Green Streams for Data-Intensive Software

Thomas W. Bartenstein and Yu David Liu
SUNY Binghamton
Binghamton, NY13902, USA
{tbarten1, davidL}@binghamton.edu

***Abstract*—This paper introduces GREEN STREAMS, a novel solution to address a critical but often overlooked property of data-intensive software: energy efficiency. GREEN STREAMS is built around two key insights into data-intensive software. First, energy consumption of data-intensive software is strongly correlated to data volume and data processing, both of which are naturally abstracted in the stream programming paradigm; Second, energy efficiency can be improved if the data processing components of a stream program coordinate in a "balanced" way, much like an assembly line that runs most efficiently when participating workers coordinate their pace. GREEN STREAMS adopts a standard stream programming model, and applies Dynamic Voltage and Frequency Scaling (DVFS) to coordinate the pace of data processing among components, ultimately achieving energy efficiency without degrading performance in a parallel processing environment. At the core of GREEN STREAMS is a novel constraint-based inference to abstract the intrinsic relationships of data flow rates inside a stream program, that uses linear programming to minimize the frequencies – hence the energy consumption – for processing components while still maintaining the maximum output data flow rate. The core algorithm of GREEN STREAMS is formalized, and its optimality is established. The effectiveness of GREEN STREAMS is evaluated on top of the StreamIt framework, and preliminary results show the approach can save CPU energy by an average of 28% with a 7% performance improvement.**

I. INTRODUCTION

From megabyte-scale Youtube Apps on smart phones, to gigabyte-scale Netflix applications on laptops, and to terabyte-scale NASA scientific computations on servers [1], data-intensive software is quickly becoming the new norm of modern computing. Software engineering for "Big Data" is an active area of research, with innovations addressing diverse goals, such as architectural soundness [2], [3], programmability [4], [5], performance [6], [7], and seamless database integration [8], [9].

A critical goal that has received less attention than it deserves is the *energy efficiency* of data-intensive software. According to US Environment Protection Agency (EPA), data centers in 2007 were responsible for up to 1.5% of the total US electricity consumption [10], and recent reports show the percentage has increased to 1.7-2.2% in 2011 [11]. In the consumer sphere, battery-powered hand held devices such as smart phones and tablets are experiencing explosive growth in popularity. On both high-end and low-end platforms, data-intensive software is widely used: data center applications are predominately data-intensive in nature; smart phone Apps related to video, music, and maps are among the most commonly used. In recent years, a number of software-centric solutions have been proposed to address energy efficiency, through design patterns [12], [13], programming language designs [14], [15], [16] and compiler and runtime optimizations [17], [18], [19], but none has focused on data-intensive software. This is unfortunate because the root cause of energy consumption for data-intensive software is often a combination of high-volume data processing and complex data flows, distinctive traits not sufficiently addressed by solutions built around control-flow-centric models.

In this paper, we propose GREEN STREAMS, a novel energy-efficient solution that addresses data-intensive software. At its essence, GREEN STREAMS is an energy-efficient "twist" to standard stream programming models [4], [20], [21]. Stream programming is a general-purpose paradigm where software is composed as a *stream graph*, where nodes of the graph are data processing components called *filters*, and edges of the graphs are data flows called *streams*. Compared with control-flow-centric models (*e.g.* Java and C), the streaming model exposes data processing and data flow at the forefront of programming. Its friendliness to parallelism – crucial in the multi-core era – has been well articulated. GREEN STREAMS elucidates yet another beneficial trait of the stream paradigm – its friendliness for improving energy efficiency – crucial in the "Big Data" era.

The energy efficiency solution of GREEN STREAMS is based on a key insight into stream programming: a stream graph, like a manufacturing assembly line, can be operated with more efficiency if the rates of streams can be coordinated, so that, one filter may output a data item to a stream "just-in-time" for consumption by the next filter on the receiving end of the stream. GREEN STREAMS optimizes the trade-offs between performance and energy consumption through judicious adjustment of the stream rates, a goal achieved by a novel combination of static inference and dynamic scaling of CPU frequencies through Dynamic Voltage and Frequency Scaling (DVFS). Concretely, this paper makes the following technical contributions:

- a novel *constraint-based rate inference* algorithm to statically compute the intrinsic relationships among data streams and coordinate them in an energy-efficient fashion;
- the use of *linear programming* to compute the minimal frequencies necessary to execute individual filters while at the same time maintaining the maximum output rate of the whole application;
- a formal account of the GREEN STREAMS core, and more importantly, a formal analysis of the *optimality* of all

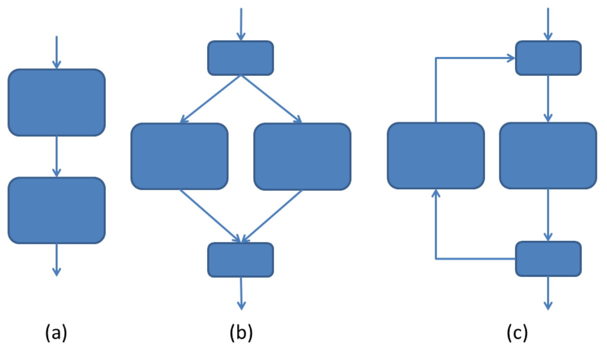

Fig. 1. Stream Composition: (a) Sequence (b) Split/Join (c) Loop

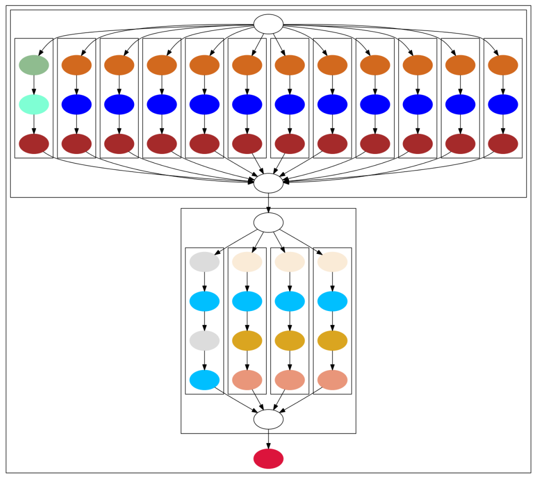

Fig. 2. An Example Stream Graph (BeamFormer)

frequency selections, which intuitively translates to the analytical optimality of energy reduction;

- a prototype implementation that involves DVFS instrumentations on top of a parallel stream processing infrastructure;
- an evaluation that demonstrates the effectiveness of our approach in saving energy without degrading performance.

Broadly, GREEN STREAMS explores the unbeaten path of constructing energy-efficient software where innovations on programming models, program analyses, and runtime systems converge. To the best of our knowledge, this unified software-centered approach is unique in addressing energy efficiency of data-intensive software.

II. BACKGROUND

A. Stream Programming Model

The stream paradigm organizes software into basic units of first-in-first-out (FIFO) data streams flowing through stream filters. A data stream is simply a list of data objects of fixed size. A stream filter is a unit of software which consumes data from a single input data stream, and produces data on a single output data stream. The description of the details of a stream filter is implementation dependent, but can be viewed abstractly like a function, with internal computation. Stream filters pop a fixed number of data items from their input stream when that data is available, process the input data, and push a fixed number of derived data items onto their output stream.

A stream program is represented as a stream graph, which can be decomposed as sub-graphs, which ultimately decomposes to simple filters. There are three common forms of composition, as represented in Figure 1. The simplest is sequential composition, in which the output stream of one sub-graph feeds the input stream of the second sub-graph (Figure 1(a)). The split/join composition (Figure 1(b)) first “splits” a single input stream to two or more streams, each of which will be fed to different sub-graphs. The output streams of these sub-graphs will be “joined” together into a single output stream. A third, less commonly used composition style is a loop configuration (Figure 1(c)). Here, the output of the ”body” sub-graph is split into an output stream and a feedback stream. The feedback stream feeds a ”loop” sub-graph, whose output is joined to the input stream, and the result feeds the body sub-graph. For instance, the full stream graph for the BeamFormer benchmark [22] is graphically represented in Figure 2. In this figure, each colored oval represents a stream filter, and sub-graphs are surrounded by rectangles.

It is well known that a major benefit of stream programming is its natural support for parallelism. For instance, two filters with different functionalities can be deployed as different threads running on different CPU cores, achieving task parallelism, whereas two instances of the same filter code can take different data and run in parallel as well, achieving data parallelism. Logically speaking, every filter – such as every oval box in Figure 2 – can be mapped to a separate thread and run in parallel. In real-world stream programming systems, optimizations exist to map muliple logical filters into one thread [4].

B. The Stream Paradigm: Expressiveness and Applicability

Popular terms such as “video streaming” are both a boon and a burden to the stream paradigm. It vividly demonstrates what *one* streaming application looks like, but at the same time may lead to misconceptions about the generality of stream programming. Is the stream paradigm a general-purpose software development paradigm? We answer this question in two dimensions.

From the perspective of language expressiveness, the stream model is a variant of the data-flow programming model [23]. The namesake difference between control-flow and data-flow programming highlights the individual strengths of each model, but in terms of language expressiveness, the two models are on par: commonly used control flows are either supported or encodable in the dataflow model, and data-flow analysis is a standard semantic analysis for the control flow model [24]. Most of the recently developed stream languages are extensions from Java/C-like languages [4], [20], [21], a hybrid of both paradigms. Below the surface, the most non-trivial semantic difference between the two models is the data-flow model generally assumes a non-shared memory model between filters: different filters only access the same memory through explicit input/output stream connections. With a growing awareness off the vulnerabilities of shared-memory models (*e.g.* race conditions and atomicity violations), non-shared memory models are becoming more popular in new languages such as Scala [25] and Go [26]. The combination of non-shared memory models and the explicit identification of parallelism in stream languages make them naturally conducive to constructing highly effective parallel software.

From the perspective of applicability, any software where the program can be decomposed as a graph of data flows will naturally fit into a stream programming paradigm. In modeling, this application style is among the most classic software architectures [27], and resonates in current research on software processes and work flows. In programming, the stream paradigm is known to be relevant to Graphical User Interface (GUI) programming ("GUI events as streams" [28]), sensor network programming ("sensing data as streams" [29]), database programming ("query results as streams" [30]), and robotics programming ("signals as streams" [31]).

C. Dynamic Voltage and Frequency Scaling (DVFS)

DVFS [32] is a common CPU feature where the operational frequency and the supply voltage of the CPU can be dynamically adjusted. Virtually all CPUs being used today – from ARM Cortex on Droid smartphones, Intel Core 2 on laptops and desktops, to high-end clusters in data centers – support DVFS. It the era of multi-core CPUs, the frequencies of individual cores can often be adjusted separately, a feature known as multiple frequency domain support. For instance, the AMD Zambezi family, a family of 8, 6, or 4-core CPUs shipped in 2011, supports this feature.

DVFS is often used as an effective power management strategy in VLSI and architecture research. In addition to small portions related to static leakage, the vast majority of a CPU's power consumption P results from its dynamic operation, which can be (roughly) computed as $P = C * V^2 * F$, where V is the voltage, F is the frequency, and C is the capacitance. The energy consumption E is an accumulation of power consumption over time, roughly $E = P * t$ where t is the operating time. Due to the innate nature of CPU VLSI design, voltage and frequency are often scaled together. In a multi-core context, it has been known that power has a somewhat cubic relation to DVFS scaling [33]. Scaling down the CPU frequency is thus effective in saving *power*. Saving *energy* however is slightly more complex, because a reduction of frequency may increase the execution time, t. DVFS-based energy management thus often deals with the trade-off between energy consumption and performance.

$$
\begin{array}{llll}
A & ::= & \langle S; P \rangle & \textit{stream application} \\
P & ::= & P \rhd P' & \textit{stream graph} \\
 & \mid & P \parallel_{\langle n_{sa}, n_{sb}, n_{ja}, n_{jb} \rangle} P' & \\
 & \mid & P \circlearrowleft_{\langle n_j, n_{fi}, n_{fo}, n_s \rangle} P' & \\
 & \mid & F_{\langle \ell, n_i, n_o \rangle} & \\
 & & & \\
F & & & \textit{filter} \\
S & ::= & \langle \overline{d}; \texttt{rt} \rangle & \textit{stream} \\
 & & & \\
\ell & \in & \mathbb{LAB} & \textit{filter label} \\
d & \in & \mathbb{INTEGER} & \textit{data element} \\
n & \in & \mathbb{NAT} & \textit{natural number} \\
\texttt{rt} & \in & \mathbb{FLOAT} & \textit{rate}
\end{array}
$$

Fig. 3. Stream Programming Core Abstract Syntax

III. The Green Streams Algorithm

This section describes the algorithm used to statically determine an optimal DVFS setting for multiple cores executing a parallelized stream process.

A. Abstract Syntax

We formalize a small core of stream programming, whose abstract syntax is represented in Figure 3. A stream application, A, is represented as a tuple, $\langle S; P \rangle$, where S is the input stream, and P is a stream graph. Each stream is represented by tuple $\langle \overline{d}; \texttt{rt} \rangle$, where $\overline{d}$ defines a sequence of data d and $\texttt{rt}$ represents *data rate*, *i.e.* the frequency the elements in $\overline{d}$ become available. For convenience, we only formalize the case where stream data are integers. A stream graph is either a filter ($F_{\langle \ell, n_i, n_o \rangle}$), or composed of two stream graphs P and P', via either sequential composition $P \rhd P'$, or split-join composition $P \parallel_{\langle n_{sa}, n_{sb}, n_{ja}, n_{jb} \rangle} P'$, or loop composition $P \circlearrowleft_{\langle n_j, n_{fi}, n_{fo}, n_s \rangle} P'$.

A filter $F_{\langle \ell, n_i, n_o \rangle}$ is the basic building block of a stream application. Each filter pops (consumes) n_i data elements from its input stream, and pushes (produces) n_o data elements on its output stream. To keep this presentation simple, we are abstract about the internal implementation of filters, but a filter is most easily pictured as a function taking n_i input elements as parameters, and returning n_o output elements as the return value. Each filter is explicitly labeled (ℓ). For a stream graph, P, the set of all filter labels appearing in P is computed by convenience function $\texttt{filters}(P)$, whose definition is obvious. Operationally, $\langle S; F_{\langle \ell, n_i, n_o \rangle} \rangle$ feeds elements in $\overline{d}$ to filter F in a FIFO fashion at the rate of $\texttt{rt}$, where $S = \langle \overline{d}; \texttt{rt} \rangle$. An output stream – the result of applying F over $\overline{d}$ – is implicit.

Stream application $\langle S; P \rhd P' \rangle$ can be viewed as having S as the input stream of P, whose output stream is then fed to P' as input. Application $\langle S; P \parallel_{\langle n_{sa}, n_{sb}, n_{ja}, n_{jb} \rangle} P' \rangle$ first splits the data elements in stream S into two streams in a round-robin fashion, following the *distribution factor* $\langle n_{sa}; n_{sb} \rangle$: the computation waits until $n_{sa} + n_{sb}$ data elements are available on its input stream, S, and then writes the first n_{sa} of these data elements to the input stream of P, and the n_{sb} data elements to the input stream of P'. This process then repeats. For instance, given $S = \langle [1, 2, 3, \dots,], \mathtt{rt} \rangle$ for some $\mathtt{rt}$, and $n_{sa} = 2$, $n_{sb} = 3$, the data elements fed to the input of P are $[1, 2, 6, 7, 11, 12, \dots]$, and those fed to the input of P' are $[3, 4, 5, 8, 9, 10, \dots]$. The second part of the parallel operation is to join the data elements output from P and P' into a single stream, again, in a round-robin fashion, following the *aggregation factor* $\langle n_{ja}; n_{jb} \rangle$. The process waits until there are at least n_{ja} data items available on the output of P, and at least n_{jb} data items available as the output of P'. When both conditions are met, n_{ja} data items from P and n_{jb} data items from P' are transferred to the output of the composed stream.

The meaning of $\langle S; P \circlearrowleft_{\langle n_j, n_{fi}, n_{fo}, n_s \rangle} P' \rangle$ is to first join S and the output stream of P' (*i.e.* the output stream of the feedback loop) following the aggregation factor $\langle n_j; n_{fo} \rangle$, then feed the joined stream as the input stream of P, and finally divide the output stream of P into two following the distribution factor $\langle n_{fi}; n_s \rangle$, one of which becomes the final output stream, while the other of which is the input stream of P'. Note that there are some restrictions on the values of n_{fi} and n_{fo} in order to ensure that a feedback loop stream graph can achieve a steady-state schedule [22], a topic out of the scope of this paper.

r	$::=$	$rv \mid \ell \mid r_I \mid r_O$	*rate variable*
Σ	$::=$	$\overline{\sigma}$	*constraint set*
σ			*linear constraint over* r
rv			*rate variable name*

Fig. 4. Inference Elements

B. Constraint-Based Rate Inference

We define a novel constraint-based algorithm to infer the intrinsic rate dependencies of different elements of a stream program. The inference algorithm represents the data stream rates as *rate variables*, defined in Figure 4.

A rate variable, r, is the abstract representation of a stream rate, used to constrain and reason about stream rates statically. The most basic form of a rate variable is *rv*, simply a name that the inference algorithm can internally generate, where every *fresh* generation specifies the creation of a distinct rate variable with a distinct name. The second form of a rate variable is a filter label ℓ. When a filter label appears in a constraint, it doubles as a rate variable that abstractly represents the *natural rate* of a filter. The natural rate is the intrinsic rate at which a filter can process a single set of data, *i.e.* the inverse of time required to take n_i data items from the input stream, process it through the filter, and put n_o data items onto the output stream. When a filter executes at its natural rate, the rate at which items are consumed from the input stream is $\ell \times n_i$, and the rate at which items are added to the output stream is $\ell \times n_o$. For convenience, we further provide two pre-defined rate variables, r_I and r_O to represents the input rate and output rate of the entire stream graph, respectively.

The core inference rules for rate constraints are defined in Figure 5. Function $RC(r, P, r')$ collects the constraints for stream graph P when its input stream rate and output stream rate are represented by r and r' respectively. Each rule is defined over a particular syntactical construct, and represents a principle of GREEN STREAMS that we now elaborate.

a) Principle of Natural Bound: Clearly, the output rate of a filter is dependent on the input rate to that filter. It might be tempting to consider a filter as a pipeline whose output stream rate can be infinite given an infinite input stream rate. This naive view ignores the execution model of a stream program: a filter cannot start processing a second set of input data items until it has finished with the first. Therefore, even if data items arrive at the input to a filter very fast, the filter cannot execute faster than its natural rate. Thus, the maximum output rate of a filter is not only constrained by the rate of the input stream ($r' \leq r \times \frac{n_o}{n_i}$ in (R-Filter)), but also the natural rate of the filter itself ($r' \leq \ell \times n_o$ in the same rule).

b) Principle of Sequential Balance: Given a stream graph involving sequential composition $P_1 \rhd P_2$, it would be a waste of energy if P_1 can output data items at the rate of 100 items a second whereas P_2 can only take in data items at the rate of 10 items a second. It would also be a waste of energy if P_2 can only output 10 items a second, and P_1 could take in data items at the rate of 100 items a second. In both cases, the party with a faster rate has no positive impact on the overall output rate of the stream graph – the ultimate "throughput" that matters. GREEN STREAMS balances the output of P_1 with the input of P_2. Observe that in (R-Seq), *rv* is used both as the output rate of P_1 – as in $RC(r, P_1, rv)$ – and the input rate of P_2 – as in $RC(rv, P_2, r')$.

c) Principle of Join Balance: Given a split-join composition of two streams P_A and P_B, in the form of $P_A \parallel_{\langle n_{sa}, n_{sb}, n_{ja}, n_{jb} \rangle} P_B$, let us first assume $n_{ja} = 1$ and $n_{jb} = 1$. It would be a waste of energy if the output rate of P_A were vastly greater than that of P_B, because in this case, the two streams are "joined" together by taking items from the two streams in a round-robin fashion, 1 from P_A and 1 from P_B, and the P_A branch would have to wait for the P_B branch. More generally in (R-Par), we use rv'_a and rv'_b to represent the output rates of P_A and P_B respectively, where the balanced execution would conform to the constraint $\frac{rv'_a}{n_{ja}} = \frac{rv'_b}{n_{jb}}$. The rest of the generated constraints of the same rule describes the intrinsic dependencies of rates. Here rv_a and rv_b are the input rates of P_A and P_B respectively. The first two constraints denote how the rates of the two are "split" from the input rate of the whole stream graph, whereas the last constraint describes how the output rate of the whole stream graph is combined.

Notice that the (R-Loop) rule is simply a variation of the (R-Par) rule because a loop composition can be viewed as a "reversed" split-join configuration. In this case, rv_b is the rate

$$\begin{array}{llcl}
\text{(R-Filter)} & RC(r, F_{\langle \ell, n_i, n_o \rangle}, r') & \overset{\text{def}}{=} & \{r' \leq r \times \frac{n_o}{n_i}, r' \leq \ell \times n_o\} \\
\text{(R-Seq)} & RC(r, P_1 \rhd P_2, r') & \overset{\text{def}}{=} & RC(r, P_1, rv) \cup RC(rv, P_2, r') \\
 & & \text{if} & rv \text{ fresh} \\
\text{(R-Par)} & RC(r, P_A \parallel_{\langle n_{sa}, n_{sb}, n_{ja}, n_{jb} \rangle} P_B, r') & \overset{\text{def}}{=} & RC(rv_a, P_A, rv'_a) \cup RC(rv_b, P_B, rv'_b) \cup \left\{ \begin{array}{l} rv_a = r \times \frac{n_{sa}}{n_{sa}+n_{sb}} \\ rv_b = r \times \frac{n_{sb}}{n_{sa}+n_{sb}} \\ \frac{rv'_a}{n_{ja}} = \frac{rv'_b}{n_{jb}} \\ rv'_a + rv'_b = r_O \end{array} \right\} \\
 & & \text{if} & rv_a, rv_b, rv'_a, rv'_b \text{ fresh} \\
\text{(R-Loop)} & RC(r, P_B \circlearrowleft_{\langle n_j, n_{fi}, n_{fo}, n_s \rangle} P_F, r') & \overset{\text{def}}{=} & RC(rv_b, P_B, rv'_b) \cup RC(rv_f, P_F, rv'_f) \cup \left\{ \begin{array}{l} \frac{r}{n_j} = \frac{rv'_f}{n_{fo}} \\ rv_b = r_I + rv'_f \\ r' = rv'_b \times \frac{n_s}{n_{fi}+n_s} \\ rv_f = rv'_b \times \frac{n_{fi}}{n_{fi}+n_s} \end{array} \right\} \\
 & & \text{if} & rv_b, rv_f, rv'_b, rv'_f \text{ fresh}
\end{array}$$

Fig. 5. Constraint-Based Rate Inference

for the joined stream combining the input stream of the graph and the output stream of the feedback loop, and rv'_b is the rate of the stream to be split into the output stream of the graph and the input stream of the feedback loop.

Since any stream graph is inductively defined over the 3 forms of compositions of filters, the inductive definition in Figure 5 is sufficient to compute constraints over arbitrary stream graphs.

C. Relating Frequency and Natural Rate

Our ultimate goal is to select appropriate frequencies for individual filters, a task that we will tackle in Sec. III-D. First, we must elucidate how frequencies are related to our rate inference.

Let us abstractly represent the supported frequencies of a CPU core as a total order $\langle \mathbb{FREQ}; < \rangle$, where each element $freq \in \mathbb{FREQ}$ in the concrete scenario would be an available frequency supported by the CPU (in Hertz). For simplicity, we only consider homogeneous architectures where all cores support the same number of available frequencies for DVFS. We use $\mathtt{max}(\mathbb{FREQ})$ to compute the upper bound of $\mathbb{FREQ}$. We further define a mapping function $\Pi : \mathbb{LAB} \times \mathbb{FREQ} \rightarrow \mathbb{FLOAT}$ that, given a filter label $\ell \in \mathbb{LAB}$, and a frequency $freq \in \mathbb{FREQ}$, $\Pi(\ell, freq)$ computes the natural rate for filter labeled ℓ under operating frequency $freq$. Intuitively, Π records how fast a data item can be output by filter ℓ when the filter is running on a CPU core of a particular frequency.

We rely on profiling to compute Π. Concretely, we profile a filter to determine the natural rate of that filter at the maximum CPU frequency, and assume an inversely proportional relationship between frequency and elapsed time of the filter. This relationship can be defined with the equation:

$$\Pi(\ell, freq) = \Pi(\ell, \mathtt{max}(\mathbb{FREQ})) \times \frac{freq}{\mathtt{max}(\mathbb{FREQ})}$$

We elaborate on this implementation detail in Section IV-C.

D. Linear Programming for Optimal Frequency Selection

From an abstract perspective, GREEN STREAMS follows two steps to select frequencies for specific filters. First, we assume every filter runs at the highest CPU frequency, and compute the maximum possible output rate of the whole stream graph. This is described as Algorithm 1 below. Second, we compute the lowest possible frequency at which individual filters can execute, assuming we must maintain the maximum possible rate computed in the first step. This is conducted by Algorithm 2 below. The central idea here is both steps can be achieved by performing linear programming over the constraints we inferred earlier (Section III-B).

Before we explain the details, let us first introduce some notation related to linear programming. Notation $\underset{\Sigma}{\mathbf{min}}\ obf$ represents an instance of linear programming to minimize an objective function *obf* over constraints Σ. It computes a mapping whose domain coincides with the rate variables that appear in *obf*, and whose range is the floating point numbers for rates (`rt`). In other words, all rate variables in all constraints are now "solved", including the subset of variables we care about. For example, a typical result looks like $[r_1 \mapsto 3.3, r_2 \mapsto 4.0]$, meaning r_1 should be of rate 3.3 and r_2 should be of rate 4.0 if we wanted to achieve the minimality of *obf*. Objective function *obf* takes the form of a linear expression. For convenience, we use symbol $\oplus$ to represent the AST expansion of addition, *i.e.* $\bigoplus_{rv \in \{rv_1, rv_2\}} rv$ is equivalent to objective function "$rv_1 + rv_2$." The meaning of notation $\underset{\Sigma}{\mathbf{max}}\ obf$ is identical to $\underset{\Sigma}{\mathbf{min}}\ obf$ except that we are maximizing the objective function.

Next let us define the constraints of a stream graph assuming that all filters are executing at the maximum frequency possible. In other words, each filter can operate at its highest natural rate. The definition is a simple substitution of all rate variables that represent filter natural rates with a concrete rate when the maximum frequency is used. Notation $\Sigma\{\mathtt{rt}/r\}$ means substitute every occurrence of r in Σ with $\mathtt{rt}$.

Definition 1 (Global Constraints with Maxed-Out Filters): Given a stream graph P, $\mathtt{mofCons}(P)$ is defined as $RC(r_I, P, r_O)\{\Pi(\ell_1, \mathit{freq})/\ell_1\} \ldots \{\Pi(\ell_n, \mathit{freq})/\ell_n\}$, where $\mathtt{filters}(P) = [\ell_1, \ldots, \ell_n]$ and $\mathit{freq} = \mathtt{max}(\mathbb{FREQ})$.

With this, the maximum output rate of a stream graph P is an instance of linear programming of maximizing r_O over global constraints with maxed-out filters:

Algorithm 1 (Max Output Rate): Given a stream graph P, $\mathtt{maxOut}(P)$ denotes the maximum output rate with unbounded input rate. It is defined as $\mathtt{rt}$, where $\max\limits_{\mathtt{mofcons}(P)} r_O = [r_O \mapsto \mathtt{rt}]$.

Note that we do not bound the input rate of the stream graph here. This is not necessary because, intuitively, the natural rate of individual filters – and the constraints associated with them – will limit the output rate of the whole stream graph. Hence, linear programming cannot yield unbounded results. Obviously, for users of GREEN STREAMS who would like to artificially bound the input rate of the stream graph, a variant algorithm can be provided as follows:

Definition 2 (Max Output Rate with bounded Input Rate): Given a stream graph P and a pre-defined input rate $\mathtt{rt}_0$, $\mathtt{maxOutB}(P, \mathtt{rt}_0)$ denotes the maximum output rate with bounded input rate $\mathtt{rt}_0$. It is defined as $\mathtt{rt}$, where $\max\limits_{\mathtt{mofcons}(P)\{\mathtt{rt}_0/r_I\}} r_O = [r_O \mapsto \mathtt{rt}]$.

Finally, given that we know the maximum output rate, the issue of reducing individual frequencies of filter executions – and hence energy consumption – is a matter of minimizing the natural rate of individual filters:

Algorithm 2 (Minimal Frequency): Given a stream graph P, the minimal frequency required for filter ℓ without affecting the overall output rate, denoted as $\mathtt{minFreq}(P, \ell)$, is defined as the least value *freq* in $\mathbb{FREQ}$ such that $\Pi(\ell, \mathit{freq}) \geq \mathtt{rt}$ and $\Sigma = RC(r_I, P, r_O)\{\mathtt{maxOut}(P)/r_O\}$, and $\min\limits_{\Sigma} \ell = [\ell \mapsto \mathtt{rt}]$.

E. Global Optimality and Algorithm Optimization

Algorithm 2 leaves two issues to be resolved. First, it only says how to find the minimal frequency for *a particular* filter. Does this localized optimality – seemingly greedy to the particular filter being subjected to linear programming – also lead to global optimality? Second, the algorithm requires linear programming to be used for every filter in the stream graph, which is not an efficient solution.

The following theorem addresses the first issue with Algorithm 2 above, namely, Algorithm 2 is a globally optimal algorithm.

Theorem 1 (Natural Rate Independence for Fixed Output): Given a stream graph P and a pre-determined output rate $\mathtt{rt}$, then if $\Sigma\{\mathtt{rt}_1/\ell_1\}$ has solutions, and $\Sigma\{\mathtt{rt}_2/\ell_2\}$ has solutions, then $\Sigma\{\mathtt{rt}_1/\ell_1\}\{\mathtt{rt}_2/\ell_2\}$ has solutions, where $\ell_1, \ell_2 \in \mathtt{filters}(P)$, and $\Sigma = RC(r_I, P, r_O)\{\mathtt{rt}/r_O\}$.

This important theorem states the independence of satisfiability of filter natural rates given a fixed output rate of the stream graph. Here Σ represents the necessary constraints to allow the stream graph to maintain an output rate $\mathtt{rt}$, and ℓ_1 and ℓ_2 are two filter natural rate variables ($\ell_1, \ell_2 \in \mathtt{filters}(P)$). The fact that $\Sigma\{\mathtt{rt}_1/\ell_1\}$ has solutions implies that by setting the natural rate of the filter represented by ℓ_1 to $\mathtt{rt}_1$, the output rate of the stream graph is maintained. Similarly, the fact that $\Sigma\{\mathtt{rt}_2/\ell_2\}$ has solutions means that by setting the natural rate of the filter represented by ℓ_2 to $\mathtt{rt}_2$, the output rate of the stream graph is maintained as well. The theorem thus tells us that the settings of ℓ_1 and ℓ_2 – hence the minimal frequency selections of the two – do not interfere with each other, and by setting ℓ_1 to $\mathtt{rt}_1$ and setting ℓ_2 to $\mathtt{rt}_2$ at the same time, the stream graph can still maintain it output rate $\mathtt{rt}$.

Finally, the independence of natural rate variables intuitively tells us that minimizing the natural rates of filters one by one is no better than minimizing the sum of all of them together. This leads to an optimized algorithm where one instance of linear programming can compute all minimal natural rates of all filters, and hence compute all minimal frequencies:

Theorem 2 (Linear Programming Compositionality): Given a stream graph P and some pre-defined rate constant $\mathtt{rt}$, and $\Sigma = RC(r_I, P, r_O)\{\mathtt{rt}/r_O\}$, $\min\limits_{\Sigma} \ell_i = [\ell_i \mapsto \mathtt{rt}_i]$ for each $\ell_i \in \mathtt{filters}(P)$ and $\min\limits_{\Sigma} \bigoplus\limits_{\ell_i \in \mathtt{filters}(P)} \ell_i = [\ell_1 \mapsto \mathtt{rt}'_1, \ldots, \ell_n \mapsto \mathtt{rt}'_n]$, then $\mathtt{rt}_1 = \mathtt{rt}'_1$, …, and $\mathtt{rt}_n = \mathtt{rt}'_n$.

IV. EXPERIMENTAL RESULTS

A. Implementation

We implemented GREEN STREAMS as an extension to StreamIt[1] (version 2.1.1), a sophisticated stream programming infrastructure. There were several significant modifications to StreamIt required in order to implement GREEN STREAMS, including the following:

- The inclusion of run-time monitors for profiling natural rate (Section III-C).
- The ability to perform and manipulate DVFS for StreamIt applications. To demonstrate the effectiveness of our approach in context, support was added to select one of four different DVFS configurations:
  - A GREEN STREAMS configuration, which sets DVFS frequencies based on the results of our static analysis.
  - A "fast" configuration in which all DVFS frequencies are forced to the highest possible frequency. This configuration is likely to deliver the fastest possible performance, but also consume the most energy.
  - An "on-demand" configuration in which all DVFS frequencies are managed by the standard "on-demand" governor in Linux. The on-demand governor modulates DVFS frequency based on demand. Cores with a high work-load are run at a high DVFS frequency to improve performance. Cores with a low workload are run at a low DVFS frequency to

[1]http://groups.csail.mit.edu/cag/streamit/

save energy. The on-demand DVFS configuration is the configuration most often used in DVFS-capable microprocessors, and is considered the default configuration.
    - A "slow" configuration in which all DVFS frequencies are forced to the lowest possible frequency. This configuration is likely to deliver the slowest performance.
- The implementation of the filter frequency selection algorithm (Section III-D).
- The ability to assign a filter to a core, and set DVFS for that core at run time.

The StreamIt compiler is equipped with an optimization performed during an intermediate pass called *partitioning*. Partitioning transforms and load-balances the source-program stream graph into a stream graph that contains a small number of relatively independent filters that can be mapped to independent cores in a multi-core architecture. The load balancing optimizations of StreamIt partitioning enable high parallel efficiency on multi-core hardware. The GREEN STREAMS algorithm is implemented over the post-partitioning stream graph.

B. Experimental Environment

All experiments were performed on an AMD FX-8510 (Bulldozer) microprocessor running Debian Linux Version 6.0.5. The processor was configured as an 8-core multiprocessor with the Turbo-boost feature disabled. All normal operating system tasks were executing in the background, but all experiments were performed with no other load on the system.

Our experiments consisted of running five StreamIt benchmarks in each of the four DVFS configurations five times, while measuring the system current draw using a current meter on the CPU power cable of the microprocessor. Since the CPU is supplied at a constant voltage of 12V, the power is directly proportional to the measured current, and the energy consists of the power consumed over time. We used a Fluke©i30 AC/DC Current Clamp which accurately measures current with a resolution of 1mA using Hall Effect technology. Current measurements were taken every 1/100th of a second, and stored on an independent system. These measurements were then post-processed to isolate each trial run, where a trial run consists of a single execution of a benchmark program in a single DVFS configuration. Five trials were collected for each benchmark in each DVFS configuration to avoid intermittent errors or current draw based on external factors such as cache latencies. We automated the testing of a single benchmark by creating a loop that cycled through each DVFS state with a three second sleep between each trial, and then executing that loop five times. This ensured that there were no latencies between DVFS states.

The benchmark programs we tested were a subset of benchmarks developed for the StreamIt compiler [22]. The five benchmark programs we selected were as follows.

- Beamformer - An implementation of standard beamforming or spatial filtering. This is a signal processing technique that combines signals in such a way as to achieve constructive interference or destructive interference, depending on the spatial relationship of the signals. Beamforming is used in several applications, including seismology, radio astronomy, etc.
- BitonicSort - An implementation of Batcher's bitonic sort network for sorting power-of-2 sized key sets.
- DCT - An implementation of a two-dimensional 8x8 inverse Discrete Cosine Transfer, which transforms a 16x 16 signal from frequency domain to signal domain. DCT is used in both JPEG and MPEG-2 coding.
- DES - An implementation of a Data Encryption Standard block cipher. This implementation uses 4 stages of processing rather than the 16 required by the US government DES standard.
- Vocoder - An implementation of a the speech filter analysis portion of a source-filter model. The analysis includes Fourier analysis, a low-pass filter, a bank of band-pass filters, and a pitch detector.

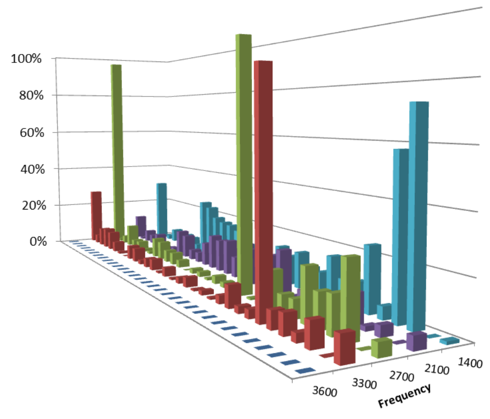

Fig. 6. Error between Computed Natural Rate and Profiled Natural Rate

C. Experimental Results

In Section III-C, we introduced a formula to calculate natural rate with the assumption that the relationship between DVFS frequency and the elapsed time of a specific filter is inversely proportional: GREEN STREAMS profiles the active elapsed time using the maximum CPU frequency, but assumes that the active elapsed time for lower frequencies can be estimated proportionally. In order to validate this assumption, we profiled each of the filters in the five benchmarks at all valid DVFS frequencies: 3.6Ghz, 3.3Ghz, 2.7Ghz, 2.1Ghz, and 1.4Ghz, monitoring the active elapsed time. We compared our estimated elapsed time with the monitored elapsed time. The graph in Figure 6 shows the percentage of error (Z axis) resulting from this assumption. Filters are on the X axis, and frequencies on the Y axis. While there are some filters at some frequencies that show significant error, the average error is well below 10%.

TABLE I
DVFS FREQUENCIES ASSIGNED TO BENCHMARK FILTERS

	Frequency in Ghz				
Benchmark	3.6	3.3	2.7	2.1	1.4
BeamFormer	4	1	2		1
BItonicSort	2		2	1	3
DCT	2				6
DES	2	3			3
Vocoder	1			4	3

The GREEN STREAMS compiler was run against the five benchmarks described above, each of which contains 8 filters that can be independently scheduled as threads after StreamIt partitioning (see Section IV-A). The GREEN STREAMS algorithm determines the optimum frequency for each of the 8 filters. Table I shows the DVFS frequencies computed according to our algorithm, and the number of filters for each benchmark assigned to each frequency.

Our 8-core CPU has 4 independently adjustable frequency domains, *i.e.* every pair of cores need to share a single frequency. Due to this hardware constraint, GREEN STREAMS needs to map the 8 filters (threads) to 4 DVFS frequency domains. In doing so, one filter in each benchmark (except DCT) was forced to a higher frequency than the one computed by our algorithm. We then collected current measurements for all benchmarks in all DVFS configurations.

The first observation is that the results show a remarkable consistency. The results from the 20 trials of the BeamFormer benchmark appear in Figure 7, grouped by different DVFS configurations. Observe that the instantaneous current fluctuations (which, given constant voltage, is proportional to power consumption) over time are very similar for different trials of the same DVFS configurations. Other benchmarks showed similar consistency. This consistency reinforces the concept that our experimental environment produced reliable results. Given this consistency, we are able to consolidate results from different trials of the same DVFS configuration and the same benchmark by taking the average current at each time.

The consolidated graph for the BeamFormer benchmark is in Figure 8. Following our discussion earlier, the instantaneous current readings (Y axis) are proportional to the instantaneous power consumption because the CPU has a constant voltage of 12V. The figure demonstrates the effect of running all cores at their top speed (the purple line), which consumes high power but finishes the fastest, versus running all cores at their lowest speed (the blue line), which incurs the lowest power consumption but takes significantly longer to complete. The GREEN STREAMS line (green) shows performance that almost matches the fast line, but with significantly less power consumption. The only surprising data in this graph is the on-demand line (in red). The on-demand DVFS configuration consumes more power than the fast state, but performs slightly slower. We speculate this is because the on-demand DVFS governor spends extra energy to switch DVFS frequencies,

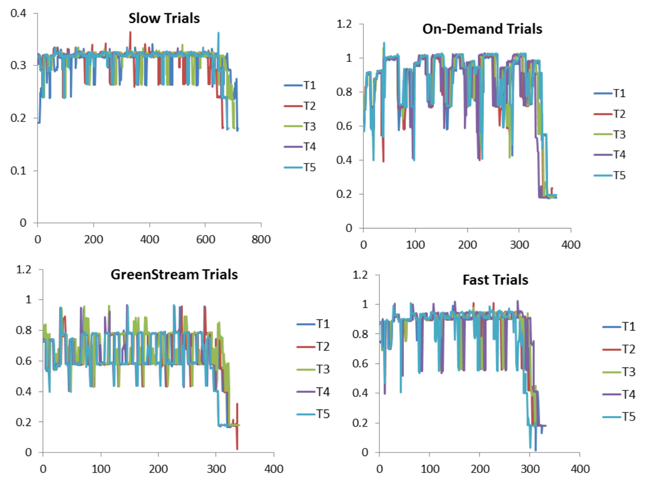

Fig. 7. BeamFormer Trial Consistency (X unit: 0.01 second; Y unit: 1mA)

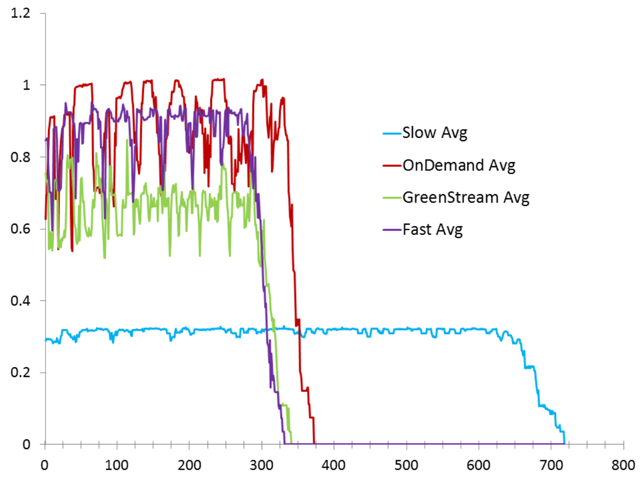

Fig. 8. BeamFormer Execution over Different DVFS Configurations (X unit: 0.01 second; Y unit: 1mA)

and the somewhat erratic resource demand of the BeamFormer benchmark may cause the on-demand governor to switch too frequently.

Figure 9 contains the DVFS configuration comparison for the other four benchmarks. In this figure, the DCT benchmark is the most interesting graph. The GREEN STREAMS DVFS configuration not only requires less power than either the fast state or the on-demand state, but also finishes significantly sooner than either the fast or the on-demand DVFS configuration. The DES graph is also interesting because it demonstrates the static nature of GREEN STREAMS. In the DES case, the on-demand DVFS configuration is better at handling an application for which the resource requirements change over time.

The energy consumption of each benchmark/DVFS configuration is proportional to the area under each curve in Figure 8 and Figure 9. Since the CPU is supplied at a

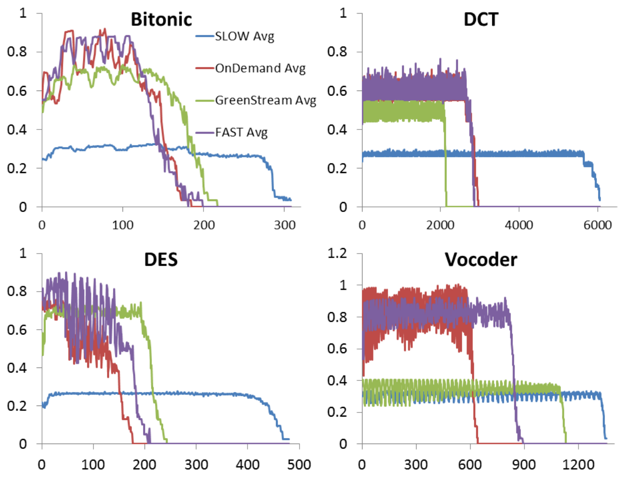

Fig. 9. Other Benchmarks over Different DVFS Configurations (X unit: 0.01 second; Y unit: 1mA)

TABLE II
ENERGY CONSUMPTION PER BENCHMARK

BenchMark	Slow	OnDem	GreenStr	Fast
BeamFormer	25.79	37.22	25.16	32.13
BitonicSort	10.02	13.62	14.76	13.94
DCT	192.74	199.59	124.82	199.76
DES	14.19	10.82	17.83	15.05
Vocoder	49.71	61.40	46.88	82.82

TABLE III
TIME CONSUMPTION PER BENCHMARK

Benchmark	Slow	OnDem	GreenStr	Fast
BeamFormer	6.88	3.64	3.29	3.23
BitonicSort	2.90	1.71	2.03	1.64
DCT	59.12	28.31	21.32	27.93
DES	4.61	1.72	2.35	2.00
Vocoder	13.45	6.30	11.23	8.63

constant voltage of 12V, the power is directly proportional to the measured current, and the energy consists of the power consumed over time. Our data collection method enabled simple energy consumption calculations, based on summing the current measurements for the entire time span of a given trial. Since current measurements were taken every 1/100th of a second, we can compute the "instantaneous energy consumption" for that measurement by multiplying the measurement value, 1/100 sec, and 12 V. The total energy consumed by the CPU for that trial is the sum of all instantaneous energy consumption values.

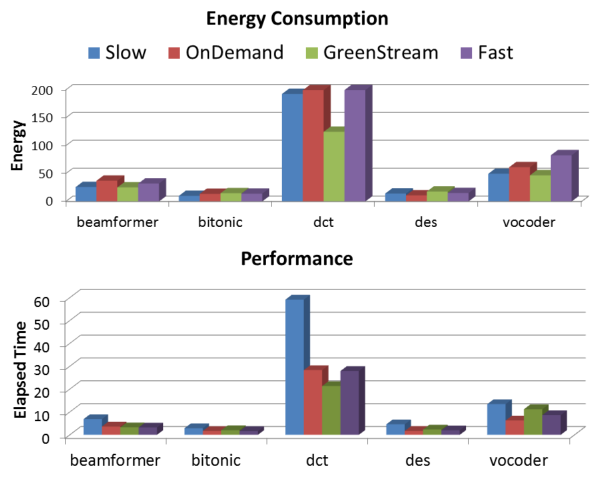

Fig. 10. Energy and Performance

Figure 10 and Tables II and III present the results of average performance and CPU energy consumption for all five benchmarks. In three of the five benchmarks (Beamformer, DCT, and Vocoder), the energy consumption with GREEN STREAMS was less than the on-demand energy consumption, and in all benchmarks except Vocoder, the performance of GREEN STREAMS was comparable or better than all other DVFS states. Clearly, the DCT benchmark, which was the most computation-intensive and had the most stable resource requirements, was the best demonstration of the advantages of GREEN STREAMS.

V. RELATED WORK

A number of energy management strategies have been proposed for stream applications, primarily from the systems community. Benoit *et. al.* [34] considers a subset of stream programs that can be modeled as serial-parallel workflows, and studies the problem of mapping such workflows to CMPs to minimize energy. Eprof [35] designs an energy-efficient scheduling algorithm for stream applications, with a non-DVFS based solution. Rangasamy *et. al.* [36] used stream programs as the context to evaluate the effectiveness of three DVFS schemes: one based on a Petri net performance model, one based on profiling, and one based on hardware. None of these efforts reduce the problem to a program analysis over stream programs as we do, nor do they perform constraint-based inference over stream rates.

DVFS as an implementation strategy has been used in compiler and run-time optimizations. Hsu *et. al.* [18] defines a compiler optimization algorithm where memory-intensive regions of the program control flow are identified, and a CPU's frequency is scaled down in these regions to reduce energy consumption. Xie *et. al.* [17] evaluates the limitations and opportunities of DVFS in a control-flow-centric setting. An operating system solution [19] is proposed to reduce energy efficiency by scheduling fixed-deadline tasks judiciously.

Several energy-aware programming models have been proposed as extensions to Java-like languages. Green [14] defines a framework where programmers can customize quality of service (QoS) to balance the trade-off QoS and energy consumption. EnerJ [15] allows data to be approximated, and applies hardware techniques to approximate data to save energy. Energy Types [16] defines a type system to reason about program energy-phase behaviors and energy states. Both EnerJ and Energy Types use DVFS as an implementation strategy; they are otherwise unrelated to our approach. Clause *et. al.* [12] explores the impact of different design patterns on energy consumption. The impact of different synchronization patterns on energy consumption was also explored [13].

Related work on stream programming and its applications was summarized in Sec. II-B.

VI. Conclusion

Green Streams provides a practical and effective solution to save energy for data-intensive software. The stream programming paradigm not only provides appropriate language abstractions for developing data-intensive software, but also offers the ideal structure and predictability for effective energy management.

In the future, we plan to extend Green Streams to support dynamic adaptability. Instead of relying on off-line profiling and static inference, frequency selections can be adaptive to the fluctuations of the run-time and changing resource requirements.

Acknowledgments

We thank Tyler Stachecki for helping us set up the experiment environment. We thank Michael Gordon, Bill Thies, and anonymous reviewers for their useful suggestions. This work is supported by NSF CAREER Award CCF-1054515 and a Google Faculty Award.

References

[1] J. J. Tran, L. Cinquini, C. A. Mattmann, P. A. Zimdars, D. T. Cuddy, K. S. Leung, O.-I. Kwoun, D. Crichton, and D. Freeborn, "Evaluating cloud computing in the NASA DESDynI ground data system," in *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, ser. SECLOUD '11, 2011, pp. 36–42.

[2] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes, "A software architecture-based framework for highly distributed and data intensive scientific applications," in *ICSE '06*, 2006, pp. 721–730.

[3] R. Mahjourian, "An architectural style for data-driven systems," in *Proceedings of the 10th international conference on Software Reuse: High Confidence Software Reuse in Large Systems*, ser. ICSR '08, 2008, pp. 14–25.

[4] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[5] K. Fisher and R. Gruber, "PADS: a domain-specific language for processing ad hoc data," in *PLDI '05*, 2005, pp. 295–304.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI'04*, 2004.

[7] Nvidia, "Compute unified device architecture programming guide," *NVIDIA: Santa Clara, CA*, 2007.

[8] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007.

[9] W. R. Cook and S. Rai, "Safe query objects: statically typed objects as remotely executable queries," in *ICSE*, 2005, pp. 97–106.

[10] E. S. P. U.S. Environmental Protection Agency, "Report to congress on server and data center energy efficiency public law 109-431," 2007.

[11] J. Koomey, "Growth in data center electricity use 2005 to 2010," August 2011.

[12] J. Clause, C. Sahin, F. Cayci, I. L. M. Gutierrez, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *Proceedings of Workshop on Green and Sustainable Software (GREENS'12)*, 2012.

[13] Y. D. Liu, "Energy-efficient synchronization through program patterns," in *Proceedings of Workshop on Green and Sustainable Software (GREENS'12)*, 2012.

[14] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *PLDI '10*, 2010, pp. 198–209.

[15] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI'11*, Jun. 2011.

[16] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu, "Energy types," in *OOPSLA '12*, October 2012.

[17] F. Xie, M. Martonosi, and S. Malik, "Compile-time dynamic voltage scaling settings: opportunities and limits," in *PLDI '03*, 2003, pp. 49–62.

[18] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction," in *PLDI '03*, 2003, pp. 38–48.

[19] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *OSDI '94*. Berkeley, CA, USA: USENIX Association, 1994, p. 2.

[20] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: high-throughput stream programming in java," in *OOPSLA '07*, 2007, pp. 211–228.

[21] J. Zhou and B. Demsky, "Bamboo: a data-centric, object-oriented approach to many-core software," in *PLDI'10*. ACM, 2010, pp. 388–399.

[22] M. I. Gordon, "Compiler techniques for scalable performance of stream programs on multicore architectures," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, May 2010.

[23] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, pp. 1–34, March 2004.

[24] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL '95*, 1995, pp. 49–61.

[25] P. Haller and M. Odersky, "Scala Actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, no. 2-3, pp. 202–220, Feb. 2009.

[26] Google, "The Go language, http://golang.org/."

[27] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[28] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: a programming language for Ajax applications," in *OOPSLA '09*, 2009, pp. 1–20.

[29] A. Manjhi, S. Nath, and P. B. Gibbons, "Tributaries and deltas: efficient and robust aggregation in sensor network streams," in *SIGMOD '05*, 2005, pp. 287–298.

[30] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *SIGMOD '02*, 2002, pp. 49–60.

[31] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *PLDI '00*, 2000, pp. 242–252.

[32] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proceedings of the 1998 international symposium on Low power electronics and design (ISLPED)*, 1998, pp. 76–81.

[33] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 347–358.

[34] A. Benoit, P. Renaud-Goud, Y. Robert, and R. Melhem, "Energy-aware mappings of series-parallel workflows onto chip multiprocessors," in *2011 International Conference on Parallel Processing (ICPP)*, 2011, pp. 472 –481.

[35] Y. Yetim, S. Malik, and M. Martonosi, "EPROF: An energy/performance/reliability optimization framework for streaming applications," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 2012, pp. 769 –774.

[36] A. Rangasamy and Y. N. Srikant, "Evaluation of dynamic voltage and frequency scaling for stream programs," in *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF)*, 2011, pp. 40:1–40:10.