Simulating a Multi-core x86_64 Architecture with Hardware ISA Extension Supporting a Data-Flow Execution Model

Nam Ho*, Antoni Portero†, Marcos Solinas‡, Alberto Scionti‡, Andrea Mondelli‡, Paolo Faraboschi§, and Roberto Giorgi ‡
* *Department of Computer Science, University of Paderborn*
Pohlweg 47-49, 33098 Paderborn, Germany, Email: namh@mail.upb.de
†*IT4Innovations, National Supercomputing Center, VSB Technical University of Ostrava*
17. listopadu 15/2172, 708 33 Ostrava Poruba, Czech Republic, Email: antonio.portero@vsb.cz
§*HP Labs*, 3000 Hanover Street. Palo Alto, CA 94304-1185. USA. Email: paolo.faraboschi@hp.com
‡*Dept. Ingegneria dell'Informazione Universita di Siena*,
Via Roma, 56 - 53100 Siena (Italy), Email:{solinas, scionti, mondelli, giorgi}@unisi.it

***Abstract*—The trend to develop increasingly more intelligent systems leads directly to a considerable demand for more and more computational power. Programming models that aid to exploit the application parallelism with current multi-core systems exist but with limitations. From this perspective, new execution models are arising to surpass limitations to scale up the number of processing elements, while dedicated hardware can help the scheduling of the threads in many-core systems. This paper depicts a data-flow based execution model that exposes to the multi-core x86_64 architecture up to millions of fine-grain threads. We propose to augment the existing architecture with a hardware thread scheduling unit. The functionality of this unit is exposed by means of four dedicated instructions. Results with a pure data-flow application (i.e., Recursive Fibonacci) show that the hardware scheduling unit can load the computing cores (up to 32 in our tests) in a more efficient way than run-time managed threads generated by programming models (e.g., OpenMP and Cilk). Further, our solution shows better scaling and smaller saturation when the number of workers increases.**

Keywords*– *many-core architecture; data-flow; fine-thread scheduling; ISA extension; model simulation;

I. Introduction

The tendency to evolve to more complex intelligent systems leads directly to an appreciable request for more and more computational powerful systems. Several applications rely on massive parallel hardware to better perform (e.g., many algorithms in machine learning such as neural networks or support vector machines, and signal processing algorithms). In future many-core systems, the number of cores will be one or two orders of magnitude larger than current multi-core systems. This tremendous increment in the number of cores is allowed by the continuous improvements in silicon manufacturing technology. However, current multi-threaded real-world applications are not fully capable to take advantage from the large parallelism exposed by the hardware. Moreover, major design constraints impose serious limitations to the ability of integrating more and more processing units on the same silicon die. Among the others, the communication and synchronization overhead becomes quickly predominant. With the aim of surpassing these limitations, new programming and execution models have been proposed [1], [29], [37], [38]. Data-flow paradigm is known to be capable of taking advantage of the full parallelism offered by the underlying hardware, by leveraging an implicit way to eliminate data dependencies and threads synchronization [5]. Although data-flow is not a new concept, only recently it has met the appropriate hardware support maturity. By introducing a hardware assisted thread scheduling unit, the efficiency and scalability of such systems can be improved. However, targeting a many-core chip embedding up to one thousand cores (as expected for future many-core processors), efficiently scheduling threads among the computing resources remains a not trivial problem.

This paper defines an abstract many-core system supporting a data-flow execution model, by resembling very closely the existing architectures (e.g., by adopting existing architectural blocks). To this end, the paper introduces a set of additional hardware units to efficiently support the scheduling and distribution of fine-grain threads. We can synthetically refer to this set of units as the hardware thread scheduler. The proposed architecture exposes the functionality of the hardware thread scheduler by means of an extension of the x86_64 instruction set architecture (ISA), called *T-Star* (T*64). Simulation results show that managing micro-threads with a specific x86_64 ISA extension provides better scaling w.r.t. standard programming models such as OpenMP and Cilk. The rest of the paper is organized as follows. Section II describes major contributions in the context of data-flow architectures and scheduling systems. Section III presents the ISA extensions we introduced to support data-flow thread scheduling at the hardware level. Section IV introduces the reference architecture chosen in this work, while section V presents experiment results. Section VI concludes the paper.

II. Related works

Data-flow principles and their architectural exploitation have been investigated in several research works. Dennis and Gao in [13] proposed a first example of the applications of the data-flow principle to design an high-efficient processor. Several data-flow projects existed during '70 and early '80s to explore the data-flow paradigm. Projects died out in the mid '80s due to the complexity of the hardware, difficulty of dealing with complex data structures, expensive of moving and copying data, communication and synchronization overheads, and programming difficulties. Etsion ets al. [31] provides an accurate vision of the current state-of-the art in data-flow programming. The authors designed a processing system composed of a standard execution pipeline augmented with a hardware unit holding information of the execution data-flow graph. Similarly, Monsoon [14], and StarT project [15] are other examples of a multi-threaded architecture employing data-flow principles. The systems use a custom pipelined processor for supporting data-flow sequencing and execution of very fine-grained threads.

More recently, other research works applied data-flow principles to modern multi-threaded multi-core architectures. These works progressively departed from the original data-flow model, with the aim of overcoming scalability limitations. Scheduled Data-Flow (SDF) architecture [16] and Decoupled Threaded Architecure (DTA) [17] decouple memory access and computations within fine-grain non-blocking threads. Properly scheduling in a distributed fashion the execution of these threads, they are able to outperform conventional superscalar architectures. Moreover, authors in [18], [19], exploited scheduled data-flow execution model on multi-threaded off-the-shelf heterogeneous processor, in contrast of the adoption of custom cores as in SDF system.

With respect to the previous works, we propose a hardware support for scheduled data-flow execution model, relying on standard off-the-shelf processing cores, with a small extension to the instruction set architecture. The underlying hardware support results in a small area cost, compared to other previous solutions, since in the proposed model, there is no need for data-flow graph storage. Similar to the DDM [1] architecture, our proposed one uses a hardware support for efficiently schedules the execution of threads. The codelet model [23], [24] has been recently proposed as an execution model targeting future exascale machine. The codelet model is similar to the one we propose. The main difference between the two models are that our scheduler is fully responsible for the scheduling operations, and that in our data-flow model the synchronization counter (the counter associated to the number of required inputs by each data-flow thread) is automatically decremented every time producer threads execute a write operation on the frame memory of consumer threads.

III. Execution model and system overview

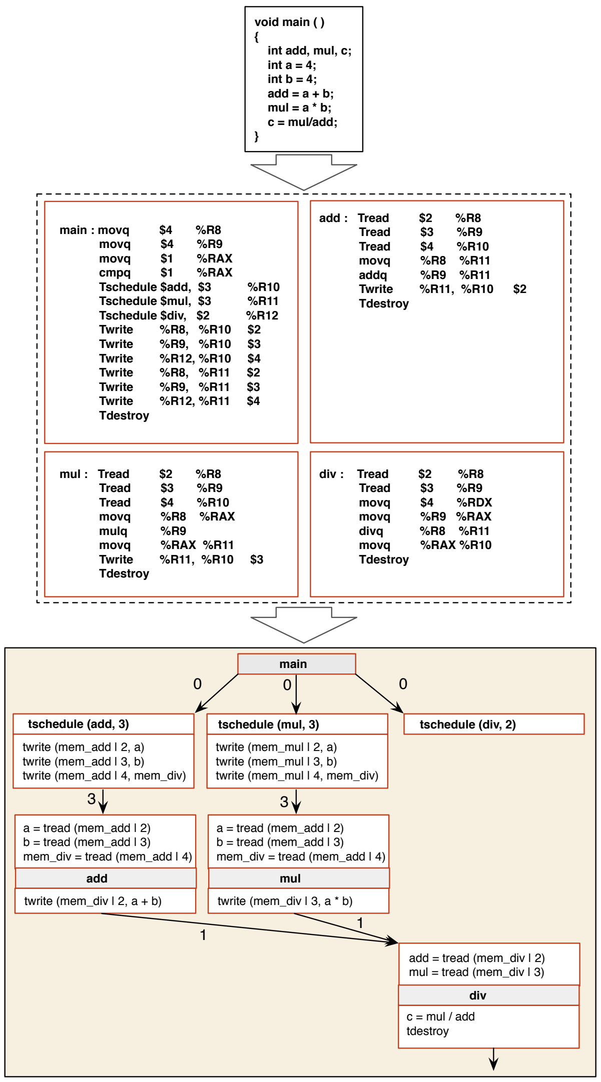
The data-flow paradigm [30], [16], [17], [20], [21], [37], [38], [40] allows applications to simply express data dependencies and synchronization among threads, mainly resorting to an explicit producer-consumer model. In our data-flow execution model the threads are organized in such a way they perform input-output operations respectively at the beginning and the end of their execution. Moreover, threads can be executed if and only if all the input data have been produced by other threads. We call this threads *Data-Flow Threads* (DF-Threads).

In our model, the execution of a DF-Thread is enabled by two conditions: (*i*) all the needed input data are available, (*ii*) the system scheduler assigns the thread to a specific computing unit. With the aim of enabling the thread execution, a counter is associated to each thread. The counter is initialized to the number of required input data, and is updated every time a producer thread generates a new input data for the consumer thread. Considering standard off-the-shelf computing cores, this mechanism becomes effective if there is a way to control the thread execution at the level of instructions. With the aim of allowing the application code to interact with dedicated scheduling units and to control thread execution (i.e., thread creation, thread removal, etc.), we introduce an extension to the instruction set architecture.

Due to the large availability and adoption of x86_64 architectures in several context (e.g., high-performance computing, general purpose desktop systems, etc.), we augmented x86_64 cores with dedicated hardware units for accelerating the scheduling and distribution of DF-Threads. These hardware units are completely aware of the above described data-flow execution model.

A. Hardware thread scheduling support

The proposed architecture is based on designs presented in other works ([7], [37], [38]). A multi-core x86_64 system is modified by augmenting the functionalities of the cores with the integration of a dedicated scheduling unit (*local scheduler* – LS). In order to ease the distribution of the threads among the cores, a *remote scheduler* (RS) is implemented. All these hardware units are responsible for allocating internal system resources during the lifetime of the threads. DF-Threads are possibly managed locally by each LS. The LS is in charge of managing input data counters for each DF-Thread that is created. In case not sufficient resources are locally available, the LS can communicate with the RS, requiring to move the thread on a different core. The RS selects a different core, trying to balance the whole system load. Similarly, every time a DF-Thread needs to write in the memory region associated to a consumer thread residing in a different core, the LS unit exchanges information with the RS unit. Programs can interact with the LS and RS using dedicate instructions. For this purpose, we created an Instruction Set Extension (ISE) called T*64 [8], [16], [7], [41]. T*64 extension consists of four instructions for generating/stopping DF-Threads, and operating on input/output data.



```
void main ( )
{
   int add, mul, c;
   int a = 4;
   int b = 4;
   add = a + b;
   mul = a * b;
   c = mul/add;
}
```

```
main : movq       $4      %R8
       movq       $4      %R9
       movq       $1      %RAX
       cmpq       $1      %RAX
       Tschedule $add,  $3          %R10
       Tschedule $mul,  $3          %R11
       Tschedule $div,  $2          %R12
       Twrite     %R8,  %R10    $2
       Twrite     %R9,  %R10    $3
       Twrite     %R12, %R10    $4
       Twrite     %R8,  %R11    $2
       Twrite     %R9,  %R11    $3
       Twrite     %R12, %R11    $4
       Tdestroy
```

```
add :  Tread      $2      %R8
       Tread      $3      %R9
       Tread      $4      %R10
       movq       %R8   %R11
       addq       %R9   %R11
       Twrite     %R11,  %R10     $2
       Tdestroy
```

```
mul :  Tread      $2      %R8
       Tread      $3      %R9
       Tread      $4      %R10
       movq       %R8   %RAX
       mulq       %R9
       movq       %RAX  %R11
       Twrite     %R11,  %R10    $3
       Tdestroy
```

```
div :  Tread      $2      %R8
       Tread      $3      %R9
       movq       $4      %RDX
       movq       %R9   %RAX
       divq       %R8   %R11
       movq       %RAX %R10
       Tdestroy
```

```
tschedule (add, 3)
twrite (mem_add l 2, a)
twrite (mem_add l 3, b)
twrite (mem_add l 4, mem_div)
```

```
tschedule (mul, 3)
twrite (mem_mul l 2, a)
twrite (mem_mul l 3, b)
twrite (mem_mul l 4, mem_div)
```

```
tschedule (div, 2)
```

```
a = tread (mem_add l 2)
b = tread (mem_add l 3)
mem_div = tread (mem_add l 4)
add
twrite (mem_div l 2, a + b)
```

```
a = tread (mem_add l 2)
b = tread (mem_add l 3)
mem_div = tread (mem_add l 4)
mul
twrite (mem_div l 3, a * b)
```

```
add = tread (mem_div l 2)
mul = tread (mem_div l 3)
div
c = mul / add
tdestroy
```

Figure 1. An example of execution of a data-flow application. The code snippet in the top box is used to generate four data-flow threads (DF-Threads – middle box). Resorting to the T*64 instructions, these four DF-Threads are dynamically scheduled by the LS/RS units as depicted in the box below.

B. Instruction Set Extension support for data-flow execution

As described in the previous section, we resort to an Instruction Set Architecture extension (ISE – Instruction Set Extension) [5] to allow programs directly interact with the hardware units introduced for accelerating DF-Thread scheduling operations. This ISE is composed of four main instructions. We overloaded the semantic of a side-effects free instruction. The advantage of using this mechanism is the absence of visible architecturally side-effects in the cores.

More specifically, managing the execution of DF-threads requires an effective mechanism to create/destroy the threads and allow them to read and write input/output data: these four operations are the core of our T*64 ISA extension. T*64 instructions are decoded locally by the computing core in the decode stage, and forwarded to the LS unit for the execution. Standard x86_64 instructions and T*64 instructions maintain separate execution paths within each core, allowing to maximize the instruction level parallelism exploitation. The four T*64 instructions are described in the following:

- *TSchedule*: this instruction allows to create a new DF-Thread by allocating the internal LS/RS resources (e.g., storage space for managing the input data counter). The instruction returns a pointer to the memory region used to store input data for the thread. Whenever a DF-Thread issues this instruction, it also specifies both the pointer to the thread code and the initial value of the input data counter. When this counter is reduced to zero, the scheduling unit changes the state of the DF-Thread from waiting to ready.
- *TDestroy*: the DF-Thread that issues this instruction is going to finish so all the resources associated to it are released.
- *TWrite*: this instruction allows DF-Threads to produce output data for other consumer DF-Threads, by writing at proper locations in the memory region associated to these threads. The producer DF-Thread specifies both a pointer to this memory region, and an offset. The scheduling unit in charge of managing the consumer DF-Thread decrements the corresponding input data counter.
- *TRead*: this instruction is executed by a DF-Thread to read input data. In order to read a specific data, the consumer thread specifies an offset within the memory region associated to it.

Figure 1 shows an example of the DF-Thread scheduling process. In the top box, a code snippet written in C language is depicted. This piece of code is parallelized, and four corresponding DF-Threads are generated. The box in the middle of the figure provides an expanded view of each DF-Thread. Several methods have been proposed to automatically extract fine-grain threads starting from a high-level description given in an imperative language. Finally, below this box there is a representation of the dynamic scheduling process applied by the LS units and the RS unit of the proposed architecture.

IV. Simulation Methodology

The experiments focus on analyzing the T*64 execution model using the COTSon simulation infrastructure [10]. COTSon uses SimNow [11] to perform behavioral simulation of the target system (i.e., functional simulation), while it implements timing models for all the components in the

TABLE I
MAIN ARCHITECTURAL PARAMETERS USED IN THE SIMULATION FRAMEWORK.

Parameter	Description
Cores	X86-64, out-of-order
Clock Frequency	1.0 GHz
L1 Cache	Private 32Kbytes I + 32Kbytes D, 2 way s.a., 3 cycle latency
L2 Cache	512 Kbytes, 4 ways s.a., 5 cycles latency
Block Size	64 bytes
Main Memory	1 GByte, 16 cycles latency
Bus	MOESI protocol
L1 TLB	32 entries. for both I-TLB and D-TLB
L2 TLB	512 entries (unified TLB)

target system. The architecture considered for this study is a single-node multi-core system. The architecture considered for this study is a single-node multi-core system. Architectural details of the simulated system are reported in table I.Standard x86_64 cores are configured to have separated instructions and data Translation Lookaside Buffer (i_TLB and d_TLB respectively) which both connect to a second level TLB (L2-TLB). Besides TLBs, the standard cores have separated first level instruction and data caches (i.e., L1-IC and L1-DC) which both connect to a second level cache (L2). Finally, L2-TLB and L2 cache are connected to a coherent bus implementing the MOESI protocol. This bus also provides the access to the memory controller of a DRAM. The clock frequency of the x86_64 cores is 1.0 GHz. In order to correctly simulate the proposed execution model, we implemented a dedicated behavioral and timing model for the local schedulers and for the remote scheduler unit. Finally, latencies of the memory components have been obtained with Cacti 6.0 tool [36]. For the simulations, we consider a system implemented using 32nm technology.

The simulation of the T*64 ISA in the x86-64 guest system is performed at cycle accurate level. We do not neglected any event or bus transaction. Deep analysis were performed to collect bus latency and the cache utilization for our T*64 execution model. For T*64 instructions, we measured the latencies occurring between the decoding of the instruction at the level of the LS unit till the completion of the operations (e.g., write operation in a specific memory location, etc.).

V. EXPERIMENTAL RESULTS

The simulation of data-flow benchmarks such as *Recursive Fibonacci* allows us to evaluate the benefit of adopting a hardware scheduling component w.r.t. software-based or mixed software-hardware based approaches for distributing and scheduling fine-grain threads. Pure data-flow applications following the exposed execution model exhibit an interesting feature: they are able to generate a high number of threads during the application lifetime. Thus, they represent a starting point for evaluating the proposed execution model and scheduling hardware support.

Our results show that the choice of deploying a hardware scheduler is most effective when the number of running threads increases. The following analysis focuses on a single-node multi-core system (i.e., 32 cores). This choice allowed us to identify potential bottlenecks with small system configuration. An efficient solution for the single-node case can be further extended to multiple nodes in future works (essentially by extending the current scheduler in order to make RS units, that are distributed among various nodes, to communicate each other). According to our estimations the area overhead is equal to 3.8%, making the solution feasible for future many-core systems.

A. *Performance evaluation of T*64*

Experiments have been performed on a single-node system configured with 1, 4, 8, 16, and 32 cores (for the kernel implementation with OpenMP and Cilk we set the number of worker threads equal to the number of cores). The results of the execution of the Recursive Fibonacci kernel with input value $n = 35$ are shown in figure 2. The execution time is normalized to the sequential execution time and compared with that of OpenMP [12] and Cilk [2]. The reference value is given by the time for the sequential execution. For the data-flow version of the kernel we considered two possible implementations: in *T*64 default* implementation each T*64 instruction takes several cycles to complete (i.e., 5, 1, 4, and 2 cycles respectively for the TSchedule, TRead, TWrite and TDestroy), while for the *T*64 optimistic* it is assumed that they complete in one cycle. The optimistic implementation relies on the assumption that the hardware units devoted to the execution of T*64 instructions are able to efficiently exploit the ILP.

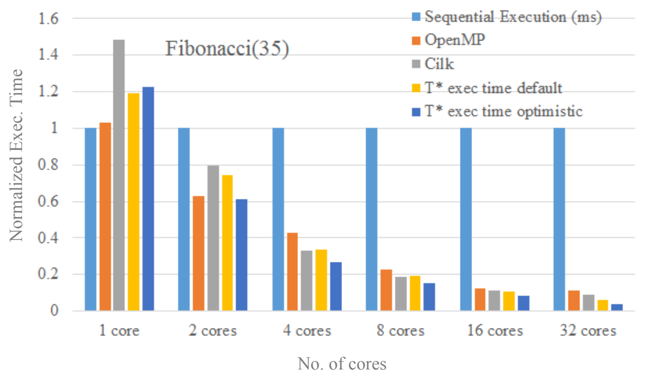


Figure 2. Normalized execution time for the Recursive Fibonacci kernel with input $n = 35$. Bars in the graph shows the relative execution time for different parallel implementations of the kernel: OpenMP, Cilk, and T*64.

The first important observation is that speed up for all solutions is similar till the number of workers arrives to sixteen. But, when the number of workers is increased to 32 the data-flow scheme presents better scaling. There is space for design exploration between *T*64 default* and *T*64 optimistic* when the number of cores becomes higher than 32 (as in the case of multi-node configurations). In fact, our execution model is build in such a way it can dynamically spawn and distribute threads among all the available cores. Thus, it is possible to say that if more cores are integrated in the architecture, the scaling can be greatly improved. We also observed that with the configuration we considered, the bus interconnection is able to handle all requests without saturation.

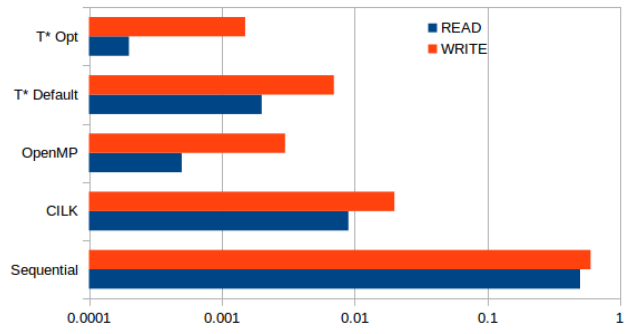


Figure 3. L1-DC miss-rate for the execution of recursive Fibonacci with input $n = 35$. The bars show the average miss-rate in the first level data cache when read and write operations are considered. The plot allows to compare the impact of different programming/execution models on L1 data cache.

As the reader can see, the OpenMP execution scales, but there is a saturation when the number of threads becomes higher than 16. Cilk execution model exhibits a scale up improvement when the number of workers is relatively high (>16), but there is still a saturation when number of threads equals 32. On the contrary, the proposed execution model exhibits a clear improvement when the number of cores increases, e.g., 32 cores or more (thus the number of threads that can be in the running state at the same time). We have also to consider that for the selected kernel application the memory regions associated to each DF-Thread are bigger than the actual used size. In fact, we managed memory regions with a minimum size of 512 bytes (64 values, each represented on 64 bit) while the benchmark generates no more than 2 input values for each thread. From this perspective, we can considered the default execution as an upper limit. We can also claim that the behavior of the default execution is similar to some well known multi-threading execution models such OpenMP and Cilk, but it has better performance in multi-cores systems when the number of cores is higher than 16.

Finally, figure 3 shows the impact of our execution model in terms of cache misses on the first level of the cache memory hierarchy. Since the data-flow paradigm is based on the explicit movement of input data between producer and consumer threads, we analyzed the impact on the L1 data cache (L1-DC). The results refer to the comparison of Recursive Fibonacci with input $n = 35$, implemented with our execution model, OpenMP and Cilk. As the reader can see, these results are promising since we obtained a lower cache miss-rate (for the optimistic execution). The main reasons are because the amount of data exchanged among DF-Threads is small enough to fit in the lower level of the cache hierarchy (L1-DC), and also because we have thousand of ready threads that are preemptively loaded in the higher cache levels.

VI. Conclusions and Future Work

In this paper a data-flow based execution model and its hardware support is presented. The paper depicts the execution model based on current x86_64 cores with some extra hardware infrastructure. We showed that the x86_64 architecture with a data-flow ISA extension (T*64 in this work) has advantages against OpenMP and Cilk for multi-core systems, when the number of cores is relatively high (>16). In these case scenarios our data-flow model exhibits a clear better scaling. Simulations indicated that T*64 is able to scale better than the current state-of-the-art programming/execution models (e.g., OpenMP and Cilk).

At the beginning of the paper we showed that T*64 relies on the execution of fine-grain threads and that the thread scheduler units can manage a huge number of these threads (referred to DF-Threads in the paper).

These results help to investigate more future architecture optimizations and design exploration. We observed that T*64 ISA extension and its related data-flow execution model have a lot of potential in dynamically spawning and distributing threads. Our future works are focusing on multi-node configuration characterization, and the exploration of the presented execution model for improving energy save and reliability.

Acknowledgment

We thank Prof. Bruce Jacob for his comments and useful suggestions. This article was supported by Operational Programme Education for Competitiveness and co-financed by the European Social Fund within the framework of the project New creative teams in priorities of scientific research, reg. no. CZ.1.07/2.3.00/30.0055, by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by the project Large infrastructures for research, development and innovation of Ministry of Education, Youth and Sports of Czech Republic with reg. no. LM2011033,and by the European FP7 projects HARPA id. 612069 and TERAFLUX id. 249013.

REFERENCES

[1] K. Stavrou and D. Pavlou and M. Nikolaides and P. Petrides and P. Evripidou and P. Trancoso and Z. Popovic and R. Giorgi, *Programming Abstractions and Toolchain for Dataflow Multithreading Architectures*. IEEE Proceedings of the Eighth International Symposium on Parallel and Distributed Computing: Lisbon, 2009.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 207–216, 1995.

[3] Bernhard Fechner, Arne Garbade, Sebastian Weis, Theo Ungerer: Fault detection and tolerance mechanisms for future1000 core systems. HPCS 2013: 552-554

[4] A. Portero, A. Scionti, M. Solinas and H. Nam, R. Giorgi, *Simulation infrastructure for the next kilo x86-64 Chips*. HiPEAC ACACES: Fiuggi (Italy), 2012.

[5] Z. Yu, A. Righi, R. Giorgi, *A Case Study on the Design Trade-off of a Thread Level Data Flow based Many-core Architecture*. Future Computing: Rome (Italy), 2012.

[6] A. Portero, Z. Yu,R. Giorgi, "TERAFLUX: Exploiting tera-device computing challenges", Procedia Computer Science 7, 146-147.

[7] A. Portero, A. Scionti, Z. Yu, P. Faraboschi, C. Concatto, L. Carro, A. Garbade, S. Weis, T. Ungerer, R. Giorgi, *Simulating the Future kilo-x86-64 core Processors and their Infrastructure*. 45th Annual Simulation Symp. (ANSS12): Orlando (FL), 2012.

[8] A. Portero Z. Yu, R. Giorgi, *T-Star (T*): An x86-64 ISA Extension to support thread execution on many cores*. HiPEAC ACACES: Fiuggy (Italy), 2011.

[9] A. Bardine, P. Foglia, F. Panicucci, J. Sahuquillo, M. Solinas, *Energy Behaviour of NUCA caches in CMPs*. 14th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools (DSD2011), Augustu 31, September 2, 2011, OULU, Finland, pp: 746-753. ISBN: 978-0-7695-4494-6. IEEE CS, Los Alamitos, CA, 2011.

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

[11] *AMD SimNow Simulator 4.6.1 User's Manual, November 2009.*. November, 2007.

[12] OpenMP Architecture Review Board, *OpenMP: Application Program Interface Version 3.0*, http://www.openmp.org/mp-documents/spec30.pdf , May, 2008.

[13] J. B. Dennis and G. R. Gao, *An Efficient Pipelined Dataflow Processor Architecture*, IEEE, 1988.

[14] G. Papadopoulos and D. Culler, *Monsoon: An Explicid Token Store Architecture*, Proc. 17th Ann. Int'l Symp. Computer Architecture (ISCA), pp. 82-91, May 1990.

[15] R. S. Nikhil, G. M. Papadopoulos and Arvind **T: A Multithreaded Massively Parallel Architecture*, Proc. Int'l Symp. Computer Architecture (ISCA), pp. 156-167, 1992.

[16] K. M. Kavi and R. Giorgi, and J. Arul *Scheduled dataflow: Execution paradigm, architecture, and performance evaluation*, IEEE Transaction on Computers, 50(8):834-846, aug 2001.

[17] R. Giorgi, Z. Popovic, and N. Puzovic, *Dta-c: A decoupled multi-threaded architecture for cmp systems*, Volume 0, pages: 263-270. Los Alamitos, CA, USA, 2007, IEEE Computer Society.

[18] R. Giorgi, Z. Popovic, and N. Puzovic, *Implementing fine/medium grained tlp support in a many-core architecture*, In Proceedings of 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulations, SAMOS 2009, pages 78-87, Samos, Greece, Jul 2009, Springer.

[19] R. Giorgi, Z. Popovic, and N. Puzovic, *Introducing hardware support for the cell processor*, In Proceedings of IEEE International IEEE Workshop on Multi-Core Computing Systems, pages 657-662, Fukuoka, Japan, march 2009.

[20] H. Hum et al., *A design Study of the EARTH Multiprocessor*. Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '95), pp. 59-68, June 1995.

[21] C. Kyriacou, P. Evripidou and P. Trancoso, *Data-Driven Multithreading Using ConventionalMultiprocessors*. IEEE Transactions on Parallel and Distributed Systems, Vol. 17, No. 10, October 2006.

[22] D. Culler et al., *TAM: A Compiler Controlled Threaded Abstract Machine*, J. Parallel and Distributed Computing, Volume 18, No. 3, pages 347-370, 1993.

[23] G. R. Gao, J. Suetterlein and S. Zuckerman, *Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond*. Technical Memo, april 2011.

[24] S. Zuckerman, J. Suetterlein, R. Knauerhase and G. R. Gao, *Using a code let 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, EXADAPT '11, pages 64-69, New-York, NY, USA, 2011, ACM.

[25] P. Radojkovic et al., *Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors*, IEEE Transactions on Parallel and Distributed Systems, 2012.

[26] A. Annamalai, R. Rodrigues, I. Koren and S. Kundu, *Dynamic Thread Scheduling in Asymmetric Multicores to Maximize Performance-per-Watt*, Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2012 IEEE 26th International , vol., no., pp.964-971, 21-25 May 2012.

[27] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, Y. Etsion, *Hybrid Dataflow/Von-Neumann Architectures*. Parallel and Distributed Systems, IEEE Transactions on Volume: PP , Issue: 99, 2013

[28] Exploiting Dataflow Parallelism in Teradevice Computing.http://www.teraflux.eu, 2010-2014.

[29] Borkar, Shekhar, "Thousand core chips: a technology perspective", Proceedings of the 44th annual Design Automation Conference, year2007 pages: 746-749

[30] Jack B. Dennis: First version of a data flow procedure language. Symposium on Programming 1974: 362-376

[31] Daniel Sanchez, Christos Kozyrakis, "ZSim: Fast and Accurate microarchitectural Simulation of Thousand-Core Systems", pages: 475-486, year 2013, ISCA.

[32] Fahimeh Yazdanpanah, et al. "Hybrid Dataflow/von-Neumann Architectures",Parallel and Distributed Systems, IEEE Transactions on (Volume:PP , Issue: 99), April 2013, doi: 10.1109/TPDS.2013.125

[33] Feng Li, Antoniu Pop and Albert Cohen, "Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs", IEEE Micro, vol. 32, num. 4, issn 0272-1732, year 2012, pages 19-31, IEEE Computer Society.

[34] Michael J. Flynn, "Computer Architecture, pipelined and parallel processor design", Jpnes and Bartlett Publishers, London UK, 1995.

[35] svn co https://svn.code.sf.net/p/cotson/code cotson (May 2014)

[36] http://www.cs.utah.edu/~rajeev/cacti6/ (August 2014)

[37] M. Solinas et al., "The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices", Proceedings - 16th Euromicro Conf. on Digital System Design, DSD 2013, Santander, Spain

[38] R. Giorgi et al., "TERAFLUX: Harnessing dataflow in next generation teradevices ", ELSEVIER Microprocessors and Microsystems , 2-s2.0-84899772882, Apr 2014.

[39] S. Weis, A. Garbade, B. Fechner, A. Mendelson and R. Giorgi, T. Ungerer, "Architectural Support for Fault Tolerance in a Teradevice Dataflow System", Springer Int.l Journal of Parallel Programming, no. 0, May 2014, pp. 1-25.

[40] R. Giorgi et al., An introduction to DF-Threads and their execution model, Parallel and Programming Models (MPP-2014), Paris, October 2014, pp. 1-6.

[41] Roberto Giorgi, "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", ACM Computing Frontiers, Cagliari, Italy, May 2012, pp. 303-304.