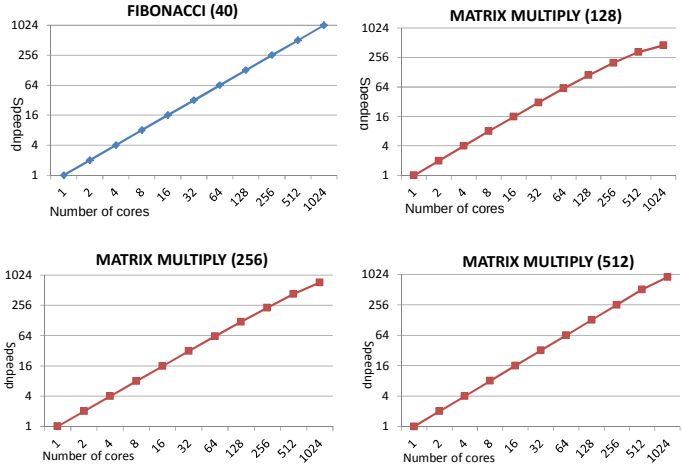

Figure 3. Speedup in case of two simple benchmarks: recursive Fibonacci with input 40 and Matrix Multiply

The initial benchmarks that we used are a recursive implementation of the Fibonacci formula, which is useful to easily generate a large number of threads (millions in our case when the input number is 40, with a cut-off threshold for recursion set to 10) and a hierarchical blocked matrix multiply to generate a larger number of data memory references (compared to the recursive Fibonacci); in this case we investigated input sizes of 128x128, 256x256, 512x512 for the two matrices to be multiplied (block size is 32).

As we can see (Figure 3), first of all the input size has to be chosen large enough to generate enough parallelism otherwise the system may seem under saturation while it is not (with smaller input number for Fibonacci, e.g., 30 we cannot reach scalability up to 1024 cores for example). This is an important indication to further investigate the scalability of the system at the 1k-core level. It has to be noted that we are increasing the input size to better investigate the architecture, rather than being limited by the parallelism of the architecture: in this way we can fairly estimate if the DF-Threads and their execution model is providing an efficient execution. Moreover, an interesting result is the possibility to scale not only within the boundary of the single core (32 cores), but also across nodes, without changing programming model or execution model, thanks to DF-Threads and their architectural support.

In particular, we also wanted to analyze how many DF-Threads we were able to generate, how many of them are waiting to receive inputs, how many of them are ready for execution. This is show in Figure 4 and Figure 5.

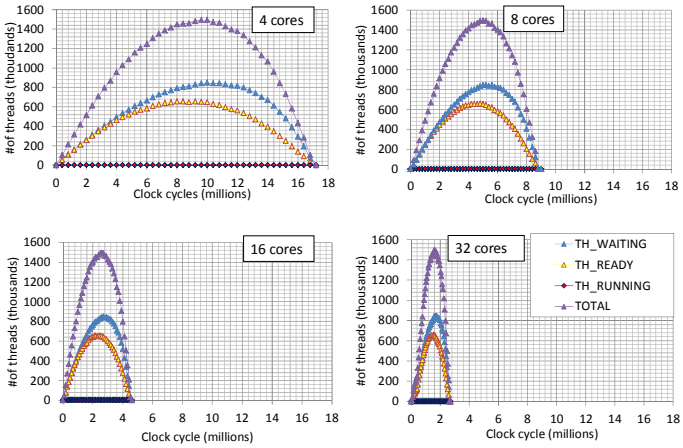

Figure 4. Fibonacci(35). From top to bottom: Total, Waiting and Ready DF-Threads (the Running ones are squashed on the horizontal axis, since the vertical scale arrives to 1.6 million threads). Core configurations are 4, 8, 16 and 32.

In particular, in Figure 4, we report the total number of DF-Threads for four architectural configuration (4, 8, 16, 32 cores). As can be seen, while the total number of threads (the highest curve with violet markers) does not change across the configuration, the availability of more cores directly translates in shorter execution time (about 17, 8.5, 4.25, 2.15 M-cycles) with an almost perfect scalability, as already noted in Figure 3. The number of executing threads in this case is completely squashed on the horizontal axis, since at most we have 32 threads running while their availability is much higher. For the Matrix-Multiply kernel we have slightly more total threads than the number of cores: still we are able to fully load the machine as can be seen in Figure 5.

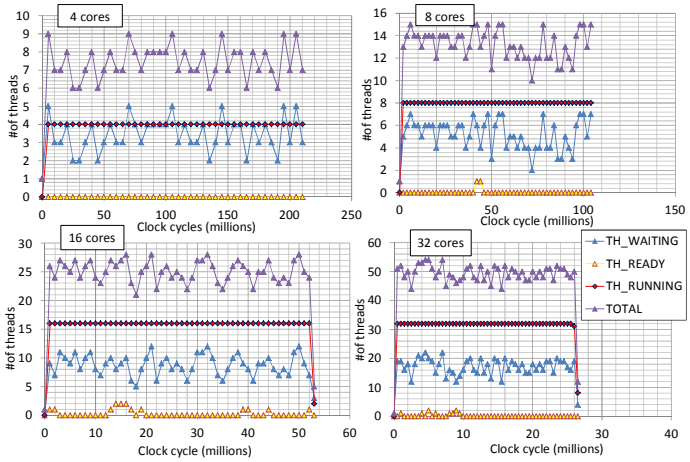

Figure 5. Matrix Multiply (256x256). From top to bottom: Total, Running, Waiting and Ready DF-Threads. The steady line at the center represents the DF-Threads effectively using the available cores. Core configurations are 4, 8, 16 and 32.

VIII. RELATED WORK

Several past efforts propose to group more instructions in a single threads, such as Macro Dataflow [5], ETS, Multithreaded Monsoon, TAM, HEP, Tera, EARTH, SDF, DDM, DTA, just to name a few (for a comprehensive and detailed discussion of them we suggest some survey papers [6][7]). DF-Threads represent a minimalistic API to support Data-Flow execution that also encompasses shared-memory computations through Transactional Memory.

The DF-Thread APIs had been proposed in the TERAFLUX project [8] for the first time and a little later in the Data-Flow Codelet model [9], and the Open Community Runtime (OCR) project [13]. In these cases, a low-level software interface to start computation on the availability of data or events has been proposed. DF-Threads provide also a definition of a general memory model while still preserving a minimalistic API consisting of only six functions.

Hierarchically Tiled Arrays (HTAs) [25] are a high-level abstraction for writing programs to execute on runtime systems based on DF-Threads or Codelets. Similarly one can use StarSs/OmpSs [15], OpenAcc [16] or OpenStream [19]. Recent proposals like OpenSPL [28] represent further effort to generate Data-Flow based execution models.

IX. CONCLUSIONS

In this paper we introduce the DF-Threads. Differently from other previous works we encompass the possibility of managing both classical Data-Flow patterns and shared memory through Transactional Memory by a unifying API that permits Data-Flow execution on hierarchical many-core architecture. The DF-Thread API and its memory model are defined. Architectural support that enables an efficient execution without redesigning completely the hardware is illustrated. DF-Threads enable to exploit Thread Level Parallelism across cores and across nodes. The initial results are quite encouraging in terms of scalability across a large number of cores such as 1k cores.

ACKNOWLEDGMENT

We acknowledge the anonymous reviewers for their precious suggestions. This work was partly funded by the European FP7 Project TERAFLUX (id. 249013).

REFERENCES

[1] J. Dongarra, et al., “The international exascale software project roadmap”, IJPCA, Vol. 25(1), pp.3-60, 2011.

[2] R. Giorgi, et l., “TERAFLUX: Harnessing dataflow in next generation teradevices”, ELSEVIER Microprocessors and Microsystems, Apr 2014.

[3] US Dept. of Energy. (Apr.14) “Advanced scientific computing research - x-stack portfolio” http://science.energy.gov/ascr/research/computer-science/ascr-xstack-portfolio/

[4] N. P. Carter, et al. “Runnemede: An architecture for Ubiquitous High-Performance Computing”, HPCA '13, IEEE Comp. Soc., pp. 198-209, 2013.

[5] V. Sarkar and J. Hennessy. “Partitioning parallel programs for macro-dataflow”, LFP '86, ACM, pp. 202-211, 1986.

[6] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, Y. Etsion, “Hybrid Dataflow/von-Neumann Architectures,” IEEE Trans. on Parallel and Distributed Systems, vol. 25, no.6, pp.1489-1509, Jun. 2014.

[7] W. M. Johnston, J. R. Paul Hanna, and R. J. Millar. “Advances in dataflow programming languages”. ACM Comp. Surv. 36,1, pp.1-34, Mar.04.

[8] R. Giorgi and et. al, “Definition of isa extensions, custom devices and external cotson api extensions,” Siena, Italy, pp. 1–78, [Online]. Available: http://www.dii.unisi.it/ giorgi/papers/Giorgi11b.pdf , Mar. 2011.

[9] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, “Using a codelet program execution model for exascale machines: Position paper,” EXADAPT 2011, pp. 64–69, June 2012.

[10] M.A., Suleman, O. Mutlu, M. K. Qureshi, Y. N. Patt, “Accelerating critical section execution with asymmetric multi-core architectures”. ASPLOS '09, ACM, pp. 253-264, 2009.

[11] Apple Grand Central Dispatch, retrieved on Aug. 2014: http://opensource.mlba-team.de/xdispatch/GrandCentral_TB_brief_20090608.pdf

[12] K. Chakraborty, P. M. Wells, G. S. Sohi, “Computation spreading: employing hardware migration to specialize CMP cores on-the-fly”,. ASPLOS’06, ACM, pp. 283-292, 2006.

[13] Building an open community runtime (OCR) framework for exascale sys.: http://sc12.supercomputing.org/schedule/event_detail.php?evid=bof219.

[14] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer, “Architectural support for fault tolerance in a teradevice dataflow system,” International Journal of Parallel Programming, pp. 1–25, 2014.

[15] A. Duran, J. M. Perez, E. Ayguad´e, R. M. Badia, and J. Labarta, “Extending the openmp tasking model to allow dependent tasks,” in OpenMP in a New Era of Parallelism. Springer, 2008, pp. 111–122.

[16] S. Bihan, “CAPS OpenACC Compilers: Performance and Portability,” Feb. 2013.

[17] M. Herlihy and J. E. B. Moss, “Transactional memory: architectural support for lock-free data structures”, ISCA '93,, 289-300, May 1993.

[18] Intel Manuals: http://developer.intel.com/products/processor/manuals/index.htm

[19] A. Pop, A. Cohen, “OpenStream: expressiveness and data-flow compilation of OpenMP streaming programs”, ACM Trans. Architec. Code Optim. 9 (4), pp. 53:1–53:25, 2013.

[20] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional Memory Coherence and Consistency”, ISCA '04, IEEE Computer Society, pp.102-111, 2004.

[21] Intel. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions. 2012.

[22] R. Giorgi, Z. Popovic, N. Puzovic, “DTA-C: A Decoupled multi-Threaded Architecture for CMP System”, SBAC-PAD-07, pp.263-270, 2007.

[23] . Sargeant, C. C. Kirkham, “Stored data structures on the Manchester dataflow machine”, SIGARCH Comput. Arch. News 14, 2 pp. 235-242, 1986.

[24] E. Argollo, P. Faraboschi, M. Monchiero, D. Ortega, COTSon: infrastructure for full system simulation. ACM SIGOPS, vol. 43, 1: 2009.

[25] B. Fraguela et al., “The Hierarchically Tiled Arrays programming approach,” in Proc. of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR), 2004, pp. 1–12.

[26] R. Giorgi, “TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", ACM Computing Frontiers, May 2012, pp. 303-304.

[27] M. Solinas et al., "The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices", DSD 2013, pp. 272-279

[28] OpenSPL, retrieved in August 2014: http://www.openspl.org/

[29] S. Weis et al., "A Fault Detection and Recovery Architecture for a Teradevice Dataflow System", DFM-2011, Oct. 2011, pp. 38-44.