E-pipeline: Elastic Hardware/Software Pipelines on a Many-Core Fabric

Xi Zhang*, Haris Javaid *, Muhammad Shafique†, Jorgen Peddersen*, Jörg Henkel†, Sri Parameswaran*
*School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{zhangx, harisj, jorgenp, sridevan}@cse.unsw.edu.au
†Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe, Germany
{muhammad.shafique, henkel}@kit.edu

***Abstract*—On-chip many-core systems are expected to be in common use in the future. A set of homogeneous processors in a many-core system can be used to implement multiple pipelines which execute simultaneously. Pipelines of processors use varying numbers of cores when their workloads vary at run time. In this paper, we show how such a system executing multiple pipelines with varying workloads can be implemented. We further show how the system can switch cores within a pipeline (intra-elasticity) and between pipelines (inter-elasticity). The method is named *E-pipeline*, and is implemented and evaluated in a commercial tool suite. Compared to reference design methods with clock gating, E-pipeline achieves the same power savings, maintains the throughput to meet throughput constraints and reduces core usage by an average of 37.7%. The adaptation overhead for switching cores is approximately 2μs.**

I. Introduction

A. Introduction

On-chip many-core systems are anticipated to become a large part of the embedded device industry [1], [2]. Hundreds and even thousands of cores on a chip are foreseen in the future. Such a homogeneous many-core system is desirable to avoid the increasing expense of fabricating, since such a chip is reusable in many different industries and applications. Much like a single FPGA design which pervades the industry, the next generation of on-chip many-core systems can be programmed in-situ and used in a wide variety of applications. In order to use such a system in embedded applications, run-time adaptation is necessary which allows the homogeneous many-core system to achieve power and performance efficiencies by dynamically adapting the system to specific applications.

The hardware/software pipeline divides a stream application into sequential stages and assigns a number of cores to different stages [3], [4]. Streaming applications are implemented as hardware/software pipelines when high throughputs are necessary. Traditionally, such pipelines in many-core systems are designed such that the throughput constraints (e.g., frames per second) are met even in the worst case of workloads. For streaming applications with workload variations, such as H.264 [4], worst-case designs may overestimate workloads by more than an order of magnitude for the average case. To adapt to workload variations, work in [5] studied run-time task mapping, and work in [6] studied run-time task duplication. However, their works focused on maximizing throughput rather than minimizing both resource usage and power consumption under the throughput constraint. Also their management overheads are typically at the level of hundreds of milliseconds, which are over 10× the workload variation intervals in many modern streaming applications. Hence their works are not suitable for for fast adaptation. Works in [4], [7] applied run-time low power techniques to reduce the dynamic power when run-time workloads were much lower than the worst case. However, their works still allocate as many cores as necessary for the worst case designs.

In this paper, for the first time we consider the scenario where multiple pipelines with workload variations are executing on the same on-chip many-core system. Since their peak times of workload may not happen at the same time, we study the system to reuse cores between pipelines, thus reducing the need for resources and reducing power. We call this Elastic computing.

Elastic computing technologies can adapt to workload variations and reuse cores between applications. In this paper, we propose *E-pipeline*, a fine-grained elastic computing methodology for streaming applications in on-chip many-core systems. Inside a pipeline, *E-pipeline* applies *intra-elasticity* which adapts core assignments to the workload variations so that the throughput constraint is met. Between separate pipelines which are executed in parallel, *E-pipeline* applies *inter-elasticity* which allows the unnecessary cores in one pipeline to be dynamically given up for use by other pipelines or applications running on the same chip. In addition, *E-pipeline* applies low-power technologies (clock gating) to necessary cores to reduce power consumption when throughput constraints of all pipelines are met.

B. Contributions

For the first time, we introduce the concept of executing multiple hardware/software pipelines on a many-core system, which are able to utilize and shed processor cores as necessary as the workload varies. The main contributions can be summarized as follows:

- Intra-Elasticity: A method to switch cores within a single pipeline (from stage to stage), as stages within the pipeline are idle (or overwhelmed), so that the pipeline can meet a given throughput.
- Inter-Elasticity: A method to switch cores from one pipeline to a pool of cores which are asleep when the cores are not necessary, and a method to obtain cores from the sleep pool to a pipeline when the pipeline needs an additional core to meet throughput.
- We further show how a distributed management system can be incorporated for inter- and intra-elasticity of pipelines and be effectively implemented.

II. Task Cloning Mode

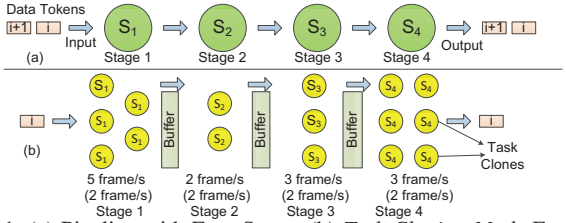

Fig. 1: (a) Pipeline with Four Stages (b) *Task Cloning Mode* Example

Fig. 1(a) describes a hardware/software pipeline with four stages (S_1, S_2, S_3 and S_4). The input data of each stage is packed as data tokens. The throughput is measured by frames per second and each frame is composed of a certain amount of data tokens. Each stage works on an input data token and then produces a processed data token, which is the input data token of the next stage. The communication between stages is through buffers. An iteration of one stage includes fetching

Haris Javaid is now affiliated with Google Inc.

978-3-9815370-4-8/DATE15/©2015 EDAA

a data token, performing the task of this stage on this data token and sending the processed data token to the buffer. The workload of a stage is measured as the average clock cycles consumed per iteration.

We focus on a particular program mode of hardware/software pipelines, named *task cloning mode*, which can be applied to various streaming applications (we applied it to 14 benchmarks ranging from media applications, such MPEG encoder, and communication applications to signal processing applications, such as FFT). In *task cloning mode*, the execution for an iteration of the stage only depends on its input data token and does not depend on other data tokens or other stages. Hence, the *task cloning mode* allows a task of a stage to be cloned several times, and each task clone works on different data tokens independently. Fig. 1(b) shows a *task cloning mode* example for the pipeline in (a). There are various numbers of task clones in each stage. The throughput of a stage is the total throughput of task clones in this stage. The stage with the heaviest workload (the *bottleneck stage*) determines the effective throughput of the pipeline (within brackets of Fig. 1(b)). Based on the discussion in work [7], [8], we can keep increasing the number of clones at the *bottleneck stage* to increase the effective throughput until this stage is no longer the bottleneck. Similarly, we can keep decreasing the number of task clones in other stages without affecting the effective throughput until the stage becomes the *bottleneck stage*. Details of discussion are explained in [7], [8].

In this paper, one core executes one task clone for the sake of simplicity thus reducing overheads. *Intra-elasticity* and *inter-elasticity* are employed by changing the numbers of cores/clones which are assigned to stages of pipelines.

A. Motivational Example

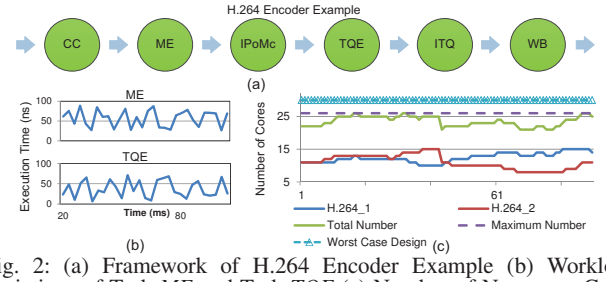

Fig. 2: (a) Framework of H.264 Encoder Example (b) Workload Variations of Task *ME* and Task *TQE* (c) Number of Necessary Cores Variations of Two Pipelines

In this section, we examine an hardware/software pipeline example of H.264 encoder with a throughput constraint of 30 Frame/s. The H.264 encoder shown in Figure 2 (a) is composed of six tasks: task *CC* for color conversion; task *ME* for motion estimation; task *IPoMC* for intra-prediction or motion compensation; task *TQE* for transform, quantization and entropy coding; task *ITQ* for inverse transform and quantization; and task *WB* for writing back. Workloads of task *ME* and task *TQE* depend on the input data (similarity between the current input frame and the reference frame).

The workload distributions of Task *ME* and *TQE* (represented by execution time for a data token) of the H.264 example is shown in Figure 2 (b), where the execution time of *ME* and *TQE* varies significantly. Note that the worst case of the workload for *ME* is not the worst case for *TQE*: When the workload of *ME* is high, the workload of *TQE* can be low, and vice versa. It necessitates the intra-elasticity to change core assignments between different worst cases of the pipeline.

With run-time workload variations, the number of cores which are necessary for the H.264 pipeline to meet the 30 Frame/s constraint also varies with time. There are two pipelines of the H264 encoder with different input data (*H.264_1* and *H.264_2*). Figure 2 (c) shows the necessary run-time numbers of cores needed by *H.264_1* and *H.264_2*. It is worth noting that, in Figure 2 (c), the total number of cores used in the two pipelines varies (the third curve from the top) and never exceeds 26 (the second dotted line from the top). Each pipeline occupies 15 cores at their peak points, however, they do not occupy 15 cores at the same time. If the pipelines are designed based on worst case designs, they will always occupy 30 cores in total (each pipeline occupies 15 cores). Similarly, if we execute two or three differing applications, then we would be able to execute on a lesser number of cores, compared to the design of allocating the worst case number of cores for each application. Thus *E-pipelines* allow switching for cores between pipelines, reducing the necessity for allocating a maximum number of cores.

III. Related Work

Hardware/software pipelines have been implemented in various architectures [9]–[13]. Traditional worst-case design methodologies can satisfy the throughput requirement, however, they usually are neither power efficient nor resource efficient when workloads of pipelines vary at run time [3], [4] as they assume worst case workloads.

Works in [14], [15] discussed run-time task mapping for multi-objective optimization in on-chip many-core systems. The work in [5] described a method of remapping tasks between cores to improve the throughput when workloads vary. The work in [6] studied dynamic task duplication at operating system level for streaming applications to improve the throughput. The management time overheads of all these works [5], [6], [14], [15] are large due to coarse-grained hardware/software managements, and the throughput decline (due to workload variation) lasts seconds [5]. Hence, their methods are not suitable for modern streaming applications under throughput constraints with fast workload variations (e.g. at millisecond or microsecond levels). Some works, such as [16], presented run-time task mapping methods in on-chip many-core systems to minimize communication latencies. They did not examine the effect on throughput.

Papers in [4], [7], [17], [18] discussed pipelines with workload variations under throughput constraint, and presented worst-case designs with dynamic low-power methods. Works in [7], [17], [18] reduced the operating frequency/voltage of each core when workloads are smaller than the worst case. The work in [4] applied knowledge of workload variation to forecast the period when the core is not utilized. The core is set to sleep to save power during un-utilized period. In [7], [17], [18] and systems can adapt to fast workload variations for modern streaming applications under throughput constraints. However, the core-to-task assignments in these works are static. They did not study changing core-to-task assignments and reusing cores across multiple pipelines which are running in parallel to optimize the total power and core usage.

Elastic computing techniques [19] such as invasive computing [20] are resource-efficient methods that can adapt to workload variations. However, their approach focuses on application level variations rather than stage level variations, which are more important in pipelines to improve throughput while reducing usage of cores. Work in [8] employed a similar program mode and applied a methodology akin to *intra-elasticity*. However, the work [8] only examines a single pipeline with a fixed number of cores, and focused on maximizing the throughput. *E-pipeline* discusses optimizing power and resources with varying numbers of cores. No run-time work exists which employs fine-grained inter-elasticity and intra-elasticity methods to improve the power- and resource-efficiencies without violating throughput constraints of streaming applications.

IV. System Overview

The on-chip many-core system assigned a core named *'manager'* for each hardware/software pipeline to be executed. Fig. 3 shows the overview of the system with three pipelines (*Pipeline A, Pipeline B* and *Pipeline C*). The *managers* perform the run-time adaptation, while *workers* are assigned to separate pipelines to execute specific tasks (*workers* for different stages

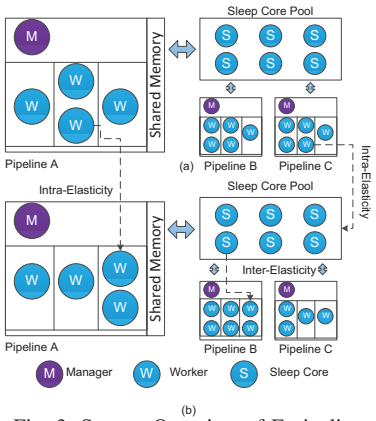

Fig. 3: System Overview of E-pipeline

are separated by vertical lines within the pipelines in Fig. 3). The communication between cores in a pipeline is based on the shared memory. There is a common *sleep core pool* where unnecessary cores - depicted by S - are set to sleep (i.e., clock gating). Fig. 3 (a) and (b) shows two scenarios before and after the adaptation, which are demonstrated by directed dotted arrows. For *Pipeline A*, an adaptation of *intra-elasticity* is performed as a core is switched from one stage to the other stage. For *Pipeline B and C*, *inter-elasticity* is performed through *sleep core pool*. A *sleep core* is assigned to *Pipeline B*, while a core in *Pipeline C* is set to sleep and added to the *sleep core pool*, indicated by arrows. Note that, in this paper, we assume that there is a sufficient number of cores in the system. If there are not enough cores, it may indicate that the system can only guarantee the throughput of important pipelines. Such a priority based pipeline system is not studied here, but is a simple extension of this work.

V. Methodology

A. Methodology Overview

Figure 4 (a) shows the program executed on the *manager* of a pipeline. The input is a pipeline with S stages. The *manager* initializes the pipeline with one core assigned to each stage. At run time, the *manager* monitors the execution information and performs an adaptation. In the adaptation, it first reads the monitor information, then finds the *bottleneck stage* and the *non-critical stage* (the *bottleneck stage* restricts the throughput and the *non-critical stage* contains one or more cores that are not necessary), performs the elasticity management (adaptation), and finally sends task assignments to cores (task assignments consist of choosing cores and sending task assignment information of cores if the cores are used for adaptation; if a core is to be sent to sleep, a sleep signal is sent to it). The execution of the worker is explained in Section V-B. After the adaptation, the *manager* monitors the pipeline again. A new adaptation will be performed after a set number of iterations.

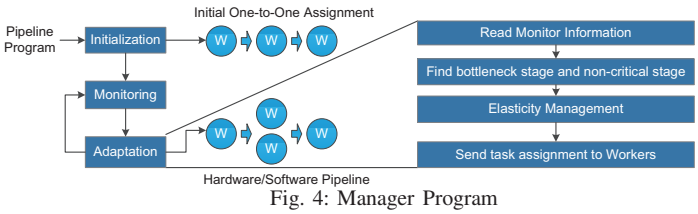

Fig. 4: Manager Program

B. Worker Program

Algorithm 1 presents the execution loop of a *worker*. In every iteration of the execution loop, the core first checks which pipeline it is assigned to by fetching *pipeline id*, and then checks which stage of the pipeline it is assigned to by fetching

Algorithm 1: Execution of A Worker

```
1 while the pipeline termination condition is false do
2     pipeline id = FetchPipelineid();
3     stage id = FetchStageid();
4     FetchDT(pipeline id, stage id);
5     ExecuteStage(pipeline id, stage id);
6     SendDT(pipeline id, stage id);
7     CollectandSendTimeInformation();
```

stage id. The *pipeline id* and *stage id* are the task assignment information sent from the *manager*. Based on the *pipeline id* and *stage id*, the core fetches one input data token (*FetchDT()*), executes one iteration of the assigned stage (*ExecuteStage()*), sends one output token using function (*SendDT()*) to the output buffer and executes function *CollectandSendTimeInformation()*. Function *CollectandSendTimeInformation()* collects time information for adaptation and stores time information in memory. After *CollectandSendTimeInformation()*, the core will check to see whether the *manager* has updated the *pipeline id* and *stage id* or whether it has asked the core to go to sleep. If the *manager* has changed the status, the core will execute the new task or go to sleep. Else it will continue with the old task.

C. Adaptation Methodology

The adaptation method of *manager* is shown in Algorithm 2. The new adaptation is triggered when *n* iterations are complete after the last adaptation (note that we have determined *n* experimentally). The input of the adaptation is an array of *t* (t_1 to t_n) and an array of *unutilized time* of each stage. Both arrays are the time information collected by *workers* (details of time information collection method are explained in Section VI-C). The t_1 to t_n denote the finishing times of each of the last n data tokens. The average throughput of the previous n-1 iterations is calculated by:

$$Throughput = \frac{t_n - t_1}{n-1} \quad (1)$$

The *unutilized time* is the average locking time per *worker* in a stage. The locking time is caused by *write/read locking mechanism*, which is employed for synchronization. For example, the reading operation on the input data is locked when the input buffer is empty, while the writing operation on output data is locked when the output buffer is full. The locking time of a stage (the time locked by *write/read locking mechanism*) reflects the workload difference between this stage and the *bottleneck stage* [8].

The adaptation is composed of *ThroughputMeasurement()*, *FindBottleneckNoncriticalStage()* (see below), and *Elasticity-Management()* (see below). Finally, the *manager* resets the time information, and then restarts another adaptation after *n* iterations.

Algorithm 2: Adaptation Methodology

```
1 while t_n > 0 do
2     throughput=ThroughputMeasurement(t);
3     FindBottleneckNoncriticalStage();
4     ElasticityManagement();
5     Reset time information;
```

1) Find Bottleneck Stage and Non-critical Stage: Algorithm 3 shows the algorithm of *FindBottleneckNoncritical-Stage()*. As mentioned in Section V-C, the *unutilized time* of a stage (the average locking time of write/read locking mechanism per *worker* in a stage) reflects the average under-utilization extent of a *worker* in this stage. The stage with the least under-utilization is considered as the *bottleneck stage* ($S_{bottleneck}$). For other stages, we can calculate the normalized number

of cores that are utilized ($N_{normalied}$), and then estimate the throughput after removing one core in this stage. If the result is still greater than the constraint, this stage is named the *non-critical stage* ($S_{noncritical}$). When there are multiple *bottleneck stages* and *non-critical stages*, the last one of them is identified as $S_{bottleneck}$ or $S_{noncritical}$. When there is no *non-critical stage* or *bottleneck stage*, the value of $S_{noncritical}$ or $S_{bottleneck}$ remains *0*.

Algorithm 3: Find Bottleneck Stage and Non-critical Stage

```
1  S_bottleneck = 0; S_noncritical = 0;
2  for each i from 1 to S do
3      if IsBottleneckStage(S_i) = 1 then
4          S_bottleneck = S_i;
5      if IsBottleneckStage(S_i) = 0 then
6          N_i = the number of cores in S_i;
7          N_normalied = N_i × (1-Unutilized Time / Time of n iterations);
8          if throughput × (N_i − 1)/N_normalied > Throughput
           Constraint then
9              S_noncritical = S_i;
10 Output S_bottleneck, S_noncritical;
```

2) Elasticity Management: Algorithm 4 shows the algorithm of *ElasticityManagement()*. According to the comparison between the measured throughput (*throughput*) and the throughput constraint, different strategies are chosen. When the throughput is smaller than the constraint, the strategy, based on the discussion in Section II, is to increase the throughput by increasing the number of *workers* in the *bottleneck stage*. The first option is to reassign a *worker* from the *non-critical stage* to the *bottleneck stage* (*intra-elasticity*). If there is no *non-critical stage*, the *manager* awakes a *sleep core* from *sleep core pool* and assigns this core to the *bottleneck stage* (*inter-elasticity*). When the throughput is greater than the constraint, the strategy, based on the discussion in Section II, is to decrease the number of *workers* in the *non-critical stage* and set the removed core to sleep (*inter-elasticity*). If there is no *non-critical stage*, no action is performed.

Algorithm 4: Elasticity Management

```
1  if throughput < constraint then
2      if S_noncritical = 0 then
3          C_i=AwakeACore(sleep core pool);
4          // inter-elasticity
5          Core-to-Stage(C_i, S_bottleneck);
6          Exit;
7      C_i = SelectCore(S_noncritical); // intra-elasticity
8      Core-to-Stage(C_i, S_bottleneck);
9  if throughput > constraint then
10     if S_noncritical = 0 then
11         Exit;
12     C_i = SelectCore(S_noncritical);
13     SleepCore(C_i); // inter-elasticity
```

VI. Implementation

This section details the implementation of the *E-pipeline*. The platform setting is presented, followed by the implementation details of memory layout, throughput/*unutilized time* measurement and core-to-stage assignments.

A. Platform Setting

Fig. 5 shows the overview of the implementation. The basic prototype of *E-pipeline* is built with 48 Tensilica's XTensa LX4 [21] cores and two memories. Each core is working in the frequency of 1 GHz and tailed with 1KB instruction cache, 1KB data cache, 1MB local memory (for use of instructions as well as local data storage). A 256MB main memory is used for general purposes, such as storing input data, experiment results and run-time measured power. A shared memory of 16MB is used for the management and communication of hardware/software pipelines. The management and communication of a pipeline in our benchmark library require no more than 8KB, hence, the size of 16MB is sufficient to run multiple pipelines in parallel. Finally, there is a global timer which is used for cores to get time stamps.

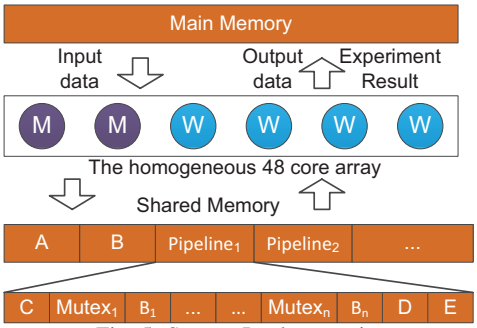

Fig. 5: System Implementation

B. Memory Layout

The shared memory are divided into blocks (Labeled by *A, B*, etc. in Fig. 5) for the management and communication of pipelines. Block *A* is the *core-to-stage assignment table*. Each core has a *core id* (C_1 to C_{48}) which corresponds to an address in the *core-to-stage assignment table*. Each address stores a two-byte data. The upper byte denotes the *pipeline id* while the lower byte denotes the *stage id* (if the core is sleeping, the number is set to *0*). Block *B* is the *sleep core table*. When a core is set to sleep, the *manager* writes its *core id* into this table, while the *manager* removes the *core id* of a core from the *sleep core table* when this core is assigned to a pipeline.

The rest part of the memory is divided into blocks for multiple pipelines ($Pipeline_1$, $Pipeline_2$, etc.). Each block (e.g. $Pipeline_1$) is composed of sub-blocks (as shown in Fig. 5): Block *C* is an *assigned core table*, where the *manager* records *core ids* of *workers* in the pipeline. Sub-blocks from B_1 to B_n are the *n* buffers that are used for communication between stages, and $Mutex_1$ to $Mutex_n$ are for the mutual exclusion (Mutex) locking mechanism for these buffers. Each *Mutex* contains *read and write pointers, empty and full signals, locks to avoid race conditions during buffer access by multiple readers/writers* and *data sequence signals*. The *data sequence signal* stores the sequential number of the last data token stored in the buffer. The new data token (e.g. data token *i*) can be sent to the buffer only when its previous data token (e.g. *i-1*) is already recorded by the *data sequence signal*. Finally sub-blocks *D* and *E* store the array of *t* (t_1 to t_n) and the array of *unutilized time* of each stage, which are used by adaptation. The *empty and full signals* of each buffer are also used to detect the *write/read locking*.

C. Throughput and Unutilized Time Measurement

When a final stage of a pipeline completes a data token, the final stage creates a time stamp and stores it in the shared memory in sequence as t_1 to t_n. Time stamps t_1 to t_n are fetched by the *manager* for throughput measurement (illustrated in Equation 1) when *n* time stamps are available. After adaptation, the time stamps are cleared by the *manager* so that new time stamps can fill this area. The *unutilized time* of each stage is stored in data structures in the area labeled as *E* in Fig. 5. Each data structure is a software accumulator, to which all *workers* of a stage send their unutilized time. When the *empty and full signal* of a buffer indicates *buffer empty*, a core which is going to read data from the buffer is locked by the *write/read locking*

mechanism, and the core creates a time stamp to record the beginning of the locking time. The core keeps checking the *empty and full signal* of this buffer periodically until the lock is released, and then creates another time stamp to record the end of the locking time. The difference between the two time stamps is the *unutilized time* for this core. When the *empty and full signal* of a stage is *full*, the *unutilized time* is similarly recorded. The *unutilized time* is accumulated in the software accumulator of the stage. The *manager* gets the average *unutilized time* of a core in this stage through dividing the sum of the accumulator by the number of cores assigned to the stage.

D. Core-to-stage Assignment Implementation

The pointers of different blocks are known to cores (for example, the addresses of A, B, etc. in Fig. 5 are known). When a *manager* performs Algorithm 4. The function *SelectCore(i)* returns a *core id* of a core from the stage *i* by checking *assigned core table*. The function *AwakeACore(S)* wakes a core from the *sleep core pool* and returns the *core id* of the core by checking *sleep core table*. The returned *core id* is used in the function *Core-to-Stage(core id, stage id)*. Function *Core-to-Stage(core id, stage id)* changes the task assignment of the core by writing *stage id* to the corresponding address of *core id* in the *core-to-stage assignment table*. At the start of one iteration, the *worker* fetches the *pipeline id* and *stage id* from the *core-to-stage assignment table*, and jumps to the new stage. The time overhead of the *worker* to switch from one stage to the other is 231ns (measured during experiments described below).

VII. Results

We first execute one benchmark at a time to verify intra-elasticity of the system, and then execute several benchmarks in parallel to verify inter-elasticity of the system. Results of *E-pipeline* are compared to reference designs (minimum number of cores needed to satisfy throughput in the worst case) with clock gating as used in [4]. In *E-pipeline*, when a core is added to the *sleep core pool*, it is set to sleep (applying the same clock gating technique as in [4]). Note that the work in [4] used a large number of cores so that worst cases could be handled, and the cores were switched off, when worst case scenario was not present. *E-pipeline*, on the other hand, allows run-time task reassignments from one stage to another or from one pipeline to another. We use identical methods to find *bottleneck stages*, *non-critical stages* and measure throughputs in both techniques so that we can make a fair comparison.

Seven benchmarks with workload variations were used. These were H.264 encoder (H264), MPEG decoder (MPEGdec), MPEG encoder (MPEGenc), finite impulse response filters (FIR), fast fourier transform (FFT), Compress Algorithm (Compress), insert-sort algorithm (Insertsort). For media applications, the throughput constraint was set to 30 frames per second; while for other applications, the throughput constraint is set to the maximum throughput that can be achieved in a hardware/software pipeline with 15 *workers* in the reference design.

A. Experiment Setting

We build our *E-pipeline* using Tensilica's cycle accurate multiprocessor simulation tool – XTensa Modeling Protocol (XTMP) [22]. The average memory delays of accessing caches, local memories, the shared memory and the main memory are set to 1 cycle, 3 cycles, 8 cycles and 64 cycles. We assume that the shared memory is non-blocking for multiple accesses. The clock gating technique is applied to both *E-pipeline* and the reference designs for fairness of comparison. Based on the LX_4 parameters given by XTMP (in 45nm technology), the switching overhead for clock gating is set to 1 clock cycle, and the dynamic power/leakage power is set to 41.23/4.25mW per core. We assume that when a core is working, it consumes both dynamic power and leakage power (45.48mW); while when a core is set to sleep, it consumes only leakage power (4.25mW). The power consumption is calculated by the number of working cores multiplied by 45.48mW and the number of sleeping cores multiplied by 4.25mW. We evaluate the average power consumption throughout the life time of executing benchmarks. In addition, the *manager* is considered to be a working core and its power consumption is included in the measurement.

The adaptation starts every *n* iterations. Differing values for *n* were explored. We change it from 4, 6, 10, 12 to 15, and measure the average throughputs of all benchmarks with different *n*. The average normalized throughputs with different *n* are 1.044, 0.967, 1.016, 0.975 and 1.029 (the normalized throughput constraint is *1*). There was no clear pattern, due to the varying benchmarks and the differing loads. However, for the sake of experimentation we used *n=10*, which provided the closest throughput compared to the throughput constraint, in experiments for which the results are provided below.

B. Experiment Results

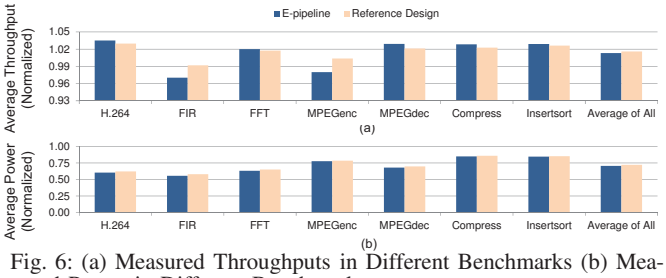

Fig. 6: (a) Measured Throughputs in Different Benchmarks (b) Measured Power in Different Benchmarks

Fig. 6 shows the execution of running single benchmarks. Fig. 6 (a) compares the normalized average throughputs between *E-pipeline* and the reference design with clock gating (we set the throughput constraint as *1*). Figure (b) shows the normalized average power consumptions of *E-pipelines* and the reference design (we set the normalized power consumption *1* when all cores of the reference designs are always working without the *manager*). The average throughput of *E-pipeline* (normalized 1.013) is slightly less than the reference designs (normalized 1.016). This is due to the switching overhead for reassigning cores to tasks in *E-pipeline* (typically 231ns in experiments). However, *E-pipeline* achieves better power saving (0.70) than the reference design (0.72) because it allows cores to be reassigned from one stage to another. For example, *E-pipeline* can reassign one core from *Stage A* to *Stage B*; while the reference designs have to awake another core in *Stage B*, and then set one core in *Stage A* to sleep, consuming extra leakage power. The higher the leakage power is, the better power saving *E-pipeline* can achieve. The management time overheads vary from 678 to 1912ns. The 678ns is the time overhead when the *manager* does not find a *non-critical stage* and the adaptation function exits (line 9-10 in Algorithm 4). The 1912ns is the maximum time overhead, including adaptation overheads and Mutex locking delays. This time only includes intra-elasticity adaptation.

Fig. 7 shows the results of running benchmark combinations in parallel. Fig. 7 (a) is the execution of the combination of FIR and FFT; (b) is the combination of H.264 and MPEGdec; (c) is the combination of MPEGenc and MPEGdec; (d) is the combination of H.264, MPEGdec and MPEGenc; (e) is the combination of H.264, MPEGdec and FFT; and (f) is the combination of Insertsort, Compress and MPEGenc. The curves at the bottom (labeled as *A* in Fig. 7) are the run-time numbers of cores used for each pipeline in *E-pipeline*. The dotted curves at the middle (labeled as *B* in Fig. 7) are the total number of cores used in *E-pipeline*. The first curves from the top (labeled as *C* in Fig. 7) are the numbers of cores used for the reference designs with clock gating, and the second curves from the top (labeled as *D* in Fig. 7) are the maximum numbers of cores used in *E-pipeline*. Note that, the numbers of cores in Fig. 7 do not count *managers*, as the number of managers do not change at run time. From Fig. 7, we can see that resources are saved since

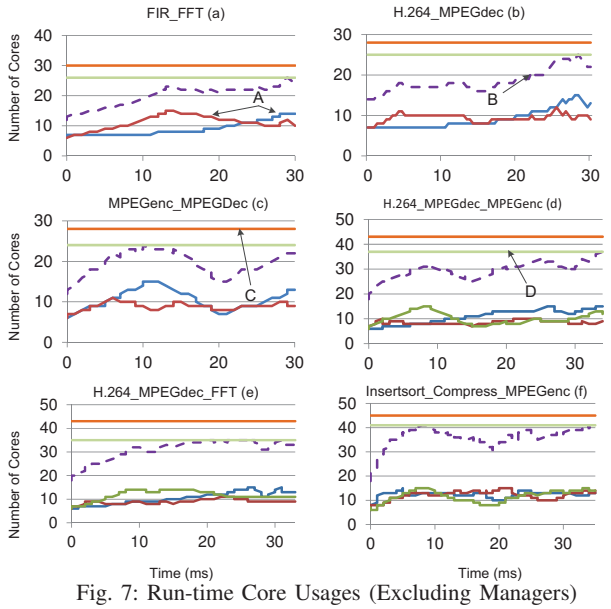

Fig. 7: Run-time Core Usages (Excluding Managers)

the worst case workloads of different pipelines normally do not occur at the same time. For reference designs with clock gating, unnecessary cores in a pipeline can only be set to sleep rather than share with other pipelines. Thus, the maximum number of cores being used by pipelines is reduced in *E-pipeline* by sharing cores between pipelines. The management time overheads vary from 694ns to 2352ns. The maximum overhead (2352ns) includes both inter- and intra-elasticity adaptation. Compared to works in traditional methodology with adaptive cores which take hundreds of milliseconds in management time overheads [5], [6], [14], [15], the management time overheads of *E-pipeline* are significantly small.

Table I summarizes system-level results of different benchmark combinations in Fig. 7 (i.e. (a) shows results of the benchmark combination in Fig. 7 (a), and (b) shows results of the benchmark combination in Fig. 7 (b), etc.). In Table I, *E-pipe* denotes *E-pipeline* and *Ref.D* denotes the reference design. Results include the total number of cores used in the system at run-time (*Number of Cores* - the range and the average number), the average power of the system (*Power*) and the energy consumption of the system (*Energy*). Note that, *mangers* are considered in system-level results. From the table, we can see that *E-pipeline* can achieve nearly the same power saving factor as the reference design with clock gating. However, the *E-pipeline* can save an average 37.7% of resources compared to reference designs (measured by the average number of cores used in *E-pipeline* and the number of cores used in the reference design). As the number of pipelines being executed increases, the number of cores saved will continue to increase, allowing additional applications to be executed in the many-core system.

TABLE I: Summary of Results (Including Managers)

	Number of Cores			Power (mW)		Energy (mJ)	
	E-pipe		Ref.D	E-Pipe	Ref.D	E-Pipe	Ref.D
	Rang.	Ave.					
(a)	16 ∼ 29	18	33	803	820	111	113
(b)	17 ∼ 28	20	31	912	925	119	123
(c)	17 ∼ 27	20	31	931	948	127	129
(d)	22 ∼ 40	28	46	1296	1321	175	183
(e)	23 ∼ 38	26	46	1199	1233	160	163
(f)	21 ∼ 44	36	48	1651	1747	241	242

C. Conclusion

The paper describes *E-pipeline*, an elastic computing method for power and resource efficiencies in on-chip many-core systems. Based on *task cloning mode*, *E-pipeline* explores *intra-elasticity* and *inter-elasticity* for multiple hardware/software pipelines operating in parallel on a chip. The *manager* cores monitor the execution of each pipeline, change task assignments of *workers* to adapt to workload variations and meet throughput constraints, reuse cores between pipelines by the use of a *sleep core pool* and set unnecessary cores to sleep. In experiments on a platform of 48 cores, the results show that *E-pipeline* can meet the throughput constraints for pipelines with fine-grained workload variations, achieve the same power efficiency as reference designs with clock gating, and save 37.7% in the use of cores for a variety of benchmarks, with roughly 2μs adaptation overhead.

REFERENCES

[1] S. Borkar, "Thousand core chips: A technology perspective," ser. DAC '07, pp. 746–749.

[2] X. Zhang and A. Louri, "A multilayer nanophotonic interconnection network for on-chip many-core communications," ser. DAC '10, pp. 156–161.

[3] J. Jahn, S. Pagani, S. Kobbe, J.-J. Chen, and J. Henkel, "Optimizations for configuring and mapping software pipelines in many core systems," in *DAC 2013*, pp. 130:1–130:8.

[4] H. Javaid, M. Shafique, J. Henkel, and S. Parameswaran, "System-level application-aware dynamic power management in adaptive pipelined mpsocs for multimedia," in *ICCAD*, Nov 2011, pp. 616–623.

[5] J. Jahn and J. Henkel, "Pipelets: Self-organizing software pipelines for many-core architectures," in *Date 2013*, pp. 1516–1521.

[6] Y. Choi, C.-H. Li, D. D. Silva, A. Bivens, and E. Schenfeld, "Adaptive task duplication using on-line bottleneck detection for streaming applications," in *Proceedings of the 9th conference on Computing Frontiers*, ser. CF '12, 2012, pp. 163–172.

[7] A. Alimonda, S. Carta, A. Acquaviva, A. Pisano, and L. Benini, "A Feedback-based Approach to Dvfs in Data-Flow Applications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pp. 1691–1704, 2009.

[8] "Omitted for blind review."

[9] M. Ruggiero, A. Acquaviva, D. Bertozzi, and L. Benini, "Application-specific power-aware workload allocation for voltage scalable mpsoc platforms," in *2005, ICCD*, 2005, pp. 87–93.

[10] F. Bouwens, M. Berekovic, B. De Sutter, and G. Gaydadjiev, "Architecture Enhancements for the ADRES Coarse-Grained Reconfigurable Array," in *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, ser. HiPEAC'08. Springer-Verlag, 2008, pp. 66–81.

[11] J. T. Zhai, M. A. Bamakhrama, and T. Stefanov, "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems," in *the 50th DAC*, 2013, pp. 170:1–170:8.

[12] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 370–380. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669160

[13] L. Zhang, J. A. Ambrose, J. Peddersen, S. Parameswaran, R. Ragel, S. Radhakrishnan, and K. K. Saluja, "Drma: Dynamically reconfigurable mpsoc architecture," ser. GLSVLSI '13, pp. 239–244.

[14] P. Bellasi, G. Massari, and W. Fornaciari, "A Rtrm Proposal for Multi/many-core Platforms and Reconfigurable Applications," July 2012, pp. 1–8.

[15] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," ser. CASES '12, 2012, pp. 71–80.

[16] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, "Smart hill climbing for agile dynamic mapping in many-core systems," ser. DAC '13, 2013, pp. 39:1–39:6.

[17] P. Bogdan, R. Marculescu, S. Jain, and R. Gavila, "An Optimal Control approach to power management for multi-voltage and frequency islands multiprocessor platforms under highly variable workloads," in *IEEE/ACM International Symposium on Networks on Chip (NoCS)*, 2012, pp. 35–42.

[18] T. da Rosa, V. Larrea, N. Calazans, and F. Moraes, "Power consumption reduction in mpsocs through dfs," in *25th Integrated Circuits and Systems Design (SBCCI)*, 2012, pp. 1–6.

[19] J. R. Wernsing and G. Stitt, "Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *LCTES '10*, 2010, pp. 115–124.

[20] J. Teich, A. Weichslgartner, B. Oechslein, and W. Schrder-Preikschat, "Invasive computing - concepts and overheads," in *2012 FDL*, 2012, pp. 217–224.

[21] "Xtensa Customizable Processor," 2012-06-24.

[22] G. Martin, "The configurable processor view of platform provision," in *Platform Based Design at the Electronic System Level*. Springer Netherlands, 2006, pp. 59–70.