Hybrid Dataflow/von-Neumann Architectures

FAHIMEH YAZDANPANAH, CARLOS ALVAREZ-MARTINEZ,
DANIEL JIMENEZ-GONZALEZ, YOAV ETSION, *Member, IEEE,*

Abstract—General purpose hybrid dataflow/von-Neumann architectures are gaining attraction as effective parallel platforms. Although different implementations differ in the way they merge the conceptually different computational models, they all follow similar principles: harness the parallelism and data synchronization inherent to the dataflow model, yet maintain the programmability of the von-Neumann model. In this paper, we classify hybrid dataflow/von-Neumann models with two different taxonomies: one based on the execution model used for inter and intra block execution, and the other based on the integration level of both control and dataflow execution models. The paper reviews the basic concepts of von-Neumann and dataflow computing models, highlights their inherent advantages and limitations, and motivates the exploration of a synergistic hybrid computing model. Finally, we compare a representative set of recent general purpose hybrid dataflow/von-Neumann architectures, discuss their different approaches, and explore the evolution of these hybrid processors.

Index Terms—Dataflow Architectures, von-Neumann Model, Parallel Processors, Hybrid Systems, Scheduling and Task Partitioning

1 Introduction

Power-efficiency is today one of the main challenges in the computer architecture. One of the trends to overcome this challenge is the use of homogeneous and heterogeneous multi-core architectures that help to: (1) use more power-efficient cores, and (2) exploit the existing parallelism on the applications. This multi-core architectures, conventionally, are based on the von-Neumann (traditional control flow) computing model, which is inherently sequential because of its use of a program counter and an updateable memory. Nevertheless, the von-Neumann computing model can exploit some limited instruction level parallelism (ILP), data level parallelism (DLP), and thread level parallelism (TLP). However, DLP and TLP should be explicitly expressed by the programmer and/or compiler, and ILP is limited by the sequential execution of the instructions.

The dataflow model is a recurrent alternative to the von-Neumann execution model. The dataflow computing model is known to overcome the limitations of the traditional control flow model by fully exploiting the parallelism inherent in programs. In the dataflow model, the operands trigger the execution of the operation to be performed on them. In other words, dataflow architectures use the availability of data to fetch instructions rather than the availability of instructions to fetch data. Unlike the von-Neumann model, the dataflow model is neither based on memory structures that require inherent state transitions, nor does it depend on history sensitivity and program counter to sequentially execute a program. These properties allow the use of the model to represent maximum concurrency to the finest granularity, and facilitate dependency analysis among computations.

In this sense, the dataflow model holds the promise of an elegant execution paradigm with the ability to exploit the inherent parallelism available in applications. Furthermore, this model is self-scheduled and more power-efficient than the control flow model, which shows inefficiencies [42], [52], [57]. However, although the benefits of the dataflow have been known for a long time, this model has not yet been fully exploited for commercial systems. In fact, implementations of the model have failed to deliver the promised performance because the dataflow model has some inefficiencies and limitations. One significant problem of the dataflow model is its inability to effectively manage data structures, memory organizations and traditional programming languages.

Therefore, in order to increase the performance and power efficiency of multi-core systems, those systems can be designed as hybrid architectures that combine the dataflow and von-Neumann models of computation. The convergence of the dataflow and control flow execution models allows for the incorporation of conventional control flow execution into the dataflow approach, or exploiting a dataflow approach in von-Neumann architectures. This alleviates the inefficiencies associated with both methods. Hybrid dataflow/von-Neumann models, therefore, bind the power of the dataflow model for exposing parallelism together with the execution efficiency of the von-Neumann model in order to overcome the limitations of both models. While different hybrid implementations differ in the way they merge the two conceptually different execution models, they all follow similar principles.

The objective of this paper is to provide a better understanding of the evolution of the hybrid models and their main characteristics. We classify them with two taxonomies: one based on the execution model used for inter and intra block execution, and the other based on the integration level of both the control flow and the dataflow models. Using those taxonomies, we classify a representative set of recent (works on 2000 year or later) general purpose hybrid models, not presented on other dataflow surveys [101], [109], [110] to the best of our knowledge, summarize their main features, and compare their benefits and issues. However, in order to have a complete historical point of view, we also describe some of the previous main contributions on hybrid models. On the other hand, to keep the length of this survey at bay, software frameworks and specific purpose dataflow accelerators are left beyond the scope of the paper.

The rest of the article is organized as follows: Section 2 discusses the von-Neumann (control flow) computing model.

- *F. Yazdanpanah, C. Alvarez-Martinez, and D. Jimenez-Gonzalez are with the Universitat Politècnica de Catalunya (UPC) and Barcelona Supercomputing Center (BSC), Barcelona 08034, Spain. E-mail: {fahimeh,calvarez,djimenez}@ac.upc.edu*
- *Y. Etsion is with the Electrical Engineering and Computer Science Departments, Technion – Israel Institute of Technology, Haifa 32000, Israel E-mail: yetsion@tce.technion.ac.il*

Section 3 overviews the dataflow computing model as well as different dataflow architectures. Section 4 presents hybrid dataflow/von-Neumann models, and classifies them with two taxonomies. In Section 5, we describe some recent general purpose hybrid dataflow/von-Neumann architectures. A comparison and discussion on main features of recent hybrid architectures and their common trends are done in Section 6. Finally, we conclude in Section 7.

2 The von-Neumann Computing Model

The von-Neumann computation model [128] is the most common and commercially successful model to date. The main characteristic of this model is a single separate storage structure (the memory) that holds both program and data. Another important characteristic is the transfer of control between addressable instructions, using a program counter (PC). The transfer is either implicit (auto-increment of PC) or through explicit control instructions (jumps and branches, assignment to PC). That is the reason the von-Neumann model is commonly referred to as control flow model.

A key tenet of the model is the set of memory semantics it provides in which loads and stores occur in the order in which the PC fetched them. Enforcing this order is required to preserve true (read-after-write), output (write-after-write), and anti (write-after-read) dependences between instructions.

Also, the serial execution of instructions is a hallmark of the von-Neumann architecture. However, this simplistic sequential execution along with data, control and structural hazards during the execution of instructions may be translated into an under-utilization of the hardware resources. In that sense, exploiting parallelism at different granularities: instruction level parallelism (`ILP`), data level parallelism (`DLP`), and thread level parallelism (`TLP`), is a mechanism to increase hardware resources utilization.

Pipelined (IBM Stretch 1959 [13]) and superscalar [3] processors that try to process several instructions at the same time are the most common examples of **ILP**. Arguably the most notable class of superscalar processors is that of the dynamically scheduled Out-of-Order processors [92] that maintain a window of pending instructions dispatching them in dataflow manner. In all these processors, parallelism is further enhanced using a set of techniques such as register renaming, branch prediction and speculative execution, that are used in addition to dynamically dispatching independent instructions, in parallel, to multiple functional units (see details in Section 5.1). Another way of exploiting ILP is that of the very long instruction word (VLIW) processors [37]. The explicitly parallel instructions sets for VLIW enable the compiler [32] to statically express instruction independence in the binary code, reducing the necessary hardware support for dynamically managing data and control hazards in Out-of-Order processors.

Architectures with **DLP** apply a single operation to multiple, independent data elements. Probably the most common examples of DLP are the single instruction multiple data (SIMD) extensions. SIMD extensions are mechanisms that statically express parallelism in the form of a single instruction that operates on wide, multi-element registers (a method that is sometimes referred to as sub-word parallelism). These extensions appeared in supercomputers such as the Thinking Machines CM-1 [56] and CM-2 [20], and are now ubiquitous in all general purpose processors. A derivative of SIMD processors, known as the single instruction multiple thread (SIMT) architecture, is nowadays common in graphics processing units (GPUs) [87].

Finally, **TLP** architectures, or multi-threading, is applied by executing parallel threads on separate processing units. Nevertheless, some architectures utilize this coarse-grain parallelism to hide memory latencies and improve the utilization of hardware resources by interleaving multiple threads on a single physical processor. This technique is known as simultaneous multi-threading (SMT) [124], [130] and has been implemented in large machines such as HEP [112] and Tera [5] (as well as many others [2], [76], [129]). SMT has even made it to consumer products, starting with the Pentium 4 [81] and Power 5 [18] processors. However, despite all these efforts, effective utilization of parallel von-Neumann machines is inherently thwarted by the need to synchronize data among concurrent threads. Thread synchronization and memory latencies were identified by Arvind and Iannucci [8] as the fundamental limitations of multiprocessors.

The need for efficient data synchronization has grave programmability implications, and it has put emphasis on the cache coherency and consistency in shared-memory machines, particularly as the number of processing units continuously increase [15]. Transactional memory architectures [54] aim to somewhat alleviate that problem by providing efficient and easy-to-use lock-free data synchronization. Alternatively, speculative multithreading architectures exploit TLP dynamically scheduling the threads in parallel [114], as Out-of-Order architectures for instructions, masking the synchronization issues. Experience shows that multithreaded control flow machines are feasible, though some doubt their scalability due to two major issues that limit their parallel processing capabilities: memory latency and synchronization.

In summary, improvements of memory system, ILP, DLP and TLP significantly reduce the memory latency issue of von-Neumann architectures but those are still limited by the execution in control flow manner. On the other hand, the dataflow architectures can overcome this limitation thanks to the exploitation of the implicit parallelism of programs [8], [24].

3 The Dataflow Computing Model

The dataflow computing model represents a radical alternative to the von-Neumann computing model. This model offers many opportunities for parallel processing because it has neither a program counter nor a global updatable memory, i.e., the two characteristics of the von-Neumann model that inhibit parallelism. Due to these properties it is extensively used as a concurrency model in software and as a high-level design model for hardware.

The principles of dataflow were originated by Karp and Miller [66]. They proposed a graph-theoretic model for the description and analysis of parallel computations. Soon later, in the early 1970s, the first dataflow execution models were developed by Dennis [27] and Kahn [65]. Dennis originally applied the dataflow idea to the computer architecture design while Kahn used it in a theoretical context for modelling concurrent software.

The dataflow model is self-scheduled since instruction sequencing is constrained only by data dependencies. Moreover, the model is asynchronous because program execution is driven only by the availability of the operands at the inputs to the functional units. Specifically, the firing rule states that an instruction is enabled as soon as its corresponding operands are present, and executed when hardware resources are available. If several instructions become fireable at the same time, they can be executed in parallel. This simple principle provides the

potential for massive parallel execution at the instruction level. Thus, dataflow architectures implicitly manage complex tasks such as processor load balancing, synchronization, and accesses to common resources.

A dataflow program is represented as a directed graph, referred to as a *dataflow graph* (DFG). This consists of named nodes and arcs that represent instructions and data dependencies among instructions respectively [25], [67]. Data values propagate along the arcs in the form of packets, called *tokens*. A DFG can be created at different computing stages. For instance, it can be created for a specific algorithm used for designing special-purpose architectures (common for signal processing circuits). However, most dataflow-based systems convert a high-level code into DFG at compile time, decode time, or even during execution time, depending on the architecture organization. Unlike control flow programs, binaries compiled for a dataflow machine explicitly contain the data dependency information.

In practice, real implementation of the dataflow model can be classified as static (single-token-per-arc) and dynamic (multiple-tagged-token-per-arc) architectures. The first dataflow architecture [29] followed the static model. This approach allows at most one token to reside on any arc. This is accomplished by extending the basic firing rule as follows: A node is enabled as soon as tokens are present on its input arcs and there is no token on any of its output arcs [30]. To implement the restriction of having at most one token per arc and to guard against non-determinacy, extra reverse arcs carry acknowledge signals from consuming to producing nodes [30].

The implementation of the static dataflow model is simple, but since the graph is static, every operation can be instantiated only once and thus loop iterations and subprogram invocations can not proceed in parallel. Figure 1-a shows an example of static dataflow graph for computing a loop which is executed N times sequentially (note that in this figure, the graph for controlling iteration of the loop is not illustrated). Despite this drawback, some machines were designed based on this model, including the MIT Dataflow Architecture [29], [31], DDM1 [26], LAU [96], and HDFM [125].

The dynamic dataflow model tries to overcome some of the deficiencies of static dataflow by supporting the execution of multiple instances of the same instruction template, thereby supporting parallel invocations of loop iterations and subprogram. Figure 1-b shows the concurrent execution of different iterations of the loop. This is achieved by assigning a *tag* to each data token representing the dynamic instance of the target instruction (e.g. a_1, a_2, ...). Thus, an instruction is fired as soon as tokens with identical tags are present at each of its input arcs. This enabling rule also eliminates the need for acknowledge signals, increases parallelism, and reduces token traffic. Dynamic dataflow machines employ two types of control instructions: *Data-steering instructions* and *Tag management instructions*. *Data-steering instructions* explicitly guide data values to the correct path after a branch, which is a control flow instruction. Each live value requires its own data-steering instruction [27]. *Tag management instructions* are inserted into tagged-token dataflow programs to differentiate between multiple dynamic instances of named program values (e.g. variables in simultaneously executing iterations of a loop). Notable examples of this model are the Manchester Dataflow Machine [51], the MIT Tagged-Token [7], DDDP [71] and PIM-D [63].

The dynamic dataflow can execute out-of-order, bypassing any token that requires complex execution and that delays the rest of the computation. Another noteworthy benefit of the tagged-token model is that little care must be taken to ensure that tokens remain in order.

The main disadvantage of the dynamic model is the extra overhead required to match tags on tokens. To reduce the execution time overhead of matching tokens, dynamic dataflow machines require expensive associative memory implementations [51]. One notable attempt to eliminate the overheads associated with the token store is the *Explicit Token Store* (ETS) [23], [55]. The idea is to allocate a separate memory frame for every active loop iteration and subprogram invocation. Since frame slots are accessed using offsets relative to a frame pointer, the associative search is eliminated. To make that concept practical, the number of concurrently active loop iterations must be controlled. Hence, the condition constraint of k-bounded loops was proposed [10], which bounds the number of concurrently active loop iterations. The Monsoon architecture [90] is the main example of this model.

The dataflow model holds the promise of an elegant execution paradigm with the ability to exploit inherent parallelism available in applications. However, implementations of the model have failed to deliver the promised performance due to inherent inefficiencies and limitations. For one, the static dataflow is unable to effectively uncover large amount of parallelism in typical programs. On the other hand, dynamic dataflow architectures have been limited by prohibitive costs linked to associative tag lookups, both in terms of latency, silicon area, and power consumption.

Another significant problem is that dataflow architectures are notoriously difficult to program because they rely on specialized dataflow and functional languages. Dataflow languages are required in order to produce large dataflow graphs that expose as much parallelism to the underlying architecture. However, these languages have no notion of explicit computation state, which limits the ability to manage any non-degenerate data structures. To overcome these limitations, some dataflow systems include specialized storage mechanisms that preserve the single assignment property, such as the I-structure [9]. Nevertheless, these storage structures are far from generic and their dynamic management complicates the overall design.

In contrast, imperative languages such as C, C++, or Java explicitly manage machine state through load/store operations. This modus operandi decouples the data storage from its producers and consumers, and thereby conceals the flow of data and makes it virtually impossible to generate effective (large) dataflow graphs. Furthermore, the memory semantics of C and C++ support arithmetic operations on memory pointers, which results in memory aliasing — where different semantic names can refer to the same memory location. Memory aliasing cannot be resolved statically, thus further obfuscating the flow of data from between producers and consumers. Consequently,

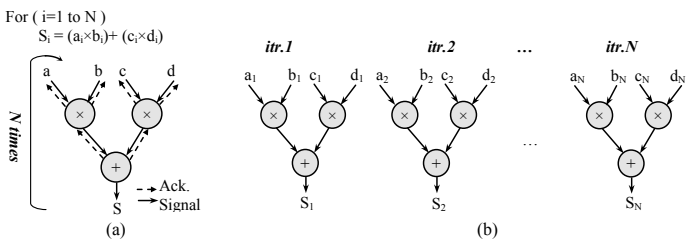

Fig. 1. DFG of a loop using (a) the static dataflow model and (b) the dynamic dataflow model.

dataflow architectures do not effectively support imperative languages.

In summary, the dataflow model is effective in uncovering parallelism, due to the explicit expression of parallelism among dataflow paths and the decentralized execution model that obviates the need for a program counter to control instruction execution. Despite these advantages, programmability issues limit the usefulness of dataflow machines. Moreover, the lack of a total order on instruction execution makes it difficult to enforce the memory ordering that imperative languages require. Although this section has presented key features and characteristics of the dataflow model and its limitations, a complete survey of the model is beyond the scope of this paper. For further reference, we refer the interested reader to more complete literature on the subject [85], [115], [126].

4 Hybrid Dataflow/von-Neumann Models

The inherent limitations of both dataflow and von-Neumann execution models motivate the exploration of a convergent model that can use synergies to leverage the benefits of both individual models.

Therefore, the hybrid models try to harness the parallelism and data synchronization inherent to dataflow models, while maintaining existing programming methodology and abstractions that are largely based on von-Neumann models. While different hybrid implementations differ in the way they merge the two conceptually different models, they all follow similar principles.

Most notably, hybrid models alleviate the inefficiencies associated with dataflow model either by increasing the basic operation granularity, or by limiting the size of the DFG. Additionally, they incorporate control flow abstractions and shared data structures. As a result, different hybrid architectures employ a mix of control flow and dataflow instruction scheduling techniques, using different partial scheduling methods. Also, in the hybrid models, nodes of a DFG vary between a single instruction (fine-grain) to a set of instructions (coarse-grain).

Another important aspect in which the benefits of hybrid models are most evident is that of their memory models. Hybrid models combine single assignment semantics, inherent to dataflow, with consistent memory models that support external side-effects in the form of load/store operations. This relieves one of the biggest (if not the biggest) restriction of pure dataflow programming: the inability to support a shared state, and specifically shared data structures [85]. Therefore, hybrid models are capable to execute imperative languages. As a result, combining dataflow and von-Neumann models facilitates designing efficient architectures that benefit from both computing models, while the remaining question is the best granularity-parallelism trade-off.

4.1 Evolution of Hybrid Architectures until 2000

The first idea of combining dataflow and control flow arose in early of 1980s [64], [99], [112], [123]. That combining idea included data and memory structure management (e.g., Multithreaded Monsoon (MT. Monsoon) [91]), self-scheduling and asynchronous execution to simplify thread synchronization (e.g., HEP [64], [112]; Tera [5]; MT. Monsoon [91]), and the ability of executing both conventional and dataflow programs in the same machine [8], [15]. Some hybrid models [15], [61] even included a program counter to a dataflow architecture in order to execute sequential instructions in control flow manner. In the same direction, other studies explored the *threaded dataflow* model [101], [109], in which partial data subgraphs are processed as von-Neumann instruction streams. In particular, given a dataflow graph (program), each subgraph that exhibits a low degree of parallelism is identified and transformed into a sequential thread of instructions. Such a thread is issued consecutively by the matching unit without matching further tokens except for the first instruction of the thread. Data passed between instructions in the same thread is stored in registers instead of being written back to memory. These registers may be referenced by any succeeding instruction in the thread. This improves single-thread performance because the total number of tokens needed to schedule program instructions is reduced, which in turn saves hardware resources. In addition, pipeline bubbles caused by runtime overhead associated with token matching are avoided for dyadic (two-operand) instructions within a thread. Two threaded dataflow execution techniques can be distinguished: (1) direct token recycling technique, which allows cycle-by-cycle instruction interleaving of threads in a manner similar to multithreaded von-Neumann computers (e.g. MT. Monsoon architecture), and (2) consecutive execution of the instructions of a single thread technique (e.g. Epsilon [47], [48] and EM-4 [6] architectures). In the second technique, the matching unit is enhanced with a mechanism that, after firing the first instruction of a thread, delays matching of further tokens in favor of consecutive issuing of all instructions of the started thread. In addition, some architectures based on threaded dataflow use instruction pre-fetching and token pre-matching to reduce idle times caused by unsuccessful matches. EM-4 [6], EM-X [72] and RWC-1 [108] are examples of this kind of architectures, which are also referred as *macro-dataflow* [78].

Up until the late 80s and early 90s the common wisdom was that fine-grain execution was much more suited at masking network and memory latencies than a coarse-grain one; and obviously would achieve a much better load leveling across processors and hence faster execution. However, it has been demonstrated that coarse-grain is as suited to exploit parallelism as fine-grain [83], [86], [122]. On one hand, Gao's group [40], [59], [121] was the first to develop a coarse-grain data flow simulator and compiler from scratch and report on very extensive evaluations of very complex applications. On the other hand, Najjar's group [14], [35], [36], [85], [102] focused on modifying the Sisal compiler [1] in two ways: (1) generate coarse-grained data flow code from a fine-grained one, and (2) generate coarse-grained data flow code from scratch using the Sisal compiler.

In addition to the coarsening of nodes in the DFG, another technique for reducing dataflow synchronization frequency (and overhead) is the use of complex machine instructions, such as vector instructions. With this instructions, structured data is referenced in block rather than element-wise and can be supplied in bursts introducing, also, the ability to exploit parallelism at the sub-instruction level. This technique introduces another major difference with conventional dataflow architectures: tokens do not carry data (except for the values `true` or `false`). Data is only moved and transformed within the execution stage. Examples of such machines are Stollman [43], ASTOR [133], DGC [35], [36], and Sigma-1 multiprocessor [132].

In parallel, the Out-of-Order model [60], [92], that emerged in the late 80s, incorporated the dataflow model to extract ILP from sequential code. This approach has been further developed by Multiscalar [114] and thread level speculation

(TLS) [98], [103] that can be viewed as coarse-grain versions of Out-of-Order.

Efforts have been done to survey hybrid models up to 2000 year [101], [109], [110] and dataflow multithread models [28], [62], [69], [78]. However, a comprehensive survey describing recent (since year 2000) hybrid architecture has been lacking, to the best of our knowledge. Hence, the main focus of this paper is on classifying recent hybrid dataflow/von-Neumann architectures, that mainly have tried to improve the conventional architectures exploiting dataflow concepts in several aspects [33], [68], [77], [105], [118] or utilize dataflow approach as accelerators [45], [84], [127]. Most of those recent works are classified and compared in the following sections.

4.2 Taxonomy Based on Block Execution Semantics

The inherent differences between dataflow and von-Neumann execution models seemingly puts them on two ends of a spectrum that can accommodate a wide variety of hybrid models. However, under our point of view, the coarsening of the basic operation granularity, from a single instruction to a block of instructions, together with the inter- and intra-block execution semantics, enables us to partition the spectrum into four different classes of hybrid dataflow/von-Neumann: *Enhanced Control Flow*, *Control Flow/Dataflow*, *Dataflow/Control Flow* and *Enhanced Dataflow* class. This taxonomy is based on whether they employ dataflow scheduling inside and/or between code blocks. Block is defined in base of the boundary between where the two scheduling models (inter and intra-block scheduling) are mainly applied. This way, the number of instructions in a block (block granularity) depends on the specific model. Figure 2 illustrates inter- and intra-block scheduling of conventional organizations of hybrid dataflow/von-Neumann architectures.

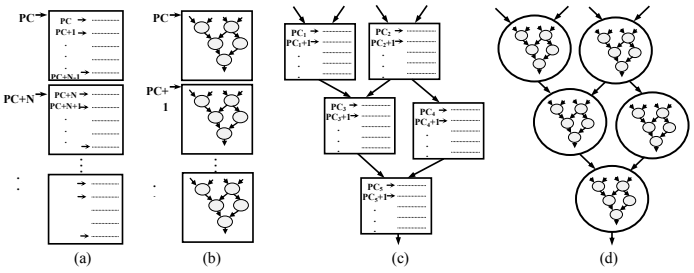

Fig. 2. Inter- and intra-block scheduling of organizations of hybrid dataflow/von-Neumann architectures. a) Enhanced Control Flow, b) Control Flow/Dataflow, c) Dataflow/Control Flow, and d) Enhanced Dataflow. Blocks are squares and big circles.

4.2.1 Enhanced Control Flow Class

Models in this class schedule blocks in control flow manner, whereas the instructions within a block are scheduled in a mixed approach of control flow and dataflow manner. Figure 2-a) illustrates the organization of this class.

The main example of this class is the Out-of-Order (restricted dataflow) model [60], [92]. The Out-of-Order model, as an extension of superscalar processors, incorporates the dataflow model only in the issue and dispatch stages to extract ILP from sequential code. It is also referred to as local dataflow or micro dataflow architecture [101], [109], [110].

4.2.2 Control Flow/Dataflow Class

Models in this class schedule the instructions within a block in dataflow manner, whereas blocks are scheduled in control flow manner (Figure 2-b). This method is used in RISC dataflow architectures, which support the execution of existing software written for conventional processors.

Main examples of this class are TRIPS [105], [106], Tartan [84], Conservation cores (C-Cores) [127], DySER [45] and other architectures that rely on domain specific dataflow accelerators. TRIPS was a new design that tried to overcome the foreseen limitations of large cores' architectures by adding new layers of flexibility to the hardware. Explicit dataflow execution within blocks was a necessary way to improve fine-grain ILP while keeping hardware complexity at bay. TRIPS unifies dataflow and von-Neumann into a single execution model. However, other architectures in this class, essentially use dataflow to accelerate parts of the code (hyperblocks in Tartan; kernels in C-Cores; phases in DySER). Their decision on which parts of the code to accelerate is mostly static while TRIPS uses dynamic scheduling decisions to map hyperblocks to dataflow cores. Tartan, C-Cores and DySER use profiling to determine the parts of the code to be accelerated in a dataflow unit (or units), mapped on a reconfigurable hardware coupled to a classical von-Neumann processor. Unlike Tartan and C-Cores, DySER also supports reconfigurations at runtime. This behavior allows the DySER architecture to capture a significant percentage of computation from a single application as multiple accelerated phases can be mapped to the same accelerator.

4.2.3 Dataflow/Control Flow Class

Models in this class employ dataflow rules between blocks and control flow scheduling inside the blocks (Figure 2-c). Under these restrictions, blocks are issued by the matching unit, and token matching needs only to be performed on a block basis. As a result, the total number of tokens needed to schedule program instructions is reduced, which in turn saves hardware resources. A block may be a set of sequential instructions, where data is passed between instructions using register or memory (coarse-grain dataflow models [101], [109], [110]), a complex machine instruction, or a combination of both strategies. Main examples of coarse-grain dataflow models are: Star-T (*T) [89], TAM [22], ADARC [117], EARTH [58], [121], P-RISC [88], MT. Monsoon [91], Pebbles [102], SDF[1] [68], DDM [77], and Task Superscalar [33]. Sigma-1 multiprocessor [132] is the main example for complex machine instructions, and Stollman [43], ASTOR [133] and DGC [35], [36] are examples of the combination of coarse-grain and complex machine instructions.

Figure 3 shows a further decomposition of this class based on the number of cores and number of instructions in a block (i.e., size of block) targeted by every specific model, as well as the year in which it was first published. Figure 3-a depicts the relationship between core granularity and the proposed architectures' publication year. First hybrid designs tend to have few number of cores, while recently proposed architectures tend to use a larger number of cores. Figure 3-b shows the variance in core granularity in hybrid design. Architectures with a larger number of cores typically use fewer number of instructions per block, and designs with fewer number of cores tend to use larger blocks (with more than 1000 instructions per block).

4.2.4 Enhanced Dataflow Class

Models in this class use dataflow firing rules both for instructions inside the blocks, and for the blocks themselves.

1. Please note that here SDF is acronym for scheduled dataflow, as opposed to synchronous dataflow (SDF) [79]. The latter is a dataflow based execution model for signal processing algorithms and does not include any von-Neumann properties.

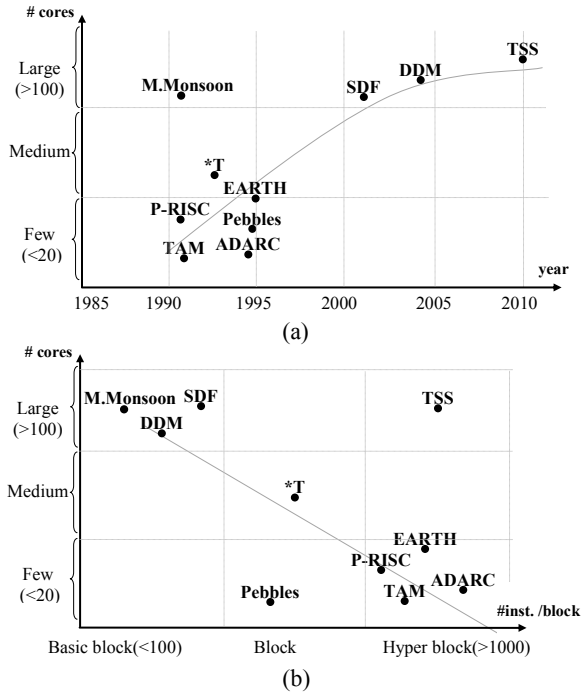

Fig. 3. Different architectures of Dataflow/Control Flow class (a) number of cores and year, (b) number of cores and size of blocks

Effectively, this class consists of two-level dataflow models (Figure 2-d) utilizing some concepts of the von-Neumann model (e.g., storage management) to add the abilities of running imperative languages and managing data structures. Cedar [75] and WaveScalar [118], [120] are main examples of this class.

4.2.5 Comparison of Hybrid Classes

Every one of the four classes presents advantages and drawbacks. *Enhanced Control Flow class* machines can very naturally execute control flow codes and uncover more ILP than the strict von-Neumann models. However, as the actual technology only allows them to address small to medium block sizes, the amount of parallelism that they can expose is typically limited (some architectures such as like Kilo-instruction Processors [21] try to overcome this problem targeting much larger block sizes).

Control Flow/Dataflow class machines try to overcome the limitations of the previous class by forcing the pure dataflow execution of the instructions inside a block. These models attempt to statically expose ILP at the block level, deferring memory operations to inter-block synchronization. Indeed, the *Control Flow/Dataflow* general strategy has shown a great potential in both performance and power savings [45], [84], although it presents the same problems as the previous class (e.g., smaller block sizes than desirable for fully exploiting dataflow advantages at ILP level).

For their part, *Dataflow/Control Flow class* models have taken advantage of the recent growth in the number of parallel hardware structures in cores, chips, machines and systems. As models in this class address parallelism at a coarse grain they can exploit all these resources more effectively than conventional (von-Neumann) models while retaining the programming model inside the blocks.

Finally, *Enhanced Dataflow class* models are a complete rethinking of the execution problem. Because they do not use a program counter, they face several difficulties to execute conventional codes and manage memory organizations and, therefore, need more hardware resources to be effectively used. On the other hand, *Enhanced Dataflow class* models can be viewed as an addition of both *Dataflow/Control Flow* and *Control Flow/Dataflow* classes, and in this sense, they present great potential.

4.3 Taxonomy Based on Execution Model

Hybrid models can also be classified from an execution model point of view; *unified-hybrid* models versus *accelerator* models. In an unified-hybrid architecture, a program must be executed using both dataflow and control flow scheduling since both models are intimately bound in the architecture. Although the majority of the models presented belong to this group, it presents some drawbacks. The additional hardware needed by the interconnection and synchronization mechanisms (e.g., hardware of Out-of-Order architectures) leads to more complexity and power consumption. Furthermore, as every program should be executed with the same hybrid scheduling schema, they are not able to adapt to specific cases where a pure dataflow or von-Neumann model would be better.

On the other hand, in accelerator based architectures, the decision on which parts of the code to accelerate is mostly static (made by the programmer or compiler, and sometimes based on profiling). In addition, a whole program may be executed without the use of the accelerator.

Within this category, another subdivision can be done based on the type of the accelerator: dataflow inside von-Neumann based architectures and von-Neumann inside dataflow based architectures. Tartan, C-Cores and DySER are architectures that use dataflow to accelerate kernels (or hyperblocks) and thus belong to the former group. Sigma-1 multiprocessor and all the other dataflow architectures that use complex instructions belong to the later.

As in both cases the selected and accelerated parts are executed with a specifically designed hardware, those architectures can achieve big performance improvements and power savings on these parts.

5 Examples of Recent Hybrid Dataflow/von-Neumann Architectures

In this section, we describe recent examples of hybrid dataflow/von-Neumann architectures for each of the above mentioned taxonomy classes, in chronological order.

Out-of-Order (restricted dataflow) architectures [60], [92], [113] are presented for *Enhanced Control Flow class*. Although Out-of-Order appeared before 2000, we have included it because its popularity, significant contribution to the class and to better highlight how the introduction of dataflow execution into an otherwise control flow model can dynamically extract parallelism. TRIPS [105], [106], WaveScalar [118], [120] and Task Superscalar [33], [34] are presented for *Control Flow/Dataflow*, *Enhanced Dataflow*, and *Dataflow/Control flow classes* respectively. DySER [45], although is also an architecture of *Control Flow/Data flow class* as TRIPS, has been included as a recent representation of a whole range of architectures that use dataflow or control flow accelerators inside an otherwise pure control flow or dataflow processor.

There are other relevant architectures that have been included in the supplementary file due to space constraints. For instance, MT. Monsoon and Sigma-1 multiprocessor are very relevant, but non-recent, representations of *Dataflow/Control flow class* architectures. And, in the case of Sigma-1, it is also an example of the *accelerator* model class. Also, DDM and

SDF architectures are relevant and recent *Data flow/Control Flow* architectures that have been included in the supplementary file.

Main characteristics of all mentioned architectures are described and discussed in Section 6.

5.1 Out-of-Order Execution Model

The *Out-of-Order* architecture (Restricted Dataflow) [60], [92], [113] is a fine-grain hybrid architecture that belongs to the *Enhanced Control Flow class*. The Out-of-Order architecture is also referred to as local dataflow or micro dataflow architecture [101], [109].

Execution Model

Out-of-Order processors employ dataflow principles to extract instruction level parallelism (ILP) and optimize the utilization of the processor's resources. The processor relies on hardware mechanisms that determine dynamically data dependencies among the instructions in the instruction window. In other words, in this paradigm, a processor executes instructions in an order governed by the availability of input data, rather than by their original order in a program[2]. In doing so, the processor can both extract ILP and hide short data fetch latencies by processing subsequent instructions that are ready to run. Each instruction window of Out-of-Order processor is a block granularity for the intra-block scheduling.

Architecture Organization

Figure 4 illustrates the general scheme of the Out-of-Order execution pipeline. Instructions are fetched in order, decoded and placed, after register renaming, into a pool of pending instructions (the instruction window) and the reorder buffer. The reorder buffer saves the program order and the execution states of the instructions. To increase the effective instruction window size, those architectures rely in branch prediction and speculation. As a a result, they require complex checkpointing mechanisms to recover from branch mis-predictions and mis-speculated executions (not shown in the Figure).

Dispatch and Issue determine the out of order and dataflow execution of the microprocessor. The matching of the exectuable instructions in the microprocesor is restricted to the pending instructions of the instruction window. Therefore, the matching hardware can be restricted to a small number of instructions slots. In addition, because of the sequential program order, the instructions in this window are likely to be executable soon.

Once the instructions are executed, they are retired to permanent state machine (memory) in source program order (commit in the Figure). Another advantage of those architectures is its sequential execution of the instructions, exploiting the spatial locality of the program. That locality allows to employ a memory hierarchy that stores the instructions and data, potentially executed in the next cycles, close to the executing processor.

Implementation Examples

Arguably the first machine of Out-of-Order execution was the CDC 6600 (1964), which used a scoreboard to resolve conflicts. The IBM 360/91 (1966) introduced Tomasulo's algorithm, supporting full Out-of-Order execution. In 1990, the first Out-of-Order microprocessor appeared, the POWER1, but its Out-of-Order execution was limited to floating point instructions.

As mentioned above, Out-of-Order microprocessors have an instruction window that is restricted to a sequence of instructions. Thread-level speculation (TLS) processors can be seen as an extension of Out-of-Order hybrid dataflow/von-Neumann architecture that increases the instruction window and potentially uncover more ILP. Thread-level speculation is a technique which empowers the compiler to identify potential parallel threads, despite uncertainty as to whether those threads are actually independent [116]. TLS allows to speculatively execute those threads in parallel, while squashing and re-executing any thread that suffers dependence violations. Therefore, the instruction window is the addition of the sequence of instructions of all non-speculative and speculative threads executing in parallel, potentially larger than the Out-of-Order processor's instruction window.

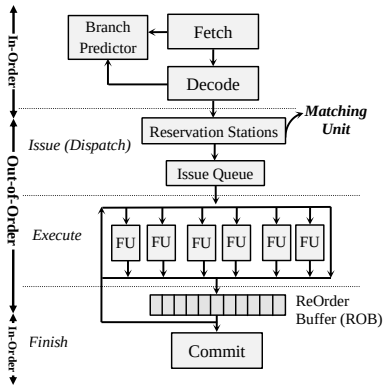

Fig. 4. The Out-of-Order execution pipeline

Thread creation, and the mechanism for buffering speculative state and tracking data dependences between speculative threads, are important features of the different TLS approaches. Some of them are implemented entirely in software [50], [98], [104], others in hardware [4], [80], [103] and others are a combination of software and hardware [17], [38], [44], [53], [73], [74], [97], [134]. Two relevant works are the LRPD test [98] (software-only support) and the Multiscalar architecture [114]. LRPD test allows the compiler to parallelize loops without fully disambiguating all memory references, and applies only to array-based codes. Disambiguation is done with the use of shadow arrays to detect any cross-iteration Read-after-Write dependence. Multiscalar architecture was the first complete evaluation of an architecture designed specifically for supporting TLS. The compiler statically performs the distribution of the instructions among tasks (potential speculative threads). Address resolution buffer (ARB) [39], forward and release bits, and CFG (control flow graph) information are the mechanisms used for tracking control and data dependences between speculative threads.

5.2 TRIPS

TRIPS (Tera-op, Reliable, Intelligently adaptive Processing System) [105], [106] was designed at University of Texas at Austin as a grid architecture that implements the EDGE (Explicit Data Graph Execution) ISA [16], [105], [111]. It is an example of *Control Flow/Dataflow class* models.

Execution Model

TRIPS combines control flow execution across hyperblocks of code consisting of up to 128 instructions with a dataflow execution inside them. In TRIPS, a hyperblock is equivalent to the block granularity. This scheme enforces conventional memory semantics across hyperblocks and so it allows imperative code (as C or Fortran) to be executed without major modifications.

2. The memory accesses are done in order

The TRIPS architecture is fundamentally block oriented. The compiler is responsible for statically scheduling each block of instructions onto the computational engine such that inter-instruction dependences are explicit. Therefore, the compiler role is key on the final performance of the application [19]. Each block has a static set of state inputs, and a potentially variable set of state outputs that depends upon the exit point from the block. At runtime, the basic operational flow of the processor includes fetching a block from memory, loading it into the computational engine, executing it to completion, committing its results to the persistent architectural state if necessary, and then proceeding to the next block.

TRIPS has a block-atomic execution mode, and direct communication for instructions within a block. On one hand, block-atomic execution means a block of instructions must be fetched and executed as though it was a single unit providing interruptions at block level. On the other hand, direct communication implies that instructions within a block can directly send values to dependent instructions within the same block. This behavior allows the architecture to have very large windows (up to 1024 instructions) that execute in dataflow order.

TRIPS provides three modes of execution that enable *polymorphous parallelism*: Desktop-morph (D-morph), Thread-level morph (T-morph) and Stream-level morph (S-morph). Each of these modes is aimed at exploiting one of the three types of parallelism: instruction level parallelism (ILP) in D-morph mode, thread level parallelism (TLP) in T-morph mode, and data level parallelism (DLP) in S-morph mode. How TRIPS works in these different modes is explained in detail in the next subsection.

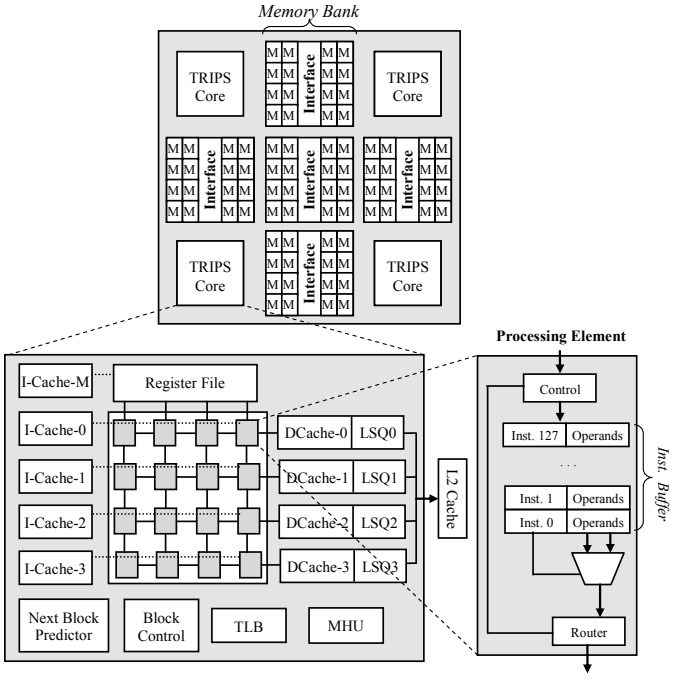

Fig. 5. The TRIPS architecture (Figure based on [105], [106]). **DJG: to be changed by the adapted figure I have sent it to you**

Architecture Organization

TRIPS is a tiled and distributed architecture. Figure 5 shows the general TRIPS architecture. TRIPS processor consists on four TRIPS cores and a tiled secondary memory (M tiles in the Figure 5.a), surrounded by a tiled newtwork (N tiles in the Figure) that acts as translation agents for determining where to route memory system requests.

Each of the TRIPS cores is implemented using five unique tiles: one global control tile (GT), 4x4 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT), as shown in Figure 5.b. Each tile only interacts with its immediate neighbours through microarchitectural networks (micronets). Micronets have roles such as transmiting operands between instructions, distributing instructions from the instruction tiles to the execution tiles, or communicating control messages from the program sequencer [107]. The major processor core micronetwork is the operand network (OPN), that handles transport of all data operands along ETs, DTs, GTs, and RTs.

Global Control Tile contents the blocks' PC running in the TRIPS core, the instruction cache tag arrays, the I-TLB, and the next-block predictor. The GT handles TRIPS block management, including prediction, fetch, dispatch, completion detection, flush (on mispredictions and interrupts), and commit. In addition, GT is used to set up the control register that configure the processor into different speculation, execution, and threading modes. GT also maintains the state of all in-flight blocks (maximum 8) running in the ETs of the TRIPS core. When a block finishes, the block predictor (tournament local/ghsare predictor based) provides the predicted address of the next target block. The block is fetched and loaded into the execution units' reservation stations.

Each execution unit consists of a fairly standard single-issue pipeline, a bank of 128 reservation stations (instructions and two operands per instruction), an integer unit, a floating point unit, and operand router (shown in Figure 5.c). When a reservation station contains a valid instruction and a pair of valid operands, the node can select the instruction for execution. After execution, the node can forward the result to any of the operand slots in local or remote reservation stations within the ALU array (4x4 ETs' ALUs). The nodes are directly connected to their nearest neighbors, but the routing network can deliver results to any node in the array.

Instructions are statically placed into the locations of the ET, and executed in dataflow manner using the direct instruction communication between intra-block producers and consumers, specified by the TRIPS ISA. Therefore, the instructions are statically placed at compile time and dynamically issued at runtime.

Instruction Cache is also tiled into five banks to increase the memory bandwidth. Each IT acts as a slave of the GT which holds the single tag array, and can hold a 128-byte chunk, for a total of 640 bytes (128x5) for the maximum-size block. There is one instruction bank per row of ETs, and a additional one to issue fetches to values from registers for injection into the ALU array.

Register file is divided in four 32-register banks (tiles) that are nodes of the OPN micronet, allowing the compiler to place critical instructions that read and write from/to a given bank close to that bank. There are a total of 128 registers for the four threads that can run in parallel. Each bank has two read ports and one write port, and contains a write queue and a read queue. Those queues allow to have write-to-read forwarding and renaming capabilities. The registers file holds a portion of the architectural state. Thus, values passed between hyperblocks, where direct instruction communication is no possible, are transmitted through the register file.

Primary memory is also divided in four data tiles (DT).

Each DT holds one L1 data cache bank, a load/store queue, a dependence predictor (for previous stores), one-entry back side coalescing write buffer, a data TLB, and a MSHR that supports up to 16 requests for up to four outsanding cache lines. It can be accessed by any ALU through the local grid routing network.

Some of those hardware resources can be configured, using the GT, to operate differently depending on the mode: D-morph, T-morph and S-morph. For instance, the reservation stations can be managed differently depending on the execution mode. A physical frame is formed by the reservation stations with the same index accros all of the execution unit nodes (e.g. combining the first slot for all nodes in the grid forms frame 0). Frames that contain one hyperblock form an architectural frame (A-frame). Thus, direct instruction communication is only possible within a A-frame.

In D-morph, all the frame space of a TRIPS core can be used, as it was a large, distributed, instruction issue window, by only one thread allowing it to achieve maximum ILP. In addition, in order to increase the potential ILP, the hardware fills empty A-frames with speculatively mapped hyperblocks, predicting which hyperblock will be executed next, mapping it to an empty A-frame, and so on. The A-frames are treated as a circular buffer where the first is non-speculative, and the rest are speculative. When the non-speculative A-frame finishes, first speculative A-frame becomes the non-speculative first A-frame of the cicular buffer.

In T-morph, the frame space is statically partitioned so that each thread can have its own frame space partition. Within each thread, speculation is also used but extra prediction registers are needed, as block control state for each of the hardware threads.

In S-morph, only one thread can be run and no speculation is done. Instead, inner loops of a streaming application are unrolled to fill the reservation stations within multiple A-frames fused in a *super* A-frame. In this case, to reduce the power and instruction fetch bandwidth overhead of repeated fetching of the same code block across inner-loop iterations, the S-morph employs mapping reuse, in which a block is kept in the reservation stations and used multiple times. In this case, the L2 cache memory can be configured to be used as a stream register file (SRF) [106], so that direct data array access and DMA transfer capabilities are allowed. Otherwise, the secundary memory works as a non-uniform cache access (NUCA) on-chip memory system.

Implementation Examples

Some studies have been done on different aspect of TRIPS. Sankaralingam et al. [107] describe the control protocols in the TRIPS processor. They detail each of the five types of reused tiles that compose the processor, the control and data networks that connect them, and the distributed microarchitectural protocols that implement instruction fetch, execution, flush, and commit. They also describe the physical design issues that arose when implementing the microarchitecture in a 170M transistor, 130nm ASIC prototype chip composed of two 16-wide issue distributed processor cores and a distributed 1MB non-uniform cache access (NUCA) on-chip memory system.

Gratz et al. [49] presented design, implementation and evaluation of the TRIPS on-chip network (OCN) which is a wormhole routed, 4x10 2D mesh network with four virtual channels. It provides a high bandwidth, low latency interconnect between the TRIPS processors, the L2 cache banks and the I/O units. They discussed the tradeoffs made in the design of the OCN, in particular, why area and complexity were traded off against latency. A full evaluation of a real TRIPS ASIC prototype and an EDGE compiler [41] demonstrates that the TRIPS machine is feasible. The work also shows that TRIPS is competitive with a Pentium 4 system in the number of cycles needed to execute an application. It is an impressive outcome for a new machine fully developed in an academic environment.

TFlex is another architecture based on the EDGE ISA. It is an implementation of the composable lightweight processor (CLP) [70] which are proposed to eliminate the problem of fixed-granularity processors, and consist of multiple simple, narrow-issue processor cores that can be aggregated dynamically to form more powerful single-threaded processors. TFlex adds four capabilities to TRIPS in a distributed fashion: I-cache management, next-block prediction, L1 D-cache management, and memory disambiguation hardware. Robatmili et al. [100] present a hardware implementation of mapping blocks to a distributed substrate of composable cores for the TFlex.

5.3 WaveScalar

WaveScalar [118], [120], is an example of the *Enhanced Dataflow class*. It is a dynamic, general purpose, decentralized superscalar dataflow architecture that is product of a research performed in the University of Washington. WaveScalar is also the name of the dataflow instruction set and the execution model.

Execution Model

WaveScalar execution model is basically a dataflow model enhanced to support imperative languages. The key tenet of the Wavescalar execution model is that programs execute in *waves*, that are sets of connected instructions of the program graph. The wave name may come after the way data flows from the initial instruction to the subsequent ones in parallel.

Formally, a wave is a connected, directed acyclic portion of the control flow graph with a single entrance. The WaveScalar compiler partitions an application into maximal waves and adds wave management instructions. In fact, waves are similar to hyperblocks, but they can contain control flow joins and are generated using loop unrolling to make them larger (all instructions within a wave are partially ordered, so waves can not contain loops). In order to allow instructions to operate on different dynamic waves, all data elements travel with their wave number that increases as the data goes out of a wave and enters a new one (or the same) using a special *waveadvance* instruction.

Therefore, in order to execute an imperative program in Wavescalar, it is compiled into an special code that contains the dataflow graph (i.e. the wave) and also the memory order. An example of the memory order problem is illustrated in Figure 6. Assume that the `Load` instruction must execute after the `Store` instruction to ensure correct execution because the two memory addresses are identical. In a pure dataflow graph this implicit dependence between the two instructions (the dashed line in the Figure 6) can not be expressed.

However, Wavescalar supports a wave-ordered memory mode in which the compiler annotates memory access instructions within each wave to encode the ordering constraints between them forming a chain of memory instructions. Therefore, a memory request can only be executed if the previous request in the chain and all memory requests from the previous wave have already been executed. In order to be successful, the compiler must ensure that there is a complete chain of

memory operations along every path through a wave. So, if there are no memory operations in one of the paths of a branch, a `MemNop` instruction must be inserted in that path to maintain the chaining. Also, in order to increase the parallelism (i.e. in loops), non-dependent memory accesses can also be annotated with an additional *ripple* number that allow loads to execute in parallel and even out of order if all previous stores have finished.

By implementing wave-ordered memory, a total ordering of memory instructions can be achieved with little dynamic overhead. This feature, alongside indirect jumps for object linking, allows traditional von-Neumann models of computation to execute just as fast — if not faster — on the dataflow architecture. Its main advantage is that it is a dataflow hardware that runs programs written in standard programming languages, by efficiently providing the sequential memory semantics that imperative languages require.

In addition to wave-ordered memory, a second memory scheme in WaveScalar (*standard data firing rule* mode), allows the programmer to omit any unnecessary ordering and intertwine memory operations into the program graph by using the standard data firing rule. The unordered memory scheme introduces a new store instruction, store-unordered-ack, that returns zero to signify when it has completed. Using this value as an input arc to other instructions enforces memory ordering while providing greater flexibility to the programmer.

Both wave-ordered and unordered memory can be used interchangeably within the same program or even within the same wave to take advantage of fine-grain (unordered) and coarse-grain (wave-ordered) threads, resulting in significant performance improvements [118], [120].

Figure 6 should be modified. The code should read: A[j+k]=x; y=A[i];

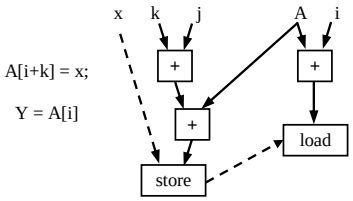

Fig. 6. Dataflow graph and wave-ordered memory (Figure based on [120])

Architecture Organization

To execute WaveScalar programs, a scalable, tile-based processor architecture called *WaveCache* has been designed. Figure 7 shows the WaveScalar architecture. Each basic processing element (PE) is a five-stage (Input, Match, Dispatch, Execute, and Output), dynamically scheduled execution pipeline. In WaveScalar, pairs of PEs are coupled into `pods` sharing ALU results via a common bypass network. Four `pods` (8 PEs) are grouped into a `domain` that communicate over a set of pipelined buses. Four domains (32 PEs) form a `cluster` supported by conventional memory hierarchy. To build larger machines, multiple clusters can be connected by a 2D mesh interconnection network.

Wave-ordered memory lies in the WaveCache's store buffers (one per cluster), which are responsible for implementing the wave-ordered memory interface that guarantees correct memory ordering.

To reduce the communication costs, the PEs are connected through a hierarchical interconnection infrastructure.

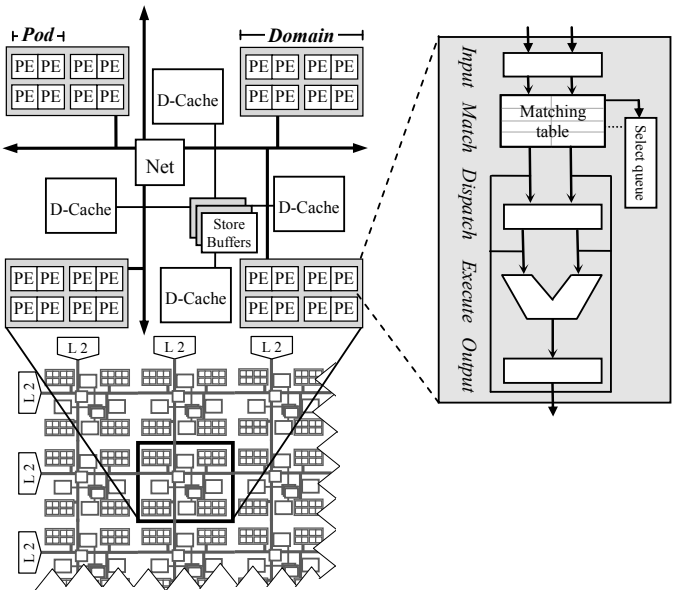

Fig. 7. The WaveScalar architecture (Figure based on [118], [120]).

WaveScalar's hierarchical interconnect plays an important role in overall scalability. Swanson et al. [119] have studied the area-performance trade-offs for WaveScalar.

The placement scheme of the instructions of a program has a compile-time and a runtime component. The compiler is responsible for grouping instructions into segments. Those segments have up to 64 instructions. As a program executes, the WaveCache maps the program's instructions onto its array of PEs, placing a whole segment of instructions at the same PE. The instructions remain at their PEs for many invocations, and as the working set of instructions changes, the WaveCache removes unused instructions and maps new ones in their place. The instructions communicate directly with one another over a scalable, hierarchical on-chip interconnect, obviating the need for long wires and broadcast communication.

Implementation Examples

The only implementation of WaveScalar is WaveCache. WaveScalar uses a regular native DEC compiler for converting source code to Alpha binary, and a binary translator is used for translating an Alpha binary to a WaveScalar binary. Petersen et al. [95] present and analyze three compiler optimizations for wavescalar C compiler that significantly reduce control overhead with minimal additional hardware. The basis of the solution lies in recognizing that overhead instructions are relatively simple to implement in hardware and can generally execute in parallel with computation. Hence, the microarchitecture can be tuned to execute overhead instructions in parallel with computation instructions. Merzulo et al. [82] proposed the transactional WaveCache to expliot speculative execution of memory operations. Pei et al. [93] exploited speculative multithreading (SpMT) based on WaveScalar.

5.4 Task Superscalar

Task Superscalar [33], [34] is a task-based dataflow architecture which generalizes the operational flow of dynamically scheduled Out-of-Order processors. It was designed at Barcelona Supercomputing Center (BSC) and belongs to the *Dataflow/Control Flow class*. The Task Superscalar combines the effectiveness

of Out-of-Order processors in uncovering parallelism together with the task abstraction, thereby provides a unified management layer for CMPs, which effectively employs processors as functional units.

Execution Model

The Task Superscalar processor combines dataflow execution of tasks with control flow execution within the tasks (i.e., the block granularity is a task). As ILP pipelines uncover parallelism in a sequential instruction stream, similarly, the Task Superscalar uncovers task level parallelism among tasks generated by a sequential thread. Utilizing intuitive programmer annotations of task inputs and outputs, the Task Superscalar pipeline dynamically detects inter-task data dependencies, identifies task-level parallelism, and executes tasks out-of-order. That design thus enables programmers to exploit many-core systems effectively, while simultaneously simplifying programming model.

Architecture Organization

The high-level operational flow of the Task Superscalar is illustrated in Figure 8. A task generator thread resolves the inter-task control path and sends non-speculative tasks to the pipeline frontend for dependency decoding. The task window can consist of tens of thousands of tasks, which enables it to uncover large amounts of parallelism [34]. The pipeline asynchronously decodes the task dependencies, generates the data dependency graph, and schedules tasks as they become ready. Finally, ready tasks are sent to the execution backend, which consists of a task scheduler and a queuing system.

As shown in Figure 8, the frontend employs a tiled design, and is managed by an asynchronous point-to-point protocol. The frontend is composed of four module types: *pipeline gateway*, *task reservation stations* (TRS), *object renaming tables* (ORT), *object versioning tables* (OVT).

The gateway is responsible for controlling the flow of tasks into the pipeline including allocating TRS space for new tasks, distributing tasks to the different modules, as well as stalling the task generator thread whenever the pipeline fills. TRSs store the in-flight task information and track the readiness of task operands. Inter-TRSs communication is used to register consumers with producers, and notify consumers when data is ready. The TRSs store the meta-data of all in-flight tasks and their parameters, and thereby effectively embed the task dependency graph. In this graph, nodes are tasks and arcs are dependencies between tasks.

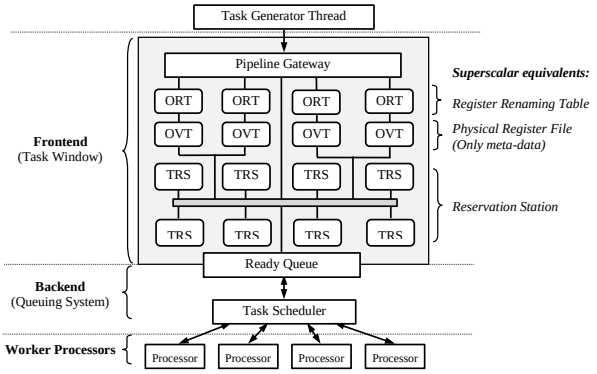

Fig. 8. The Task Superscalar architecture (Reprint from [34]).DJG: Superscalar equivalents should be OoO equivalents since they are for OoO execution explanation

The ORTs map parameters to the most recent task accessing the same memory object, and thereby detect object dependencies. Storing all data users (either producer or consumer), rather than only storing real data for producers, facilitates TRS consumer chaining. The OVTs track live operand versions, created whenever a new data producer is decoded. Each OVT is associated with exactly one ORT. The functionality of the OVTs is similar to a physical register file, but only for maintaining operand meta-data. Effectively, the OVT manages data anti- and output-dependencies, either through operand renaming, or by chaining different output operands and unblocking them in-order by sending a ready message when the previous version is released.

Figure 8 also shows the Out-of-Order components equivalent to the Task Superscalar modules. In Out-of-Order processors, dynamic data dependencies operates by matching each input register of a newly fetched instruction (consumer), with the most recent instruction that writes data to that register (producer). The instruction is sent to a reservation station to wait until all its inputs become available. Hence, the reservation stations effectively store the instruction dependency graph, which consists of all in-flight instructions. In the Task Superscalar, the mechanism of decoding tasks identifies all possible effects a task may have on the shared processor state, so producers and consumers are identified correctly. Moreover, tasks are decoded in-order to guarantee correct ordering of producers and consumers, and specifically, that the decoding of a task producing a datum updates the renaming table, before any task consuming the datum performs a lookup.

Implementation Examples

Etsion et al. [33] presented a design for a distributed Task Superscalar pipeline frontend which can be embedded into virtually any many-core fabric, and manages it as a Task Superscalar multiprocessor. The Task Superscalar architecture uses StarSs programming model [11], [94]. This programming model supports Out-of-Order execution of tasks, by enabling programmers to explicitly expose task side-effects, using annotating operands of kernel functions as input, output, or inout. The model can thus decouple the execution of the thread generating the tasks, from their decoding and execution. At runtime, whenever the task generator thread reaches a call site to one of the kernels, task creation code (injected by a source-to-source compiler) packs the kernel code pointer and all the operands, and writes them to the task pipeline.

Yazdanpanah et al. [131] presented a FPGA-based prototype of the Task Superscalar architecture. The implemented hardware is based on a tiled design that can operate in parallel and is easily scalable to manage hundreds of cores in the same way that Out-of-Order architectures manage functional units. The prototype operates at near 150Mhz and can maintain up to 1024 in-flight tasks, managing the data dependencies in few cycles.

5.5 DySER

DySER (Dynamically Specialized Execution Resource) [45], [46] is an architecture based on dataflow accelerators that belongs to the *Control Flow/Dataflow class*. It was designed at the University of Wisconsin-Madison as the hardware substrate of the dynamically specialized execution (DySE) model.

Execution Model

DySER integrates dataflow accelerators (DySER block) into a control flow processor's pipeline as functional units. To achieve this goal, the program is explicitly partitioned by the compiler (profile-guided to determine common path trees of control flow) into phases (i.e. program sections). After that, for each phase, the compiler determines its kernels and tries to accelerate them using the DySER block. The DySER block is basically a big reconfigurable functional unit composed by different arithmetic units whose connections are reconfigured at runtime creating specialized dataflow blocks that can be pipelined. The model hinges on the assumption that only a few dataflow blocks are active during a given phase of an application and they are invoked several times. Thus, setting up the static routes once amortizes the execution of the DySER unit over many invocations.

To be able to execute each kernel within a phase with a pure dataflow accelerator, kernels are divided into a load-back slice and a computation slice. A load-back slice includes all the memory accesses, while a computation slice consists of computation operations without memory accesses that are grouped and executed in the dataflow accelerator. With this separation between slices, the usual processor's memory disambiguation optimizations can proceed unhindered. Therefore, in DySER, these computation slices are the block granularities for the intra-block scheduling.

To take better profit of the reconfigurable unit, when the control flow execution reaches a program phase, the DySER block is dynamically configured (specialized for the phase). Furthermore the execution model allows multiple DySER blocks, where each block is configured differently. With multiple DySER blocks, the next block can be predicted and configured before its inputs are produced by the processor. The large granularity of the phases allows easy predictability.

Architecture Organization

Figure 9 illustrates the DySER attached to a processor pipeline. The DySER block consists of a circuit-switched network of heterogeneous functional units. The functional units (FUs) form the basic computation fabric. Each FU is connected to four neighbor switches from where it gets data and control input values and injects outputs. Each FU also includes a configuration register that specifies which function to perform, and one data register and one status register for each input switch. The status registers indicate the validity of values in the data registers. The data registers match the word-size of the machine. The switches (Ss) contain data and status registers, and include a configuration register which specifies the input to output port mappings.

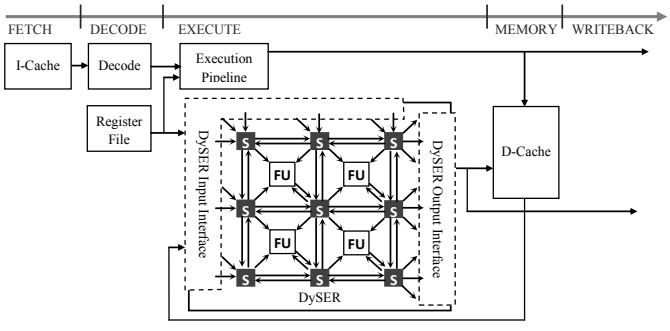

Fig. 9. Processor pipeline with DySER (Reprint from [45])

DySER blocks are configured by writing into configuration registers at each functional unit and switch. After configuration, the switches in the DySER block form a circuit-switched network that creates explicit hardware paths from inputs to the functional units, between functional units, and from functional units to outputs. The functional units are configured to perform the operation that is needed to execute the desired dataflow graph. The idea is that for a given application phase, DySER blocks are configured once and re-used many times.

The basic execution inside a DySER block is dataflow driven by values arriving at a functional unit. When the valid bits for both left and right operands are set, the functional unit consumes those inputs, and a fixed number of cycles later produces the output, writing into the data and status register of the output switch.

All the inputs to a DySER block are fed through a logical FIFO, which delivers register inputs and memory values. Each entry specifies a switch and a port. As a DySER block uses circuit-switched routing, this effectively decides where the value will be delivered in the block. Outputs follow a similar procedure. Each port in the output switches corresponds to one possible DySER block output. Since for each output port, the DySER produces outputs in order, no FIFOs are required on the output side. When values arrive at the output ports, an output interface writes them to the corresponding register or memory.

DySER can be easily integrated into conventional in-order and Out-of-Order pipelines as an accelerator. Integration with an in-order pipeline is simple and the DySER block interfaces with the instruction fetch stage for obtaining the configuration bits, the register file stage and the memory stage of the pipeline. A state machine must be added to the instruction cache to read configurations bits for a DySER block and send them to the input interface of that DySER block.

DySER integration with an Out-of-Order pipeline requires more careful design. The processor views DySER as a functional unit but the input ports should be exposed to the issue logic to ensure two send operations are not executed out-of-order. Since loads can cause cache misses, when a load executes in the processor, the corresponding input port is marked *busy* in the input buffers. When the data arrives from the cache, the input port is marked *ready*, which prevents subsequent loads values from entering the DySER block earlier.

Implementation Examples

Govindaraju et al. [45] implemented the DySER block in Verilog and synthesized it using Synopsys compiler with a 55nm standard cell library. They developed path-tree, a program representation for application phases, in order to find the most frequently executed basic blocks for mapping on DySER. For evaluating DySER, they developed extensions to the GCC toolchain which operates on the SPARC backend and performs path-profiling and DySER mapping. Benson et al. [12] described the integration of DySER into a commercial processor designing an FPGA prototype based on the OpenSPARC T1 processor called *OpenSPlySER*. Govindaraju et al. [46] studied challenges for DySER on data parallel workloads.

6 Discussion of the Recent Hybrid Models

This section highlights the main features of recent hybrid architectures, compares and discusses them, and shows their common trends. Table 1 introduces the main features of the architectures described in Section 5 and in the supplementary file, sorted by the *year* the architecture appeared.

TABLE 1
Comparison of the recent hybrid dataflow/von-Neumann architectures

	SIGMA-1	Out-of-Order	MT. Monsoon	DDM	SDF	TRIPS	Wavescalar	Task Superscalar	DySER (w/ inorder GPP)
Year	1982	1985	1991	2000	2001	2003	2003	2010	2011
ISA	RISC	RISC / CISC	RISC	RISC / CISC	RISC (Preload/ store + computation)	EDGE	WaveScalar	RISC / CISC	RISC /CISC + Ultra-wide insts.
Main features	Vector processing	Out-of-order instruction execution	ETS, MT	Decoupled non-blocking, CacheFlow policy (prefetching)	Decoupled non-blocking multithreading	Polymorphous (multiple parallelism modes), memory ordering, branch prediction	Wave-ordered and unordered memory, hierarchically interconnection	Out-of-order task execution	Profiling based detection of ultra-wide insts. compiler support to execute them on DySER blocks
Core Granularity	SIGMA-1 PE and SE	PE agnostic	Monsoon PE	PE agnostic	Simple processor	1 TRIPS processor = 4x4 ALU (TRIPS core)	5 stage dynamically scheduled pipeline	PE agnostic	DySER block: 8x8 FU / GPP: PE gnostic
Scalability	> 100 PEs, >100 SEs	~ 10 FUs	> 1000PEs	~ 100 PEs	~ 100 PEs	~ 100 PEs	> 1000 FUs	>> 100 PEs	< 10 DySER Blocks
Parallelism level	ILP, DLP, TLP	ILP, DLP, TLP (dual threaded)	TLP	TLP	TLP	ILP, TLP, DLP	TLP, ILP	TLP	ILP, DLP, TLP (dual threaded)
Block Granularity	Vector length	Instruction Window size	Thread (basic block)	BB size (code block, more than one thread, in TSU graph memory)	BB size, <128-inst. blocks (27 (15 in EP) up to 51 (39 in EP))	128-inst. block (EDGE)	Up to 64 insts. per PE cache, any number of insts. in a wave	Task size (any size) > 10K	Equivalent to few hundreds of ISA insts.
Inter-block Scheduling	Dataflow	Control flow	Dataflow	Dynamic dataflow (dependencies specified in programs)	Static dataflow (programmer/ compiler)	Static control flow (compile time)	Dynamic dataflow (dependencies detected at execution time)	Dynamic dataflow (dependencies specified in programs)	Control flow
Intra-block Scheduling	Static control flow	Hybrid control flow / dataflow	Control flow	Control flow	Control flow (scheduled dataflow)	Static dataflow (compile time)	Dynamic dataflow (execution time)	Control Flow	Static Dataflow (compile time)
Inter-block Communication	Direct inter-connection	Register /cache /memory	Register / memory	Cache	Frame memory and registers	Registers	Memory /direct interconnection	Memory	Register /memory / FIFO
Intra-block Communication	Register	Register /cache /memory	Register / memory	Register / memory	Register	Memory / direct interconnection	Memory / direct interconnection	Register / memory	Direct interconnection
Examples	SIGMA-1	Many	MT. Monsoon	D2NOW, Flux, DDM-VMc	SDA	TRIPS, TFlex	WaveCache	Task Superscalar	DySER

6.1 Main Features

Out-of-Order, DDM, Task Superscalar and DySER are based on RISC/CISC ISA. DySER, in addition, has the ultra-wide instructions, used to run part of the program in the reconfigurable DySER blocks. SDF is based on a RISC ISA defined for the execution and synchronization processors. MT.Monsoon is based on CISC ISA. TRIPS and WaveScalar are based on dataflow ISAs: EDGE ISA and WaveScalar ISA, respectively.

The main feature of Out-of-Order is the dataflow execution of a sequential instruction stream. ETS and multithreading are main features of MT.Monsoon architecture. For DDM the main feature is the introduction of the CacheFlow policy, that implies the execution of a DDM thread (basic block of instructions - BB) only if its data is already placed in the cache. Decoupling computation and synchronization, and non-blocking threads are also main features of SDF and DDM. However, the computation in the DDM is carried out by an off-the-shelf processor while in the SDF it is carried out by a custom designed processor. Another difference is that in SDF data is preloaded in registers while in DDM data is pre-fetched in the cache. Polymorphism is one of the main features of TRIPS providing three modes of execution for exploiting one of the three types of parallelism ILP, TLP and DLP. The main feature of the WaveScalar is the wave-ordering execution. In the wave-ordered memory of WaveScalar, memory instructions are annotated with extra information that orders them relative to other instructions of a block. The main feature of Task Superscalar is Out-of-Order task execution. DySER architecture introduces the idea of generic dataflow accelerators integrated in a general purpose processor through ultra-wide instructions. Those generic dataflow accelerators are dynamically configured at execution time.

The computational core granularity varies from any processing element (PE) or core size in the case of DDM, Task Superscalar and Out-of-Order processors to a small SDF core. MT.Monsoon uses its PE as core granularity. Each TRIPS processor consists of 16 ALUs (the basic core of TRIPS). DySER blocks consists on 8x8 FUs circuit-switched networks. The scalability also varies from more than 1000 PEs in the case of MT. Monsoon and much more than 100 PEs in the case of the Task Superscalar down to less than 10 DySER blocks in the DySER architecture.

6.2 Comparison and Discussion

6.2.1 Enhanced Control Flow Class

Out-Of-Order architectures (restricted dataflow architectures) are the main representation of Enhanced Control Flow class. Out-of-Order processors support ILP, DLP, and TLP in the form of dual threaded cores. The number of instructions of a block (*block granularity*) is that of the instruction window, created at runtime. Out-of-Order processors use cache, memory and registers to communicate data between the blocks. Also, Out-of-Order processors use hybrid control flow/dataflow intra-block scheduling and the same communication mechanisms than for inter-block communication.

The main difference between dynamic dataflow architectures and restricted dataflow pipelines is that the latter are designed to dynamically reconstruct the dataflow graph from a sequential instruction stream. The success of such reconstruction, relies on the ability to view a window of sequential code without control instructions, is largely attributed to accurate branch prediction and speculative execution. However, such processors are also susceptible to the prohibitive costs of branch misprediction that require unrolling the execution of the wrongly predicted paths. This operation is especially costly in deeply pipelined microprocessors. On the other hand, the restricted size of the instruction window limits the number of in-flight instructions and thus, to some extent, avoids the scalability issues associated with token stores in dynamic dataflow processors.

Nevertheless, the ILP achieved by Out-of-Order microprocessors is limited by the size of the instruction window and the

amount of parallelism available in the instruction stream. In this sense, thread level speculation (TLS) may increase ILP by using speculative thread execution and a large instruction window. Indeed, thanks to the fact that each processor or processing unit only works with a limited part (i.e., instructions of thread) of the large instruction window, the complexity of concurrently monitoring the instruction issue of all the pending instructions, the data dependency cross check complexity among the instructions, and the overall branch miss-prediction are reduced. Unlike dataflow models, TLS does not require large waiting-matching store, but, it may suffer from costly checkpointing of memory accesses, squashing and re-executing threads.

6.2.2 Control Flow/Dataflow Class

Main representation of this class, TRIPS, presents a major effort to rethink the computation of conventional codes, while trying to overcome the limitations of architectures based on big cores — that is, large communication delays inside ever growing control structures. The key is dataflow execution inside 128-instruction hyperblocks (intra-block dataflow scheduling) because it allows large instruction windows to be executed with reasonable hardware resources. That intra-block scheduling is static and defined at compile time. TRIPS uses direct interconnection and also memory for intra-block communication. For inter-block communication, TRIPS uses registers. Furthermore, when not enough ILP is available, TRIPS can use its polymorphous nature that allows different modes of execution. Therefore, it can also exploit DLP or TLP through loop-unrolling or parallel thread execution. On the other hand, commercial processors can obtain similar performance results by exploiting TLP through simultaneous multithreading, and DLP through SIMD instructions [41]. Indeed, small Out-of-Order instruction windows are sufficient to efficiently extract the available ILP in conventional codes. Therefore, TRIPS can be seen as an efficient architecture that obtains similar results as classical processors with a different approach.

Another group of processors within this class use, inside an otherwise classical von-Neumann processor, a dataflow accelerator statically defined by the compiler. Within dataflow accelerators, DySER stands out because it is general purpose and presents some amount of runtime reconfiguration. Designed with power-efficiency in mind, the DySER execution model is based on the idea that a limited number of dataflow accelerators are enough to capture highly reused sections of the applications. TRIPS and DySER differ in that the former unifies dataflow and von-Neumann into a single execution model and the latter essentially use dataflow to accelerate parts of the code.

DySER architecture can also support ILP, DLP, and TLP in the form of dual threaded cores. It supports DLP and TLP based on the general purpose processor (GPP) that it incorporates and the DySER blocks integrated with the GPP. For DySER, a block is a part of the program up to hundreds of GPP ISA instructions. Unlike TRIPS, DySER uses FIFOs to communicate input data with the DySER block, and static dataflow intra-block scheduling and direct interconnection is provided. Therefore, DySER requires profiling analysis of applications in order to pre-define the instructions that are going to be accelerated with the use of the DySER blocks. Once those sets of instructions are defined, the DySER execution model dynamically reconfigures the switched-network of functional units on the DySER block for each phase of the application. This dynamic reconfiguration provides area efficiency (rather than dynamically arbitrated networks) and programmability, although it requires compiler support and a phase predictor that tries to reconfigure the DySER block before it is needed to hide the reconfiguration time. The need of profiling and the limited amount of runtime adaptability are the main disadvantages of this subclass. On the other hand they can obtain significant improvements in both performance and power efficiency over von-Neumann approaches, specially for computation intensive kernels.

6.2.3 Dataflow/Control Flow Class

In these architectures, blocks are scheduled in dataflow manner while control flow scheduling is used within the blocks. Therefore, models in this class tend to provide specific support only to TLP. In particular, DDM and Task Superscalar perform dynamic dataflow inter-block scheduling, based on dependencies specified on the program, using cache and memory respectively for inter-block communication. SDF performs static dataflow inter-block scheduling and uses frame memory and registers for inter-block communication. DDM/SDF block is equivalent to a BB, being up to 128 instructions in the case of a SDF block. Task Superscalar may have blocks of any size.

This large class can be further divided in two groups regarding the size of the blocks: small and large. The size of the blocks of DDM and SDF models tends to be small, a decision that allows large amount of parallelism to be discovered and executed but also increases the cost of the synchronization. In the case of DDM, this characteristic makes the thread scheduling unit as important as the workstation duplicating the number of necessary processing elements. Another key point of this model is that in order to be efficient, it needs more information about the program than the classical control flow model. Programs should thus be annotated either by the compiler or by the programmer, which increases the complexity of the toolchain needed to develop new applications. Unlike DDM, SDF executes the instructions within a block in-order, obtaining less ILP but allowing the execute processor of its architecture to be simpler and smaller. Another characteristic of the SDF paradigm is that, although it can take profit of annotated code, it can execute actual code as is, automatically extracting the available parallelism.

Task Superscalar is another instance of Dataflow/Control Flow class architectures, but in this case, the blocks were designed to be as large as desired. The Task Superscalar pipeline is designed as a generalization of Out-of-Order processors to the task-level. Nevertheless, its scalability goals, which target dynamically managing very large graphs consisting of tens of thousands of nodes, require an alternative design to that of Out-of-Order processors. This redesign is the result of Out-of-Order pipelines' use of reservation stations and bypass networks, whose operation is similar to that of associative token stores and are known not to scale.

The designers of the Task Superscalar pipeline thus opted for a distributed structure that, through careful protocol design that ubiquitously employ explicit data accesses, practically eliminates the need for associative lookups. The benefit of this distributed design is that it facilitates high levels of concurrency in the construction of the dataflow graph. These levels of concurrency trade off the basic latency associated with adding a new node to the graph with overall throughput. Consequently, the rate in which nodes are added to the graph enables high task dispatch throughput, which is essential for utilizing large manycore fabrics.

In addition, the dispatch throughput requirements imposed on the Task Superscalar pipeline are further relaxed by the

use of tasks, or von-Neumann code segments, as the basic execution unit. The longer execution time of tasks compared to that of instructions means that every dispatch operation occupies an execution unit for a few dozen microseconds, and thereby further amplifies the design's scalability.

The main disadvantage of models in this class is the need for annotating the actual codes in order to be able to extract a significant amount of parallelism from them. In this sense it can be observed in the designs of the programming models a trend towards simplifying the annotations as much as possible. Another common trend of this class is the increasing in both the number of processing elements and the size of the blocks.

6.2.4 Enhanced Dataflow Class

WaveScalar is the main example of the Enhanced Dataflow class. WaveScalar supports ILP and TLP. Unlike DySER and TRIPS, which need compiler support, WaveScalar performs dynamic dataflow intra-block scheduling since the dependences are detected in execution time. WaveScalar, as TRIPS, uses memory and direct interconnection for intra-block communication. For inter-block communication, WaveScalar uses memory and direct interconnection. A WaveScalar block is equivalent to a wave of instructions, although every PE caches up to 64 instructions, called segments.

The fact that WaveScalar is the only example of a mainly dataflow architecture able to execute imperative codes explains by itself the difficulties of such challenge. On the other hand this uniqueness makes WaveScalar present a very interesting set of properties. Probably the main characteristic of this model is that it was designed with Moore's Law in mind to profit from the increase in transistor density and count. Therefore, ideally the whole application would be mapped to the PEs at the same time and, in this scenario (i.e. using kernels), it is expected to clearly outperform Out-of-Order processors [118]. However, to achieve this goal the processor should have a larger number of PEs than what is possible to date. In the meantime, the need for "loading and discarding" instructions in the PEs along the program execution is one of its main bottlenecks. On the other hand, it is expected that as technology evolves this problem would diminish and the WaveScalar architecture would offer an approach than can benefit from the increasing transistor count, while keeping power consumption at bay.

6.3 Common Trends

In addition to individual features of the discussed classes, they share common properties and advantages. Moreover, they face similar challenges in their design. Recent hybrid architectures can handle imperative programming languages and data structures, as well as memory operations. This fact makes them stand out amongst other hybrid dataflow/von-Neumann architectures. It seems scheduling and memory management are key challenges in the design of hybrid architectures. One common theme among these architectures is their attempt to improve traditional processors, using dataflow principles at various levels, in order to increase the capability of providing high levels of parallelism and performance. As the matter of fact, several features of the dataflow model such as static single assignment, register renaming, dynamic scheduling and Out-of-Order instruction execution, I-structure-like synchronization and non-blocking threads are used in modern processor architectures and compiler technology. Moreover, many studies on hybrid models replace large, centralized processor cores with many simpler processing elements. In fact, all of these architectures, except WaveScalar, are von-Neumann machines, rely on a program counter between blocks (inter-block) or inside blocks (intra-block), and with some concepts of dataflow scheduling. WaveScalar eliminates both the program counter and the register file and relies completely on the dataflow program graph, allowing the arcs between waves to define interactions between them.

The hybrid architectures discussed in this paper were developed as general purpose processors, although some of them may have not achieved their goals as they failed to deliver the expected performance. Some of the hybrid architectures have limited scalability (e.g., Out-of-Order processors). In other cases, performance improvement was less than expected (e.g., TRIPS), while some of the hybrid architectures rely on new programming models (e.g., Task Superscalar and DDM). Most of them are not focused on power saving, although some dataflow based accelerators integrated with general purpose processors have been designed for energy efficiency such as C-Cores, Tartan and DySER.

7 Conclusions

This work surveys the recent general-purpose hybrid dataflow/von-Neumann architectures. To this end, we review the benefits and drawbacks of the von-Neumann and the dataflow computing models. Then, we present the common characteristics of the different hybrid models classifying them with two different taxonomies that allow to better understand their features. After that, we describe, compare and discuss a representative set of recent general-purpose hybrid dataflow/von-Neumann models. Finally, we present an insight discussion that tries to find the trends of the next generation hybrid architectures.

Nowadays, the majority of current computer systems are based on the von-Neumann model. Such processors use a program counter to sequence the execution of instructions of a program and global updatable memory. Consequently, the von-Neumann machines have two fundamental limitations: memory latencies and thread synchronization. The dataflow model has no program counter and global updatable memory so that dataflow architectures have the potential for exploiting all the parallelism available in programs. Since instructions in the dataflow models do not impose any constraints on sequencing except real data dependencies in programs, the dataflow model is asynchronous and self-scheduled.

However, although the dataflow model has been investigated since 1970s, no commercially viable global pure dataflow system has been implemented. The amount of parallelism discovered by the model becomes an implementation issue due to token matching and memory resources limitations. In theory, the dataflow model offers better performance and power efficiency than the von-Neumann model. The main reasons are the parallelism inherent to this model and that there is no overhead on pipeline control structures and temporary state (i.e., register file). Nevertheless, the efficient parallel programming of the dataflow architectures is difficult due to the fact that dataflow and functional languages do not easily support data structures, and they are not popular. On the other hand, imperative languages cannot be compiled to dataflow architectures, mainly because of issues associated with memory semantics.

Research on modern microprocessor architectures revealed the advantages of dataflow concepts in the use of instruction level parallelism. Indeed, in order to build efficient dataflow based machines, the dataflow model has to exploit some

concepts from the von-Neumann computing model. Similarly, most von-Neumann based architectures borrow concepts and mechanisms from the dataflow world to simplify thread synchronization and tolerate memory latency. As a result, the dataflow and von-Neumann models are not orthogonal, but are at two ends of a continuum. Combination, or even unification of von-Neumann and dataflow models is possible and preferred to treating them as two unrelated, orthogonal computing paradigms. Recent dataflow research incorporates more explicit notions of state into the architecture, and von-Neumann models using many dataflow techniques improve the latency hiding aspects of modern multithreaded systems.

Hybrid architectures exploit the benefits of dataflow while preserving von-Neumann capabilities and imperative languages, in order to have a high performance and low power architectures. We found that most studies of hybrid designs exploit dataflow concepts in von-Neumann based architectures, particularly in superscalar and VLIW systems, to increase the capability of providing high levels of parallelism. On the other hand, some architects of the hybrid models have attempted to increase the efficiency of dataflow based architectures by using some ideas of von-Neumann models.

Designing a general-purpose architecture is a common goal, and all hybrid architectures discussed in this paper were developed as general-purpose processors. Moreover, it is also clear that modern hybrid architectures are designed to have the ability of handling imperative programming languages and data structures as well as memory organizations. Another observed trend is that architects of recent hybrid models have attempted to replace centralized processors by several simpler processing elements, as scheduling and memory management pose key challenges in their designs. It can be observed an increase in the number of processing elements. Also it can be observed that all the architectures try to use the dataflow principles at the level (ILP, DLP or TLP) envisioned by its designers with the most potential parallelism. At the same time, the von-Neumann scheduling is maintained at the other levels to keep the needed resources at bay.

ACKNOWLEDGMENTS

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the Generalitat de Catalunya (contract 2009-SGR-980), and by the European FP7 project TERAFLUX id. 249013.The authors wish to thank Mark Oskin for his insightful comments on earlier drafts of this document.

REFERENCES

[1] "J. mcgraw and s. skedzielewski, "sisal - streams and iteration in a single assignment language reference manual (version 1. 0)", livermore national laboratory, livermore, ca, 1983."

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing," in *Intl. Symp. on Computer Architecture*, 1990, pp. 104–114.

[3] T. Agerwala and J. Cocke, "High performance reduced instruction set processors," IBM T.J. Watson Research Center Technical Report RC12434, Tech. Rep., 1987.

[4] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Intl. Symp. on Microarchitecture*, 1998, pp. 226–236.

[5] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *Intl. Symp. on Supercomputing*, 1990, pp. 1–6.

[6] Arvind, L. Bic, and T. Ungerer, "Evolution of dataflow computers," in *Advanced Topics in Data-Flow Computing*, J.-L. Gaudiot and L. Bic, Eds. Prentice Hall, 1991.

[7] Arvind and D. E. Culler, "Dataflow architectures," in *Annual Review of Computer Science vol.1*, 1986, pp. 225–253.

[8] Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessing," in *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, 1988, pp. 61–88.

[9] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Trans. on Programming Languages and Systems*, vol. 11, no. 4, pp. 598–632, 1989.

[10] P. Barahona and J. R. Gurd, "Simulated performance of the Manchester multi-ring dataflow machine," in *Parallel Computing '85*, 1985, pp. 419–424.

[11] P. Bellens, J. Perez, R. Badia, and J. Labarta, "CellSs: A programming model for the Cell BE architecture," in *Supercomputing*, 2006.

[12] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," in *Intl. Symp. on High Performance Computer Architecture*, 2012, pp. 1–12.

[13] E. Bloch, "The engineering design of the stretch computer," in *IRE-AIEE-ACM (Eastern) computer conference*, 1959, pp. 48–58.

[14] W. Bohm, W. Najjar, B. Shankar, and L. Roh, "An evaluation of coarse grain dataflow code generation strategies," in *Programming Models for Massively Parallel Computers*, 1993, pp. 63–71.

[15] R. Buehrer and K. Ekanadham, "Incorporating data flow ideas into von Neumann processors for parallel execution," *IEEE Trans. Comput.*, vol. 36, no. 12, pp. 1515–1522, 1987.

[16] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. K. John, C. Lin, C. R.Moore, J. Burrill, R. G. McDonald, W.Yoder, and the TRIPS Team, "Scaling to the end of silicon with EDGE architectures," *IEEE Computer*, vol. 37, no. 7, pp. 44–55, 2004.

[17] M. Cintra, J. F. Martínez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *Intl. Symp. on Computer Architecture*, 2000, pp. 13–24.

[18] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson, "Design and implementation of the POWER5 TM microprocessor," in *annual Design Automation Conference*, 2004, pp. 670–672.

[19] K. Coons, X. Chen, S. K. Kushwaha, D. Burger, and K. McKinley, "A Spatial Path Scheduling Algorithm for EDGE Architectures," *SIGPLAN Not.*, vol. 41, no. 11, pp. 129–140, 2006.

[20] T. M. Corp., "Connection machine model CM-2 technical summary," Thinking Machines Corp. Technical Report TR89-1, Tech. Rep., 1989.

[21] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero, "Kilo-instruction processors: Overcoming the memory wall," *IEEE Micro*, vol. 25, no. 3, pp. 48–57, 2005.

[22] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. Eicken, "TAM: A compiler controlled threaded abstract machine," *Journal of Parallel & Distributed Computing*, vol. 18, no. 3, pp. 347–370, 1993.

[23] D. E. Culler and G. M. Papadopoulos, "The explicit token store," *Journal of Parallel & Distributed Computing*, vol. 10, no. 4, pp. 289–308, 1990.

[24] D. E. Culler, K. E. Schauser, and T. Eicken, "Two fundamental limits on dataflow multiprocessing," in *Proceeding IFIP WG 10.3 Conf. on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism*, 1993.

[25] A. L. Davis and R. Keller, "Data flow program graphs," *IEEE Computer*, vol. 15, no. 2, pp. 26–41, 1982.

[26] L. Davis, "The architecture and system method of DDM1: A recursively structured data driven machine," in *Intl. Symp. on Computer Architecture*, 1978, pp. 210–215.

[27] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, ser. Lecture Notes in Computer Science, B. Robinet, Ed. Springer Berlin/Heidelberg, 1974, vol. 19, pp. 362–376.

[28] J. B. Dennis and G. R. Gao, "Multithreaded architectures: Principles, projects, and numbers," School of Computer Science, McGill University, Montreal, Quebec, CA, Tech. Rep., 1994.

[29] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Intl. Symp. on Computer Architecture*, 1975, pp. 126–132.

[30] J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, vol. 13, no. 11, pp. 48–56, 1980.

[31] ——, "The varieties of data flow computers," in *Advanced computer architecture*, 1986, pp. 51–60.

[32] J. R. Ellis, "Bulldog: a compiler for VLIW architectures (parallel computing, reduced-instruction-set, trace scheduling, scientific)," Ph.D. dissertation, Yale University,New Haven, CT, USA, 1985.

[33] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *Intl. Symp. on Microarchitecture*, 2010, pp. 89–100.

[34] Y. Etsion, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: Using processors as functional units," in *Hot Topics in Parallelism*, 2010.

[35] P. Evripidou and J. L. Gaudiot, "A decoupled graph/computation data-driven architecture with variable-resolution actors," in *Intl. Conf. on Parallel Processing*, 1990, pp. 405–414.

[36] ——, "The USC decoupled multilevel dataflow execution model," in *Advanced topics in data-flow computing*, J.-L. Gaudiot and L. Bic, Eds. Prentice Hall, 1991, pp. 347–379.

[37] J. A. Fisher, "Very long instruction word architectures and the ELI-512," *SIGARCH Comput. Archit. News*, vol. 11, no. 3, pp. 140–150, 1983.

[38] M. Frank, C. A. Moritz, B. Greenwald, S. Amarasinghe, and A. Agarwal, "SUDS: Primitive mechanisms for memory dependence speculation," Cambridge, UK, Tech. Rep., 1999.

[39] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, pp. 552–571, 1996.

[40] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller, "The Sisal model of functional programming and its implementation," in *Intl. Symp. on Parallel Algorithms / Architecture Synthesis*, 1997, pp. 112–.

[41] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley, "An evaluation of the TRIPS computer system," in *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems*, 2009, pp. 1–12.

[42] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Intl. Symp. on Computer Architecture*, 2011, pp. 235–246.

[43] E. Gluck-Hiltrop, M. Ramlow, and U. Schurfeld, "The Stollman dataflow machine," in *Lect. Notes Comput. Sc.*, 1989, pp. 433–457.

[44] S. Gopal, T. N. V. James, E. Smith, and G. S. Sohi, "Speculative versioning cache," in *Intl. Symp. on High Performance Computer Architecture*, 1998, pp. 195–205.

[45] V. Govindaraju, C. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Intl. Symp. on High Performance Computer Architecture*, 2011.

[46] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, pp. 38–51, 2012.

[47] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. Holmes, "The Epsilon dataflow processor," in *Intl. Symp. on Computer Architecture*, 1989, pp. 36–45.

[48] V. G. Grafe and J. Hoch, "The EPSILON-2 multiprocessor system," *Journal of Parallel & Distributed Computing*, vol. 10, no. 4, 1990.

[49] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger, "Implementation and evaluation of on-chip network architectures," in *Intl. Conf. on Computer Design*, 2006, pp. 477–484.

[50] M. Gupta and R. Nim, "Techniques for speculative run-time parallelization of loops," in *Supercomputing*, 1998.

[51] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Comm. ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[52] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Intl. Symp. on Computer Architecture*, 2010, pp. 37–47.

[53] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems*, 1998, pp. 58–69.

[54] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Intl. Symp. on Computer Architecture*, 1993, pp. 289–300.

[55] J. Hicks, D. Chiou, B. S. Ang, and Arvind, "Performance studies of Id on the Monsoon dataflow system," *Journal of Parallel & Distributed Computing*, vol. 18, no. 3, pp. 273–300, 1993.

[56] D. Hillis, "The connection machine," Ph.D. dissertation, Department of Electrical Engineering and Computer Science (EECS), MIT, 1988.

[57] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Intl. Symp. on Computer Architecture*, 2010, pp. 280–289.

[58] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryky, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnany, A. Marquez, S. Merali, S. S. Nemawarkarz, P. Panangaden, X. Xue, and Y.Zhu, "A design study of the EARTH multiprocessor," in *Intl. Conf. on Parallel Arch. & Compilation Techniques*, 1995, pp. 59–68.

[59] H. H. J. Hum, O. Maquelin, K. Theobald, X. Tian, G. Gao, and L. Hendren, "A study of the EARTH-MANNA multithreaded system," *Parallel Programming*, vol. 24, no. 4, pp. 319–348, 1996.

[60] W. Hwu and Y. N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," in *Intl. Symp. on Computer Architecture*, 1986, pp. 297–306.

[61] R. A. Iannucci, "Toward a dataflow/von neumann hybrid architecture," in *Intl. Symp. on Computer Architecture*, 1988, pp. 131–140.

[62] R. A. Iannucci, G. R. Gao, R. H. H. Jr., and B. Smith, *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, 1994.

[63] N. Ito, M. Sato, E. Kuno, and K. Rokusawa, "The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D," in *Intl. Symp. on Computer Architecture*, 1986, pp. 149–156.

[64] H. F. Jordan, "Performance measurements on HEP- a pipelined MIMD computer," in *Intl. Symp. on Computer Architecture*. ACM, 1983, pp. 207–212.

[65] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, vol. 74, 1974.

[66] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM Journal of Applied Mathematics*, vol. 14, no. 5, pp. 1390–1411, 1966.

[67] K. M. Kavi, B. Buckles, and U. Bhat, "A formal definition of data flow graph models," *IEEE Trans. on Computers*, vol. 35, no. 11, pp. 940–948, 1986.

[68] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *IEEE Transactions on Computers*, vol. 50, pp. 834–846, 2001.

[69] C. Kim and J. L. Gaudiot, *Dataflow and Multithreaded Architectures*. Wiley, New York, 1997.

[70] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *Intl. Symp. on Microarchitecture*, 2007, pp. 381–394.

[71] M. Kishi, H. Yasuhara, and Y. Kawamura, "DDDP — a distributed data driven processor," in *Intl. Symp. on Computer Architecture*, 1983, pp. 236–242.

[72] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi, "The EM-X parallel computer: Architecture and basic performance," in *Intl. Symp. on Computer Architecture*, 1995, pp. 14–23.

[73] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Transactions on Computers*, vol. 48, pp. 866–880, 1999.

[74] V. Krishnan and L. J. Torrellas, "The need for fast communication in hardware-based speculative chip multiprocessors," in *Intl. Conf. on Parallel Arch. & Compilation Techniques*, 1999.

[75] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C. Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, and J. Andrews, "The Cedar system and an initial performance study," in *Intl. Symp. on Computer Architecture*, 1993, pp. 213–223.

[76] K. Kurihara, D. Chaiken, and A. Agarwal, "Latency tolerance through multithreading in large-scale multiprocessors," in *Intl. Symp. on Computer Architecture*, 1991, pp. 91–101.

[77] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, 2006.

[78] B. Lee and A. Hurson, "Dataflow architectures and multithreading," *Computer*, vol. 27, no. 8, pp. 27–39, 1994.

[79] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[80] P. Marcuello and A. González, "Clustered speculative multithreaded processors," in *Intl. Symp. on Supercomputing*, 1999, pp. 365–372.

[81] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, and M. U. J. A. Miller, "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 1–12, 2002.

[82] L. A. J. Marzulo, F. M. G. Franca, and V. S. Costa, "Transactional WaveCache: Towards speculative and out-of-order dataflow execution of memory operations," in *Intl. Symp. on Computer Architecture & High Performance Computing*, 2008, pp. 183–190.

[83] W. M. Miller, W. A. Najjar, and A. P. W. Böhm, "A quantitative analysis of locality in dataflow programs," in *Intl. Symp. on Microarchitecture*, 1991, pp. 12–18.

[84] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems*, 2006, pp. 163–174.

[85] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, no. 13-14, pp. 1907–1929, 1999.

[86] S. S. Nemawarkar and G. R. Gao, "Measurement and modeling of EARTH-MANNA multithreaded architecture," in *Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 1996, pp. 109–.

[87] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[88] R. S. Nikhil, "Can dataflow subsume von neumann computing?" in *Intl. Symp. on Computer Architecture*, 1989, pp. 262–272.

[89] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "*T: A multithreaded massively parallel architecture," in *Intl. Symp. on Computer Architecture*, 1992, pp. 156–167.

[90] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," in *Intl. Symp. on Computer Architecture*, 1990, pp. 82–91.

[91] G. M. Papadopoulos and K. R. Traub, "Multithreading: A revisionist view of dataflow architectures," in *Intl. Symp. on Computer Architecture*, 1991, pp. 342–351.

[92] Y. N. Patt, W. M. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Intl. Symp. on Microarchitecture*, 1985, pp. 103–108.

[93] S. Pei, B. Wu, M. Du, G. Chen, L. A. J. Marzulo, and F. M. G. Franca, "SpMT WaveCache: Exploiting thread-level parallelism in wavescalar," in *Congress on Computer Science and Information Engineering*, 2009.

[94] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *IEEE International Conference on Cluster Computing*, 2008, pp. 142–151.

[95] A. Petersen, A. Putnam, M. Mercaldi, A. Schwerin, S. Eggers, S. Swanson, and M. Oskin, "Reducing control overhead in dataflow architectures," in *Intl. Conf. on Parallel Arch. & Compilation Techniques*, 2006, pp. 182–191.

[96] A. Plas, D. Comte, O. Gelly, and J. Syre, "LAU system architecture: A parallel data-driven processor based on single assignment," in *Intl. Conf. on Parallel Processing*, 1976, pp. 293–302.

[97] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas, "Removing architectural bottlenecks to the scalability of speculative parallelization," in *Intl. Symp. on Computer Architecture*, 2001, pp. 204–215.

[98] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," in *Conf. on Programming Language Design and Implementation*, 1995, pp. 218–232.

[99] J. E. Requa, "The piecewise data flow architecture control flow and register management," in *Intl. Symp. on Computer Architecture*, 1983, pp. 84–89.

[100] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley, "Strategies for mapping dataflow blocks to distributed hardware," in *Intl. Symp. on Microarchitecture*, 2008, pp. 23–34.

[101] B. Robic, J. Silc, and T. Ungerer, "Beyond dataflow," *Computing and Information Technology*, vol. 8, no. 2, pp. 89–101, 2000.

[102] L. Roh and W. Najjar, "Design of storage hierarchy in multithreaded architectures," in *Intl. Symp. on Microarchitecture*, 1995, pp. 271–278.

[103] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace processors." in *Intl. Symp. on Microarchitecture*, 1997, pp. 138–148.

[104] P. Rundberg and P. Stenstrom, "Low-cost thread-level data dependence speculation on multiprocessors," in *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, 2000.

[105] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Intl. Symp. on Computer Architecture*, 2003, pp. 422–433.

[106] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. Mcdonald, and C. R. Moore, "TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP," *ACM Trans. on Arch. & Code Optim.*, vol. 1, no. 1, pp. 62–93, 2004.

[107] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P.Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed microarchitectural protocols in the TRIPS prototype processor," in *Intl. Symp. on Microarchitecture*, 2006, pp. 480–491.

[108] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *Intl. Symp. on Microarchitecture*, 2000, pp. 42–53.

[109] J. Silc, B. Robic, and T. Ungerer, "Asynchrony in parallel computing: From dataflow to multithreading," *Journal of Parallel & Distributed Computing*, pp. 1–33, 1998.

[110] ——, *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer-Verlag, 1999.

[111] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. C. Burger, and K. S. McKinley, "Compiling for EDGE architectures," in *Intl. Symp. on Code Generation & Optimization*, 2006, pp. 185–195.

[112] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system." in *Real Time Signal Processing IV, Proceedings of SPIE*, 1981, pp. 241–248.

[113] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Intl. Symp. on Computer Architecture*, 1998, pp. 291–299.

[114] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Intl. Symp. on Computer Architecture*, 1995, pp. 414–425.

[115] V. P. Srini, "An architectural comparison of dataflow systems," *IEEE Computer*, vol. 19, no. 3, pp. 68–88, 1986.

[116] J. G. Steffan, "Hardware support for thread-level speculation," Ph.D. dissertation, Computer Science Department,Carnegie Mellon University, Pittsburgh,PA, USA, 2003.

[117] J. Strohschneider, B. Klauer, S. Zickenheimer, and K. Waldschmidt, "ADARK: A fine grain dataflow architecture with associative communication network," in *EUROMICRO Conf.*, 1994, pp. 445–450.

[118] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Intl. Symp. on Microarchitecture*, 2003, pp. 291–302.

[119] S. Swanson, A. Putnam, M. M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. J. Eggers, "Area-performance trade-offs in tiled dataflow architectures," in *Intl. Symp. on Computer Architecture*, 2006, pp. 314–326.

[120] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. Eggers, "The WaveScalar architecture," *ACM Trans. on Computer Systems*, vol. 25, no. 2, pp. 4:1–4:54, 2007.

[121] K. B. Theobald, "EARTH: An efficient architecture for running threads," Ph.D. dissertation, McGill University, Montreal, Quebec, CA, 1999.

[122] X.-M. Tian, S. Nemawarkar, G. R. Gao, H. Hum, O. Maquelin, A. Sodan, and K. Theobald, "Quantitive studies of data-locality sensitivity on the EARTH multithreaded architecture: preliminary results," in *Intl. Conf. on High-Performance Computing*, 1996, pp. 362–.

[123] P. Treleaven, R. Hopkins, and P. Rautenbach, "Combining data flow and control flow computing," *Computer Journal*, pp. 207–217, 1982.

[124] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Intl. Symp. on Computer Architecture*, 1995, pp. 392–403.

[125] R. Vedder and D. Finn, "The Hughes data flow multiprocessor: Architecture for efficient signal and data processing," in *Intl. Symp. on Computer Architecture*, 1985, pp. 324–332.

[126] A. H. Veen, "Dataflow machine architecture," *ACM Computing Surveys*, vol. 18, no. 4, pp. 365–396, 1986.

[127] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems*, 2010, pp. 205–218.

[128] J. von Neumann, "First draft of a report on the EDVAC," Between the United States Army Ordnance Department and the University of Pennsylvania Moore, School of Electrical Engineering, Tech. Rep., 1945.

[129] W.-D. Weber and A. Gupta, "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results," in *Intl. Symp. on Computer Architecture*. ACM, 1989, pp. 273–280.

[130] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky, "Performance estimation of multistreamed, supersealar processors," in *Hawaii Intl. Conf. on System Sciences*, 1994, pp. 195–204.

[131] F. Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia, "Fpga-based prototype of the task superscalar architecture," in *Proceedings of the 7th HiPEAC Workshop on Reconfigurable Computing*, 2013.

[132] T. Yuba, K. Hiraki, T. Shimada, S. Sekiguchi, and K. Nishida, "The SIGMA-1 dataflow computer," in *Computer Conference on Exploring technology: today and tomorrow*, 1987, pp. 578–585.

[133] E. Zehender and T. Ungerer, "The ASTOR architecture," in *Intl. Conf. on Distributed Computing Systems*, 1987, pp. 424–430.

[134] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors," in *Intl. Symp. on High Performance Computer Architecture*, 1999, pp. 135–.