Asynchronous Dataflow De-Elastisation for Efficient Heterogeneous Synthesis

Mahdi Jelodari Mamaghani*, Danil Sokolov† and Jim Garside*
*School of Computer Science, The University of Manchester, Manchester M13 9PL, UK
†uElectronics Group, Newcastle University, Newcastle-upon-tyne, UK
Email: *Mahdi.Jelodari@manchester.ac.uk, †Danil.Sokolov@newcastle.ac.uk, *Jim.Garside@manchester.ac.uk

***Abstract*—Algorithmic synthesis provides flexibility in design-space exploration and improves design productivity by separating the concerns of system timing and functionality. This enables a designer to cope with the rapid increase of SoC complexity and to employ different computation and communication models with various timing constraints. De-elastisation emerged as a technique that transforms timing-free concurrent dataflow models to synchronous circuits while offering selective timing flexibility in the design.**

We adopt De-elastisation in an in-house EDA flow: it starts from a system specification in the Balsa language and uses eTeak to generate an elastic network of macro-modules. Based on structural analysis of the obtained network some of its portions are selectively transformed into synchronous circuits, in a supervised fashion, targeting better power and performance in the computation domain, whilst preserving fine-grained elasticity between communicating modules to handle timing uncertainties. We evaluate De-elastisation and compare it against some popular high-level synthesis technologies, namely LegUp, Bluespec, Chisel and Balsa using a set of benchmarks from the domain of Database Management Systems (DBMS) accelerators. Our experiments demonstrate the efficacy of Dataflow Decomposition and De-elastisation on the selected range of applications and its advantages in exploring the design trade-offs: a twofold increase in performance and 15% decrease in power consumption can be achievable at the expense of moderate area overhead.

I. Introduction

Extreme heterogeneity is believed to act as a cure for the power issue in the domain of the general purpose processors through coprocessor-dominant architectures [1]. In recent years, industry has been successful in adopting heterogeneity at the coarse level (e.g. ARMs big.LITTLE architecture) that enables efficient utilisation of the energy budget [2]. A follow-up research has shown that energy efficiency can be further improved by pushing heterogeneity into a core [3] and exploring it at a finer level of granularity [4]. The rapid growth of demand for high-performance, energy-efficient devices calls for advanced design and synthesis technologies. High-Level Synthesis (HLS) addresses this demand and offers a flexible, productive and customisable design methodology for the realisation of complex systems. It allows the designer to focus on specifying the system functionality and postpone the issues of timing to the subsequent stages of the synthesis flow. It is claimed that raising the abstraction level enables designers to have almost 10× higher chance to improve power closure by architectural exploration [5]. Moreover, synthesis from a behavioural specification in an algorithmic language is shown to be more productive than a low-level realisation of a complex system using Hardware Description Languages (HDLs) [6].

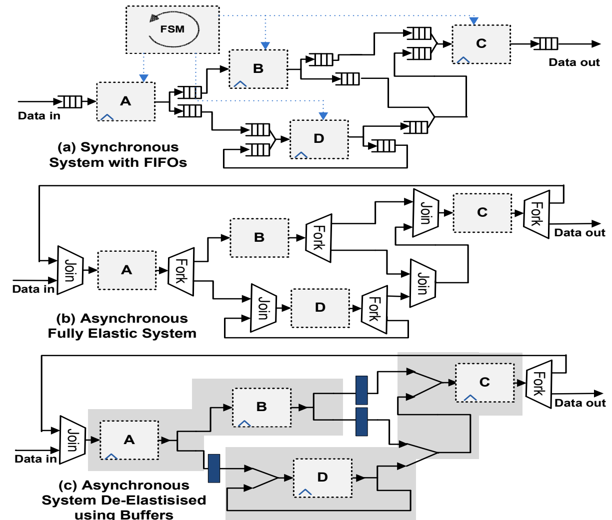

Fig. 1. A system consist of four synchronous blocks: a) shows a conventional synchronous design elastisised by insertion of FIFOs between the blocks, b) depicts a fine-grained asynchronous elastic dataflow system made of primitive components (Buffer, Join, Fork, Steer and Merge), communicating through channels and c) is a locally De-elastisised version of (b) where the dataflow model is re-timed by buffers insertion to make synchronous 'islands' within a globally elastic ecosystem.

In the past decade, Bluespec [7] and, lately, Chisel [8], have emerged as powerful synthesis frameworks. These leverage the power of functional programming languages (Haskell and Scala) to realise concurrent structures in the form of dataflows. Another example is an open-source HLS framework, LegUp [9], that is based on the C language and aims to abolish hardware/software boundaries, providing the designer with flexible design space exploration.

Asynchronous circuits, on the other hand, are well-known for their capability in clock-less computation and elastic communication. In contrast with synchronous circuits, asynchrony rely on handshake signals for local interactions. This facilitates designing a system at a higher level of abstraction, describing *dataflows* rather than thinking in terms of RTL and time-budgeting sequential tasks. The need for handshake circuits, however, may result in significant (up to 4 times) area and performance overheads [10]. Also, as feature sizes shrink, sub-threshold leakage poses low-power challenges so fully-interlocked larger circuits may be detrimental here, too.

ASIC implementation still largely relies on D-type flip-flops and a synchronous Register Transfer Level (RTL) model where the data and control paths are separated. For many

reasons, not least performance, algorithms are mapped onto synchronous circuits where 'communication' is based on the assumptions – verified by the design tools – that data will move within well-controlled time steps. This has long proved a good abstraction but there is a tension between this expression of the algorithm and a timing-free dataflow description.

De-elastisation [11] has emerged as a technique to transform asynchronous dataflows to synchronous circuits whilst *selectively* preserving elasticity in the design. Figure 1 shows the difference between data and control driven architectures and how static scheduling through buffer insertion can optimise away the handshake circuitry.
Unlike de-synchronisation [12] which aims to enable synchronous designs to benefit from asynchronous timing discipline, De-elastisation provides a mechanism for self-timed systems to exploit the advantages of rigid timing behaviour. De-elastisation alleviates the communication overheads by identifying islands of synchrony and replacing their fine-grain handshakes by a rigid clock. Between the synchronous islands, the elasticity is still preserved by coarse-grain handshakes, thus forming a Globally-Asynchronous Locally-Synchronous (GALS) [13] ecosystem.

We explore Dataflow De-elastisation together with Decomposition (D3) to efficiently transform dataflow cycles selectively into synchronous circuits *with respect to their high-level behaviour*. The proposed exploration in this work paves the way for industrial adoption of elasticity at fined grain level. D3 is evaluated against RTL and three popular HLS frameworks, namely network of atomic rules (Bluespec), synthesisable by construction (Chisel) and C functions to FSM (LegUp). Our case study is based on a set of data mining algorithms from the domain of database management system [14]. The experimental results show that D3 can outperform the synchronous models by 2× whilst improving the energy efficiency by 15%.

The contributions of this paper can be summarised as follows:

- De-elastisation is adapted in an in-house EDA flow. It is benchmarked against three popular HLS frameworks, namely network of atomic rules (Bluespec), synthesisable by construction (Chisel) and C functions to FSM (LegUp).
- A representative set of algorithms (including Database sort and Search) [14] is synthesised using the established HLS schemes and De-elastisation technology. The generated circuits are compared in terms of throughput, energy efficiency and size.
- The Architectural capabilities of the selected HLS frameworks are analysed based on the levels of granularity and abstraction. This provides the designer with an insight into elasticity in the design space and is generalisable to more complicated models at system level.

II. Popular Synthesis Flows

This section overviews the most popular synthesis flows used for system-level design.
HDL-Verilog is the most popular language in IC design that benefits from well-established libraries and EDA tools. However, the RTL abstraction of HDL-Verilog is fundamentally timed and forces designers to squeeze tasks into clock boundaries. The majority of HLS flows raise the abstraction to a timing-free level. As output, RTL Verilog is generated to reuse the state-of-the-art logic synthesis flow and established techniques for individual optimisation of data and control paths.

For the sake of productivity it is desirable to abstract hardware design at a higher level, describing dataflows rather than RTL interactions. Much like a software description, these high-level flows neglect implementation details such as timing and are, naturally, asynchronous and elastic. Below is an overview of the most popular HLS flows developed both in academia and industry:

BlueSpec [7] is an atomic rule-based synthesis scheme capable of synthesising dataflow networks of rules. It uses an HDL-like Bluespec Verilog (BSV) language that exploits SystemVerilog as a front-end and adopts Haskell's state-based computation model for the scheduling rules (these are embedded in higher-order functions that are guarded by conditions). This model is fundamentally parallel, in contrast to sequential programming models (such as C++) which leverage an extension to model parallelism. Rules in BSV are defined by the user and employ the all-at-once updating approach similar to Verilog when a clock executes the *always* blocks forcing the state-holding elements (registers) to update simultaneously. BSV uses *Atomic Transactional Memories* to handle communication between modules/rules/processes and follows the *Latency-Tolerant Shared-Memory* model [15].

Chisel [8] is a synthesisable-by-construction graph-based synthesis scheme that allows meta-programming features (e.g. recursion and parameterisation) to be exploited for scalable hardware design. It uses an extended version of the Scala programming language for defining modules, wires and registers. Although a recent contribution proposes an elastic NoC fabric [16] for large-scape multiprocessor chips and SoCs using Chisel, its coarse-grained nature prevents intra-process exploration of elasticity and adopting it at fine-grained level towards energy efficient design.

LegUp [9], similar to other C-to-Verilog compilation paradigms, takes C functions and transforms them into a mid-level representation using the LLVM compiler. It exploits advanced compilation features, such as loop unrolling, vectorising and parallel realisation. Thereafter, LLVM code of each function, is translated into a Finite State Machine (FSM) that has access to the registers implementing global and local variables. The communication between the FSMs is through a bus that also provides shared access to a general purpose processor. This processor is employed to run hard-to-synthesise or non-synthesisable functions of the C code. LegUp leaves the tasks of decomposing the code into functions and considering parallelism using *pthread* to the designer. Unlike domain-specific languages and tools, C-based synthesis flow provide a facility for software developers to reach hardware level implementation without involving timing. However, as C was not designed for IC synthesis, some functions can result in an inefficient implementation. To improve cycle time within pipelines a De-pipelining technique has been proposed [17], which could be interpreted as a move from synchrony to asynchrony and considering hybrid architectures.

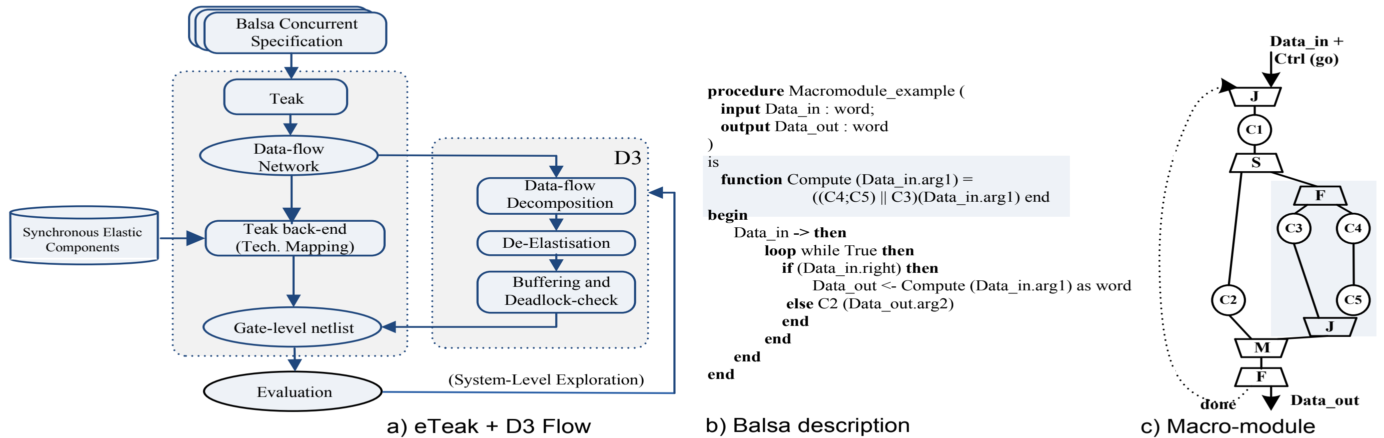

Fig. 2. Synchronous Teak flow with the D3 technique incorporated to it for power and performance enhancement; dataflow synthesis example: b) System level description of an iterative function, like MPEG decoder, in the Balsa language; c) macro-module implementation of the Balsa input using primitive components (**B**uffer, **J**oin, **F**ork, **M**erge, **S**teer and **C**omputation) for data manipulation.

III. Elasticity and De-elastisation

Elasticity emerged as a solution for tolerating uncertainty or non-determinism in computation and communication. The idea was introduced by Carloni et al. [18] at system level and was formalised by Carmona et al. [10] for exploitation in CAD flows, which made it applicable from transistors to the system level. Elasticity can also be viewed as a technique that realises 'On-demand Computing', such that the absence of data implies that computation is not necessary, therefore it can be 'switched off'. This feature has significant impact on power and nowadays is being used as *clock gating* and *dynamic power gating* in various contexts. Elasticity comes with costs: if the designer chooses to apply elasticity at a fine-grained level, its communication overhead may prohibitively dominate computation costs.

De-elastisation *selectively* removes elasticity from an asynchronous circuit, adopting a clocked protocol without handshaking in selected parts of the circuit without forcing synchronisation on the circuit as a whole. This results in a fine grained GALS structure where regions may be run by different clocks or the intervening elastic buffers may use synchronous handshakes within a region. This process should be considered with some circuit level restrictions.

To safeguard functionality loops are categorised into *blocking* and *non-blocking* architectures. These models are commonly used at different levels of abstraction. For instance, at operating system level for handling IO devices the 'polling' and 'interrupt' mechanisms are considered which follow the exact communication pattern at higher level between devices. These mechanism are categorised under asynchronous IO or a non-blocking IO processing model that allows other computations to continue before the communication has terminated. Dataflow systems are well-known for their concurrent nature which is in a close relation with the implementation style that exploits push channels – where data tokens instigate a transfer. In contrast to conventional dataflows [19], Teak provides pull channels (triggered by a *want* of data) to read data which significantly simplifies its storage structures.
Accordingly loops in the Teak dataflow networks are categorised into two types; Type 1: the loops that pull data on-demand become non-blocking (suitable for modelling determinism) and Type 2: which receive data through push channels emerge as blocking (suitable for modelling non-determinism). In Type 1 data is pulled so a bounded degree of latency is expectable and deterministic, therefore the on going computation will not be blocked; Unlike Type 1, in Type 2 data is pushed into the loop and it may be blocked by Join/Fork guard until the on going computation terminates.
The proposed classification is leveraged to mark determinism and non-determinism in eTeak dataflows so that De-elastisation can be applied *selectively*.

IV. Dataflow Decomposition & De-elastisation

In this section we describe the eTeak framework and next propose a set of patterns for exploring elasticity at different levels of granularity. An approach to HLS that neglects rigid clocking was taken in the asynchronous community [20][21][22]. These exploit the Communicating Sequential Processes (CSP) model to implement concurrency wherein the processes communicate through message passing which has to be realised using handshaking. The obtained circuits, however, suffered from performance penalties (due to handshake communications when transferring data rather than relying on evaluating within a prescribed cycle time) and area overheads (due to the need for the extra handshake circuitry). These drawbacks were (partially) addressed by improving the control synthesis flow. Recently, Teak [23] and Click [24] have been proposed as dataflow synthesis frameworks to tackle the control overhead issue. Although Teak exploits the dataflow model of computing to overcome the control overhead, its fine-grained elastic nature remains a major bottleneck for performance improvement.

A. The Base Framework: A CSP-based Synthesis Flow

Teak [23] is a dataflow synthesis backend for Balsa which is a CSP-like language that allows defining sequential processes with channels between them for message passing. Taylor et al. [19] demonstrated that the asynchronous data-driven synthesis can overcome the performance bottlenecks

of control-driven circuits [20] which has provided sufficient evidence to support dataflow synthesis. As with any HLS flow, Teak has three major steps for transforming compiled high-level code, in form of control-data-flow graph (CDFG), into hardware logic: a) Scheduling, b) Allocation and c) Binding. Unlike conventional synthesis flows, Teak uses a syntax-directed synthesis mechanism. Therefore the binding and allocation phases are simplified but a set of peep-hole optimisations are considered to remove redundancy in the circuit instead. Initially in the compilation phase, an extracted CDFG is implemented as macro-modules where control signals are encoded in data (Fig. 2). Macro-module logic enables designers implement complex circuits using simple data processing building blocks [25]. Later, this concept was used to simplify the asynchronous control design [26]. Teak uses this technique to perform control interactions locally instead of employing a separate, central unit, which has significant performance implications. After generating macro-modules, the scheduling phase is simplified to a register insertion problem (aka retiming in the synchronous domain) which handles deadlocks in the design.

With the emergence of the synchronous elastic protocol (SELF) [27], asynchronous synthesis shifted towards synchrony to leverage its advantages as synchronous circuits are efficient, well understood and well supported by modern EDA tools. eTeak [28], as a synchronous/GALS extension to Teak, exploits SELF at the binding phase where a synchronous elastic library of components are used such as controllers. Adapting SELF not only permits fine-grained elasticity to be preserved in the design but also simplifies the buffer insertion problem at the scheduling phase. Moreover, when allocating the computation blocks commercial synchronous EDA can use the RTL library to optimise the circuit which improves the area by a factor of four [29]. Regarding the *Slack Elastic* property [30], Teak circuits can be pipelined with any degree of storage on their communication channels. This property provides a flexible communication environment for the computational blocks. eTeak preserves this property and safely optimises the computational blocks without affecting the overall functionality. We use the eTeak system as our baseline synthesis platform for exploring De-elastisation. Its cycle-accurate nature allows easier timing analysis over the generated circuits.

B. eTeak+D3 Overall Flow

D3 adopts De-elastisation together with Dataflow Decomposition to explore elasticity in a bottom-up fashion at three levels of granularity: *component-level*, *stage-level*, and *loop-level*. The overhead of elasticity at gate and transistor level imposes an area overhead of 4×, exhibits profound power drawbacks, and therefore is out of interest for this work.

Figure 2(a) depicts D3 incorporated into the eTeak framework. D3 is the main step toward exploring synchronous versus asynchronous/elastic heterogeneity in this work. The channel-based communication between the high-level procedures in the Balsa language facilitates functional Decomposition of computation in our dataflow model. Respecting that eTeak inherits slack elasticity, it allows selective De-elastisation to be applied onto an individual procedure without affecting its successor or predecessor procedures. However, to avoid unnecessary stalls in the system, proper number of buffers have to be inserted on every channels after D3 takes place.

De-elastisation enjoys an architectural degree of freedom for establishing synchronous localities/domains in the design. Regarding the slack elastic nature of the eTeak networks, it is possible to insert any bounded number of buffers on every link without affecting the functionality. In a FIFO based design, FIFOs, which are basically elastic buffers, are inserted at the boundaries. To determine their size, the worst-case scenario is usually considered. This may impose area and energy overhead to the design. De-elastisation at finer level of granularity can avoid this by reconsidering the elastic boundaries with regard to the intra-domain buffering requirements which may lead to more appropriate FIFO sizing, and thus less expense.

The combinational nature of the elastic Joins, especially those with high fan-in, contributes overhead to the critical paths. By removing elastic Joins in the course of De-elastisation, eager forks become redundant and are safe to be replaced with non-elastic components. The transformed circuits have reduced area and boosted clock frequency due to the removal of unnecessary combinatorial logic. Subsequent re-synthesis using synchronous EDA may give further improvements.

C. The Intuition

An intuitive understanding of the Decomposition and De-elastisation technique can be obtained by analysing the example shown in Figure 3, where (a) is a dataflow realisation of the high-level expression of C1 ; [(C2 ||C3) ? C4] in Balsa; where C2 and C3 are specified as parallel computations and are merged with C4. C1 is also considered in sequence with the rest. Each computation could represent a simple arithmetic unit or a complicated module, depending on the design's abstraction level. For performance analysis blocks are annotated with arbitrary delays in this example. The question mark in the expression implies data-dependency and infers a Steer-Merge pair (aka Choice) with α and β probabilities on its a and b branches, respectively. The associated throughput is denoted by θ and is proportional to:

$$\theta \propto \frac{\sum m(e)}{\gamma \cdot \delta} \tag{1}$$

where m denotes the sum of active tokens on the edges (e) of the loop, which is assumed to be one in this example; δ represents the critical path delay which, in this example, equals to the associated delay of C3 (4ns) plus the delay of the combinatorial components, which are assumed to be 0.1ns each. Therefore the overall delay is 4.4ns (the red dotted path). γ also represents the Local Cycle Time (LCT) for every single cycle in the design. LCT is defined as the time between the arrival of a token and the time that the stage is ready to receive another token [31]. In this example γ equals to $4.\alpha$ + $3.\beta$. Figure 3(b) depicts a decomposed implementation of (a) into two shorter cycles. Decomposition may improve the overall throughput by increasing the number of active tokens in the system and decreasing LCT. Since in the synchronous elastic circuits of eTeak time is discretised, we measure LCT according to the clock. In this example the number of active tokens is doubled after decomposition and the new LCT is:

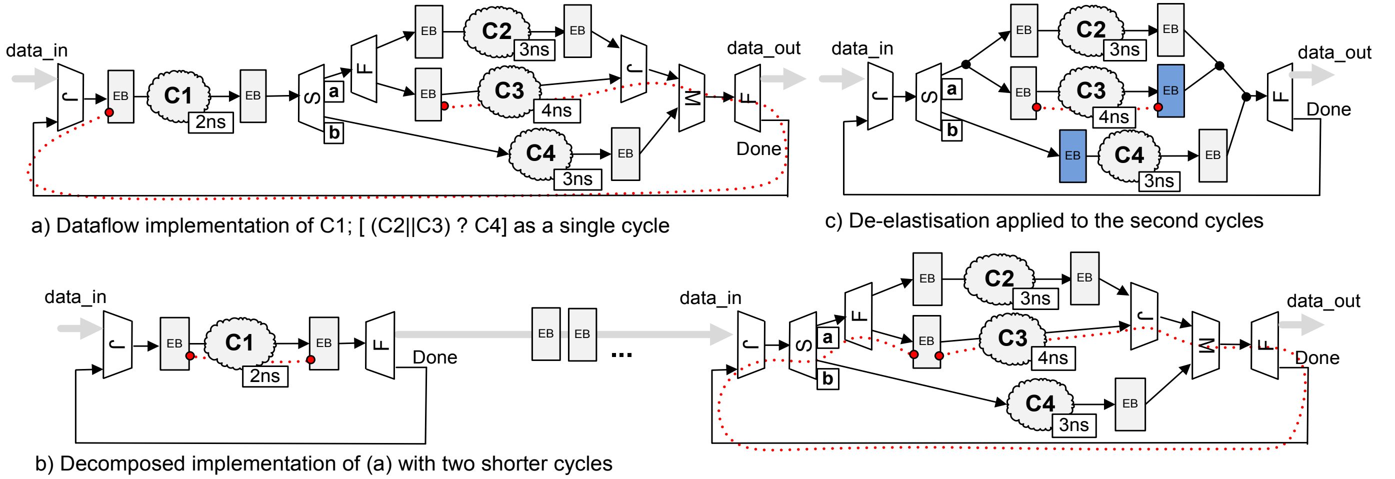

Fig. 3. The energy and performance impact of Decomposition and De-elastisation on a eTeak-generated dataflow circuit: a) shows the dataflow realisation of the C1 ; [(C2 ||C3) ? C4] expression as a single cycle, b) is a decomposed implementation of (a) into two shorter cycles and c) is the de-elastisised version of a cycle from (b) where extra elastic blocks are inserted to balance the pipelines and remove the Join/Merge components. The critical path associated with each cycle is shown with red dots.

$$LCT_{new} = max\{(\alpha \cdot 2 + \beta \cdot 1), 2\} \tag{2}$$

However Decomposition does not guarantee an improvement on the critical path. In this example δ associated with the second cycle has increased to 4.6ns which forces the system to run at slower clock rate unless a multiple-clock discipline is exploited. In this way, each sub-system can run at different rate whilst exploiting channels for communication. Adopting this scheme has energy advantages and is a subject for our future work. To ensure that Decomposition yields an improved throughput it is accompanied with De-elastisation to boost the clock rate by re-timing the system. Figure 3(c) shows a re-timed version of the second cycle where elastic components are *selectively* removed by balancing the pipelines via buffer insertion. Outer Join/Fork pair is untouched as De-elastisation preserves elasticity at the boundaries. Similar to the synchronous circuits, re-timing invalidates the data-dependant property ($\alpha = \beta$) and forces every cycle in the design to have identical delays. In other words, the worst-case scenario is considered for every cycle in the design. It should be noted that De-elastisation is a selective process and calls for an appropriate decomposition at first place to avoid unnecessary buffering which may have prohibitive impact on the throughput of the elastic circuits [29]. After all, in this example D3 yields a new throughput:

$$\Theta(1.1 \cdot 3) \leq \theta_{new} \leq O(1.1 \cdot 4) \tag{3}$$

This example demonstrates how critical rate at higher level of abstraction and critical path at circuit level are connected in the dataflow context. To explore elasticity at different levels of granularity architectural information has to be taken into account. The following section explores elasticity with respect to the higher level behaviour.

D. Exploration Patterns for De-Elastisation

An abstract dataflow example comprising concurrency and data-dependent choice is illustrated in Figure 4(a). It depicts a synchronous dataflow generated using eTeak in macro-module style with activation (go) and termination (done) signals. In this model control is implicit and propagates with data. Therefore, contrary to RTL models, a separate control unit does not necessarily exist in dataflows. Initially the model is extracted from a software-like timing-free description and is fully elastic. A notion of time is introduced into the model by adopting the SELF protocol. This way data moves in separate time intervals. Without loss of generality, it is assumed that the computation blocks take one clock cycle each to produce output and the primitives are all combinational.

Fig. 4(b) shows a dataflow model where De-elastisation is applied at the level of individual stages/functions. The inner pipelines between Fork and Join are balanced (or rescheduled) by the insertion of a buffer, thus the associated elastic Fork and Join can be safely removed. Note that the input branches of Merge can have different arrival times which allows the loop to exploit a data-dependent behaviour. Removing the Merge and balancing the branches would boost the clock frequency, but would also degrade the shorter path that can deliver data one clock cycle earlier.

Fig. 4(c) shows the result of De-elastisation at the level of algorithmic loops. The shaded Join and Fork components serve as data guards and are preserved to handle inter-loop elasticity. In order to maintain architectural non-determinism, the elasticity is kept at coarse-grained level. We exploit the distributed behaviour of local control in dataflows to apply De-elastisation within macro-modules. This enables local optimisation of the modules without affecting the timing discipline of the whole circuit.

As mentioned earlier eTeak adapts SELF. This has power advantages since it offers clock gating over every register (two-phase latches) in the design (Fig. 5). In addition, cell area can benefit from this protocol compared to asynchronous

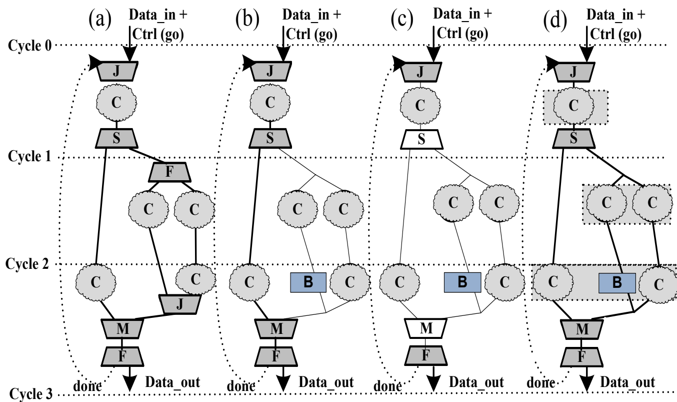

Fig. 4. Elasticity at different levels of granularity: a) fully elastic; b) stage-level De-elastisation; c) loop-level De-elastisation (preserves elasticity only at boundaries); d) step-persistent bundling of concurrent computations (steps are depicted as shaded boxes).

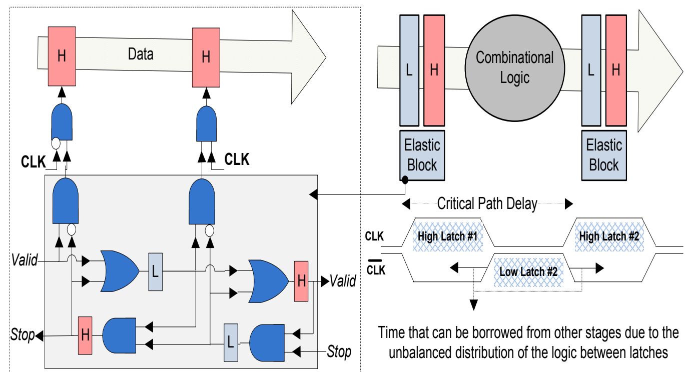

Fig. 5. Synchronous elastic controllers able to instrument a pair of latches operating in opposite clock phases and the associated timing digram demonstrating the performance advantage of these latches through time borrowing. Note that this implementation is an alternative structure to the original SELF for dataflows where data/control race is plausible to occur.

fully-interlocked protocols as it's identical to the bundled data handshake protocol [32]. However it still suffers from a 40% area overhead versus a rigid clocked synchronous circuit. Regarding performance, SELF controller's latch-based structure, offers *time (slack) borrowing* which is supported by commercial EDA flow. To take advantage of time borrowing, SELF needs to be applied at a coarse level to overcome its area overhead. This architecture independent feature of the latch-based designs has lately been leveraged by the industry to achieve a significant performance gain [33]. By applying De-elastisation the handshake components are locally removed from the design, and rigid synchronous islands are established. This makes SELF area overhead more tolerable.

To tackle the area overhead, *Step Persistence* [34] is leveraged to 'bundle' the concurrent computations, so that they can be scheduled to the same clock tick. This technique is proposed as a formalism for Petri net models of GALS systems where a reachability graph of the model is extracted and pruned to establish maximally concurrent bundles. We exploit Step Persistence in our dataflow models where data and control propagate together; thus identifying concurrent signals in both data and control paths is more straightforward. On the other hand, this idea fits very well with the De-elastisised circuits where pipelines are balanced by buffer insertion. Consequently the architectural modifications to the design contribute to an enhanced recognition of concurrent bundles and hence a reduction in area overhead. Fig. 4(d) depicts a step-persistent dataflow where the concurrent computational blocks and their corresponding elastic controllers are bundled to reduce area overhead whilst preserving their advantages regarding performance and power.

V. Experimental Results

To evaluate the proposed technique against the popular synchronous synthesis flows a good benchmark is required to which the dataflow De-elastisation and decomposition can be applied. The benchmark should realistic applications in real-life. It should include a variety of computing models, such as parallelism and concurrency, determinism and non-determinism. Finally the benchmark suite and the corresponding programs has to be obvious to display potential power and performance gains from De-elastisation.

Benchmarks. In our comparison we used a database acceleration benchmark [14], including some time-consuming operations, such as sorting, aggregation and search. Since the purpose of this work is to compare state-of-the-art HLS flows against De-elastisised dataflow transformation regarding their computation and communication models, we have selected these algorithms to role as candidates for multi-threaded fine-grained GPU-like, pipelined and producer/consumer computation models. Partitioning the designs into multiple-clock domains is out of the scope of this work and is subject for future work.

- Bitonic Sorter as Fine-grained GPU-like Computation Model: The Bitonic sorter (Figure 6) is a pipelined sorting network comprising $O(log^2 n)$ stages each with $n/2$ comparators which results in throughput (θ) bounded by:

$$\theta \leq O(w \cdot n \cdot f) \qquad (4)$$

  where, w denotes the bit-width of input elements, n is the number of elements and f is the clock frequency which equals to inverted δ. A dataflow architecture of a 16-input 64-bit bitonic sorter is implemented in Balsa and synthesised using eTeak. While this hardware-friendly sorting algorithm is capable of achieving a high throughput, its resource usage can be dominant. The Bitonic Sorter is a multi-threaded, pipelined architecture and uses push-only channels to fetch data and write back from/into the memory.

- Spatial Sorter as Pipelined Computation Model: The Spatial sorter consists of n sorting nodes each comprising a comparator, two registers and two multiplexers is shown in Figure 7. The asynchronous, synchronous elastic and De-elastisised implementations of the dataflow architecture of a 16-input 32-bit Spatial sorter is synthesised and experimented. The sorter inputs elements every clock cycle. An input element goes either through the bypass route or the comparator within each node, which takes one or two cycles, respectively. This particular example can exhibit a data-dependent performance when implemented as

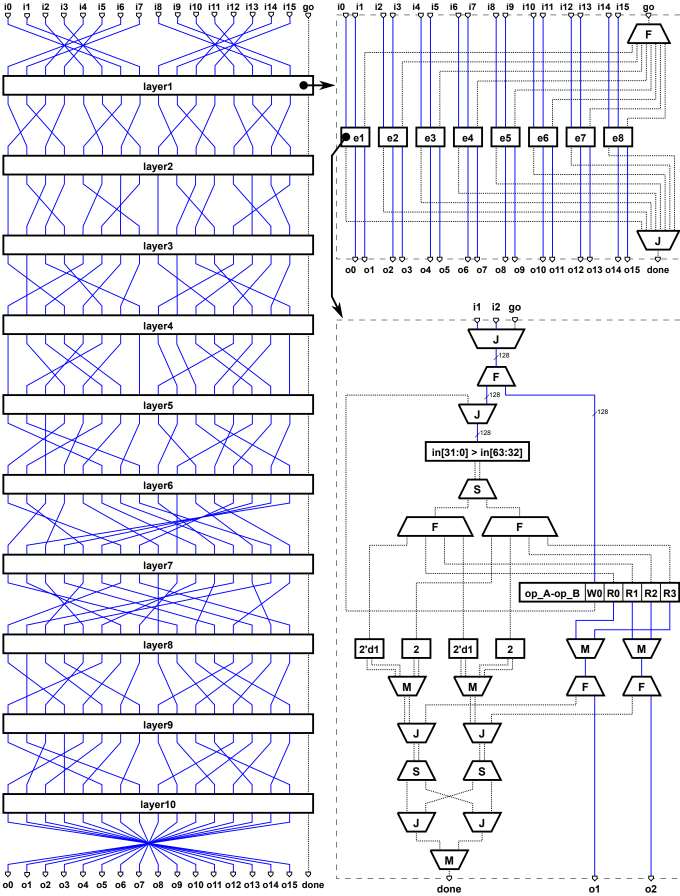

Fig. 6. Teak-generated dataflow for a 16-input Bitonic Sorter; all its 880 Joins and Forks can be removed by De-elastisation.

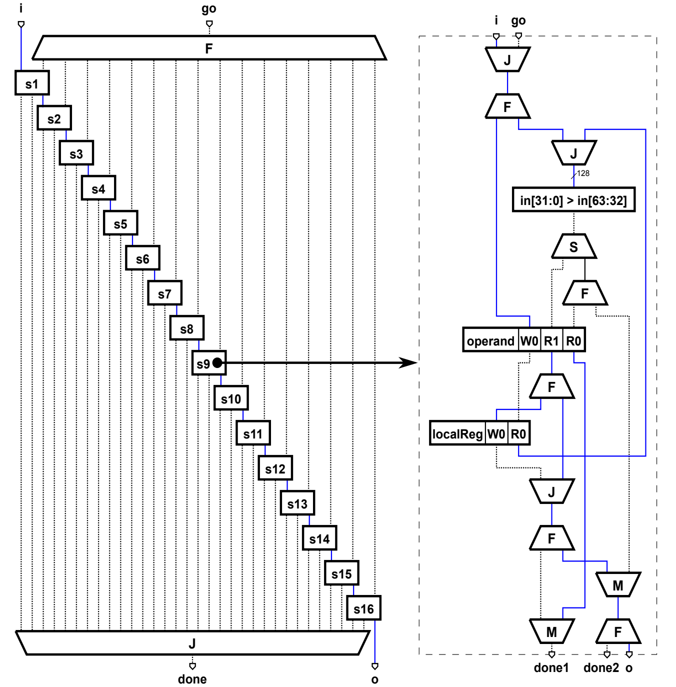

Fig. 7. Teak-generated dataflow for a 16-stage Spatial Sorter; all its 130 Joins and Forks can be removed by De-elastisation.

dataflow such that its execution time (η) is bounded by:

$$\Theta(n) \leq \eta \leq O(2 \cdot n) \tag{5}$$

To preserve this feature De-elastisation is not applied inside the nodes as it balances the routes by inserting an extra buffer in the bypass route.where n cycles as best-case (when the input stream is already sorted) and $2 \cdot n$ cycles as worst-case (when the input stream is sorted in reverse order) the synchronous elastic implementation is selected for the study. The sorting array also allows a parallel access to the sorted elements to be read in a single cycle. In contrast to Bitonic sorter, it requires $O(log^2 n)$ times less resource.

- Median Operator as Aggregation Computation Model: This algorithm abstracts the data sets with their median values and is widely used in database queries specially in finance applications. To implement this operator, a sliding window, implemented as a shift register with parallel-read enabled is connected to a 16-input Bitonic sorter which fetches the elements in every cycle and outputs the median of the 8th and 9th elements after $O(log^2 n)$ cycles with a throughput of one element per cycle.
- Hash Probe as producer/consumer Computation Model: The hash join function is one of the popular methods leveraged by database search engines to look up the entries of a hash table in a much larger repository and build a new output table. The dataflow implementation of a hash join comprises two loops, one used for probing the larger table (Fig. 8:lines 1-14) and the other one is a 16-iteration LFSR-based function in our case, to map queries from an smaller table (T2) onto the large table (hTable) (Fig. 8:lines 15-22). If the hash indexes of two keys collide, the latter key is inserted in the next consecutive empty slot of hTable. If an empty slot is found, the current query is skipped.

Experimental Setup. In this study, all the designs are using UMC 130nm technology at the typical operation conditions. To measure area *Synopsys DesignCompiler* is used along with *DesignWare* library to synthesise the synchronous functional units (operations). For sorting algorithms, the input data is a pair of 32-bit key and a 32-bit value. The median operator takes 32-bit key inputs. The hash probe uses pairs of 16-bit keys and 32-bit values, a hash table with 64K 48-bit buckets and a load factor of 0.6. The size of T2 is 400 MB and the size of hTable is 600 MB. In eTeak's case, we produce RTL code for the datapath units, such as comparators and arithmetic units, so that the generated netlist can use Design Compiler library for data manipulation units as for the other three RTL synthesis. To measure the maximum achievable frequency *Fmax*, the target clock period was turned down by 0.05ns steps until the synthesis fails to proceed. To measure performance *ModelSim SE 6.3a* from *Mentor Graphics* is used for simulations. We run our experiments on a Laptop with a 2.6 GHz Intel Core i5 processor and 8 GB of RAM.

Results. Table I summarises the results of our experiments. It presents area, performance, power and Lines of Code (LoC) obtained for each of the benchmarks which are synthesised using the high-level flows explained in this work. The hand-coded Verilog implementations are considered to represent traditional RTL flow in our evaluations. To measure performance,

```
1: procedure PROB(hashT2, result, tableAddress, tableRead)
2:     ...
3:     hashT2 −> then
4:         Stopped := 0 || Index := hashT2.key;
5:         loop while (not Stopped) then
6:             ReadTable () ;
7:             if (hashT2.key = hTableEntry.key)
8:                 then result <− #(hTableEntry.ptr) @ #(hashT2.rin) as OutRes
9:                 || Stopped := 1
10:            else IncrementIndex ()
11:            end
12:        end
13:    end
14: end procedure
15: procedure HASHFUNCTION(keyIn, Index)
16:    ...
17:    keyIn −> then
18:        xBit := (((#keyIn[15] xor #keyIn[13])
19:            xor #keyIn[12]) xor #keyIn[10]) as bit
20:        ; index <− #keyIn[14..0] @ #xBit as EntryT2
21:    end
22: end procedure
```

Fig. 8. Probe loop and the 16-iteration LFSR-based function implemented as producer/consumer model using the Balsa language

we use the throughput metric in a way that number of cycles are counted in simulation and multiplied by critical path delay which is achieved by compiling, mapping and synthesising the generated RTLs using Synopsis Design Compiler. The same approach is leveraged to synthesise eTeak netlist as its data filters/operator(O) components are behavioural Verilog modules synthesisable by commercial EDA. For the other components Teak's technology mapping service is used. For power analysis we have considered two separate scenarios to perform a fair comparison between the synchronous and elastic designs: a) when the pipeline is full which means all of the stages are kept busy every cycle and b) when a single stage is busy every cycle and the other stages are idle. Power measurements associated with these scenarios are averaged and reported in the table. It should be noted that power consumption is normalised at 500MHz to perform a fair comparison between the tools.

Discussion. According to LoC which is one of the measurable factors of designer's productivity, LegUp is ranked first (on the software extreme) whilst HDL-Verilog is last (on the hardware extreme). Between these extremes are Balsa, BSV and Chisel, in that order. Comparatively speaking, the learning curve for Balsa is quite smooth compared to BSV and Chisel.
Another measure for productivity could be the hardware details reflected to the designer. For instance, expecting the designer to define clock types and handle timing at higher abstraction level reduces the degree of productivity. Balsa exploits eTeak to virtualise timing such that clock is introduced automatically [35] whilst BSV and Chisel need the designer to define clock in the code. LegUp produces single clocked hardware, however this work could be a possible approach toward timing exploration in LegUp as well.

Regarding synthesis, eTeak inherits predictability and transparency from its syntax-directed translation scheme whilst applies a set of peephole optimisations in further levels to generate the Verilog netlist. This light-weight mechanism accomplishes almost $2\times$ faster synthesis and compilation time. This gives the tool a notion of data communication rates between the procedures, that can be used for binding high-level entities with the low-level physical characteristics automatically. Fast compilation mechanism paves the way for exploration tools like SynTunSys of IBM to take architectural refinements into account [36].

To compare the tools regardless of the input descriptions, elasticity is considered at RTL. The tools associated with each benchmark are sorted based on the level of elasticity. According to the theoretical discussions in section II on the architectural differences, the hand-coded HDL-Verilog implementations is on the inelastic extreme whilst eTeak fine-grained elastic circuits are on the elastic extreme. Next to eTeak is Bluespec with its coarse-grained elasticity: every input/output channel in the design has *enable* and *ready* signals which can be interpreted as a handshake. Thus Bluespec design entities can exploit coarse-grained elasticity which enables data-dependent computation by activating rules to fire whenever data becomes available. Chisel and LegUp do not exploit elasticity and occupy the third and fourth places, respectively.

The experiments conducted confirm that in De-elastisised dataflows time borrowing contributes to about $2\times$ speed-up in every case expect hash probe. The latter has a non-uniform logic distribution so its speed-up is limited to 11%. Time borrowing, considered by the synchronous CAD tool, demonstrates maximum efficiency when the combinatorial logic is distributed equally between the pipeline stages of a latch-based circuit. Meanwhile, bundling the elastic controllers, as explained in the previous section, reduces the associated area overhead by 12% whilst preserving the other power and performance related features.

As in the dataflow model, data and control propagate together and get latched at the same time, switching off a latch not only prevents unnecessary activity in the datapath but also disables the corresponding activities in the control path. This feature improves power efficiency in the control-dominant context (hash probe example) by 24% compared to BSV implementation and by 11% compared to HDL-Verilog implementation. Circuits obtained by LegUp run slower, but exhibit better power consumption compared to other inelastic implementations (i.e. Verilog and Chisel). The majority of power consumption is attributed to memory controllers that frequently perform memory read/write operations. It should be noted that the hash probe example exhibits non-determinism influenced by the number of cycles taken to fetch data from memories and the loading factor which indicates the application statistics in the hash table. The results show that elastic dataflow models are more efficient in terms of power and performance compared to other HLS models considered in this study for handling non-determinism.

Compared to the Teak generated dataflows [29] [37] that leverage component-level elasticity, De-elastisised dataflows achieve 15% higher performance in loop-free architectures (bitonic sorter and median operator) where De-elastisation is applied at stage-level. This improvement is even more significant ($3\times$) for the iterative architectures (spatial sorter and hash probe) where loop-level De-elastisation is considered. Note that iterative dataflows can encounter deadlock without proper buffering. In the asynchronous domain, deadlocks can be avoided by buffer insertion which may impose prohibitive area overhead. In terms of power, on average, the asynchronous dual-rail version consumes 40%

TABLE I. EVALUATION OF THE FOUR ALGORITHMS CONSIDERING DIFFERENT COMPUTATIONAL MODELS

Implementation	Total Area (um^2)	Synthesis Time(sec)	Fmax (GHz)	Power@500 MHz (mW)	Throughput (MB/sec)	LoC
bitonic Verilog	451411	525	1.00	61.3	128000	134
bitonic LegUp	257132	385	0.64	50.6	4140	101
bitonic Chisel	522082	429	0.91	65.2	116480	114
bitonic BSV	417625	497	0.91	63.4	116480	57
bitonic DeElastic	396184	231	2.30	25.7	294400	107
spatial Verilog	77311	93	0.91	21.1	3640	98
spatial LegUp	48954	73	0.83	8.3	48	28
spatial Chisel	100461	207	0.91	36.5	3640	87
spatial BSV	90345	201	0.95	28.5	3800	181
spatial DeElastic	92253	31	2.01	20.0	8040	37
median Verilog	310452	430	0.95	73.0	3800	159
median LegUp	220998	359	0.62	51.8	338	97
median Chisel	318148	269	0.91	70.1	3640	132
median BSV	433760	328	0.95	68.3	3800	70
median DeElastic	319789	173	2.03	63.8	8120	127
hashProbe Verilog	37743	130	1.18	20.6	6749.6	66
hashProbe LegUp	35733	70	0.95	4.5	194.2	50
hashProbe Chisel	40031	132	0.80	16.4	4577.6	83
hashProbe BSV	39354	145	1.25	24.3	7152.5	124
hashProbe DeElastic	37241	83	1.30	18.4	7436.4	64
Average						
LegUp	140704	223	0.76	28.8	1180.1	69
DeElastic Teak	211367	138	1.91	32.0	79499.1	84
Bluespec	245271	294	1.02	46.1	32808.1	108
Chisel	245181	264	0.99	47.5	32084.4	104
Verilog RTL	219229	295	1.01	44.0	35547.4	114

more power (at nominal voltage) compared to the synchronous counterparts due to the circuit size and switching activity of its fully-interlocked 4-phase protocol both in computation and communication.

Multiple Clocking The elastic nature is potentially convenient in that another problem with large SoCs is that synchronous clock distribution is difficult and, as different units may want to process at different rates, not always appropriate. Removing the global clock allows the units to dynamically adjust their frequency and supply voltage on demand.

To explore multiple-clocking using eTeak, De-elastisation can be exploited to transform the parts to fully synchronous netlists without relying on a third-party tool (e.g. Cadence) for RTL synthesis, which contributes to a faster compile time compared to the other synthesis schemes, whilst preserving the concurrent model of dataflow. Moreover, it enables local transformations to take place to handle the overheads imposed by communication.

Unlike eTeak, Bluespec supports the clock type at language level so the designer can specify multiple clock domains as the rule-based network is a natural fit for this purpose [38]. Although a large variety of synchronisers are supported in BSV, the issues with resolving domain crossing, adjusting clock frequencies and insertion of synchronisers are left to the designer which can influence the design efficiency and development effort.

Similar to BSV, Chisel provides the designer with multiple-clock definition at Scala level. Since this is a recent development, there is no case study taking this feature into account. However, a similar HLS in this domain, Lime [39]: a recent work by IBM, exploits an extended synchronous dataflow model for synthesis. Lime introduces 'Matcher' elements at Java level to let the designer define input/output rates for modules which reflects timing issues to the user who must have knowledge of the physical bottlenecks (critical path) in the design to make effective decisions on partitioning. This information is not visible until full synthesis takes place. The major target of this tool is to propose a heterogeneous compilation infrastructure which aims at CPU + GPU + FPGA integrity rather than applying an efficient clocking scheme to boost performance.

Multiple-clock synthesis from C can be modelled using deterministic formalisms, such as Kahn Process Networks [40], to analyse the production/consumption rates. For instance, in LegUp one can use an asynchronous/synchronous elastic bus to tolerate the data-dependent delays imposed by the FSMs.

VI. CONCLUSION AND CHALLENGES AHEAD

This work explores elasticity at different levels of granularity and abstraction using a novel synthesis framework upon which the answer for determining the degree of elasticity can be pursued. The proposed flow is compared against a set of popular synchronous HLS flows in terms of productivity, power, performance and area in the context of DBMS accelerators. Our experiments demonstrate that elastic designs enjoy short cycles; they are perfectly suitable for producer-consumer realisations and exhibit better energy utilisation, almost 30% against their synchronous counterparts, in a non-deterministic data-dependent situation, whereas the synchronous model is proper for critically timed systems where the control has to be squeezed into clock cycles. On the other hand, elasticity is expensive and may result in significant (up to 4$\times$) area overhead. Also, as feature sizes shrink, sub-threshold leakage poses low-power challenges so fine-grained elasticity may be

detrimental here, too. De-elastic design borrow rigidity from RTL to lower the overhead of elasticity, whilst inheriting the powerful features of asynchrony, such as better energy usage (almost 20%) and ability to handle system-level non-determinism with 100% performance improvement.

VII. Acknowledgment

This work is supported by EPSRC Grant "Globally Asynchronous Elastic Logic Synthesis (GAELS)" (EP/I038306/1). The authors would like to thank Alex Yakovlev, Geoffrey Ndu, John Mawer and the anonymous reviewers for their valuable remarks in refining the ideas presented in this paper.

References

[1] I. T. R. for Semiconductors (ITRS), 2009. [Online]. Available: http://www.itrs.net/links/2009ITRS/Home2009.htm

[2] P. Greenhalgh, "Big.LITTLE processing with ARM Cortex-A15 and Cortex-A7," 2011.

[3] A. Lukefahr *et al.*, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*. IEEE Computer Society, 2012, pp. 317–328.

[4] B. Keller, "Opportunities for fine-grained adaptive voltage scaling to improve system-level energy efficiency," Master's thesis, EECS Department, University of California, Berkeley, Dec 2015.

[5] Z. Zhang *et al.*, "High-level synthesis for low-power design," *IPSJ Transaction on System LSI Design Methodology*, vol. 8, pp. 12–25, 2015.

[6] K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '04. Piscataway, NJ, USA: IEEE Press, 2004, pp. 344–348.

[7] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions." High-Level Synthesis, 2008, pp. 129–146.

[8] J. Bachrach *et al.*, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[9] A. Canis *et al.*, "LegUp: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[10] J. Carmona *et al.*, "Elastic circuits," *IEEE Transaction on Computer Aided Design Integr. Circuits Systems*, vol. 28, no. 10, pp. 1437–1455, 2009.

[11] M. Jelodari Mamaghani *et al.*, "De-elastisation: From asynchronous dataflows to synchronous circuits," in *Conference Exhibition on Design, Automation and Test in Europe (DATE)*. IEEE/ACM, March 2015, pp. 273–276.

[12] J. Cortadella *et al.*, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, 2006.

[13] D. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Stanford University, 1984.

[14] A.-A. Oriol *et al.*, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *Proceedings of Field Programmable Logic and Applications (FPL)*, 2014.

[15] J. D. Kubiatowicz, "Integrated shared-memory and message-passing communication in the alewife multiprocessor," Ph.D. dissertation, Massachusetts Institute of Technology, 1997.

[16] F. Fatollahi-Fard *et al.*, "OpenSoC Fabric: On-chip network generator: Using chisel to generate a parameterizable on-chip interconnect fabric," in *Proceedings of Network on Chip Architectures*, 2014, pp. 45–50.

[17] S. Hadjis *et al.*, "Profiling-driven multi-cycling in fpga high-level synthesis," in *Proceedings of Design, Automation & Test in Europe (DATE)*. EDA Consortium, 2015, pp. 31–36.

[18] L. P. Carloni *et al.*, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[19] S. Taylor *et al.*, "Asynchronous data-driven circuit synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 7, pp. 1093–1106, July 2010.

[20] D. A. Edwards *et al.*, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, 2002.

[21] S. Nielsen *et al.*, "A behavioral synthesis frontend to the haste/tide design flow," in *Proceedings of Asynchronous Circuits and Systems (ASYNC)*, 2009.

[22] A. J. Martin *et al.*, "CAST: Caltech asynchronous synthesis tools," in *Asynchronous Circuit Design Working Group Workshop, Turku, Finland*, 2004.

[23] A. Bardsley *et al.*, "Teak: A token-flow implementation for the balsa language," in *Proceedings of Application of Concurrency to System Design (ACSD)*, 2009, pp. 23–31.

[24] A. Peeters *et al.*, "Click Elements: An implementation style for data-driven compilation," in *Proceedings of IEEE Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2010.

[25] M. Stucki *et al.*, "Logical design of macromodules," in *Proceedings of the Joint Computer Conference*. ACM, 1967.

[26] J. Cortadella *et al.*, *Logic Synthesis for Asynchronous Controllers and Interfaces*. Springer, 2002.

[27] J. Cortadella *et al.*, "SELF: Specification and design of synchronous elastic circuits," in *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 2006.

[28] M. Jelodari Mamaghani *et al.*, "eTeak: A data-driven synchronous elastic synthesiser," in *Proceedings of 13th International Conference on Application of Concurrency to System Design (ACSD), PhD Forum*. IEEE, 2013, pp. 134–137.

[29] M. Jelodari Mamaghani *et al.*, "Optimised synthesis of asynchronous elastic dataflows by leveraging clocked EDA," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, Aug 2014, pp. 607–614.

[30] R. Manohar *et al.*, "Slack elasticity in concurrent computing," in *Proceedings of Int. Conf. Mathematics of Program Construction*. Springer-Verlag, 1998, pp. 272–285.

[31] P. A. Beerel *et al.*, *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010.

[32] J. Sparsø *et al.*, *Principles of asynchronous circuit design: a systems perspective*. Springer-Netherlands, 2001.

[33] M. Fojtik *et al.*, "Bubble razor: Eliminating timing margins in an arm cortex-m3 processor in 45 nm cmos using architecturally independent error detection and correction," *Solid-State Circuits, IEEE Journal of*, vol. 48, no. 1, pp. 66–81, 2013.

[34] J. Fernandes *et al.*, "Persistent and nonviolent steps and the design of GALS systems," in *Fundam. Inform.*, vol. 137, no. 1, 2015, pp. 143–170.

[35] M. J. Mamaghani *et al.*, "Automatic clock: A promising approach toward GALSification," in *Proceedings of 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC) - Industrial Track*, 2016.

[36] M. Ziegler *et al.*, "A synthesis-parameter tuning system for autonomous design-space exploration," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2016.

[37] L. T. Duarte, "Performance-oriented Syntax-directed Synthesis Of Asynchronous Circuits," Ph.D. dissertation, University of Manchester, 2010.

[38] M. Vijayaraghavan *et al.*, "Bounded dataflow networks and latency-insensitive circuits," in *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE, 2009.

[39] J. Auerbach *et al.*, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 89–108, 2010.

[40] S. Suhaib *et al.*, "Dataflow architectures for GALS," *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 1, pp. 33–50, 2008.