Elastic CGRAs

Yuanjie Huang[¶§]
huangyuanjie@ict.ac.cn

Paolo Ienne[†]
paolo.ienne@epfl.ch

Olivier Temam[‡]
olivier.temam@inria.fr

Yunji Chen[¶]
cyj@ict.ac.cn

Chengyong Wu[¶]
cwu@ict.ac.cn

[¶]CARCH, ICT, CAS, China [†]EPFL, Switzerland [‡]INRIA, Saclay, France [§]GUCAS, China

ABSTRACT

Vital technology trends such as voltage scaling and homogeneous multicore scaling have reached their limits and architects turn to alternate computing paradigms, such as heterogeneous and domain-specialized solutions. *Coarse-Grain Reconfigurable Arrays (CGRAs)* promise the performance of massively spatial computing while offering interesting trade-offs of flexibility versus energy efficiency. Yet, configuring and scheduling execution for CGRAs generally runs into the classic difficulties that have hampered *Very-Long Instruction Word (VLIW)* architectures: efficient schedules are difficult to generate, especially for applications with complex control flow and data structures, and they are inherently static—thus, inadapted to variable-latency components (such as the read ports of caches). Over the years, VLIWs have been relegated to important but specific application domains where such issues are more under the control of the designers; similarly, statically-scheduled CGRAs may prove inadequate for future general-purpose computing systems. In this paper, we introduce *Elastic CGRAs*, the *superscalar processors* of computing fabrics: no complex schedule needs to be computed at configuration time, and the operations execute dynamically in the CGRA when data are ready, thus exploiting the data parallelism that an application offers. We designed, down to a manufacturable layout, a simple CGRA where we demonstrated and optimized our elastic control circuitry. We also built a complete compilation toolchain that transforms arbitrary C code in a configuration for the array. The area overhead (26%), critical path overhead (8%) and energy overhead (53%) of Elastic CGRAs over non-elastic CGRAs are significantly lower than the overhead of superscalar processors over VLIWs, while providing the same benefits. At such moderate costs, elasticity may prove to be one of the key enablers to make the adoption of CGRAs widespread.

Categories and Subject Descriptors

C.1.3 [**Computer Systems Organization**]: Processor Architectures — Other Architecture Styles; B.6.3 [**Logic Design**]: Design Aids — Automatic synthesis

General Terms

Design, Performance

Keywords

CPGA, elastic circuit, dataflow, reconfigurable computing

1. INTRODUCTION

Coarse-Grained Reconfigurable Arrays (CGRAs) are an appealing paradigm to implement accelerators in these days and times when both voltage scaling and homogeneous multicore scaling approach their ultimate limits [30, 16]: On one hand, as it is the case for *Field Programmable Gate Arrays (FPGAs)*, CGRAs are good candidates to create massively spatial application-specific accelerators. On the other hand, and in contrast to FPGAs, they are word-oriented and hence naturally more efficient in area, timing, and energy when used for complete applications originally designed for pure software implementation. Yet, despite such appeal and despite more than a decade of focused research, they have not yet made it into widespread commercial use.

CGRAs aggressively exploit spatial parallelism, much as *Very Long Instruction Word (VLIW)* processors [18] exploit *Instruction Level Parallelism (ILP)*, with CGRAs being vastly more complex, and thus potentially powerful, than VLIWs due to the affordable number of operators, routing, and configuration control [36]. Yet, one of the biggest impediments to the diffusion of VLIWs has always been the compiler technology required to efficiently schedule programs with complex control and memory behavior. Thus, superscalar processors, which also aim at leveraging ILP, albeit using a purely automated hardware approach (on-the-fly resolution of register dependences, memory dependences and extraction of ILP), have become the de facto standard for general-purpose computing (epitomized by Intel x86 architecture), and they are now making in-roads in energy-conscious embedded systems as well, e.g., ARM Cortex A9 [2], Intel Atom Z2460 [19]. The difficulty of compiling efficient VLIW code has ultimately relegated VLIWs to some important but niche applications such as digital signal-processing – far away from general-purpose computing.

Existing CGRA designs, such as ADRES [26], MATRIX [27], Tartan [28], MorphoSys [34], or HSRA [40], similarly rely on complex compiler technology to schedule operations. In fact, considerable research efforts have already gone in adapting for CGRAs the *classic* VLIW compilation techniques [31, 41]. To improve the chances of CGRAs to become a viable architecture for reconfigurable accelerators, we introduce an *Elastic CGRA*, where scheduling occurs entirely in hardware and control signals follow computation in a dataflow manner, adapting to the actual execution of the application: Our scheme requires practically no compiler scheduling effort, much like the hardware issuing of operations in superscalar processors removes the burden of VLIW code scheduling. And, while superscalar processors come with a significant hardware overhead which has incited decades of research on VLIW processors, we show that the hardware overhead of Elastic CGRAs, with respect to a statically scheduled CGRA, is comparatively low, even though they bring the same benefits as superscalar processors.

As the name suggests, our idea is based on *elastic circuits* [11], which are data paths where the exact scheduling of operations is not fixed in advance but is dynamically determined at runtime by the availability of operands, much like in asynchronous circuits. Yet, despite a strong conceptual similarity with asynchronous circuits, elastic circuits are perfectly synchronous circuits which can be designed with any modern EDA flow. Starting from a *Control and Data Flow Graph (CDFG)*, we show that it is relatively straightforward to generate two strongly interconnected circuits, one processing the data and the other controlling the flow of operation. These two circuits seamlessly map onto the units of an Elastic CGRA. As a result, creating a configuration for an Elastic CGRA requires less assumptions on the exact device implementation, such as the latency of the individual components or the unit topology. This property also provides some form of "binary compatibility" of the code mapping onto the CGRA across different Elastic CGRA microarchitectures, much like superscalar processors achieve compatibility across successive generations of microarchitectures whereas VLIWs do not.

In this paper, we design a tile of a possible Elastic CGRAs, which we synthesize and place and route with standard cells in a common 65nm technology. In order to demonstrate the programmability of Elastic CGRAs, we implement a GCC-based tool chain which performs a simple conversion of programs into elastic circuits. We retarget VPR [6], typically used for placement and routing on FPGAs, to place and route the combined data and control networks onto an Elastic CGRA. We automatically generate the RTL description of the elastic circuits of five benchmarks, which are then mapped and executed on the Elastic CGRA. We also implement a more classic statically-scheduled version of our CGRA to assess the cost of "elasticity" in terms of area, critical path, and energy consumption. We conclude that the area overhead is 26%, the critical path delay overhead is 8% and the energy overhead is 53%. Overall, the energy and area overhead of Elastic CGRAs over non-elastic CGRAs is significantly lower than the overhead of superscalar processors over VLIWs, while providing the same dynamic schedule and portability benefits.

In Section 2, we present the method for converting a program into elastic circuits. In Section 3, we explain how to generalize the elastic circuit concepts into an Elastic CGRA. In Section 4, we present the different elements of the tool chain. We present the experimental methodology in Section 5, the performance evaluations in Section 6, and the related work in Section 7.

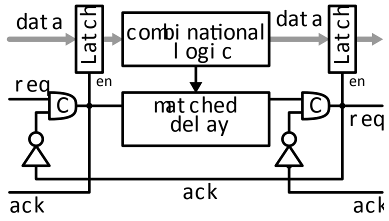


Figure 1: *Asynchronous handshaking control circuit.*

2. FROM SOURCE CODE TO ELASTIC CONTROL CIRCUITS

We first present the concept of elastic control, then we describe the conversion process from source code to elastic circuits (for both ASICs and Elastic CGRA), we outline the interface of such circuits with the memory hierarchy, we explain why such circuits are deadlock-free, and we conclude with a code conversion example.

2.1 Elastic Control

The principle of elastic control is to replace statically scheduled control with dynamic dataflow-like control in a circuit. The benefit of dataflow-like control is the avoidance of a statically precomputed schedule: data propagates down the circuit with the respective control token. As a result, complex scheduling cases occurring when the latency of certain operators varies—e.g., complex operators or memory accesses, or where multiple operators must all complete before moving to the next operation—are seamlessly handled with token propagation, as we will later see. This notion of dataflow-like control is not new but, until recently, it was mostly accomplished using asynchronous circuits or through ad-hoc solutions. Budiu et al. [7] had already noted the great potential of dataflow-like asynchronous control for easily converting programs into circuits. However, there is no mature industrial tool chain for designing asynchronous circuits yet, even if promising tool chains are emerging [14]. The key contribution of elastic control [11] is to propose dataflow asynchronous-like control in perfectly synchronous circuits, at a very low energy and area cost. We first briefly recap a few notions about asynchronous control and then we present the main principles of elastic control.

Asynchronous control. In an asynchronous pipeline, the control parts of two consecutive pipeline stages are connected through an asynchronous circuit, see Figure 1. The main component of this circuit is the Muller C gate [29] (see gate C in Figure 1) which implements a behavior similar to an SR latch. In an asynchronous circuit controlling a pipeline, the output of that gate is 1 only if the previous pipeline stage is valid (i.e., it has a data to pass) and the next pipeline stage is not stalled (i.e., it can accept the data). This output signal controls the pipeline latch between two pipeline stages in the data path of the circuit (see Figure 1).

The circuit in Figure 1 has the typical issue of needing a delay closely matched to that of the functional unit—something that is unfeasible and simply requires taking large margins.

```
int i,t0;
int x1=0;
for (i=0;i<N;i++)
{
  t0=x[i];
  y[i]=t0*C0+x1*C1;
  x1=t0;
}
```

Figure 8: *FIR code.*

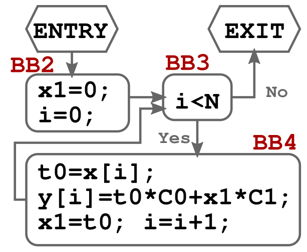


Figure 9: *Control flow graph of FIR.*

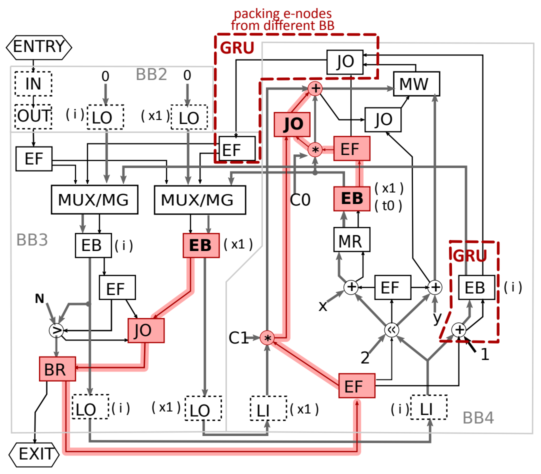


Figure 10: *FIR circuit. LI and LO stand for live-in and live-out respectively, and are only wire annotations. MR and MW are memory read and write ports.*

Elastic control. Elastic control aims at emulating a similar behavior, with similar signals handshaking, except that it is based on clocked circuits, and thus it is fully compatible with existing EDA toolchains. Cortadella et al. [11] proposed the SELF protocol and a structure named Elastic Buffer to connect pipeline stages. In our design, Elastic Buffers are modified to use flip-flops for implementation convenience, see Figure 2. Elastic control circuits are always tied to a data path component, forming a *data-control pair*. For instance, the Elastic Buffer in the control path (bottom) is always tied to a pipeline register in the data path (top). This circuit uses the same *valid* and *stall* signals as the corresponding asynchronous circuit. The completion detection signal is replaced by a standard stall signal at a clock-cycle resolution. A data path implemented using elastic control circuits, including loops, has been proved to be deadlock and race free with proper initialization [22].

2.2 Conversion Process

Three elastic operators have been previously introduced [20, 11] for implementing elastic control circuits: *Elastic Buffer* (EB), *Eager Fork* (EF), which corresponds to the control token forking to two or more targets, and the reciprocal circuit *Join* (JO), which merges multiple control paths into one. In order to implement program control flow statements such as `if` or `switch`, we also need *Branch* (BR) and the reciprocal *Merge* (MG) [35] elastic control circuits, where Branch corresponds to an operator with multiple potential control flow paths, and only a single token is issued into one of the paths.

In the remainder of this section, we describe these operators in detail, and we illustrate the conversion process with the *Finite Impulse Response* (FIR) filter code shown in Figure 8. The first step consists in converting the program into a control flow graph of its basic blocks, as shown in Figure 9.

2.2.1 *Across Basic Blocks*

By definition, a basic block is a control flow graph node with a single *control flow* entry point and a single *control flow* exit point. Note that “single control flow entry point” or “single control flow exit point” should not be confused with “single control flow predecessor” or “single control flow successor”: a basic block can have multiple predecessors all entering the basic block at the same program statement—i.e., entry point—and similarly, a single exit point can potentially fork out to multiple successors. Consider, for instance, BB3 in the control flow graph of Figure 9: It has a single entry point (statement `i < NP`), but two predecessors, BB2 and BB4. Also, BB3 has a single exit point, but it can branch out to two successors: BB4 and the exit of the function.

Merge. If a basic block has several predecessors, the control circuit of the corresponding entry point is an elastic *Merge* circuit, shown in Figure 6. For instance, the control flow token can reach BB3 from two possible predecessors, BB2 and BB4: the entry basic block, and the body of the loop (see Figure 9). However, at any given time, only one of the two successors can have the control flow token.

Branch. Similarly, if a basic block has several successors, the control circuit of the corresponding exit point is an elastic Branch circuit, shown in Figure 5. Across basic blocks, the control flow token necessarily flows out to only one of several successors. For instance, one can notice that the exit point of BB2 in Figure 9 can branch out to two successors: either BB4 or exit.

2.2.2 *Within Basic Blocks*

In this section, we explain how the elastic operators Eager Fork and Join, previously introduced by Cortadella et al. [11], can be used for basic block conversion. The control flow subgraph corresponding to a basic block is always a directed acyclic graph of the basic block statements, by definition of a basic block.

Variables. The first step consists in converting scalar accesses. There are three types of scalar accesses within a basic block: live-ins, live-outs, and temporary values. Both live-ins and live-outs must exist beyond the basic block data flow graph, and thus, they are implemented using Elastic Buffers. Temporary values require no physical storage. For instance, variables `x1` and `i` in Figure 9 correspond to both live-ins and live-outs in BB3 and BB4, and, thus, they are each implemented with an Elastic Buffer, as shown in Figure 10; `t0` corresponds to a basic block temporary value and has no corresponding register in Figure 10.

Eager Fork. When the same value flows from one statement to two or more statements in parallel, the control flow token splits into multiple tokens which, similarly to the corresponding data, flow in parallel. This can only happen *within* a basic block but never across basic blocks, since basic blocks are executed sequentially. This control flow token multiplication is implemented with the Eager Fork elastic

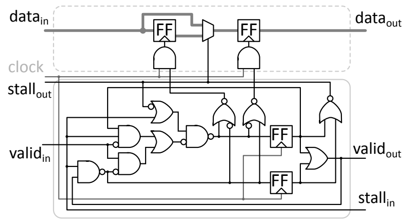


Figure 2: *Elastic Buffer (EB).*

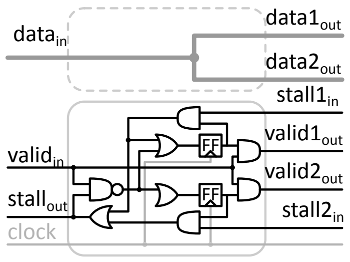


Figure 3: *Eager Fork (EF).*

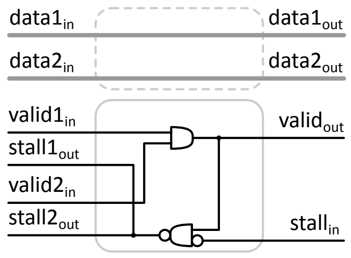


Figure 4: *Join (JO).*

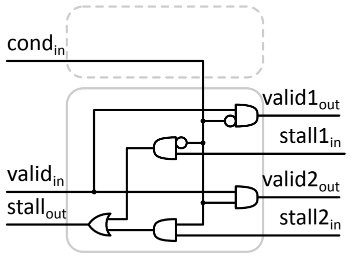


Figure 5: *Branch (BR).*

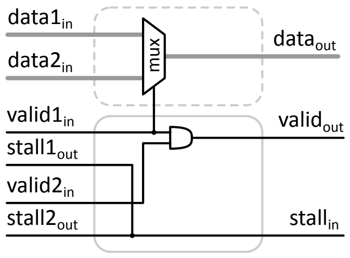


Figure 6: *Merge (MG).*

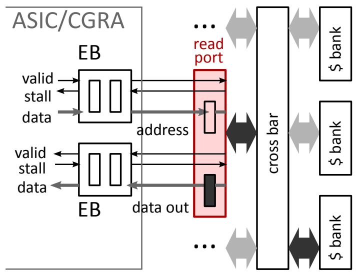


Figure 7: *Virtual memory port.*

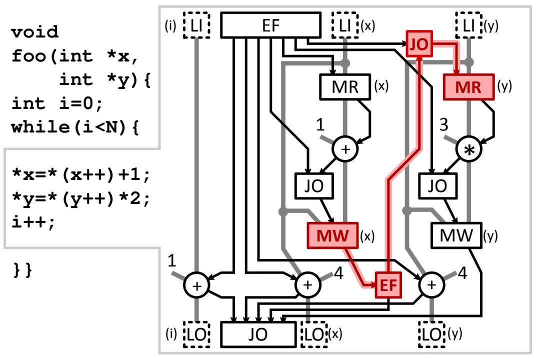
control circuit (see Figure 3). This fork is called "eager" because the control flow token can be sent to each branch as soon as the branch can accept it. Consider for instance the entry point of the body of the innermost loop in Figure 8: the statements `x1*C1`, `i<<2`, `i++` can all be executed in parallel and, as a result, the entry point is implemented with an Eager Fork, instead of an Elastic Buffer, as one can see with the EF at the bottom of BB4 in Figure 10.

Join. When two or more parallel statements are followed by a sequential flow of execution, or reach the end of the basic block, the multiple control tokens join back to become a single token as soon as they are all available. Again, this can only occur within a basic block. This is implemented with an elastic Join, see Figure 4. Consider for instance the JO at the top of BB4 in Figure 10.

Dependent statements. Conversely, dependent statements are implemented with serially connected elastic control circuits. Consider for instance the data flow dependence due to the + operation between `t0*C0` and `x1*C1`: these two statements are respectively implemented with control handshake from an EB through an operator then connected to an elastic Join—i.e., the two control tokens join back to become a single token, as shown by the shaded boxes connected by highlighted lines in Figure 10.

2.3 Memory Interface

In order to seamlessly integrate memory accesses within the translation process—and thus easily convert load and store instructions, we introduce the notion of *elastic virtual memory ports*. Each read/write memory access is replaced with a read/write elastic virtual memory port. This memory port supports the same control interface as an Elastic Buffer—i.e., the same two-signal handshake channel. The write port has two data flow inputs (`data` and `address`), and the read port has one data flow input (`address`) and one output (`data`), as shown in Figure 7. For instance, in a read port, the outgoing *valid* signal becomes true when the data has been fetched from memory and is ready. The

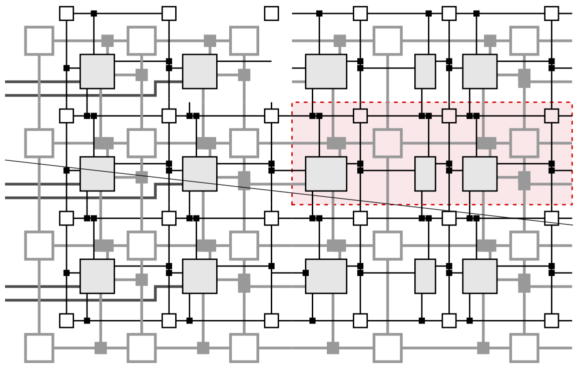


```
void
foo(int *x,
    int *y){
int i=0;
while(i<N){

*x=*(x++)+1;
*y=*(y++)*2;
i++;

}}
```

Figure 11: *Memory dependences: the four different accesses to x and y are implemented using 2 read ports (MR) and 2 write ports (MW); the highlighted elastic circuit corresponds to the dependence between the x write and the y read.*

outgoing *stall* signal becomes false when the read port can accept a new memory request—i.e., a new address.

Memory Dependences. Distinct references are mapped to different memory ports, so two ports could potentially access the same address. The memory dependence is implemented by creating a lockstep between the corresponding elastic virtual memory ports. Consider the example of Figure 11. Read and write references to pointer `x` and `y` are mapped to four memory ports, two for read and two for write. Now, for instance, due to pointer aliasing, the `y` read can depend upon the `x` write. So we create a lockstep between the two ports using the highlighted JO and EF: one `x` write must occur before the next `y` read takes place. In superscalar processors, such dependences are detected by the *Load/Store Queue* (LSQ). Although nothing would prevent us from implementing an LSQ too, for the sake of simplicity and without loss of generality, we imple-



signal can be combinatorially connected to the output stall signal through the JO elements at the entrance of a unit, and consequently the inward and outward routing networks for these two control signals are merged into one long timing path. Though it is possible to split the critical path with an extra EB, it is costly solution with the current CGRA structure because one full unit must be used.

In order to overcome that timing limitation, we developed an optimized version where the input valid signal and the output stall signal are separated by registers; this is accomplished by moving the EB to the entrance of the unit; as a result, the EB must be duplicated for each handshake pair. The resulting unit organization is shown in Figure 14. Its critical path is illustrated in Figure 15. Since an ALU unit has only two data inputs, we eliminated data registers in all but two input EBs in order to save area. Another specialized EB with a 1-bit data register is placed before the BR in order to split the long path that goes through the ALU, BR and JOIN and ends up at the input EB.

This optimization trades a small increase of area and energy for a significantly higher clock rate, as later shown in Section 6.

3.3 Grid

The Elastic CGRA grid is shown in Figure 12. The high-level structure of the grid is strongly inspired from that of FPGAs. The routing network is composed of switches (or switch boxes, also called *S-boxes*) connecting horizontal and vertical lines, and the units themselves are connected to the network through connect boxes (*C-boxes*) shown as solid black and grey squares. Our Elastic CGRA routing network has some differences from an FPGA routing network, though: Firstly, it is composed of two networks, not one—the elastic control network shown in thin black lines (S_c stands for control switch) and the data network shown in thick grey lines (S_d is a data switch). Moreover, the data network consists of 32-bit channel bundles, whereas FPGAs usually have 1-bit channels; 32-bit data channel bundles share configuration bits. Finally, the valid and stall signals of elastic control share a single 1-bit channel. All channels are unidirectional as in modern FPGAs [23]. For the switches, we have used the popular pattern proposed by Wilton [43]. Based on the benchmarks later presented in Section 5, we empirically found (after exploration with VPR) that six (32-bit) data channels and sixteen (1-bit) control channels are sufficient. Note that, because VPR (used for placement and routing) imposes that all units in a column to be identical, we arrange the three blocks of the unit (ALU, elastic control, and MUX) in a row, see Figure 12.

The elastic control and data configuration is implemented using several multiplexers shown in Figure 12. Note that the configuration of JO and EF requires to indicate how many tokens they should respectively wait upon or issue. No such configuration is needed for MGs and BRs.

4. TOOL CHAIN

The overall structure of the tool chain is described in Figure 16. The tool chain first generates a circuit intermediate representation from the source code, then converts it into an elastic circuit. This elastic circuit can either be mapped into Verilog and used to create an ASIC, or mapped onto a CGRA and then placed and routed. We briefly describe the different tool chain components below.

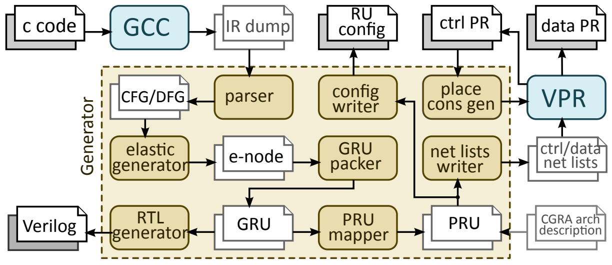


Figure 16: *Work flow and structure of the tool chain.*

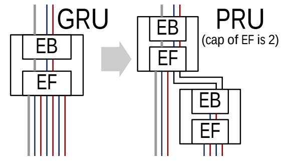


Figure 17: *GRU mapping to PRU.*

The front-end of the infrastructure is the GCC compiler (version 4.6.1). In order to generate our intermediate representation, we piggyback on the GCC 32-bit MIPS back-end. We generate the intermediate representation just before the hardware register allocation pass, because a CGRA can potentially implement a far greater number of registers than a core, and the code should not be hampered by the side-effects of a limited number of registers (especially by spill code).

4.1 Elastic Circuit

The elastic circuit generator is implemented outside GCC, communicating through the generated intermediate representation instead of being plugged into GCC, only to limit the overhead of keeping pace with the rapid evolution of GCC, and still to benefit from the most recent high-level optimizations.

The elastic generator builds the data flow graph of each basic block, and the control flow graph between basic blocks. Then it generates the elastic components using the elastic operators (stored as templates) described in Section 2. Several of these templates have a configurable number of inputs or outputs (Join, Eager Fork, Merge, and Branch). The output is a directed graph of *e-nodes*, where one e-node corresponds to an elastic operator. For instance, in Figure 10, each solid box or circle stands for an e-node corresponding to a physical elastic operator, while dotted boxes are virtual e-nodes which keep track of various information. For example, the two e-nodes labeled `LO` in BB2 indicate that the initial value of `i` and `x0` are set in BB2 and that they are live-outs. Thin black edges are control handshakes (arrow showing the valid signal direction), while bold gray edges are 32-bit data channels.

4.2 CGRA Generator

The CGRA generator is significantly more complex. It first maps every elastic node onto a *Generic Reconfigurable Unit* (GRU). A GRU is a virtual unit, i.e., an ALU, an Elastic Buffer and the set of elastic control primitives. However, at that stage, the Merge, Join, and Eager Fork primitives are assumed to have an infinite number of inputs and outputs respectively. Moreover, the number of GRU units is

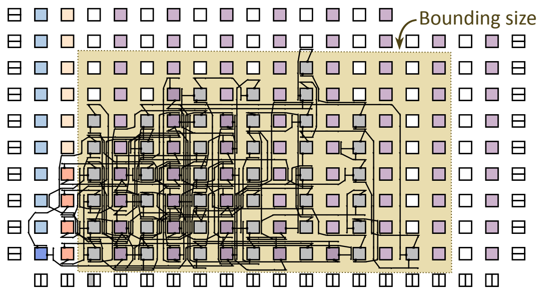


Figure 18: *VPR routing result of Susan_s' data nets.*

also considered infinite. The GRU abstract unit representation allows to perform several optimizations independently of the number of units and the capacity of each unit. For instance, an e-node storing a constant does not consume one unit because it is stored in the constant register contained in each unit, see Section 3.1. A GRU is obtained from one or several e-nodes. For instance, if an EB stores the result of an adder, both are packed into the same GRU. Or, if an EF follows a JO, both are also packed into a single GRU. These two examples are marked in Figure 10 with dashed boxes. Finally, GRUs are transformed into *Physical Reconfigurable Units* (PRU) which correspond to the target CGRA unit architecture. For instance, if one GRU contains an Eager Fork with more outputs than a PRU, it is split accordingly and mapped onto several PRUs, see Figure 17. A netlist of the PRUs is then generated for the placement and routing phases.

4.3 Place and Route

We use VPR [6] for the placement and routing phases, VPR being the most popular open-source tool for that task. Normally, VPR is designed to place LUTs and route 1-bit channels. Replacing LUTs with PRUs is trivially done. On the other hand, we have two networks to route: the 32-bit data network, and the 1-bit elastic control network (which contains the valid/stall signals). There is no notion of dual networks in VPR and we therefore perform a 2-phase routing: First, we let VPR place the PRUs and route the 1-bit control channels. After that phase, we obtain a placement of all PRUs, because all PRUs must have control connections. Then, we impose this placement and we perform a second routing for data channels, presenting them as 1-bit channels to VPR. In Figure 18, we show the placement of PRUs for *susan_s* and the routing of its data network.

5. METHODOLOGY

5.1 Synthesis

CGRA tiles are synthesized with Synopsys Design Compiler Ultra 2009 onto a TSMC 65nm library, and layout was done using Synopsys IC Compiler 2009. We employed 9 metal layers, with 1 to 7 for routing, and 8 and 9 dedicated to power distribution. The energy measurements are obtained using Synopsys PrimeTime PX 2009with post-layout activities captured by by Synopsys VCS 2010. All data presented in Section 6 are collected in the Normal Case COMmercial (NCCOM) condition.

Benchmark	Domain	Description	Hot function
blowfish_e (Cbench)	security	Symmetric block cipher with a variable length key.	BF_encrypt
fir (UTDSP)	signal processing	256-tap Finite Impulse Response filter.	fir
fft (EEMBC)	telecom	Fast-Fourier-Transformation.	fxpfft
latnrm (UTDSP)	signal processing	32nd-order Normalized Lattice filter.	latnrm
susan_s (Cbench)	auto-motive	An image recognition algorithm, it can smooth the image.	susan_smoothing

Table 1: *Benchmarks description.*

5.2 Non-Elastic CGRA

To quantitatively assess the overhead of elastic control, we derived a non-elastic CGRA from the elastic one by substituting elastic control with a distributed statically-scheduled controller as in B. Mei et. al's paper[26]. The elastic control of each unit, shown in Figure 14, is replaced with a distributed local storage containing the contexts of each unit, where one context corresponds to the configuration bits of the ALU, input MUXes and output register write control. In addition, the unit contains the simple context address generation logic allowing to iterate through, or jump to, contexts. As each non-elastic CGRA unit corresponds to an ALU-unit or a MUX-unit in elastic CGRA, we extract contexts for them from the activity of corresponding units in the simulation of the same benchmark on elastic CGRA. Note that, unlike elastic controlled CGRAs, such statically scheduled control can only be used to map codes with regular control flow, e.g., loops, so we focus the comparison on the main (most time-consuming) loop of each benchmark The data routing network remains unchanged.

5.3 Benchmarks

We use five benchmarks from the embedded benchmark suites UTDSP [33], MiBench [17] and EEMBC [15] to evaluate our proposal. The five benchmarks are chosen from three different domains: automotive, security, and telecom. For each of these benchmarks, we use gprof to identify the hot function, and convert this function into a circuit.

We simulated these benchmarks using the data sets provided in their benchmark suite. Data sets of *susan_s* were slightly trimmed to achieve acceptable post-layout simulation time. The memory subsystem is emulated in the test bench using a memory array preloaded with a memory image, obtained after running each benchmark on a MIPS CPU. The memory subsystem is assumed to be perfect (no cache misses) and all requests return in one cycle because detailed memory behavior is out of scope of this study.

6. PERFORMANCE EVALUATION

In this section, we compare the area, critical path delay and energy of elastic vsňon-elastic CGRAs. The non-elastic CGRA is the *baseline*, the naive elastic control of Section 3.1 is called *naive*, and the optimized elastic control of Section 3.2 is called *optimized*.

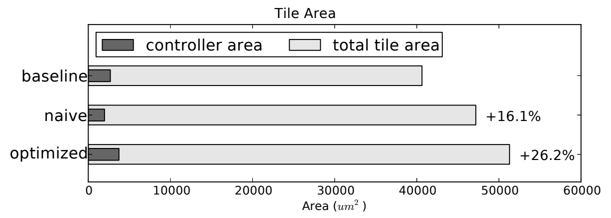


Figure 19: *Tile area of non-elastic , naive elastic and optimized elastic implementations.*

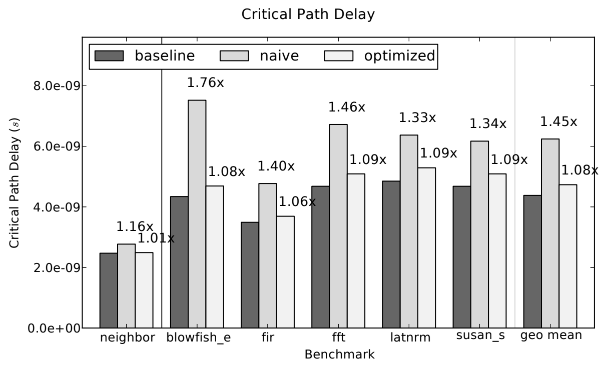


Figure 20: *Critical Path Delay of non-elastic CGRA, naive and optimized elastic CGRAs.*

6.1 Area

In Figure 19, we compare the area of the non-elastic CGRA (baseline), the optimized tile version of Section 3.2, and the naive tile version of Section 3.1. The naive version has an area overhead of 16.1% over the baseline, and most of this overhead is due to the 1-bit elastic control routing network. The optimized version has an overhead of 26.2% due to the addition of registers, see Section 3.2. In all cases (non-elastic or elastic), the controller logic itself accounts for a small fraction of the overall tile area.

6.2 Critical Path Delay

We report the critical path delay in Figure 20. On average, the naive implementation is 1.45× slower than the baseline, while the optimized one is only 1.08× slower. For neighbor-to-neighbor communications, the optimized elastic unit can achieve the same clock rate as the non-elastic unit, as shown in bar *neighbor*. The control handshake wires increases the total wire density which, in turn, degrades the wire quality and results in slightly longer wires, hence the slight slowdown of the optimized elastic CGRA.

6.3 Energy

The energy for executing each benchmark on each CGRA is shown in Figure 21, with the number on the top showing the energy ratio with respect to the non-elastic CGRA. On average, the naive and optimized elastic CGRAs respectively require 43.2% and 53.8% more energy than the non-elastic CGRA. We also provide a breakdown into leakage and dynamic energy. Leakage is higher for elastic CGRAs due to the higher area, and dynamic energy is higher due to the elastic control routing network; the dynamic energy of the optimized version is even higher due to the higher register switching activity. We further highlight the fraction of dy-

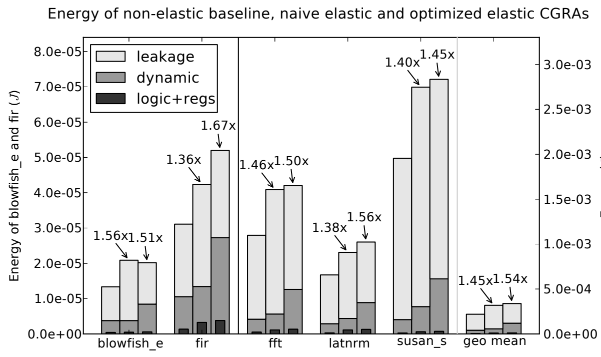


Figure 21: *Total energy and breakdown.*

namic energy due to the logic and registers (the rest being the clock tree and routing network), and we show that the dynamic energy due to the clock tree and routing network dominates.

7. RELATED WORK

Schedules generated for CGRA have been universally static, as discussed by De Sutter et al. in their recent survey [36]. This naturally limits the chances to convert efficiently arbitrary programs and adds constraints to the source code: ADRES [26], like others, assumes no control flow in loop bodies to generate VLIW-like static schedules. There exists a considerable body of work on adapting classic VLIW compilation techniques to the mapping and scheduling problems of CGRA (for instance, Park et al. developed variations of modulo scheduling targeted to reconfigurable accelerators [31]). Elastic CGRA is quite different because its elastic nature considerably simplifies program conversion with essentially no constraints. There is a large body of works on compiling programs onto dataflow architectures, e.g. Arvind et .al's TDDA [3], or more recently, Wavescalar [37] and Ambric [39]. While our design follows similar dataflow principles, our focus is on the detailed circuit implementation of such dataflow constructions based on elastic circuits, it relies on a more simple FPGA-like static routing network, and an almost direct translation from C code. Recently, Budiu et. al. [7] have proposed to implement dataflow circuits using asynchronous elements, but such approaches are limited by some classic issues of asynchronous designs; our Elastic CGRA benefits from being fully synchronous. We are also significantly different from Achronix's pipoPIPE [1], which is an asynchronous platform which promises to improve over current FPGAs but remains a fine-grain bit-level device, at a completely different granularity. Park et al. [32], on the other hand, have used data tokens in a CGRA, but their control is only partially distributed: their CGRA is still driven by instructions, stored in a central structure, the goal of the tokens is limited to driving the fetching of instructions into the functional units.

Program translation into arbitrary circuits (both targeting ASICs and FPGAs) is the focus of *High-Level Synthesis (HLS)*, widely researched since the nineties. One of the classic phases of HLS is scheduling, and some ideas in this area are similar to what is or could be done in CGRAs. Academic tools, such as ROCC [42], and commercial tools, such as CatapultC [25], are emerging and can convert programs, with certain limitations, into circuits, mostly target-

ing FPGAs. To our knowledge, most or all of these tools generate architectures limited to static schedules and cannot accommodate efficiently variable latency components, such as the ports of typical general-purpose memory hierarchies. In fact, from early notable examples of translation of high-level programming constructs into circuits [8] until more recent experiences [24, 38], generated controllers have almost universally taken the form of horizontally microcoded sequencers, single mo(n)-1.1327(o)-0.920322l3483(t)-0.491474(e)0.2927617(h)-1.1327(i)-0.0653c FS17(h)-1.1Ms,or networks of smaller FS17(h)-407(M)-0.517341(s)-374.91([)-0.0646676(1)-0.920322(2)-0.919641(])-0.0646676(.)-560.142(H)-0.879479(e)0.294067(r)-0.505088(e)-37
trollers for such variable-latency units has been revisited in

[17] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: a free, commercially representative embedded benchmark suite. In *2001 IEEE International Workshop on Workload Characterization*, pages 3–14. IEEE, 2001.

[18] J. L. Hennessy and D. A. Patterson. *Computer*