StreaMorph: A Case for Synthesizing Energy-Efficient Adaptive Programs Using High-Level Abstractions

Dai Bui
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-143
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-143.html

August 12, 2013

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards \#0720882 (CSR-EHS: PRET), \#0931843 (CPS: Large: ActionWebs), and \#1035672 (CPS: Medium: Ptides)), the Naval Research Laboratory (NRL \#N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota.

StreaMorph: A Case for Synthesizing Energy-Efficient Adaptive Programs Using High-Level Abstractions

Dai Bui
University of California, Berkeley
daib@eecs.berkeley.edu

Edward A. Lee
University of California, Berkeley
eal@eecs.berkeley.edu

Abstract

This paper presents the concept of *adaptive programs*, whose computation and communication structures can *morph* to adapt to environmental and demand changes to save energy and computing resources. In this approach, programmers write one single program using a language at a higher level of abstraction. The compiler will exploit the properties of the abstractions to generate an adaptive program that is able to adjust computation and communication structures to environmental and demand changes.

We develop a technique, called StreaMorph, that exploits the properties of stream programs' Synchronous Dataflow (SDF) programming model to enable runtime stream graph transformation. The StreaMorph technique can be used to optimize memory usage and to adjust core utilization leading to energy reduction by turning off idle cores or reducing operating frequencies. The main challenge for such a runtime transformation is to maintain consistent program states by copying states between different stream graph structures, because a stream program optimized for different numbers of cores often has different sets of filters and inter-filter channels. We propose an analysis that helps simplify program state copying processes by minimizing copying of states based on the properties of the SDF model.

Finally, we implement the StreaMorph method in the StreamIt compiler. Our experiments on the Intel Xeon E5450 show that using StreaMorph to minimize the number of cores used from eight cores to one core, e.g. when streaming rates become lower, can reduce energy consumption by 76.33% on average. Using StreaMorph to spread workload from four cores to six or seven cores, e.g. when more cores become available, to reduce operating frequencies, can lead to 10% energy reduction. In addition, StreaMorph can lead to a buffer size reduction of 82.58% in comparison with a straightforward inter-core filter migration technique when switching from using eight cores to one core.

1. Introduction

Real-time streaming of media data is growing in popularity. This includes both capture and processing of real-time video and audio, and delivery of video and audio from servers; recent usage number shows over 800 million unique users that visit YouTube to watch over 3 billion hours of video each month [1]. As a result, YouTube data centers consume huge amounts of energy. Given that traffic from mobile devices to YouTube tripled in 2011, energy-efficient stream computing is increasingly important, especially since battery life is a big concern for mobile devices. In addition, as computing devices are often now equipped with high-resolution displays and high-quality speakers, streaming quality is expected to rise accordingly. Processing high-quality streams requires proportionally higher computing energy. While several efforts [15, 22] aim at using parallel computing to meet the demand of stream computation, not much attention has been paid to energy-efficient parallel stream computing. These efforts mainly focus on devising scalable compilation techniques to speed up stream computation. While multicore can address the rising computing demand for stream computation, it also can result in an unnecessary waste of energy due to low core utilization when more cores are used than needed at low streaming rates.

In this paper, we present StreaMorph, a programming methodology that exploits the high-level abstractions of stream programs to generate runtime adaptive ones for energy-efficient executions. Based on one single specification of a stream program, the compiler exploits domain-specific knowledge to generate an adaptive program that is able to adjust its occupied computing resources, processors, memory and so on, at runtime. Such adaptive programs help reduce energy consumption by adapting their computing power to demand changes at runtime. This programming methodology comes from our new perspective on stream programs. The previous work on stream compilation assumes stream programs are isolated entities. This assumption misses an important optimization opportunity. As isolated entities, stream programs are set to run as fast as possible by using as much resources as possible. Running stream programs aggressively may result in a considerable waste of energy. In contrast, we treat stream programs as integrated components within certain systems. For example, an MPEG2 encoder/decoder does not need to run faster than its camera/display frame rates. Digital signal processing (DSP) programs, such as Radar, FM radio, and Vocoders, do not need to run faster than their input signals' sampling rates. Because using more cores than needed for certain input rates to run these programs will waste energy due to static energy leakage and inter-core data communication overhead, it is beneficial to find the minimal number of cores required for a specific workload rate to reduce energy consumption.

Determining the optimal number of cores used for a stream program at compile-time is often not possible due to several factors. These factors include varied processor speed used to run stream programs, users' control on stream quality and speed, varied runtime signal rates, etc. As a result, compiled stream applications are necessary to be able to adapt computing speed accordingly to external changes at runtime to save energy. Expressing stream programs as synchronous dataflow (SDF) [24] models presents an opportunity for such an energy optimization. In SDF, a program is decomposed into multiple autonomous filters connected using FIFO channels. As a result, stream programs can dynamically relocate filters and channels between cores at runtime to minimize the number of cores used while maintaining required throughput demands. Using fewer cores reduces leakage energy as well as inter-core communication energy overhead.

A prior inter-core filter migration technique, the one implemented in Flextream [17] by Hormati et al. in the context of changing processor availability, can adjust the number of cores used, but it is not optimal as it does not adjust memory usage accordingly. In cloud computing as well as in multicore embedded systems, processors are not the only resource that applications share. In these computing environments, it is also desirable to reduce the memory usage of each application to improve the overall utilization of a whole system [11, 12].

To improve inter-core filter migration technique, our StreaMorph technique not only migrates filters between cores but also *transforms* stream graphs, thereby adjusting the set of filters, inter-filter channels and the number of cores used. Runtime stream graph transformation entails mapping and transferring data on inter-filter channels of different stream graph configurations. This process is non-trivial because the inter-filer channel states become complicated when stream programs execute through several stages such as initialization and software pipelining [14].

We tackle the problem by proposing an analysis that helps reduce the number of tokens copied between optimized configurations of a single stream program. The main idea of the proposed analysis is to derive sequences of filter executions to drain tokens on channels as much as possible. This StreaMorph scheme helps optimize energy consumption in several ways. Either minimizing the number of cores used at a fixed frequency or lowering operating voltages and frequencies of processors by using more cores to handle the same workload rates can help reduce energy consumption. In addition, high processor temperatures due to running at high utilization for long periods can degrade processors' performance and lifespan [6, 13]. As a result, increasing the number of cores used to lower core utilization, thereby reducing processor temperatures, can mitigate the problem.

We apply the StreaMorph scheme to a set of streaming benchmarks [15] on the Intel Xeon E5450 processors. The results show that, on average, using StreaMorph to minimize the number of cores used whenever possible from eight cores to four reduces energy consumption by 29.90%, and from eight cores to one reduces energy consumption by 76.33% on average. Using StreaMorph to spread workload on more cores when they become available from four cores to six or seven cores and applying DVFS can reduce energy consumption by 10%. Our StreaMorph scheme also leads to an 82.58% buffer size reduction when switching from an eight-core configuration to one compared to Flextream.

This paper makes the following contributions

- We present the concept of adaptive programs that are beneficial in cloud computing and embedded systems. Our work demonstrates a case for employing high-level abstractions to design programs that adapt to demand and resource changes at runtime.
- We identify an energy optimization opportunity by taking into account external settings, e.g. input data rates or the number of available cores, of stream programs with a task-level program transformation technique.
- We present an analysis that helps simplify program state copying processes when switching between configurations.
- We implement the method in the StreamIt compiler and demonstrate experimentally the effectiveness of the method using a set of StreamIt benchmarks.
- We also derive an approximate energy model analysis for stream programs on multicore and experimentally validate the model. The model can serve as a guidance for finding optimal energy-efficient configurations.

2. Adaptive Stream Programs

Stream programs are often compiled to maximize speed [15, 22] assuming fixed numbers of cores. This approach implicitly assumes that sources/sinks of data for stream programs can always produce/consume data. This assumption is not often true in practice. For example, an audio source cannot produce data faster than its sampling rate. A sink, e.g. a monitor, does not consume faster than its designated frame rate. To understand this situation better, we discuss where stream programs fit into external settings.

Figure 1 shows a general structure of DSP systems. In the figure, the analog-to-digital converter (ADC) samples analog signals at a sampling frequency f and outputs digital signals to the DSP module, which executes stream programs. The DSP module manipulates digital signals and subsequently feeds processed digital signals to the digital-to-analog converter (DAC). The DAC converts input digital signals to analog signals at a conversion frequency g.

This general structure of DSP systems suggests DSP modules need not process more slowly than sampling f and/or conversion g frequencies. It is generally not possible to get ahead of the source of data, when that source is a real-time source, and it is generally not necessary to get ahead of the sink. Even with real-time sources and sinks, there may be some benefit to getting ahead, because results can be buffered, making interruptions less likely in the future when there is contention for computing resources. However, getting ahead produces no *evident* benefit unless such contention occurs, and it comes at an energy cost. When both the source and the sink are operating in real time, it also comes at a cost in latency. Having more cores than necessary can waste energy due to static energy leakage when idle. In addition, inter-core data transfer overhead when using more cores than needed can be another source of energy inefficiency. Even when buffering is used, in steady state, it is beneficial for processing speed to match IO rates, otherwise buffers will overflow or underflow.

We identify two scenarios that can lead to the mismatch between IO rates and processing speed.

- **Varied IO rates at runtime:** Users can use fast forward functionality to quickly browse through a video or audio file. Graphic applications need to render at faster/slower frame rates. When users want to increase/decrease songs' tempos in Karaoke systems, sound synthesis engines have to adjust accordingly. Sampling rates are increased/decreased to alter quality.
- **Different processor speeds:** Even though compilers can match between processing speed and certain IO rates at compile time, this optimization can only be done for specific processor models. Processors of the same instruction set architecture can vary in several dimensions such as clocking frequencies, microarchitecture, fabrication processes. These variations lead to wildly varied execution times of a single program on different processors with the same instruction set.

2.1 Energy Optimization

To get the energy benefits of matching processing speed to IO rates, two common techniques are: 1) To consolidate tasks by relocating tasks to fewer cores to improve core utilization and enable turning off idle cores [2, 31]; 2) To vary processing speed by applying dynamic voltage and frequency scaling(DVFS) [3, 9, 16, 35] or processor composition [8, 21] or heterogeneous processors of different speeds [23, 26, 36].

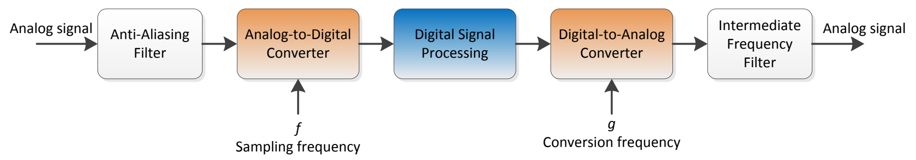

Analog signal
Anti-Aliasing Filter
Analog-to-Digital Converter
Digital Signal Processing
Digital-to-Analog Converter
Intermediate Frequency Filter
Analog signal
f
Sampling frequency
g
Conversion frequency

Figure 1. Basic digital signal processing structure.

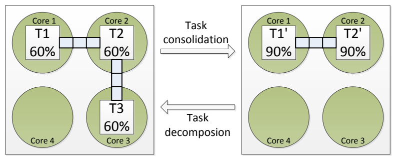


Figure 2. Task consolidation and decomposition.

2.1.1 Task Consolidation

Especially for stream programs, task consolidation can help reduce inter-core communication overhead. However, applying the task consolidation technique to stream programs optimized for speed is not straightforward. For example, in the left of Figure 2, a stream program is composed of three tasks `T1`, `T2` and `T3` connected through two FIFO channels. Suppose that each task resides on a separate core in a quad-core processor and utilizes 60% of its core at current IO rates. Relocating normally one of the three tasks to another occupied core will overload the core. Consequently, the processing system cannot guarantee the program's required IO rates. Instead, the program should switch to a different configuration composed of two tasks `T1'` and `T2'`; each utilizes 90% of its residing core as in the right of Figure 2.

This configuration switching is complicated because the two configurations may be composed of different sets of filters and channels due to speed optimization techniques that depend on the number of cores used. For example, the configuration composed of `T1`, `T2` and `T3` is optimized for 3 cores, while the configuration composed of `T1'` and `T2'` is optimized for 2 cores.

2.1.2 Task Decomposition

Task consolidation is necessary as discussed in the previous section, however, there are a number of scenarios where it is useful to decompose tasks.

- Suppose that running `T1'` and `T2'` on cores 1 and 2 at 90% utilization drastically increases those cores' temperatures [13, 25]. At this point, we want to run 2 tasks on 3 cores to reduce utilization as in Figure 2, and thereby consequently reducing the cores' temperatures.
- If the processors posses a DVFS capability, and there are more cores available because some other applications terminated, decomposing two tasks `T1'` and `T2'` into three tasks `T1`, `T2` and `T3` and reducing the operating frequencies of the processors can lead to energy reduction; power is proportional to the cube of frequency.

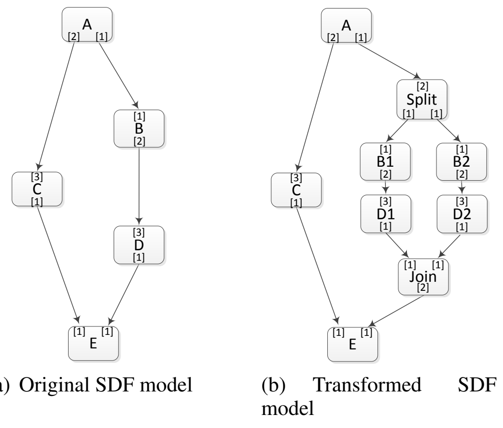
2.2 Adjusting Shared Resources Proportionally

Section 2.1 suggests that inter-core filter migration to adjust processor utilization and speed can help reduce energy consumption. While the inter-core filter migration implemented in Flextream [17] can adjust the number of cores used at runtime, it is not optimal, because the technique does not reduce memory usage accordingly. This limitation reduces the value for cloud computing, where applications are allocated cores, memory, networking bandwidth, and so on, *proportionally* [11, 12]; for example, one application using two cores is allocated 4GB of memory while another application using four cores is allocated 8GB. As a consequence, just adjusting the number of cores used by relocating filters between cores without adjusting memory usage as in Flextream [17] is no longer sufficient. In particular, reducing memory usage is crucial in embedded systems where memory is scarce.

In contrast to Flextream [17], our StreaMorph technique adjusts both the number of cores used and memory usage simultaneously. We illustrate the difference using an example. To speed up stream computation, stateless filters are often replicated as in Figure 3(b) to utilize available cores, where filters $\mathtt{B}_1$, $\mathtt{B}_2$, $\mathtt{D}_1$ and $\mathtt{D}_2$ are duplicated from the respective ones in the original stream graph in Figure 3(a). Duplicating filters is necessary when stream IO rates are high and we need to use more cores to handle such high IO rates. When IO rates become low, a Flextream inter-core filter migration technique retains the stream graphs in Figure 3(b) and relocates filters to reduce the number of core used. For example, originally the application in Figure 3(b) runs on two cores, but now the input rate is lowered and one core can handle the workload. In this case, the filters are relocated in Figure 3(b) and the stream graph is not modified. In contrast, our StreaMorph technique modifies the stream graph into the one in Figure 3(a). As a result, our technique would be more memory-efficient because it eliminates the buffers between filters `Split`, $\mathtt{B}_2$, $\mathtt{D}_2$, and `Join`. In addition, we will show that our technique also reduces the buffer sizes of the other buffers.

3. StreaMorph: Designing Adaptive Programs with High-Level Abstractions

Section 2 suggests that adapting stream programs to external changes can lead to energy and resource reduction. In this section, we will show how to attain such adaptivity by exploiting stream program abstractions.



(a) Original SDF model (b) Transformed SDF model

Figure 3. Stream graph transformation

3.1 Stream Abstractions and Data Parallelism

3.1.1 Model of Computation

The SDF [24] model of computation is often used to model executions of stream programs [33]. This abstraction enables several compiler optimization techniques [15, 22] for buffer space, scheduling and mapping to underlying architecture. In the SDF model of computation, a stream program is given as a graph composed of a set of actors, called filters in this paper, communicating through FIFO channels. Each filter has a set of input and output ports. Each channel connects an output port of a filter to an input port of another actor. Each filter consumes/produces a fixed number of tokens from each of its input/output port each time it executes.

Figure 3(a) shows an example of an SDF stream graph. Filter `A` has two output ports. Each time `A` executes, it produces 2 tokens on its left port and 1 token on its right port. Filter `B` consumes 1 and produces 2 tokens each time it executes, and so on. The theory of the SDF programming model provides the algorithm to compute the number of times each actor has to execute within one *iteration*, of the *whole* stream graph, so that the total number of tokens produced on each channel between two actors is equal to the total number of tokens consumed. In other words, the number of tokens on each channel between actors remains unchanged after one iteration of the whole stream graph. For example, in one iteration of the stream graph in Figure 3(a), filters `A`, `B`, `C`, `D`, `E` have to execute 3, 3, 2, 2, 2 times respectively. Repeating this basic schedule makes the number of tokens on each channel remain the same after one iteration of the whole stream graph. For instance, in the channel between `B` and `D`, in one iteration, `B` produces 3×2 tokens while `D` consumes 2×3 tokens.

3.1.2 Sliding Windows

The StreamIt language [33] is a language for writing stream applications extending the SDF model of computation with a sliding window feature, in which filters can *peek* (read) tokens ahead without consuming those tokens. As a result, many states of programs are stored on channels in the form of data tokens instead of being stored internally within filters. Consequently, many filters become *stateless* and eligible for the filter replication technique to speed up computation as in Section 3.1.3. As many states of programs are stored on channels, it is easier for compilers to migrate states between different configurations during the morphing process.

3.1.3 Data Parallelism Exposed by Stream Abstractions

In this section, we will discuss how the stream abstractions in the previous section can help adjusting parallelism in stream programs, thereby adjusting computing speed. To speed up a stream application, we can replicate *stateless* filters so that multiple instances of one stateless filter execute in parallel. This filter replication technique is feasible because each filter in an SDF application consumes/produces a known numbers of tokens whenever it executes at each of its input/output port. As a result, after replicating filters, the compiler knows how to distribute/merge data tokens to/from each replicated filter.

Let us take the example in Figure 3(a) to illustrate the problem. Suppose that `B` and `D` are stateless, as a result, we can duplicate the filters to obtain the configuration in Figure 3(b). Because `B` consumes one token and `D` produces one token each time they execute, we can distribute data tokens to each duplicated `B` and collect data tokens from each `D` evenly in a round-robin fashion using `Split` and `Join` filters in Figure 3(b). With this duplication, we can get 2x speed-up for the computation of the two filters. Stateless filter replication is a popular technique that has proved to achieve significant speed-up for several StreamIt benchmarks [15, 22]. Stateless filters can be replicated many times to fill-up all available cores, consequently, this technique is dependent on the number of cores used.

Execution Scaling: The filter replication technique may require changing the number of times each filter executes within one iteration. For example, `A`, `C`, and `E` in Figure 3(b) now execute 6, 4, and 4 times respectively in one iteration. This means those filters now execute twice as often within one iteration. This effect is called execution scaling.

3.2 Implementing Adaptive Stream Programs

In the previous section, we discussed how the stream abstractions can help improving stream program speed by exploiting data parallelism. However, switching between configurations optimized for different numbers of processors is complicated by several compilation techniques applied to stream programs [14]. We will show how to mitigate the complication in this section.

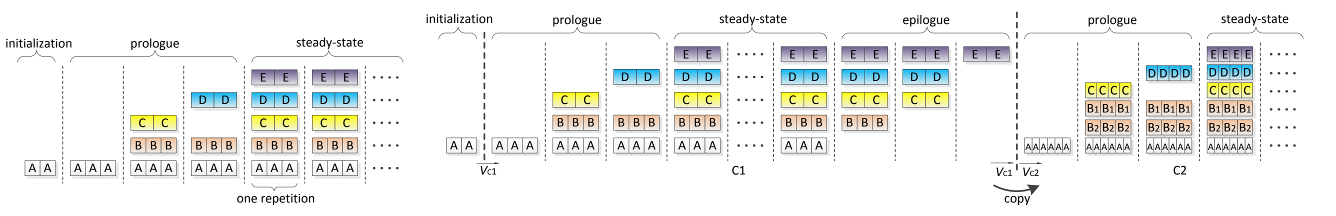
3.2.1 Execution Stages

Initialization Stage: As downstream filters can peek tokens without consuming them, upstream filters have to execute a number of times initially in the *initialization* stage to supply more tokens to downstream peeking filters. For example, suppose that filter `B` in Figure 3(a), each time it executes, peeks ahead 3 tokens and consumes only the first one of the 3. Consequently, it requires at least 2 additional tokens on the channel between filters `A` and `B`. To satisfy this requirement, `A` has to execute twice in the initialization stage to load the channel with two tokens.

Software Pipelining: Gordon et al. use software pipelining to address a drawback of hardware pipelining [15]. In hardware pipelining, only contiguous filters should be mapped onto one core, e.g. in Figure 3(a), filters `A` and `D` should not be mapped into a core when filter `B` is mapped to another core. This mapping restriction can potentially lead to unbalanced allocation. Software pipelining allows mapping any filters to any core, e.g. filters `A` and `D` can be mapped into a core even when filter `B` is mapped to another core. This is done by constructing a *loop prologue* in which each filter executes a certain number of times to make data tokens available for downstream filters to run in one iteration. In other words, within one iteration, filters do not communicate directly to each other, instead, they output tokens into buffers that will be read by downstream filters in the next iteration. Figure 4(a) shows how the prologue stage is executed and its transition to the steady-state stage. In the steady-state stage, filters execute one stream graph iteration forming a steady-state *repetition*. Each filter is able to execute completely independently within one steady state repetition without the need for waiting for upstream filters to produce data in that repetition. For example, for the model in Figure 3(a), in order for `E` to execute 2 times independently, `C` and `D` have to execute twice. Consequently, `B` and `A` have to execute 3 times. Similarly, for `C` and `D` to execute twice independently, `B` and `A` have to execute 3 times. Finally, for `B` to execute 3 times independently, `A` has to execute 3 times. Adding up, in the prologue stage, `A` executes (3+3+3) = 9 times, `B` executes (3+3)=6 times, `C` and `D` execute 2 times. After the loop prologue `A`, `B`, `C`, `D`, `E` can execute independently within one steady-state repetition.

3.2.2 Deriving Reverse Sequences

Normally, a compiled program repeatedly executes its steady-state repetitions regardless of external environment changes as in Figure 4(a). This conventional execution model may no longer be efficient in cloud computing, mobile computing and cyber-physical systems where applications run in various environments or have to interact with physical environments. For example, let us consider the situation when there are more cores becoming available in the system because some other applications terminated. One possible way to save energy is to utilize the newly available cores to process a part of the workload of the running application at lower frequencies to save energy. Suppose utilizing the available cores would require the application to switch from configuration $\mathcal{C}_1$ in Figure 3(a) to the configuration $\mathcal{C}_2$ in Figure 3(b). This transition takes place at the end of a repetition in the $\mathcal{C}_1$'s steady-state stage. Note that, in software pipelining, the prologue stage fills the channels of $\mathcal{C}_1$ with tokens, for example, the channel between `B` and `D` contains 6



(a) Conventional execution stages.

(b) Configuration switching strategy.

Figure 4. Execution strategies for stream programs

tokens; the channel between D and E contains 2 tokens. As B and D are duplicated, determining how to copy and distribute those tokens on the corresponding channels in Figure 3(b) is complicated. It is even more problematic to derive from such a state of $\mathcal{C}_2$ a sequence of filter executions such that, after executing such a sequence, filters in Figure 3(b) can execute in a software pipelining fashion. Deriving such a switching procedure for each pair of configurations may be costly.

We simplify the switching process using the strategy in Figure 4(b). Instead of deriving complicated inter-configuration token copying procedures, we use an *epilogue* stage to reduce the number of copied tokens. The epilogue stage is derived to undo the effect of the prologue stage. As a result, we only need to copy the fixed and known number of tokens across the configurations produced by the initialization stage. If there is no peeking filter, only initial tokens and states of stateful filters may still require copying.

DEFINITION 1. *Sequences of executions that undo the effect of the program's current configuration's prologue stage are called reverse sequences.*

Let $\vec{V}_{\mathcal{C}}$ be the vector of the number of tokens on each channel of configuration $\mathcal{C}$ right before its prologue stage.

DEFINITION 2. *States of a configuration $\mathcal{C}$, whose vectors of the numbers of tokens on the channels are equal to $\vec{V}_{\mathcal{C}}$, are called $\mathcal{C}$'s pre-prologue states.*

Note that, for any configuration $\mathcal{C}$, from a pre-prologue state, if each filter executes the same number of iterations, $\mathcal{C}$ is again in a pre-prologue state. As a result, a reverse sequence can be derived by finding $d_{\mathtt{A}}$, $\forall \mathtt{A} \in \mathcal{C}$, the additional number of times each filter has to execute more so that all the filters will have executed the same number of iterations. Suppose that number of iterations is J, counting from $\mathcal{C}$'s first pre-prologue state (right before executing $\mathcal{C}$'s first prologue stage). Suppose that $\mathcal{C}$ has n filters $\mathtt{A}_i$ where $i = 1, \ldots, n$, and each filter $\mathtt{A}_i$ executes $p_{\mathtt{A}_i}$ and $s_{\mathtt{A}_i}$ times in its prologue stage and one iteration respectively. We have:

$$p_{\mathtt{A}_i} + XI\, s_{\mathtt{A}_i} + d_{\mathtt{A}_i} = J s_{\mathtt{A}_i} \;\forall \mathtt{A}_i \in \mathcal{C} \tag{1}$$

where I is the current repetition of the steady-state stage and X is the execution scaling factor. From equation (1), since X, I , and J are all integers, $p_{\mathtt{A}_i} + d_{\mathtt{A}_i}$ has to divide $s_{\mathtt{A}_i}$. Let $\alpha_{\mathtt{A}_i}$ be an integer such that $p_{\mathtt{A}_i} + d_{\mathtt{A}_i} = \alpha_{\mathtt{A}_i} s_{Ai}$. Equation (1) becomes:

$$\begin{aligned} \alpha_{\mathtt{A}_i} s_{\mathtt{A}_i} + XI\, s_{\mathtt{A}_i} &= J s_{\mathtt{A}_i} \\ \Leftrightarrow \alpha_{\mathtt{A}_i} &= J - XI \end{aligned} \tag{2}$$

Note that equation (2) holds $\forall \mathtt{A}_i \in \mathcal{C}$. Expanding the equation for all actors in the configuration $\mathcal{C}$, we arrive at: $\alpha_{\mathtt{A}_1} = \alpha_{\mathtt{A}_2} = \ldots = \alpha_{\mathtt{A}_n}$. Because $\alpha_{\mathtt{A}_i} = \frac{p_{\mathtt{A}_i} + d_{\mathtt{A}_i}}{s_{Ai}} = \frac{d_{\mathtt{A}_i} + (p_{\mathtt{A}_i} \bmod s_{Ai})}{s_{Ai}} + \lfloor \frac{p_{\mathtt{A}_i}}{s_{A1}} \rfloor$, then:

$$\frac{d_{\mathtt{A}_1} + (p_{\mathtt{A}_1} \bmod s_{A1})}{s_{A1}} + \lfloor \frac{p_{\mathtt{A}_1}}{s_{A1}} \rfloor = \ldots = \frac{d_{\mathtt{A}_n} + (p_{\mathtt{A}_n} \bmod s_{An})}{s_{An}} + \lfloor \frac{p_{\mathtt{A}_n}}{s_{An}} \rfloor \tag{3}$$

As it is desirable to switch between configurations as soon as possible, we find the smallest $d_{\mathtt{A}_i} \geq 0$ satisfying equation (3) by finding the j such that:

$$j = \underset{i \in [1..n]}{\operatorname{argmax}} \left(\frac{p_{\mathtt{A}_i} \bmod s_{Ai}}{s_{\mathtt{A}_i}} + \lfloor \frac{p_{\mathtt{A}_i}}{s_{Ai}} \rfloor \right) \tag{4}$$

Because $\alpha_{\mathtt{A}_i}$ is an integer and $\alpha_{\mathtt{A}_i} = \frac{d_{\mathtt{A}_1} + (p_{\mathtt{A}_1} \bmod s_{A1})}{s_{A1}} + \lfloor \frac{p_{\mathtt{A}_1}}{s_{A1}} \rfloor$, we can conclude that $(d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})) \bmod s_{Aj} \equiv 0$. Hence,

- If $p_{\mathtt{A}_j} \bmod s_{Aj} = 0$, we find the smallest $d_{\mathtt{A}_j} = 0$.
- If $p_{\mathtt{A}_j} \bmod s_{Aj} > 0$, we find the smallest $d_{\mathtt{A}_j} = s_{Aj} - (p_{\mathtt{A}_1} \bmod s_{A1})$. It is easy to prove that $d_{\mathtt{A}_j} \geq 0$.

Now we derive other $d_{\mathtt{A}_i}$ from equation (3) as follows:

$$\begin{aligned} d_{\mathtt{A}_i} &= \left(\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}} + \lfloor \frac{p_{\mathtt{A}_j}}{s_{Aj}} \rfloor - \lfloor \frac{p_{\mathtt{A}_i}}{s_{Ai}} \rfloor \right) s_{Ai} - (p_{Ai} \bmod s_{Ai}) \quad (5) \\ &= \left(\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}} + \lfloor \frac{p_{\mathtt{A}_j}}{s_{Aj}} \rfloor \right) s_{Ai} - p_{Ai} \quad (6) \end{aligned}$$

As $\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}}$ is equal to either 0 or 1, from equation (6), it is easy to prove that $d_{\mathtt{A}_i}$ is an integer. Now we need to prove $d_{Ai} \geq 0 \;\forall i = 1, \ldots, n$. From (3) and (4):

$$\lfloor \frac{p_{\mathtt{A}_j}}{s_{Aj}} \rfloor - \lfloor \frac{p_{\mathtt{A}_i}}{s_{Ai}} \rfloor \geq \frac{p_{\mathtt{A}_i} \bmod s_{Ai}}{s_{\mathtt{A}_i}} - \frac{p_{\mathtt{A}_j} \bmod s_{Aj}}{s_{\mathtt{A}_j}} \tag{7}$$

Plugging into equation (5), we arrive at:

$$\begin{aligned} d_{\mathtt{A}_i} &\geq \left(\frac{d_{\mathtt{A}_j} + (p_{\mathtt{A}_j} \bmod s_{Aj})}{s_{\mathtt{A}_j}} + \frac{p_{\mathtt{A}_i} \bmod s_{Ai}}{s_{\mathtt{A}_i}} - \frac{p_{\mathtt{A}_j} \bmod s_{Aj}}{s_{\mathtt{A}_j}} \right) s_{Ai} - (p_{Ai} \bmod s_{Ai}) \\ &\geq \frac{d_{\mathtt{A}_j} s_{Ai}}{s_{Aj}} \geq 0 \quad (8) \end{aligned}$$

Being able to find d_{Ai} does not necessarily mean that reverse sequences always exist; there may be additional data dependency constraints. Within the SDF literature, even if we can find numbers of times filters execute within one iteration, it is still possible that the stream graph is not schedulable, e.g. when the stream graph has loops forming circular dependencies. The following theorem implies that it is always possible to undo the effect of prologue stages.

THEOREM 1. *A reverse sequence always exists for a configuration $\mathcal{C}$ if the configuration has prologue and steady-state schedules (an executed stream graph).*

Proof: We prove this by contradiction. Suppose that we cannot derive a concrete reverse sequence based on the values d_{Ai} found as above. The only reason would be because of data dependency between the executions of the filters. Let $\mathtt{A}_i^e$ denote the e^{th} execution of filter $\mathtt{A}_i$ from the beginning right before the prologue stage. There exist two possible cases:

1) There exists a data dependency loop between filter executions $\mathtt{A}_{j1}^{e_{j1}} \prec \mathtt{A}_{j2}^{e_{j2}} \prec \ldots \prec \mathtt{A}_{jm}^{e_{jm}} \prec \mathtt{A}_{j1}^{e_{j1}}$ of m filters. As the

existence of this dependency loop depends solely on the property of $\mathcal{C}$, consequently, the execution of $\mathcal{C}$ will be stalled due to the dependency loop. This contradicts with the fact that $\mathcal{C}$ can be repeated in the steady-state stage forever.

2) Some filter B is at its b^{th} execution and has not completed d_B executions of its epilogue stage to reach its execution $(Js_B)^{\text{th}}$; in other words, $b < Js_B$. B cannot proceed further because it requires more data tokens from its upstream filter A, while A has completed d_A executions in its epilogue stage to reach its $(Js_A)^{\text{th}}$ execution. This implies that A, having executed J iterations, still does not produce enough tokens for B to execute J iterations. This contradicts the property of SDF that the number of tokens A produces in one iteration is equal to the number of tokens B consumes in one iteration. □

We now need to derive concrete reverse sequences based on d_{Ai}. It is desirable to use current buffers without increasing buffer sizes for executing reverse sequences. As each buffer is large enough to store tokens from the upstream filter for the downstream filter to execute at least one iteration even if the producing filter has not produced any more tokens, it is safe to execute upstream filters for at most one iteration continuously. After that, downstream filters have to execute to free up buffers. Algorithm 1 derives reverse sequences when stream graphs do not contain loops. However, most of the stream benchmarks do not contain loops [33] and all the benchmarks used in [15, 22] are loop-free. When stream graphs have loops, the classical symbolic execution method of SDF [24] can be used. The classical method may yield more complicated sequences as it randomly selects filters to execute. This random execution ordering can cause severe performance degradation during configuration switching due to losing cache locality. To mitigate the potential negative effect of the classical symbolic execution method, the following algorithm seeks to execute each filter several times in a row by traversing filters in dataflow order.

```
Data: $reverseMap$: Map from filters to numbers of reverse executions
$schedule$: containing prologue and steady-state schedules
Result: $reverseSeq$: reverse sequence of filter executions
$reverseSeq \leftarrow$ new List()
$filters \leftarrow$ getFiltersInDataflowOrder()
while $reverseMap.size() > 0$ do
    $thisStepMap \leftarrow$ new Map()
    for $f \in filters$ do
        $nReverseExes \leftarrow reverseMap$.get($f$)
        $nIterExes \leftarrow schedule$.getNumberExesInOneIteration()
        $m = \min(nReverseExes, nIterExes)$
        $thisStepMap$.put($f$, $m$)
        $nReverseExesLeft \leftarrow nReverseExes - m$
        if $nReverseExesLeft > 0$ then
            $reverseMap$.put($f$, $nReverseExesLeft$)
        else
            $reverseMap$.remove($f$)
        end
    end
    $reverseSeq$.add($thisStepMap$)
end
return $reverseSeq$
```

Algorithm 1: Deriving reverse execution sequences when stream graphs are loop-free.

3.2.3 Illustrative Example

We illustrate the analysis method using the configuration in Figure 3(a). For the program: p_A=9,p_B=6, p_C=p_D=2, p_E=0; s_A=s_B=3, s_C=s_D=s_E=2; X= 1. As $\frac{p_A \mod s_A}{s_A} + \lfloor \frac{p_A}{s_A} \rfloor$=3 is max and $p_A \mod s_A \equiv 0$, we set $d_A = 0$. Finally, from (6), we find d_B=3, d_C=d_D=4, d_E=6. Readers can verify that condition (1) is satisfied. Now, applying Algorithm 1, we find that, in the first reverse step, B, C, D, E execute 3, 2, 2, 2 times respectively. B has executed all its reverse executions, so it is removed from the $reverseMap$. In the second step, C, D, E execute 2, 2, 2 times respectively, and C, D are removed from the $reverseMap$. In the last step, only E executes 2 times and is removed from the $reverseMap$. Now the $reverseMap$ is empty so the algorithm terminates. Applying the reverse sequence will drain all the tokens on the channels from B to D and from D to E. Figure 4(b) displays how the reverse execution sequence is executed.

3.2.4 Peeking Token Copying

As reverse sequences can undo the effect of prologue stages, if stream programs do not contain peeking filters, then after epilogue stages, the effect of respective prologue states is undone. As a result, we only need to copy filter states, which are small, and initial tokens, which often do not exist. When peeking filters exist, for example, B peeks 3 tokens and only consumes 1 each time it executes, the channel between A and B will always contain tokens after the initialization stage. Copying those peeking tokens requires further elaboration about how stream graphs with peeking filters are optimized.

Efficient Sliding Window Computation: Gordon, in his PhD thesis [14], presents a method to reduce inter-core communication that degrades performance in SMP and Tilera machines for applications with peeking filters. We will use an example to illustrate Gordon's method.

Consider the configuration in Figure 3(b), where filter B is duplicated into B_1 and B_2. Because B peeks tokens, it is necessary to send more tokens to the input channels of replicating filters B_1 and B_2 than the number of tokens B_1 and B_2 consume. A fine-grained data parallelism approach where B_1 and B_2 alternatively consume tokens from A at the step size of 1 will double the amount of communication between A and B as in Figure 5(a). For example, suppose that in one iteration A produces $2n$ tokens on its output channel, and there are 2 more tokens, numbered 1 and 2, produced by A in the initialization stage. B_1 consumes token 1 and reads tokens 2 and 3; B_2 consumes token 2 and reads tokens 3 and 4; B_1 consumes token 3 and reads tokens 4 and 5; and so on. At the end of the iteration in the steady-state stage, B_2 consumes token $2n$ and reads tokens $2n + 1$ and $2n + 2$. As a result, both B_1 and B_2 require $2n + 1$ tokens out of $2n + 2$. The total amount of traffic is therefore $4n + 2$ tokens in comparison with $2n + 2$ tokens for the original stream graph. This communication overhead may degrade performance significantly for applications with peeking filters in SMP and Tilera machines [14].

To reduce the communication overhead, a coarse-grained data parallelism approach [14] is used instead as in Figure 5(b). Now B_1 consumes the first n tokens and reads tokens $n + 1$, $n + 2$; B_2 consumes tokens from $n+1$ to $2n$ and reads tokens $2n+1$ and $2n+2$. As a result, A only needs to send $n + 2$ tokens to both B_1 and B_2 within one iteration of the steady-state stage. The amount of communication is therefore $2n + 4$ tokens in comparison with $2n + 2$ tokens for the original graph. If n is large, the overhead becomes insignificant. As $2n$ is the number of tokens A produces within one iteration, we can scale up the number of times each filter executes within one iteration to increase n. Note that in Figure 5(a)(b) and (c), colored tokens are peeked only and never consumed. This coarse-grained data parallelism approach does not increase output(joining) buffer sizes over the fine-grained one because there are no duplicated produced data. As a result, this coarse-grained data parallelism approach also does not change the real-time behavior of a program as it does not requires additional buffer space.

Copying Peeking Tokens: The StreamIt compiler enforces an additional constraint [14] that the number of tokens on the input channel of a replicated filter right after the initialization stage has to be smaller than the number of tokens the filters will consume within

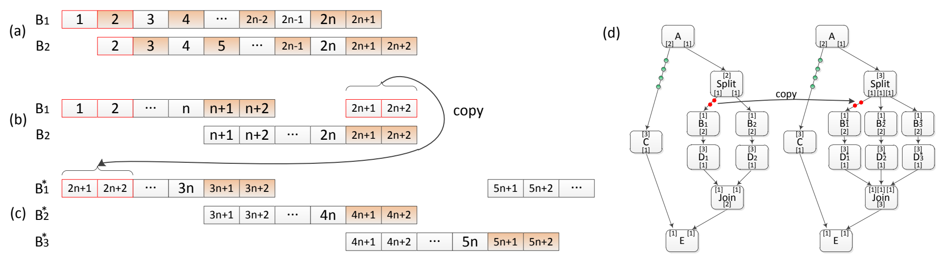


Figure 5. Peeking token copying in sliding window computation. Each cell denotes a token. Colored tokens are peeked only and never consumed. (a) Data token distribution in fined-grained interleaving execution when `B` is replicated by 2; (b) Data token distribution in coarse-grained interleaving execution when `B` is replicated by 2; (c) Data token distribution in coarse-grained interleaving when execution `B` is replicated by 3; (d) Copy peeking tokens when switching configurations.

one iteration. This can be achieved by scaling up the same number of times each filter executes in one iteration. This constraint implies that, in pre-prologue states, only the input channels of the first replicated peeking filters contain tokens regardless of configurations. As a result, after applying reverse sequences to bring a configuration to a pre-prologue state, we only need to copy tokens from the input channel of the first replicated filters in the current configuration to the input channel of the respective first replicated filters in the incoming configuration. For example, only the tokens in the input channel of $\texttt{B}_1$ need to be copied to the input channel of $\texttt{B}_1^*$ as in Figure 5(d). The tokens on the channel between `A` and `C`, due to the executions of `A` in the initialization stage, only need to be copied if the two configurations contain different versions of `A` and `C` or the channel between `A` and `C` needs to be reallocated for a larger buffer size. As the number of peeking filters and the number of peeking tokens are small [33], it would take an insignificant amount of time to copy.

4. Task Decomposition for Low-Power

As discussed in Section 2.1.2, when more cores become available because some other programs terminated, if processors possess DVFS capability, we can reduce energy consumption by decomposing tasks, thereby spreading workload on more cores, and lowering operating voltages and frequencies. We will derive an analysis that helps justify the hypothesis supported by the experimental results in Section 5.2.

We employ the energy model for one CPU from [29]:

$$P_{cpu} = P_{dynamic} + P_{static} = (C_e V_{cpu}^2 f_{cpu}) + (\alpha_1 V_{cpu} + \alpha_2) \quad (9)$$

Based on equation (9), the dynamic energy consumption P_n if computation is spread on n cores is:

$$P_n \approx \left(\sum_{i=1}^{n} C_i \right) V_n^2 f_n \quad (10)$$

If we assume that inter-core communication consumes very little energy in comparison with computational energy or the same as intra-core communication, spreading stream graphs on more cores enables lowering operating frequencies and voltages, although it increases $\sum_{i=1}^{n} C_i$. Suppose that we can run a stream application at a specific IO rate with two configurations of n and m cores, where $n > m$. From equation (10), we arrive at:

$$\frac{P_n}{P_m} \approx \frac{n}{m} \left(\frac{V_n}{V_m} \right)^2 \frac{f_n}{f_m} \quad (11)$$

Note that the n-core configuration is supposed to be as fast as the m-core configuration. As a result, it requires that the number of instructions delivered by n cores in one second be equal to that of m cores: $n f_n IPC_{avg} \approx m f_m IPC_{avg}$, where IPC_{avg} is the average number of instructions per cycle and it should be the same for the two configurations because both configurations run the same workload targeting the same IO rates. Equivalently, $\frac{n}{m} \approx \frac{f_m}{f_n}$. Plugging into (11), we arrive at:

$$\frac{P_n}{P_m} \approx \left(\frac{V_n}{V_m} \right)^2 \quad (12)$$

From [18], frequencies relate to voltages as follows:

$$f \propto \frac{(V_{cpu} - V_t)^\gamma}{V_{cpu}} \quad (13)$$

where V_t is the threshold voltage of transistors and γ depends on the carrier velocity saturation and lies between 1.2 to 1.6 for the current technologies. As a result, running on more cores with lower frequencies can reduce energy consumption. For example, $m < n \Rightarrow f_n < f_m \Rightarrow V_n < V_m \Rightarrow P_n < P_m$.

Note that this is a simplified energy model for stream applications on multicore machines. This model assumes that computation-communication ratios are large enough to approximately ignore inter-core communication energy or inter-core communication consumes the same amounts of energy as intra-core communication. The first assumption depends on benchmarks' characteristics while the second assumption is often not true in practice. This approximation analysis serves as a predictive model to explained the results in Section 5.2. In addition, using more cores would increase overall static power and reduce operating voltages. Because reducing operating voltages lowers static energy consumption as in equation (9), as a consequence, when taking both static and dynamic power into account, we need to make sure that the static power increment due to additional used cores is lower than the static and dynamic power saved by lowering voltage and frequency. A detailed processor power model can help determine optimal configurations.

5. Evaluations

We implement the StreaMorph scheme in the StreamIt compiler [14]. To model input data token streams controlled externally at certain rates, we instrument code of source filters with the token-bucket mechanism [32]. Whenever a source filter wants to send out a number of data tokens, it has to acquire the same number of rate control tokens in a bucket. If the bucket does not have enough rate control tokens, the thread of the source filter will sleep and wait until there are enough rate tokens in the bucket. A timer interrupt periodically fills the bucket with rate control tokens at rate r and wakes up the waiting thread. Our coarse-grain level implementation of this mechanism has a negligible effect on performance.

We run our experiments on a server with two Intel Xeon E5450 quad-core CPUs operating at two frequencies 2GHz and 3GHz. We use a power meter to measure the power consumption of the whole system. The system has a typical idle power consumption $P_{idle} \approx$ 228 watts. For each benchmark, we measure the dynamic power consumption, P_{load}, computed using the following equation:

$$P_{load} = P_{measure} - P_{idle} \quad (14)$$

We use the same set of benchmarks in other papers [15, 22]. Most of the benchmarks are in the DSP domain. Each benchmark is compiled into a program composed of several configurations, where each configuration is specific to a number of cores.

5.1 Energy Reduction by Task Consolidation

In this section, we evaluate the effectiveness of the task consolidation scheme using the StreaMorph technique. Suppose that when the IO rates of a stream program are reduced, the processors become under-utilized. As a result, we can morph the stream graph of the application to minimize the number of cores used to reduce P_{load}; cores are run at 3GHz. Figure 6(a) and Figure 6(b) show the effectiveness of consolidating tasks using StreaMorph to minimize the number of cores used to one and four cores respectively when input rates become low enough. Concretely, morphing from eight cores to four cores reduces energy consumption by 29.90% on average. Morphing from eight cores to one core reduces energy consumption by 76.33% on average.

5.2 Energy Reduction by Task Decomposition

This section demonstrates experimentally the effectiveness of the task decomposition scheme with StreaMorph. When there are more available cores in the system, because some other program terminates, the analysis in Section 4 suggests we should transfer a part of the workload to the newly available cores to lower operating voltages and frequencies of all the cores to save energy. For each benchmark, we use the workload rate that can be handled by 4 cores at 3GHz. We use the StreaMorph technique to switch each application to a new configuration using more cores, such that the new configuration can still handle the same workload rate at a lowered frequency, say 2GHz. Figure 7(a) shows the measured energy consumptions and Figure 7(b) shows the energy reduction percentage gained by task decomposition. On average, the energy reduction percentage is around 10%.

Our experiment also shows that this method is only effective for the benchmarks that have high computation-communication ratios [14]; in this experiment, the ratio is greater than 100. The reason for this result is that the benchmarks with small computation-communication ratios would incur significant inter-core communication energy overhead compared to computational energy.

5.3 StreaMorph vs. Flextream Task Migration

In the previous sections, we demonstrated how StreaMorph can help save energy. However, how is StreaMorph compared to the straightforward filter migration scheme implemented in Flextream [17] by Hormati et al.? Figure 8 shows the advantage of StreaMorph over a Flextream filter migration scheme in reducing buffer sizes when switching from multiple cores to one core. For example, when switching from eight cores to one core, because StreaMorph transforms the stream graph structures of applications, it reduces the buffer sizes by 82.58% on average over Flextream, which does not modify the stream graph structures. Even when switching from two cores to one core, StreaMorph can help reduce buffer sizes by 57.62%. Especially, for benchmarks `ChannelVocoder`, `FMRadio`, `FilterBank`, StreaMorph can help reduce buffer sizes more substantially, while it is not useful in the case of `TDE` because `TDE` does not require the filter-replication technique even for eight cores.

In addition, Flextream does not allow the optimizations dependent on the number of cores as described in Section 3.1.3. Furthermore, in Flextream, filters are not fused to allow fine-grain filter migration to avoid the situation in Figure 2, while filter fusion is beneficial to reduce synchronization and communication between filters [15]. As a consequence, Flextream suffers around 9% performance penalty from the optimal configurations [17]. Our experiment comparing the performances of StreaMorph and the Flextream filter migration technique for switching from five cores to three cores also result in the same result. To save space, we do not present the result here.

5.4 Switching Time

We have shown that switching between configurations can help reduce energy consumption of multicore systems. However, if it takes too long to switch between configurations, QoS of stream programs may suffer. It is desirable to measure the switching time of the system. A configuration switch is composed of three steps: the epilogue stage of the current configuration, state copying, and the prologue stage of the incoming configuration. The following equation shows how switching time is broken down:

$$t_{switching} = t_{epilogue} + t_{state-copying} + t_{prologue} \quad (15)$$

	Prologue time (μs)		Epilogue time (μs)		Copy size (bytes)	
Benchmarks	Max	Average	Max	Average	Token	State
BitonicSort	716	126	27	22	0	4
ChannelVocoder	2999	2477	45703	39271	8820	252
DCT	28	21	104	68	0	4
DES	210	153	148	115	0	4
FFT	52	38	37	31	0	4
FilterBank	1442	1121	12385	7217	1984	508
FMRadio	1582	988	56575	26974	492	508
MPEG2Decoder	116	75	176	125	0	4
Radar	123	104	347	305	0	1032
Serpent	2303	853	648	548	0	0
TDE	5	3	770	361	0	4

Table 1. Switching time statistics.

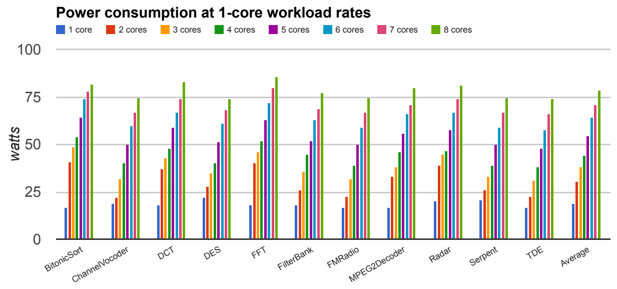
We run each benchmark and measure $t_{epilogue}$ and $t_{prologue}$ on 1 to 8 cores. To save space, we only report the maximum and average measured values for each benchmark in Table 1. As, $t_{state-copying} \approx 0$, it is often too small to measure exactly, we instead report the number of bytes to copy for each benchmark in Table 1. We can use the data from the table to compute maximum amounts of time to switch from one configuration to another configuration. For example, for the `DES` to switch from 3 cores to 7 cores, $t_{epilogue}$ for 3 cores is bounded by 148μs and $t_{prologue}$ for 7 cores is bounded by 210μs, and $t_{state-copying} = 0$ because there is no token that needs to copy. The total switching time is smaller than or equal to $148 + 0 + 210 = 358\mu$s for `DES`. We can see that switching times are small enough not to degrade user experience.

In addition, note that during the configuration switching process, processors still do useful work such as running epilogue and prologue stages, and as a result, the overall performance would not be affected too much.

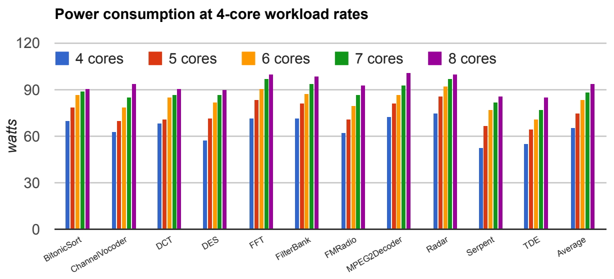
6. Lessons Learned

Designing the StreaMorph scheme, we realized a number of principles for designing adaptive programs:

- *Modularity:* Applications should be decomposed into subprocesses to allow task migration between cores.
- *Functional subprocesses:* Ideally, the subprocesses should be designed to be stateless to expose parallelism, e.g. by replicating stateless processes.
- *Internal state exposure:* States of programs should be exposed by storing on external queues instead of within subprocesses to facilitate state migration while morphing programs.

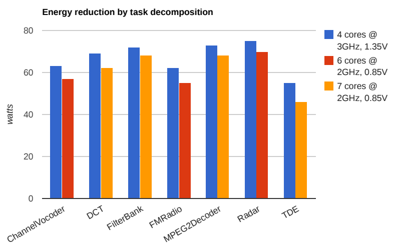


(a) Energy consumption when input rates saturate one core.

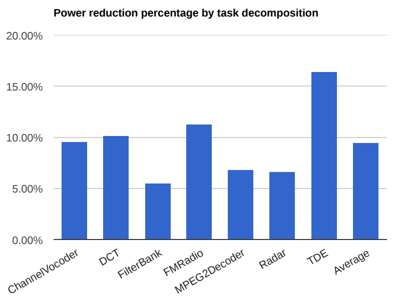


(b) Energy consumption when input rates saturate four cores.

Figure 6. Energy reduction by task consolidation.



(a) Measured energy consumption.



(b) Energy reduction percentage

Figure 7. Energy reduction by task decomposition.

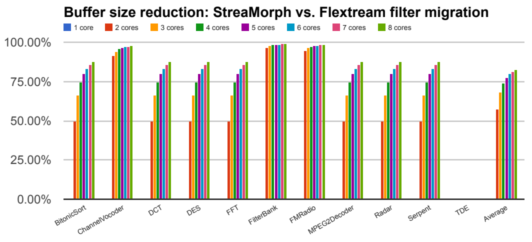


Figure 8. Buffer reduction.

- *Predictable inter-process communication rates:* This property eases program state transferring processes, e.g. by reducing the amount of copied state.

These principles point to other domains for adaptive programs. We find we can apply the adaptive program concept to programming models such as the Cilk [10], PetaBricks [4] and SEDA [34].

7. Related Work

The Hormati et al.'s Flextream [17] work is closely related to our work. Hormati et al. present a method that can efficiently repartition stream programs to adapt dynamically to environment changes such as the number of available cores. The major drawback of the Flextream is that it does not reduce memory usage proportionally when reducing the number of cores used as shown in Section 5.3 as well as 9% performance degradation. In addition, we not only present a method enabling the above adaptations but also show how to use the method to reduced energy consumption of stream programs by putting stream programs into external settings. Aleen et al. [2] propose a method to dynamically predict running times of portions of streaming programs based on input values. Portions of programs are dynamically relocated across cores based on prediction results to balance workload on multicore. Task consolidation has been deployed in the Linux kernel to reduce energy consumption by relocating workloads to fewer cores to allows other cores to turn into deep sleep states [31]. Besides the task consolidation technique, adjusting processors' speed using the DVFS capability to computation demands is another popular technique. Choi et al. [9] present a method to reduce energy consumption of a MPEG decoder program by adapting processor frequencies to MPEG video frame rates. The dynamic knobs framework [16] dynamically adjusts processor frequencies based on QoS requirements to save energy. In [5], Baek and Chilimbi present a framework that compile programs into adaptive ones that can return approximate results based on QoS requirements to reduce energy consumption.

Hardware energy-efficient research has been focusing designing processors that can adapt themselves to QoS requirements. Executing code regions of certain characteristics to suitable cores in heterogeneous multicore systems composed of cores with different points of energy/performance sharing the same ISA to save energy has been explored in [23, 26, 36]. Another approach to energy-efficient computing is to exploit the DVFS capability of modern processors [3, 35]. Burger et al. proposed an Explicit Data Graph Execution (EDGE) architecture [8] to reduce energy consumption by getting rid of complicated speculation circuits inside modern processors and using compiler techniques instead.

Our work is also related to the fair multiple resource sharing problem [11, 12] in cloud computing. Our StreaMorph scheme, when reducing the number of cores used, also reduces memory usages accordingly. This feature makes our StreaMorph scheme more suitable for cloud computing than the Flextream approach [17].

In [28], Parhi and Messerschmitt present a method for finding multiprocessor rate-optimal schedules for data flow programs. Rate-optimal schedules help programs achieve minimal periods for iterations given an infinite number of cores. Renfors and Neuvo's framework [30] focuses on determining the minimal sampling periods for a given digital filter structure assuming the speed of arithmetic operations is known and the number of processing units is infinite. This line of work is different from our work in the sense that, they assume static streaming rates and applications are optimized for specific hardware platforms while in cloud computing, mobile computing and cyber-physical systems, applications run on a wide variety of underlying hardware.

Finally, our work derives from the work by the StreamIt compiler group [14, 15, 33]. However, we focus on the energy-efficient aspect instead of speed optimization [15, 22]. Our analysis depends on the static properties of the SDF to derive sequences of filter executions that drain tokens on channels. Deriving such reverse sequences for more expressive stream models of computations such as Kahn process networks [19] is problematic due to unknown traffic patterns between processes. Although this work is within the SDF domain, the process migration technique to improve core utilization and reduce energy consumption is applicable to other streaming languages as well [7, 20, 27].

8. Conclusion

We have made a case for exploiting high-level abstractions to design adaptive programs by presenting our StreaMorph technique for stream programs. We have shown that high-level abstractions can help design *adaptive programs*. The concept of adaptive programs proposed in this paper is important in cloud computing, mobile computing, and cyber-physical systems when applications can be dynamically deployed and migrated on a wide variety of environments. Morphing programs can also help isolate performance of applications, thereby improving QoS of applications.

This work can be extended in several directions. Our next step would be predicting the number of cores necessary for a given IO rate. We plan to apply the adaptive program concept to applications with implicit parallelism with multiple algorithm choices, e.g. in PetarBricks [4], so that applications can execute adaptively under fair-multiple-resource constraints [11, 12].

Our evaluations use Intel Xeon processors, it would be more interesting if our evaluations are done using multimedia processors, however, at the time the paper is written, we do not have a multicore multimedia processor platform at hand. We also have not explored the idea of using configuration switching to lower core utilization to protect processors from overheating.

References

[1] http://www.youtube.com/t/press_statistics/. YouTube Statistics.

[2] F. Aleen, M. Sharif, and S. Pande. Input-driven dynamic execution prediction of streaming applications. PPoPP '10. 2010.

[3] A. Anantaraman, K. Seth, K. Patil, et al. Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems. ISCA '03. 2003.

[4] J. Ansel, C. Chan, Y. L. Wong, et al. PetaBricks: a language and compiler for algorithmic choice. PLDI '09. 2009.

[5] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. PLDI '10. 2010.

[6] P. Bailis, V. J. Reddi, S. Gandhi, et al. Dimetrodon: processor-level preventive thermal management via idle cycle injection. DAC '11. 2011.

[7] I. Buck, T. Foley, D. Horn, et al. Brook for GPUs: stream computing on graphics hardware. SIGGRAPH '04. 2004.

[8] D. Burger, S. W. Keckler, K. S. McKinley, et al. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37(7):44–55, July 2004.

[9] K. Choi, K. Dantu, W.-C. Cheng, et al. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. ICCAD '02. 2002.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. PLDI '98. 1998.

[11] A. Ghodsi, V. Sekar, M. Zaharia, et al. Multi-resource fair queueing for packet processing. SIGCOMM '12. 2012.

[12] A. Ghodsi, M. Zaharia, B. Hindman, et al. Dominant resource fairness: fair allocation of multiple resource types. NSDI'11. 2011.

[13] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. ASPLOS '04. 2004.

[14] M. I. Gordon. Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2010.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. ASPLOS '06. 2006.

[16] H. Hoffmann, S. Sidiroglou, M. Carbin, et al. Dynamic knobs for responsive power-aware computing. ASPLOS '11. 2011.

[17] A. H. Hormati, Y. Choi, M. Kudlur, et al. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. PACT '09. 2009.

[18] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. ICCAD '02. 2002.

[19] G. Kahn. The semantics of a simple language for parallel programming, Aug 1974.

[20] U. J. Kapasi, S. Rixner, W. J. Dally, et al. Programmable Stream Processors. *Computer*, 36, August 2003.

[21] C. Kim, S. Sethumadhavan, M. S. Govindan, et al. Composable Lightweight Processors. MICRO 40. 2007.

[22] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. PLDI '08. 2008.

[23] R. Kumar, K. I. Farkas, N. P. Jouppi, et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. MICRO 36. 2003.

[24] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), 1987.

[25] Y. Li, D. Brooks, Z. Hu, et al. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. HPCA '05. 2005.

[26] Y. Luo, V. Packirisamy, W.-C. Hsu, et al. Energy efficient speculative threads: dynamic thread allocation in Same-ISA heterogeneous multicore systems. PACT '10. 2010.

[27] W. R. Mark, R. S. Glanville, K. Akeley, et al. Cg: a system for programming graphics hardware in a C-like language. SIGGRAPH '03. 2003.

[28] K. K. Parhi and D. G. Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs Via Optimum Unfolding. *IEEE Transactions on Computers*, 40(2):178–195, February 1991.

[29] J. Park, D. Shin, N. Chang, et al. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. ISLPED '10. 2010.

[30] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed constraints. *IEEE Transactions on Circuits and Systems*, 28(3):196 – 202, Mar 1981.

[31] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, et al. Energy-Aware Task and Interrupt Management in Linux. volume 2 of *Proceedings of the Linux Symposium*. Aug 2008.

[32] A. Tanenbaum. Computer Networks. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[33] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. PACT '10. 2010.

[34] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. SOSP '01. 2001.

[35] Q. Wu, M. Martonosi, D. W. Clark, et al. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. MICRO 38. 2005.

[36] Y. Wu, S. Hu, E. Borin, et al. A HW/SW co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing. CGO '11. 2011.