Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables

Cheng Tan, Manupa Karunaratne, Tulika Mitra and Li-Shiuan Peh
School of Computing, National University of Singapore
Email: {tancheng,manupa,tulika,peh}@comp.nus.edu.sg

***Abstract*—Wearable devices are now leveraging multi-core processors to cater to the increasing computational demands of the applications via multi-threading. However, the power, performance constraints of many wearable applications can only be satisfied when the thread-level parallelism is coupled with hardware acceleration of common computational kernels. The ASIC accelerators with high performance/watt suffer from high non-recurring engineering costs. Configurable accelerators that can be reused across applications present a promising alternative. Autonomous configurable accelerators loosely-coupled to the processor occupy additional silicon area for local data and control and incur data communication overhead. In contrast, configurable instruction set extension (ISE) accelerators tightly integrated into the processor pipeline eliminate such overheads by sharing the existing processor resources. Yet, naively adding full-blown ISE accelerators to each core in a many-core architecture will lead to huge area and power overheads, which is clearly infeasible in resource-constrained wearables. In this paper, we propose *Stitch*, a many-core architecture where tiny, heterogeneous, configurable and fusible ISE accelerators, called *polymorphic patches* are effectively enmeshed with the cores. The novelty of our architecture lies in the ability to stitch together multiple *polymorphic patches*, where each can handle very simple ISEs, across the chip to create large, virtual accelerators that can execute complex ISEs. The virtual connections are realized efficiently with a very lightweight compiler-scheduled network-on-chip (NoC) with no buffers or control logic. Our evaluations across representative wearable applications show an average 2.3X improvement in runtime for *Stitch* compared to a baseline many-core processor without ISEs, at a modest area and power overhead.**

***Keywords*-Manycore architectures; accelerators; low-power; customization; network-on-chip; wearables**

I. Introduction

We are witnessing the transition from single-core to multi-core architectures [1, 2, 3] in the wearable domain to support the increasing computational requirements of the applications. Many-core architectures enable improved power-performance by exploiting the available thread-level parallelism in many wearable applications. Still they often need to be supplemented with application-specific hardware accelerators, with prohibitively high non-recurring engineering (NRE) cost, to reach the target power-performance objectives. In order to effectively amortize the NRE cost through re-use, configurable accelerators can be incorporated with general purpose CPUs. However, loosely-coupled accelerators with rich computing resources, local storage and local control consume large on-chip area that is precious for wearables. On the other hand, tightly-coupled accelerators eliminate these overheads by closely integrating into the processor pipeline and leveraging the processor resources. Instruction Set Extensions (ISE) is one form of tightly-coupled accelerator, which is normally implemented in customized circuits [4, 5, 6, 7, 8] for efficient execution of application-specific custom instructions; but suffers from limited flexibility. Fortunately, ISE accelerators can be implemented with a configurable fabric and coupled with a general-purpose processor core [9, 10, 11, 12] to reconcile the conflicting demands of performance, area, and flexibility. A custom instruction encapsulates an application-specific computational pattern and an application is typically accelerated with a few custom instructions implemented on the configurable fabric.

In this paper, we revisit the ISE support in the context of many-core architectures for wearables. We observe that wearable applications offer substantial potential for acceleration with application-specific custom instructions. The most straightforward approach to take advantage of both thread-level parallelism and custom instructions is to add an ISE accelerator fabric per core. The common computational patterns within a thread mapped to a particular core can then be accelerated by the configurable fabric associated with that core. Unfortunately, the overhead to add a complete, complex accelerator per core is too expensive both from power and area perspectives. In particular, as we are focusing on resource-constrained wearable devices, it is simply infeasible to add a per-core accelerator to a many-core architecture.

Instead, we leverage the many-core nature of the architecture itself to introduce application-specific custom instructions efficiently, bootstrapping ISEs with the thread-level parallelism of multi-cores. Instead of a huge complex ISE accelerator at each core, we propose the insertion of tiny, heterogeneous, fusible and configurable accelerators, called *polymorphic patches*, one per core. Each *patch* handles very simple custom instructions. We introduce heterogeneous *patches* where each *patch* supports a unique set of computational patterns. The choice of these *patches* is obtained through comprehensive profiling of the representative wearable applications. Also the proportion of each *patch* type in the architecture, i.e., number of cores with a particular *patch* type and their placements, are carefully orchestrated to cover a wide range of potential computational patterns.

The novelty of our proposed *Stitch* architecture is that the *patches* can be stitched (fused) together to form virtual, complex, configurable accelerator fabric through software

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00054

directives. We exploit the observation that different threads in wearable applications have imbalance in acceleration needs; so the idle patches associated with threads/cores that do not need acceleration can be utilized by other threads to create large, virtual accelerators for the critical threads. The virtual connections among the *patches* are realized efficiently with a compiler-scheduled, bufferless network-on-chip (NoC) with zero control logic. The combination of patches form unique complex accelerators that can execute a set of more complex computational patterns compared to the original *patches*. Combination of different *patch* types supports a variety of complex computational patterns.

Our evaluations across representative wearable applications show that the *Stitch* architecture obtains an average of 2.3X higher performance compared to a baseline many-core processor without application-specific custom instructions, at a modest area (only $0.17mm^2$, 0.5% of the entire chip) and power overhead (23%). The corresponding performance/watt and area/watt improvements are 1.77X and 2.28X, respectively. Comparing with state-of-the-art wearable smartwatches, *Stitch* achieves on average 1.65X throughput improvement (6.04X boost in performance/watt) across representative wearable applications with only 140mW power consumption.

The remainder of this paper is organized as follows. Section II presents the background. Section III describes the *Stitch* architecture in detail and the compiler support is described in Section IV. Section V presents a case study of a gesture recognition wearable application on the *Stitch* architecture. Section VI presents and discusses the architectural evaluations, and RTL timing, area and power results. Related works are discussed in Section VII and Section VIII concludes the paper.

II. Background

Conventional wearable devices leverage ultra-low power single-core SoCs (system-on-chip) that only support limited functionality, such as data sensing, data preprocessing, and result display. Most of the computation is offloaded to the high-performance paired smart phones, gateways or cloud servers. However, emerging wearable applications require real-time response with high sensing fidelity. So the computation has to be performed on the wearable device to avert the wireless communication delay. For example, real-time image recognition and augmented reality are supported by smart glasses. Most smartwatches support real-time gesture recognition, offline navigation, and transportation applications. In addition, many software development kits allow the programmers to create their own applications in wearables [13, 14, 15, 16]. As a result, more powerful processors are being leveraged inside wearable devices to provide real-time in-situ computation capability across diverse application scenarios.

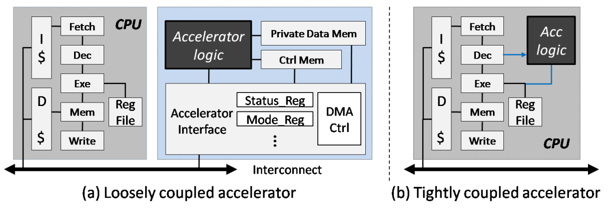


Figure 1. Accelerators with different couplings.

A. Advent of multicores and manycores in wearables

A transition from ultra-low power single-core to more powerful multi-core SoCs for wearable devices has been witnessed as compute performance requirements grow in emerging wearable applications. Before August 2013, smartwatches typically utilized the single-core ARM processors (such as the ARM Cortext-M3, M4 in Sony Smartwatch) with low power consumption. Samsung released the first standalone smartwatch utilizing dual-core ARM cortex-A7 processors with the tag-line 'leave your phone at home' in November, 2014 [17]. Nowadays, the state-of-the-art smartwatches such as Samsung Gear S3, Motorola Moto 360 2nd edition leverage quad-core ARM cortex-A7 processors (Qualcomm Snapdragon 2100 [18]) to enable real-time in-situ computation. Even with the improvement in compute performance, wearable devices are still subjected to very stringent power budget (hundreds of milliwatts) due to limited battery capacity. Hence, performance/watt is not scaling up with the increase in core count and performance.

A similar trend has also been witnessed in sensor nodes and application-specific wearables. High performance architectures for low-power sensor nodes for health-monitoring applications have been proposed [3]. [19] proposes a heterogeneous many-core architecture for object recognition. [20] proposes an ultra-low-energy convolution engine to accelerate convolutional neural networks in many-core wearables. [21] efficiently leverages many-core accelerators for faster image processing in wearables. A 64-core platform is proposed in [22] for biomedical signal processing. In contrast, our *Stitch* architecture is not application-specific and can be configured in software to work across a diverse set of wearable applications.

B. Accelerators in wearables

Application-specific ASIC accelerators are attractive from the performance/watt point-of-view; but including them sacrifices the flexibility of the architecture in handling diverse wearable applications. Due to the prohibitively high non-recurring engineering (NRE) cost and exacting time-to-market constraints, it is not practical or feasible to design an accelerator for each wearable application starting from scratch.

In order to effectively amortize the NRE cost through re-use, configurable fabrics are utilized and incorporated with general purpose CPU. The configurable accelerators can be broadly classified into two classes: loosely-coupled accelerators and tightly-coupled accelerators. As shown in

Figure 1, the loosely coupled accelerators (e.g., [23, 24, 25, 26, 27, 28, 29, 30, 31, 32]) typically require local register files, control and data memories, and high data transfer bandwidth, which significantly increases the design complexity and area overhead. On the other hand, the tightly coupled accelerators (e.g., [9, 33, 34, 35, 36, 37]) conserves precious on-chip area by closely integrating into the processor pipeline and sharing the processor resources (e.g., instruction fetch, decode, register file, and even on-chip memory). Tightly-coupled accelerators are more suitable for wearables to achieve high performance/watt with the consideration of stringent area and power budget. Instruction Set Extensions (ISE) is one form of tightly-coupled accelerator, where a configurable accelerator fabric is integrated within the general-purpose processor pipeline for efficient execution of application-specific custom instructions. A custom instruction encapsulates an application-specific computational pattern and an application is typically accelerated with just a few custom instructions. *Stitch* uses ISE accelerators for efficient execution of wearable applications.

III. Stitch System Architecture

The current prototype of *Stitch* architecture consists of 16 tiles connected through a mesh network-on-chip as shown in Figure 2. *Stitch* adopts the message-passing programming paradigm as opposed to shared memory to avoid cache coherence overhead. Each tile contains a simple in-order CPU integrated with a *polymorphic patch* (for acceleration) through a crossbar controller, separate instruction and data caches, scratchpad memory (SPM) for the accelerator, NIC (network interface controller) plus a NoC router.

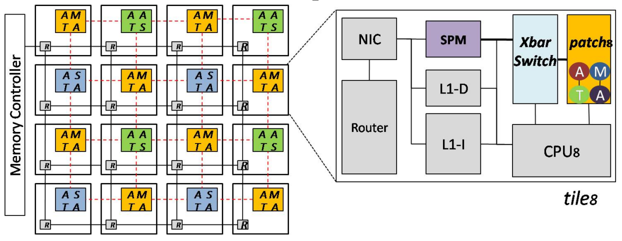


Figure 2. High-Level view of the *Stitch* architecture.

There are three different types (color coded in the figure) of *polymorphic patches* in the *Stitch* architecture. These *patches* are connected by a reconfigurable compiler-scheduled NoC, separate from the conventional NoC between the cores. During execution, the *patches* can be stitched together into larger, virtual *fused patches* to enable execution of complex custom instructions. The fusion is achieved by configuring the compiler-scheduled NoC appropriately so that the operands are delivered across multiple hops between the stitched *patches*, executing the custom instruction within a single-cycle.

A. Polymorphic Patch Design

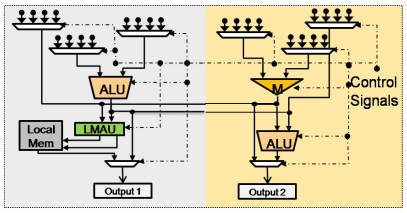
The *polymorphic patches* are configurable accelerators used to improve the execution time of commonly occurring computational patterns encapsulated as custom instructions. Figure 3 shows the design of the different *polymorphic patches*. Each *patch* contains an ALU (denoted by A) followed by a local memory access (denoted by T) unit (LMAU). The LMAU is physically a 2x1 multiplexer that directly connects to the local memory (scratchpad memory) port. The $\{AT\text{-}MA\}$, $\{AT\text{-}AS\}$ and $\{AT\text{-}SA\}$ *patches*, in addition, incorporate a multiplier (denoted by M) followed by an ALU (A), ALU (A) followed by a shifter (denoted by S) and a shifter (S) followed by an ALU (A), respectively. Each *patch* supports four input operands and two output operands. The output can be written to the register file of the corresponding core or can be transmitted to the next *patch* with which it is fused for the custom instruction execution.

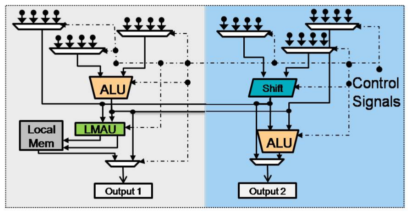
A *patch* can accelerate a number of different but simple computational patterns. The exact choice of different operations as well as the connection from the input operands to the operators is determined by the MUX settings within the *patch*. The MUX settings are configured through the control bits from the custom instruction currently executing on the *patch*. This configurability of the *patches* enables execution of different custom instructions (for example $+$ followed by $<<$ or $-$ followed by $>>$) on the same *patch* and hence the name *polymorphic patch*. Each *patch* requires 19-bits for control signals, which is carried by a two-word size custom instruction. A pair of homogeneous or heterogeneous *patches* can be fused together to create a larger, virtual *fused patch*.

Given a computational pattern, different *patches* would enable different degrees of acceleration. Figure 4 demonstrates the execution of the computational pattern (represented as a dataflow graph (DFG)) accelerated by the different *patches*. The *patch* $\{AT\text{-}MA\}$ requires 4 cycles with 4 instructions (2 custom instructions and 2 shift instructions as shown in Figure 4(b)). In contrast, it only needs 2 cycles with 2 custom instructions to execute the given computational patten by mapping it onto the *patch* $\{AT\text{-}AS\}$ (Figure 4(c)).

The main feature of *Stitch* is that it allows any *patch* to fuse with another far-away *patch*, forming a *fused patch* to execute complex custom instructions. Figure 4(e) shows that the given computational pattern can be executed in a single-cycle by the fused *patch* $\{AT\text{-}AS, AT\text{-}AS\}$. Figure 4(d) shows that the given DFG cannot be accelerated further with the fused *patch* $\{AT\text{-}MA, AT\text{-}AS\}$ when compared to the single *patch* $\{AT\text{-}AS\}$. The fused *patch* across multiple tiles finishes the execution within a single cycle strictly following the original instruction pipeline (single-cycle execution) without additional synchronization. Section III-B will detail how the *patches* are stitched.

The *polymorphic patch* design is motivated by the characteristics of 'hot' computational patterns across a set of representative wearable kernels [38]. First, we identify a pattern as 'hot' if it occurs above an occurrence rate threshold of 5%. The operations inside the 'hot' computational patterns


(a) *patch {AT-MA}*


(b) *patch {AT-SA}*

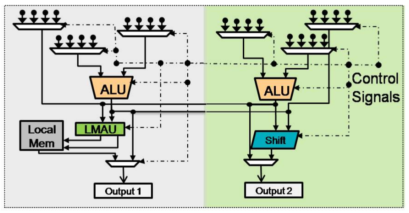

(c) *patch {AT-AS}*

Figure 3. Design of different *patches*

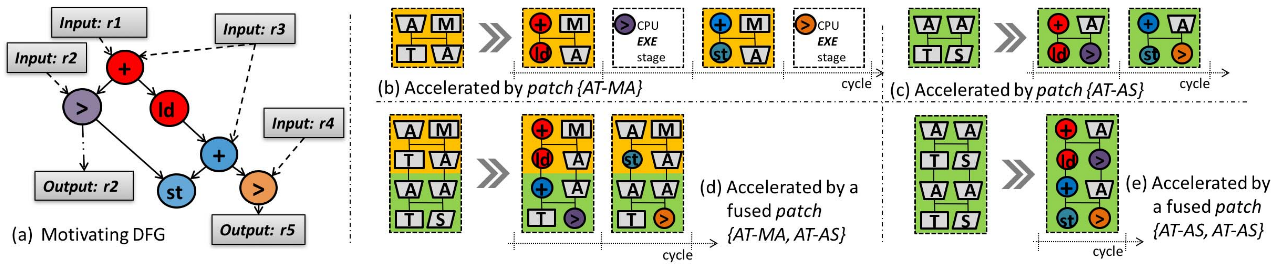


Figure 4. Computational pattern accelerated by different *patches*

are classified into four groups: arithmetic and logical operations (type A), shift (type S), multiplication operations (type M), and load/store (local scratchpad memory access) operations (type T). The move instructions are not classified into any group because they can be converted into wiring when synthesized.

Then, we design the *patches* based on the most common operation-chains representing the critical paths of the 'hot' computational patterns through multiple-rounds of Longest Common Substring (LCS) identification. The input of the LCS in round *n* is the output for round *(n-1)* excluding the most common substring generated from the LCS in round *(n-1)*. As a result, different common operation-chains are generated with the occurrence rate in each round. For example, {*AT*}: 95.7%, {*MA*}: 47.8%, {*AA*}: 34.8%, {*AS*}: 21.7%, {*SA*}: 21.7%. {*AT*} appears 95.7% across all the selected kernels. Therefore, {*AT*} should be supported by all the cores to enable the acceleration of various kernels running on different cores. {*MA*} needs to be supported by half of the cores while {*AS*} and {*SA*} should be supported by a quarter of the cores. Finally, 8 {*AT-MA*}, 4 {*AT-AS*}, and 4 {*AT-SA*} *patches* are distributed to the 16 cores in *Stitch* (Figure 2). Note that the operation-chain {*AA*} is supported by enabling the intermediate connection between {*AT*} and {*MA*} inside the *patch* {*AT-MA*}, which provides more parallelism and flexibility.

B. Single-cycle Reconfigurable Compiler-Scheduled NoC

Fused patches are enabled by the mesh NoC that comprises of only wires: interconnects and crossbar switches, but no buffers or logic. The wires are embedded with clockless repeaters, to enable single-cycle multi-hop traversal like that in SMART NoCs [39], without the complex flow control/routing logic and buffering. Instead, reconfiguration is carried out by the compiler, and the network is configured before each application initiates execution.

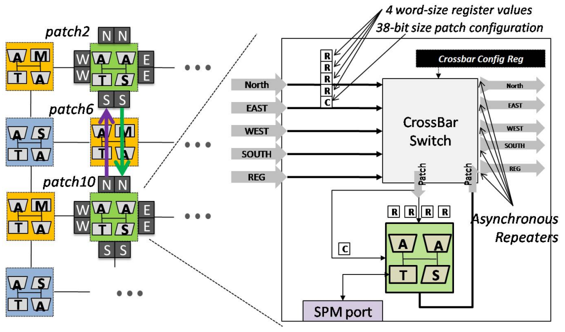


Figure 5. Reconfigurable compiler-scheduled inter-*patch* NoC.

Network. The detailed microarchitecture of the *Stitch* NoC switch can be seen in Figure 5. Each output of the crossbar switch is driven by clockless repeaters [39] that can be configured to either let the signals bypass asynchronously to the next hop (N, E, W, S), or to stop and receive the incoming data in the patch. This enables data to be sent across multiple tiles without latching. The switch has six input and six output ports. Each input/output link is 166 bits wide, carrying four 32-bit words for data delivery (corresponding to maximum four input operands) and 38 bits control signals (to support stitched *patches* where each of them requires 19-bit control). The four data values can either come from the local register file (*REG*) or another *patch* through one of the four direction inputs. The *patch* is configured according to the control signals carried by each custom instruction.

The NoC switch in each tile is configured before application execution by setting the memory mapped *crossbar configuration register* through a memory store operation. Figure 5 shows two *patches* (*patch*$_2$ and *patch*$_{10}$) are stitched together to form a larger *patch* {*AT-AS, AT-AS*} (Figure 4(e)). *patch*$_6$ is configured to provide bypass paths from *patch*$_2$ to *patch*$_{10}$ and vice versa. Figure 2 shows the placement of the patches where *patch*$_i$ belongs to tile$_i$ and the tiles are numbered starting with 1 from the top-left corner. *patch*$_2$ processes the data taken from tile$_2$ register file and sends the result via its output port to the input port of *patch*$_{10}$ (green line), which processes the data. Finally, the post-processed data is sent to *patch*$_2$ and the related registers are updated (indicated by the purple line). The entire computation plus data transfer completes within one cycle as verified by our RTL synthesis results.

C. SPM Assistance

It is well known that the inclusion of load/store operations during the ISE generation leads to significant performance boost [40, 41, 42]. *Stitch* achieves this by locating the variables involved in the custom instructions in a specific address space belonging to the local memory of the *patch*. We implement the local memory as a scratchpad memory (SPM) integrated with the CPU and accessible by both the CPU and the *patches* (Figure 2). With an in-order, single-lane pipeline, only the CPU functional unit or the patch is active at any point in time avoiding memory inconsistency. The SPM is an extension of the main memory in terms of the address space and the SPM data can never be cached. The sequencer inside the CPU identifies if the address of a memory access request belongs to the SPM address space or not. The *patches* can also access the SPM via the LMAU as shown in Figure 3. The address spaces for SPMs on different patches are disjoint from each other. Each core is restricted to access only its own SPM. The data allocated onto different SPMs belongs to different variable regions (e.g., the arrays of RealBitData[] and ImagBitData[] in FFT kernel can be mapped onto two different SPMs).

Many applications access small portions of the memory address space multiple times in a frequently executed part of the code [41]. Mapping these data to the SPM address space and keeping them always on-chip make it possible to include the load/store operations inside the custom instructions. The address mapping can be done by the compiler given the target variables [42, 43]. We analyze a set of wearable kernels [38] and it turns out that 4KB SPM space is sufficient for all the kernels (from 256 bytes in *AES* kernel to 4096 bytes in *Histogram* kernel), which leads to a 4KB SPM integrated to each core in *Stitch*. Introducing the SPM in our architecture implies reduction in the cache size per core, which may incur higher cache miss rate. However, the variables mapped into the SPM address space are usually the 'hot' ones intensively accessed during the entire execution, which in turn compensates for the cache miss overhead. Our evaluation shows only 1.5% performance degradation on average when replacing the 4KB Data Cache with a 4KB SPM. Note that *Stitch* is based on the message-passing architecture instead of conventional shared memory, which eliminates the coherence overhead both in multiple private caches and multiple local SPMs.

As the load and store operations are part of the stitched larger single-cycle custom instruction, the addresses and data will not arrive synchronized to the global clock. Thus, to enable the SPM to be accessed asynchronously, i.e., as and when the address and data arrives, we added logic to generate a valid signal that does not violate SRAM's internal timing requirements while being delayed appropriately, similar to [44].

IV. Compiler Support

Stitch provides compiler support consisting of automated ISE identification process [8], a graph-based mapper [5, 11] to synthesize ISEs onto *patches*, an ISE selector, a backend to generate the final executable binary with the corresponding control signals, and a generator for the appropriate stitching of *patches* for kernels to maximize the overall throughput of the application.

We implement an automated compiler tool chain integrated with a modified GNU Assembler to enable the compiler support of *Stitch* as shown in Figure 6. Given a multi-kernel application, it is first compiled to assembly code by the GNU gcc front-end. Meanwhile, the bottlenecked kernels and 'hot' basic blocks are detected through profiling each kernel by our tool chain. Then, the tool chain identifies the 'hot' basic blocks. Afterwards, the 'hot' basic blocks are represented as dataflow graphs (DFGs). ISE identifier then generates the custom instruction candidates from the DFGs under the 4-input/2-output (i.e., 4 input ports and 2 output ports to the register file) constraint. Given the structure of different *patches* in *Stitch* architecture, a greedy heuristic mapping algorithm [11, 45] maps each ISE candidate onto every *patch*. In this way, we can get the speedup of each kernel using each *patch* and combination of any two different *patches*. The tool chain then replaces the computational patterns with the corresponding custom instructions selected by the ISE selector in the assembly code, and generates multiple executable versions of the original kernel with the patch control signals through a modified GNU Assembler. At last, a stitching algorithm determines the appropriate kernel mapping, version selection, *patch* stitching, and inter-*patch* NoC configuration aiming for the maximal overall throughput.

Mapping a multi-kernel application onto the 16-core *stitch* architecture and stitching appropriate *patches* together for each kernel is an NP-hard problem with huge design space. Therefore, we design a simple greedy approach (Algorithm 1) to quickly generate quality, feasible solutions indicating the kernel mapping and the stitching of the *patches*. Algorithm 1 arbitrates the access to *patches* among cores at compiler time. The *patch* or stitched *patches* require preset configuration of the inter-patch NoC before an application is launched and does not change during execution. The input to the algorithm is the *Stitch* architecture and a multi-kernel application $\mathscr{A}$ with different speedup for each *kernel* $\in \mathscr{A}$ accelerated by specific *patch* or fused *patch*. The problem

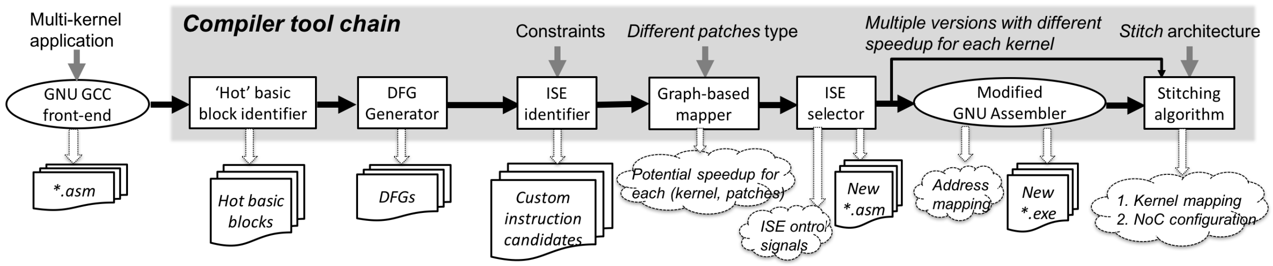


Figure 6. Compiler tool chain for the *Stitch* architecture.

is to accelerate the kernels using appropriate *patches* and construct the valid configuration for the compiler-scheduled NoC to stitch the *patches* for maximal overall throughput. Our proposed algorithm improves the application throughput by allocating the *patches* to accelerate the bottleneck kernels iteratively until all the *patches* are used up (lines 3-16) or the bottleneck kernel cannot be sped up any more (lines 6-7). The function **Bottleneck($\mathscr{A}$)** gets the current bottleneck kernel of the application $\mathscr{A}$ according to their execution time. **BestPatches(*kernel*, *kernel*.*checkedPatches*)** returns the *patch* or a pair of *patches* that can achieve the best speedup for *kernel*. Note that the returned *patches* should not appear in *kernel*.*checkedPatches* that contains all the checked *patches* for *kernel*. When the *patches* are available for a bottleneck *kernel*, **FindPath(*patches*, *stitching*)** attempts to find a valid *path* to stitch the *patches* using Dijkstra's shortest path algorithm ($\mathscr{O}(N^2)$) and finally add it into the collection of the valid paths *stitching* (lines 12). If no available path is found for the selected *patches*, the other *patches* will be considered as new candidates. After a valid *path* is generated for *kernel*, we map the *kernel* onto the tile that contains one of the *patches* (line 15).

Algorithm 1: Stitching Algorithm.

```
1  Input: $\mathscr{A}$, Stitch architecture
2  Result: stitching that connects appropriate patches for all kernels in $\mathscr{A}$
3  while there is patch available do
4      kernel = Bottleneck($\mathscr{A}$);
5      patches = BestPatches(kernel, kernel.usedPatches);
6      if patches = $\emptyset$ then
7          return stitching;
8      path = FindPath(patches, stitching);
9      if path = $\emptyset$ then
10         kernel.checkedPatches.add(patches);
11         Continue;
12     stitching.add(path);
13     foreach $k \in \mathscr{A}$ do
14         k.checkedPatches.add(patches);
15     LocateKernel(kernel);
16     UpdateExecutionTime(kernel);
```

V. Application Case Study

We present a case study to illustrate the features of the *Stitch* architecture and the impact of *Stitch* on the real-time response of an overall system.

We choose **finger gesture recognition** as our driving application, which captures accelerometer and gyroscope signals as inputs and performs *FFT* (Fast Fourier Transform), *Update feature*, *Filter*, and *IFFT* (Inverse FFT) followed by a classification to identify the finger gesture [46]. The application is implemented as multiple kernels connected in a pipelined manner (Figure 7) for maximal throughput. The processing of *FFT, IFFT* is realized as multiple kernels (threads) executing in parallel to handle the signals from two different sensors along three dimensions. The *IFFT* kernels incorporate additional processing, such as another *Update feature* processing, resulting in longer execution time compared to the *FFT* kernels. In [46], which was prototyped on the Shimmer wearable device [47], with compute offloaded to a paired smartphone through bluetooth, a sampling frequency of accelerometer and gyrometer signals of 128Hz was needed to ensure 98% gesture recognition accuracy. This translates to a 7.81ms deadline constraint for real-time response.

We implemented the finger gesture recognition application of [46] on a TI SensorTag [48] that is worn on a user's wrist (see Figure 8). When the user draws alphabets in the air, the application recognizes the gesture as representing each alphabet letter. The SensorTag has ARM Cortex-M3 as the processor, like most ultra-low-power wearables in the market today, that delivers insufficient performance for our gesture application – the SensorTag takes 577ms to detect and recognize a gesture, which is insufficient accuracy and fidelity, far from the 7.81ms real-time target deadline for this application. We thus also profile this application on an Odroid XU3 development platform, which is similar to the state-of-the-art processors (Snapdragon Wear 2100) embedded in today's high-end smartwatches, such as Huawei Watch2 and Asus Zenwatch 3, and our proposed *Stitch* architecture, to explore their impact on the application performance.

We observe that the real-time response requirement is not attainable with the quad-core (ARM Cortex-A7) processor in the Odroid XU3 development platform, but *Stitch* is able to meet it at 7.62 ms < 7.81 ms (Table I). Note that the cycle count of *Stitch* is obtained using modified gem5 (detailed in Section VI-C) simulator (that is validated against RTL with simple programs) and combined with clock frequency and power numbers generated from RTL synthesis (detailed in Section VI-D) for average power consumption estimation.

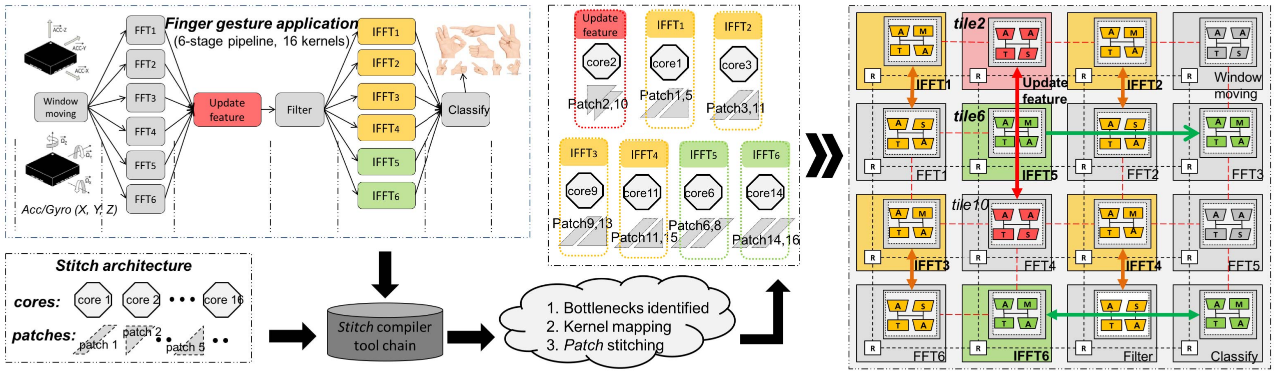


Figure 7. Finger gesture recognition applications running on *Stitch* with appropriate *patches* fused together.

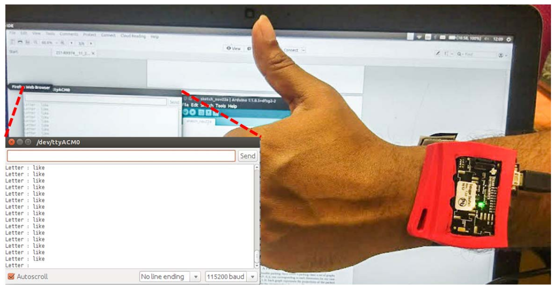


Figure 8. Finger gesture recognation application running on TI SensorTag.

	SensorTag	4-core ARM Cortex-A7	*Stitch w/o* fusion	*Stitch*
Meeting Throughput	✗	✗	✗	✓
Time per gesture (ms)	577	13	11.49	7.62
Average power consumption (mW)	8.78	469	108	139.5
Processor Freq. (MHz)	48	1200	200	200
Process technology	-	28nm	40nm	40nm

Table I
POWER-PERFORMANCE BEHAVIOR OF FINGER GESTURE RECOGNATION APPLICATION ON DIFFERENT ARCHITECTURES

Figure 7 shows why the application benefits from the *Stitch* architecture. The *IFFT* and *Update feature* kernels are identified as the bottleneck kernels and are accelerated using different pairs of the heterogeneous *patches*. Specifically, the *Update feature* kernel stitches $patch_2$ and $patch_{10}$ together while *Stitch*'s NoC is configured to bypass $patch_6$ to enable fast data transfer between them. Note that the core in $tile_{10}$ still runs an *FFT* kernel but without acceleration by $patch_{10}$. As a result, the throughput improves by 1.71X with respect to the quad-core A7 and the real-time response requirement is satisfied. The corresponding performance/watt goes up to 6.2X with respect to the state-of-the-art wearable processors. Note that if each kernel only leverages the *patch* integrated in the local tile (without fusion), the throughput and normalized performance/watt will be much lower (i.e., 11.49 ms per gesture and 4.1X w.r.t. state-of-the-art wearable processor. See *Stitch w/o* fusion in Table I).

VI. Experimental Evaluation

This section presents a detailed experimental evaluation of *Stitch* architecture for its suitability in wearable devices. RTL synthesis and high level architectural simulations with representative wearable applications are used for the evaluations.

A. Wearable Applications

We evaluate *Stitch* with representative wearable applications. A *finger gesture recognition application* [46] (**APP1**) is used in Section V. We also evaluate a *CNN-based image recognition application* [49] (**APP2**), typically used in wearables such as smart glasses, consisting of two pooling layers, one fully connected layer, and three convolutional layers. The convolutional layers are parallelized into 13 kernels (threads) and other layers are encapsulated as individual kernels as shown in Figure 9. In addition, two wearable applications from the IoT benchmark suite [38] are selected: An *SVM-based machine learning application* (**APP3**) recognizing and encrypting anomalous images; A *transportation context-detection application* [50] (**APP4**) decrypting the encrypted sensoring data and identifying the context using the dynamic time warping algorithm, before encrypting the output.

All the applications are implemented with 16 kernels (as shown in Figure 9) to maximize the throughput and fully explore the benefit of *Stitch* architecture. Note that the multi-kernel applications are multi-threaded from the single-threaded version using a subset of MPI directives (lightweight message passing library [51]) and can run on any platform supporting MPI.

B. Architectural Simulation Setup

We use the *gem5* multi-core architectural simulator [52] for the performance evaluation. We use architectural simulator rather than FPGAs because gem5 allows convenient modification of architectures, especially for the cycle-accurate emulation of the single-cycle multi-hop NoC that is difficult to be evaluated using FPGAs. *Stitch* is also implemented in RTL to evaluate power, area using Synopsys Design

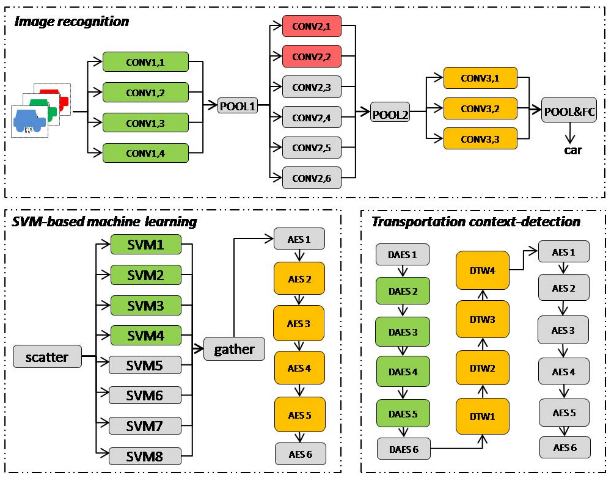


Figure 9. Representative wearable applications.

Compiler with 40nm standard cell library. *Stitch* consists of 16 in-order ARM cores (open-source ARM Amber core in RTL [53]) connected in a 2D mesh NoC (Garnet [54]) with 8KB Instruction Cache, 4KB Data Cache, 4KB SPM (Figure 2). Each tile incorporates one *patch* integrated with the core, and a compiler-scheduled NoC with no buffers and control logic (separate from the conventional NoC connecting the cores) connects all the *patches*. The detailed parameters of the simulated system of *Stitch* are listed in Table II.

General purpose	**Cores**	16 ARM in-order (single-issue) cores 200MHz
	Cache & SPM	2-way 8KB ICache, 2-way 4KB DCache, 64Bblock, LRU, 4KB SPM, 1-cycle access latency
	Inter-core Network	2-D Mesh, 16B-it, 1/5-flit control/datapackets, 5-stage router, 1-cycle link
	Memory	512MB, 30-cycle DRAM access latency
***Stitch* added**	**Inter-*patch* Network**	2-D Mesh, crossbar switch is driven by **clockless repeaters**, **bufferless**, 6 input x 6 output, 4-word/38-bit data/control (166-bit wide)
	patch	8 {*AT-MA*}, 4 {*AT-AS*}, 4 {*AT-SA*} 19-bit control signal, 4-input/ 2-output to register file

Table II
RTL AND SIMULATION PARAMETERS FOR *Stitch*.

For our *baseline*, we use a 16-core message-passing based architecture that has no customization (*patches* or compiler-scheduled NoC). The baseline has a larger 8KB data cache instead of an SPM*. In order to show the advantages of our *polymorphic patches* specifically designed for wearable applications, we compare *Stitch* with a recently proposed message-passing based many-core architecture, *LOCUS* [51], where each core is deployed with a conventional ISE accelerator. The conventional ISE accelerator used in *LOCUS* is a configurable special functional unit [11] capable of executing different custom instructions without incorporating load/store operations.

*There is no significant difference in performance between 8KB Data Cache and 4KB Data Cache + 4KB SPM in performance without custom instructions (only 1.5% performance degradation when replacing the 4KB Data Cache with SPM under appropriate data mapping strategy)

C. Architectural simulation results

Before diving into the throughput improvement of wearable applications, we first show how the individual kernels benefit from *Stitch*. Figure 11 shows the performance improvement across different representative wearable kernels running on a single core. Different kernels would benefit from different *patches*. For example, *dtw* benefits most from the *patch* {*AT-AS*}, while the *patch* {*AT-MA*} achieves the highest performance gains for *2dconv*. On an average, 1.56X performance improvement can be achieved by accelerating each kernel using single appropriate *patch*. In addition, an appropriate combination of *patches* can further improve the performance. Kernel *fft* achieves higer performance gains of 1.99X by stitching *patch* {*AT-MA*} and {*AT-SA*} together to form a fused *patch* {*AT-MA,AT-SA*} compared to the 1.37X performance improvement delivered by a single *patch*. The kernel *astar* does not show any significant improvement by stitching *patches* due to the smaller size of its identified computational patterns. Figure 11 also shows the performance of ISE accelerator in LOCUS. The *patch* performs better than the ISE in *LOCUS* because the *patches* introduce scratchpad memory to enable inclusion of load/store operations inside custom instructions.

Figure 10 demonstrates how the *patches* are fused together in the *Stitch* architecture to improve the overall throughput for different applications. Different applications lead to different stitching of the *polymorphic patches* based on Algorithm 1. In **APP2** for example, the fusion of *patch* {*AT-AS*} and *patch* {*AT-MA*} is the most suitable one for the kernel *2dconv*. However, there are seven *2dconv* kernels identified as bottlenecks, while only four pairs of {*AT-AS*} and {*AT-MA*} *patches* are available on the entire chip. Therefore, the {*AT-SA*} *patches* are also utilized and stitched together to accelerate the remaining *2dconv* kernels.

The normalized throughput boost across wearable applications with respect to the *baseline* (16-core message passing without any ISE acceleration) is shown in Figure 12. *LOCUS* (16-core message passing with conventional full-blown ISE accelerator per core) obtains an average 1.14X throughput boost while *Stitch* achieves 1.53X speedup even without fusion (*Stitch w/o Fusion*), where a kernel running on a certain tile just utilizes the local *patch* without stitching with the others. The difference between the two (*LOCUS* and *Stitch w/o Fusion*) confirms that the design of our *polymorphic patches* accommodates the characteristics of the wearable applications. By fusing together the appropriate *patches*, *Stitch* further improves the throughput by an average 2.3X. *Stitch* performs much better for **APP2** and **APP4** than **APP1** and **APP3** because the workload across the different cores in **APP2** and **APP4** is much more imbalanced. *Stitch* performs better than *LOCUS* because the stitching of heterogeneous

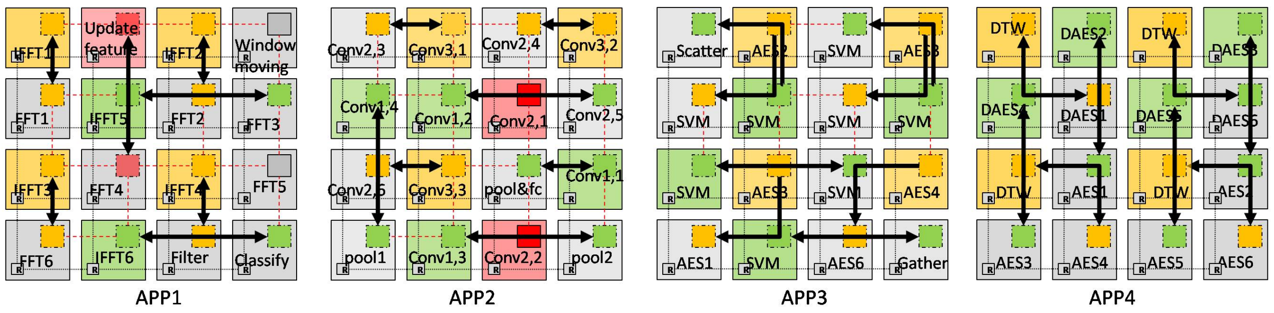


Figure 10. Stitching on *Stitch* architecture to adapt to different applications

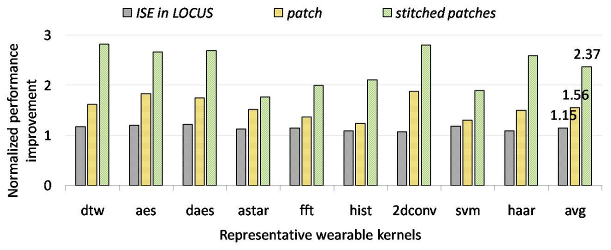


Figure 11. Normalized performance improvement of *ISE in LOCUS*, single *patch*, and stitched *patches* across representative wearable kernels compared to software-only implementation.

patches enables acceleration of larger patterns whereas the ISE in *LOCUS* employs identical per-core accelerator.

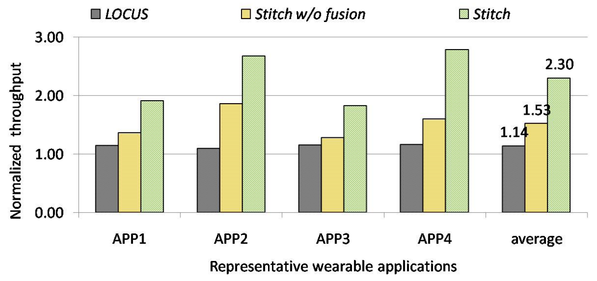


Figure 12. Normalized throughput improvement of different architectures w.r.t the *baseline* 16-core message passing architecture without any accelerator.

D. RTL timing, area and power

The *Stitch* architecture is implemented in RTL and synthesized using Synopsys Design Compiler. We tightly integrate the *polymorphic patch* with the pipeline of the open-source ARM Amber core [53], in parallel with the execute stage of the Amber processor pipeline. The design is implemented in Verilog and simulated with Synopsys VCS-MX. We use 40nm standard cell library for synthesis.

The synthesis results are shown in Figure 13. The power consumption of *Stitch* is around 140mW at 200MHz with its *polymorphic patches* and the inter-*patch* NoC accounting for 23% of the total power. Area is efficiently utilized in *Stitch* by stitching the *patches* together to accelerate the critical kernels. Figure 13 shows that the area overhead of the *polymorphic patches* and the inter-*patch* NoC in *Stitch* is only 0.5% of the chip area. We also synthesize the accelerators across different architectures (Table III). *Stitch* consumes only 0.169mm^2 area, while the accelerators in *LOCUS* consume 1.29*mm*^{2} area that is 7.64 times larger compared to *Stitch*.

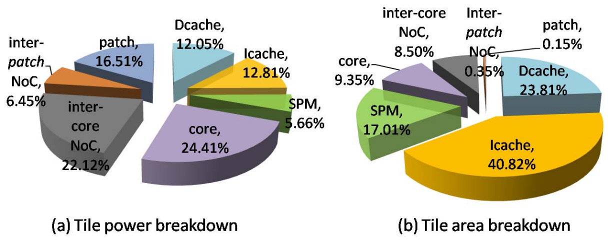


Figure 13. Power and area breakdown.

	LOCUS	*Stitch* w/o fusion	*Stitch*
actual area (μm^2)	1,288,044	49,872	168,568
area (%)	3.68%	0.15%	0.50%

Table III
AREA COST FOR ACCELERATORS IN DIFFERENT ARCHITECTURES.

NoC timing analysis. Table IV shows the synthesis results of the *polymorphic patches* and the inter-*patch* NoC. The *patch* {*AT-MA*} has the longest latency of 1.38ns among all the *patches*. [39] showed that the clockless repeated links can traverse up to 11mm within 1ns in a 45nm process. The number of hops traversed between two stitched *patches* is restricted to at most six to limit the length of the critical path. Including more hops can stretch the critical path from 1.36ns (single {*AT-SA*} including the NoC overhead : 2×0.17) to 4.63ns (one {*AT-MA*} and one {*AT-AS*} stitched together), which leads to 200MHz clock frequency for single-cycle execution of all custom instruction on the *Stitch* architecture. Specifically, the critical path of *Stitch* is in the *patch* and inter-*patch* NoC with a cumulative path delay of 4.63ns which goes through the inter-*patch* NoC switch1 (0.17ns) $\rightarrow$ *patch* {*AT-MA*} (1.38ns) $\rightarrow$ switch1 again (0.17ns) $\rightarrow$ 3 hops with 3 switches (0.3ns + 3$\times$0.17ns) $\rightarrow$ *patch* {*AT-AS*} (1.12ns) $\rightarrow$ (0.3ns + 3$\times$0.17ns) $\rightarrow$ inter-*patch* NoC switch1 (0.17ns). The 200MHz clock frequency in *Stitch* is lower than the processors in mobile and desktop platforms. However, it is appropriate for wearable devices with stringent power dudget. In Figure 12, all the architectures were set to run at 200 MHz. If we let *LOCUS* run at its maximum frequency of 400 MHz [51] and compare it with Stitch running at 200MHz, Stitch still wins in both performance (average 1.03X speedup across different kernels) and performance/watt (average 1.16X improvement across representative applications).

	patch AT-MA	*patch AT-AS*	*patch AT-SA*	NoC switch	3 hops
Delay(ns)	1.38	1.12	1.02	0.17	0.3
Area(μm^2)	4152	2096	2157	7423	-

Table IV
DELAY AND AREA OF DIFFERENT COMPONENTS.

Power-efficiency and Area-efficiency Both power and area are effectively utilized in *Stitch*. Figure 14 demonstrates the normalized power-efficiency (performance/watt) and area-efficiency (performance/area) of *Stitch* with respect to the *baseline*. On average, *Stitch* achieves 1.77X better power efficiency than the *baseline*. The average area-efficiency improvement (2.28X) is close to the throughput improvement (2.3X). This confirms that only trivial area overhead (0.5% chip area) is paid in introducing the *polymorphic patches* and the inter-*patch* NoC into *Stitch* architecture.

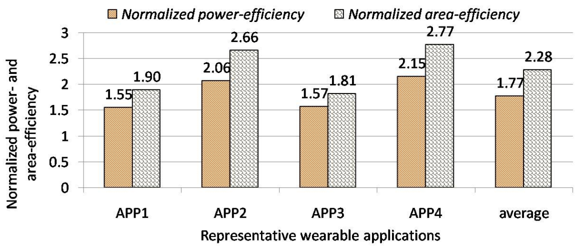


Figure 14. Normalized power- and area-efficiency of *Stitch* w.r.t the *baseline* 16-core message passing architecture without any accelerator.

E. Comparison with Processors in State-of-the-art Wearable Devices

We compare *Stitch* with the state-of-the-art wearables to illustrate its potential. Figure 15 shows the normalized throughput, power, and performance/watt with respect to quad-core ARM Cortex-A7 used in the state-of-the-art wearables. We use Ordroid XU3 board to collect the execution time and power of the quad-core Cortex-A7 running at 1.2GHz (the experiment set up is explained in Section V). The execution time of *Stitch* is obtained from *gem5* running at 200MHz, while the RTL simulation gives its power consumption. On average, *Stitch* architecture achieves 1.65X and 6.04X improvement in terms of throughput and performance/watt, respectively, compared to the state-of-the-art.

VII. RELATED WORK

We contrast *Stitch* against other configurable accelerator architectures, before delving into *Stitch*'s NoC architecture and comparing it with prior art.

A. Configurable Accelerator Architectures

A number of architectures have been proposed recently to incorporate reconfigurable fabrics with or within the processor datapath to accelerate different granularity of compute patterns. We separate prior research in this area into three

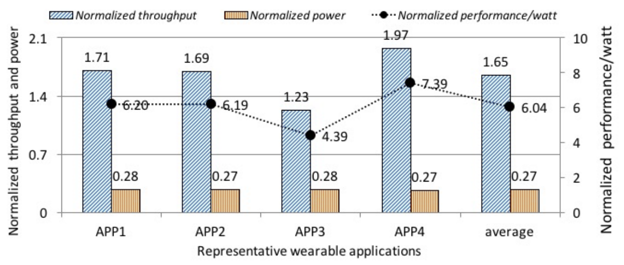


Figure 15. Normalized throughput, power, and performance/watt w.r.t quad-core ARM Cortex A7 in state-of-the-art wearables.

categories: those where the accelerator is **loosely coupled** with the processor pipeline, where the accelerator is **tightly coupled** and integrated within the processor pipeline, and where accelerators are **shared** across multiple workloads or cores.

Loosely coupled accelerators: As defined in Section 2, loosely coupled accelerator architectures have a distinct control and datapath from the main processor pipeline. Generally, a register file is needed to store the execution status, a local data memory is needed to feed the data, and a DMA controller is usually associated to transfer the data between the local data memory and DRAM or cache hierarchy. RISPP[23] architecture introduces the notion of atoms (basic datapath) and molecules (combination of atoms corresponding to specific compute kernel) implemented in a loosely coupled fabric. Different molecules are formed dynamically to accelerate specific kernels based on a combination of static and runtime analysis. RISPP dynamically adapts the tradeoff between performance and resource requirements for video codec applications. Plasticine [24] uses multiple configurable pipelined compute SIMD lanes to accelerate parallel compute patterns. Plasticine targets a specific application domain and the program must be written in a parallel pattern-based language. Both RISPP and Plasticine use complex accelerators with higher area, power budgets whereas *Stitch* uses tiny accelerators. Moreover, they are restricted to private accelerators per core, while *Stitch* can fuse together accelerators from different cores.

In addition, the Coarse-Grained Reconfigurable Arrays (CGRAs) provide word-level functional units to accelerate application loops with less configuration complexity and higher frequency compared to the FPGAs. Some of the CGRA architectures (e.g., Morphosys [25], EGRA [26], PACT-XPP [27], Elastic CGRA [28], CGRA express [29]) are loosely coupled with the processor. Morphosys [25] leverages a sophisticated interconnect network to effectively accelerate sub-tasks with different characteristics in applications. EGRA outperforms traditional CGRA design by utilizing heterogeneous cells to enable expression-grained acceleration. Similar to other loosely coupled accelerators (PipeRench [30], Polymorphic pipeline array [31], Veal [32]), both Morphosys and EGRA require local register files, control and data memories, and high data transfer

	Processor Integration	Granularity	Heterogeneity	Sharability in many-core	Technology Node	Accelerator Area Cost (mm^2)	Accelerator Area Overhead
RISPP	loose	kernel	✓	✗	FPGA-based	-	large
Plasticine	loose	kernel	✗	✗	28nm	112.8	large
MorphoSys	loose	kernel	✗	✗	350nm	180	large
EGRA	loose	kernel	✓	✗	90nm	3.7	medium
BERET	tight	traces	✓	✗	65nm	0.4	small
CCA	tight	op-chains	✓	✗	130nm	0.48	small
C-Cores	tight	kernel	✓	✗	45nm	0.326[*]	small
QsCores	tight	C-expression	✓	✗	45nm	0.77	small
DySer	tight	inner most loop	✗	✗	55nm	0.92	medium
LOCUS	tight	op-chains	✗	✗	32nm	2.3	medium
Stitch	tight	op-chains	✓	✓	40nm	0.17[†]	tiny

Table V
DIFFERENT ARCHITECTURES INCORPORATED WITH RECONFIGURABLE FABRICS

bandwidth, which significantly increases the design complexity and area overhead. In contrast, *Stitch* conserves precious on-chip real estate by closely integrating into the processor pipeline and leveraging the resources (e.g., fetch, decode, register file, and scratchpad memory) inside the CPU processor pipeline.

Tightly coupled accelerators: ADRES [33] and DySER [55] are CGRA-like accelerators tightly integrated into the processor pipeline and meant for accelerating the inner most loop bodies. Similarly, GARP [34] and Tartan [35] are tightly coupled within the processor and utilize FPGA-like and coarse-grained reconfigurable fabric, respectively, to provide the reconfigurability. All of these works target high-performance computing platforms, and thus the reconfigurable fabric area is larger, supporting very complex custom instructions compared to the tiny *polymorphic patches* in *Stitch*.

On the other hand, BERET [36], CCA [9], and C-Cores [37] tailor the reconfigurable fabrics based on the application characteristics. BERET architecture uses a set of heterogeneous subgraph execution blocks (SEBs) and a big dataflow graph (including control flow) is mapped to a subset of the SEBs. CCA contains different functional units connected in a triangular shape. The frequently occurring compute patterns (operation-chains) can be mapped onto it for acceleration. C-Cores are integrated into the processor pipeline and are patchable to adapt to different versions of the applications. Each application corresponds to different C-Cores, which would increase the area overhead when more applications are involved. ECOcores [56] improves the energy-efficiency and performance for irregular code by incorporating selective de-pipelining and cachelets on top of C-Cores. QsCores [57] are tightly integrated with the general-purpose processor via scan chains. QsCores can support large computation pattern including hundreds of operations, control flows, and irregular memory accesses prescribed as a single pattern in [58]. However, the QsCores cannot cooperate together across different tiles. Our recently proposed LOCUS [51] many-core architecture for wearables combines individual JiTC [11] cores in a 2D mesh where each core consists of a general-purpose processor and a configurable special functional unit (SFU) capable of executing different custom instructions.

All of the above architectures focus on a single core and it is expected that each core in a multi-core will be extended with the same reconfigurable logic. For example, all the SEBs in BERET are included in each core and no stitching across cores is supported. The patching in C-Cores can also only occur inside a single tile, and thus cannot deliver scalable performance with increasing core counts like *Stitch*.

Shareable accelerators for multiple cores: There have been attempts to share a reconfigurable fabric/co-processor among multiple cores [59, 60, 61] to avoid the per-core area overhead. But, in general, the co-processor is loosely coupled with the cores and is meant to accelerate large computation kernels. The sharing cannot scale well for many-core architectures and the co-processor is too large for wearable devices. Instead, *Stitch* distributes the *patches* across cores and stitches them together if necessary to offer virtual but exclusive accelerator per core. Fusion of processor cores has been explored in *CoreFusion* architecture [62] where multiple simple cores are fused together to create a wider processor pipeline that supports better instruction-level parallelism. In *Stitch*, the cores are not fused, only the *patches* are fused to accelerate certain computational patterns.

Table V summarizes and classifies the related works in terms of their level of integration with the processor, targeted code granularity, heterogeneity of accelerator units, and whether they share accelerators across cores. Loose vs. tight integration with the processor pipeline naturally leads to different acceleration granularity; Loosely coupled architectures, through taking advantage of the local data memory, enable acceleration of whole kernels [23, 24, 25, 26], while the tightly coupled architectures support fine-grain acceleration (e.g., inner most loop [55], operation-chains [9]). *Stitch* tightly integrates the *patches* with the

[*]C-Cores are customized accelerators per application. In order to be fair, we cite the highest area of the c-core (for application : vpr) that they have synthesized.

[†]This number is for the accelerators overhead of the entire 16-core instead of per-core in *Stitch*.

processor pipeline to exploit the acceleration of operation-chains. Some accelerators comprise identical functional units to simplify the design [9, 24, 25]. However, heterogenous accelerators [9, 23, 26, 36, 37] cater to the diverse acceleration requirements across applications, which is the motivation behind the heterogeneous *patches* in *Stitch*.

B. On-chip network architectures

Cong et al. [63] pointed out the need for predictable latency in NoCs that connect multiple accelerators and proposes the global management and reservation of circuit-switched paths in the NoC to match the accelerator timing. *Stitch*'s NoC provides predictable latency between the patches by relying on the compiler for setting up the NoC switches while ensuring no contention, and thus can avert the need for buffering or control logic within the NoC, resulting in an ultra-low-overhead NoC for accelerators in wearables.

There have been many prior designs of bufferless NoCs. Without buffering in the NoC, flits that contend with others can no longer be temporarily stored at the routers. Some choose to deflect the contending flits to other ports, misrouting them [64, 65] while others drop the contending flits [66, 67], with a regular NoC as the backup [67] or buffering and retransmitting the dropped flits at the network interface [66]. Like *Stitch*, [65, 68] embeds clockless repeaters to enable single-cycle across multiple hops in a bufferless NoC. However, all the above bufferless NoCs are dynamically controlled, whereas *Stitch* has no control logic as the compiler ensures flits will not contend at runtime.

Stitch's NoC is similar to other statically-scheduled NoCs. The MIT Raw [69] utilizes a compiler-scheduled NoC to transfer operands among multiple cores. The configuration of a router inside the static NoC is indicated by a 64-bit instruction word from a 8096-entry instruction memory. Hycube's NoC [70] which connects tiles within a CGRA accelerator similarly depends on the scheduler for setting up the NoC, and also uses a configuration memory to store the NoC configuration. *Stitch* uses control wires instead that can be naturally encoded by the ISE compiler, and thus needs just one configuration register per router. The register is preset before launching each application. [71] proposes a static reconfigurable NoC that also leverages clockless repeaters, that can be configured beforehand and bypass buffering and arbitration stages at intermediate routers during runtime. Instead, *Stitch* eliminates the buffers completely for realizing the ultra-low-overhead inter-*patch* NoC.

VIII. Conclusion

We propose *Stitch*, a many-core architecture where tiny heterogeneous, configurable and fusible ISE accelerators (*polymorphic patches*) are effectively enmeshed with the cores. Each *patch* can handle very simple ISEs and multiple *polymorphic patches* are able to be stitched together across the chip to create large, virtual accelerators for complex ISEs by using an ultra lightweight compiler-scheduled network-on-chip without any buffers or control logic. Our evaluations across representative wearable applications show an average 2.3X throughput boost for *Stitch* compared to a baseline many-core processor without ISEs, at a modest area and power overhead. Compared to the processors used in the state-of-the-art wearable smartwatch, *Stitch* architecture achieves average 1.65X throughput boost and 6.04X performance/watt improvement.

Acknowledgment

This work was partially funded by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2014-T2-2-129 and by Singapore National Research Foundation Research Fund NRF-RSS2016-005.

References

[1] "Huawei Watch2." https://goo.gl/cujvRb.
[2] "ASUS ZenWatch 3." https://goo.gl/Y8tBxN.
[3] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *Journal of Signal Processing Systems*, vol. 84, no. 3, pp. 339–354, 2016.
[4] "Tensilica Inc." http://www.tensilica.com.
[5] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of the 40th annual Design Automation Conference*, pp. 256–261, ACM, 2003.
[6] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–1229, 2006.
[7] P. Yu and T. Mitra, "Characterizing embedded applications for instruction-set extensible processors," in *Proceedings of the 41st annual Design Automation Conference*, pp. 723–728, ACM, 2004.
[8] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 69–78, ACM, 2004.
[9] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pp. 30–40, IEEE, 2004.
[10] R. E. Gonzalez, "A software-configurable processor architecture," *IEEE Micro*, vol. 26, no. 5, pp. 42–51, 2006.
[11] L. Chen, J. Tarango, T. Mitra, and P. Brisk, "A just-in-time customizable processor," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 524–531, IEEE Press, 2013.
[12] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 277–288, IEEE, 2009.
[13] "SmartWatch 2 APIs." https://goo.gl/IBGTmg.
[14] "Samsung Gear SDK." http://goo.gl/cT4qXJ.
[15] "AR Glasses SDK." http://goo.gl/o9Y5YM.
[16] "Google Glass SDK." https://goo.gl/jWeUh5.
[17] "Samsung Gear S." http://goo.gl/aE6ApL.
[18] "Snapdragon Wear 2100 Processor. https://goo.gl/14r8sx."
[19] S. Lee, J. Oh, J. Park, J. Kwon, M. Kim, and H.-J. Yoo, "A 345 mW heterogeneous many-core processor with an intelligent inference engine for robust object recognition," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 42–51, 2011.
[20] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pp. 683–688, IEEE, 2015.
[21] G. Tagliavini, G. Haugou, and L. Benini, "Optimizing memory bandwidth in OpenVX graph execution on embedded many-core accelerators," in *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pp. 1–8, IEEE, 2014.
[22] J. Bisasky, H. Homayoun, F. Yazdani, and T. Mohsenin, "A 64-core platform for biomedical signal processing," in *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pp. 368–372, IEEE, 2013.
[23] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: rotating instruction set processing platform," in *Proceedings of the 44th annual Design Automation Conference*, pp. 791–796, ACM, 2007.

[24] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A Reconfigurable Architecture For Parallel Paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 389–402, ACM, 2017.

[25] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE transactions on computers*, vol. 49, no. 5, pp. 465–481, 2000.

[26] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: A coarse grained reconfigurable architectural template," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 1062–1074, 2011.

[27] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPPA self-reconfigurable data processing architecture," *the Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[28] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic cgras," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 171–180, ACM, 2013.

[29] Y. Park, H. Park, and S. Mahlke, "CGRA express: accelerating execution using dynamic operation fusion," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 271–280, ACM, 2009.

[30] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[31] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 370–380, ACM, 2009.

[32] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 389–400, IEEE Computer Society, 2008.

[33] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *FPL*, vol. 2778, pp. 61–70, Springer, 2003.

[34] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[35] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 163–174, ACM, 2006.

[36] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 12–23, ACM, 2011.

[37] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 205–218, ACM, 2010.

[38] "IoT Kernels. https://github.com/iot-locus/kernels."

[39] T. Krishna, C.-H. O. Chen, W. C. Kwon, and L.-S. Peh, "Breaking the on-chip latency barrier using SMART," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 378–389, IEEE, 2013.

[40] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt, "Introduction of local memory elements in instruction set extensions," in *Proceedings of the 41st annual Design Automation Conference*, pp. 729–734, ACM, 2004.

[41] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pp. 212–217, European Design and Automation Association, 2006.

[42] A. Prakash, C. T. Clarke, and T. Srikanthan, "Custom instructions with local memory elements without expensive DMA transfers," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 647–650, IEEE, 2012.

[43] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. González, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero, "Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 720–732, IEEE, 2015.

[44] R. Manapat and K. Srinivasagam, "Method for interfacing a synchronous memory to an asynchronous memory interface and logic of same," Sept. 20 2005. US Patent 6,948,084.

[45] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for FPGAs," in *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pp. 111–117, ACM, 1995.

[46] C. Xu, P. H. Pathak, and P. Mohapatra, "Finger-writing with smartwatch: A case for finger and hand gesture recognition using smartwatch," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pp. 9–14, ACM, 2015.

[47] "Shimmer Wearable Device.." http://www.shimmersensing.com.

[48] "TI SensorTag." http://www.ti.com/ww/en/wireless_connectivity/sensortag/.

[49] J. Redmon, "Darknet: Open source neural networks in c," *h ttp://pjreddie.com/darknet*, vol. 2016, 2013.

[50] K. Sankaran, M. Zhu, X. F. Guo, A. L. Ananda, M. C. Chan, and L.-S. Peh, "Using mobile phone barometer for low-power transportation context detection," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, pp. 191–205, ACM, 2014.

[51] C. Tan, A. Kulkarni, V. Venkataramani, M. Karunaratne, T. Mitra, and L.-S. Peh, "LOCUS: low-power customizable many-core architecture for wearables," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, p. 11, ACM, 2016.

[52] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[53] "Amber ARM-Compatible Core. http://goo.gl/jshd3q."

[54] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 33–42, IEEE, 2009.

[55] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 503–514, IEEE, 2011.

[56] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor, "Efficient complex operators for irregular codes," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 491–502, IEEE, 2011.

[57] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 163–174, ACM, 2011.

[58] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 37–47, ACM, 2010.

[59] P. Garcia and K. Compton, "Kernel sharing on reconfigurable multiprocessor systems," in *ICECE Technology, 2008. FPT 2008. International Conference on*, pp. 225–232, IEEE, 2008.

[60] Y. Lu, T. Marconi, G. Gaydadjiev, and K. Bertels, "An efficient algorithm for free resources management on the FPGA," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 1095–1098, ACM, 2008.

[61] L. Chen and T. Mitra, "Shared reconfigurable fabric for multi-core customization," in *Proceedings of the 48th Design Automation Conference*, pp. 830–835, ACM, 2011.

[62] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: accommodating software diversity in chip multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 186–197, ACM, 2007.

[63] J. Cong, M. Gill, Y. Hao, G. Reinman, and B. Yuan, "On-chip interconnection network for accelerator-rich architectures," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 8, ACM, 2015.

[64] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 196–207, ACM, 2009.

[65] B. K. Daya, L.-S. Peh, and A. P. Chandrakasan, "Quest for high-performance bufferless NoCs with single-cycle express paths and self-learning throttling," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 36, ACM, 2016.

[66] M. Hayenga, N. E. Jerger, and M. Lipasti, "Scarab: A single cycle adaptive routing and bufferless network," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 244–254, IEEE, 2009.

[67] Z. Li, J. San Miguel, and N. E. Jerger, "The runahead network-on-chip," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 333–344, IEEE, 2016.

[68] B. K. Daya, L.-S. Peh, and A. P. Chandrakasan, "Low-power on-chip network providing guaranteed services for snoopy coherent and artificial neural network systems," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2017.

[69] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, *et al.*, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE micro*, vol. 22, no. 2, pp. 25–35, 2002.

[70] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 45, ACM, 2017.

[71] C.-H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L.-S. Peh, "SMART: a single-cycle reconfigurable NoC for SoC applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 338–343, EDA Consortium, 2013.