Elastic Scaling of Data Parallel Operators in Stream Processing

Scott Schneider[†,♠] Henrique Andrade[♠] Buğra Gedik[♠] Alain Biem[♠] Kun-Lung Wu[♠]

[†] Virginia Tech
Department of Computer Science
Blacksburg, VA, USA
scschnei@cs.vt.edu

[♠] Thomas J. Watson Research Center
IBM Research
Hawthorne, NY, USA
{hcma,bgedik,biem,klwu}@us.ibm.com

Abstract

We describe an approach to elastically scale the performance of a data analytics operator that is part of a streaming application. Our techniques focus on dynamically adjusting the amount of computation an operator can carry out in response to changes in incoming workload and the availability of processing cycles. We show that our elastic approach is beneficial in light of the dynamic aspects of streaming workloads and stream processing environments. Addressing another recent trend, we show the importance of our approach as a means to providing computational elasticity in multicore processor-based environments such that operators can automatically find their best operating point. Finally, we present experiments driven by synthetic workloads, showing the space where the optimizing efforts are most beneficial and a radioastronomy imaging application, where we observe substantial improvements in its performance-critical section.

1. Introduction

The area of distributed stream processing has received significant attention recently. The availability of large-scale affordable computational infrastructure is making it possible to implement large, real-world continuous streaming data analysis applications. The interest is also fueled by the existence of a plethora of *usable* academic and commercial stream processing systems [1], [2], [3], [4], [5], [6].

The implementation of continuous streaming applications where data is ingested from physical sensors (e.g., hurricane tracking stations) or consumed from business platform sources (e.g., trading information from stock exchanges), imposes interesting challenges. First, there is the *mindset* challenge. This is how we define the training of application developers and analysts alike to create and employ architectural patterns that are aligned with the *streaming* nature of the data. The design work includes ways of partitioning the incoming workload as well as the processing in functional components such that the applications can cope with the sometimes overwhelming rate of data ingest [7]. The second challenge relates to scalability and adaptation to computational infrastructure changes. In other words, how can application developers and administrators ensure that the processing infrastructure will cope with increased needs (e.g., adding new sensors for tracking weather changes) and variation in resource availability that happens once these long-running applications are deployed. Additional intelligence and appropriate abstractions in the programming model supporting these applications can help with the mindset challenge.

Another aspect is that, for streaming applications, *a priori* traditional capacity planning techniques [8], [9] can be of limited use. On one hand, spikes in data rates can happen and must be dealt with expeditiously at runtime (e.g., announcements by the Federal Reserve in the US usually affect trading patterns almost immediately). On the other hand, some of the typical streaming applications are hypothesis-driven (e.g., can I assess whether a competitor hedge fund is attempting to unload a particular asset?), which also implies spikes in processing needs. Clearly, distributed stream processing applications must include ways of *adapting* to runtime variations.

A more subtle aspect in designing streaming applications is the problem of placing its multiple pieces onto the runtime environment so that it can best utilize the computational resources. In many cases, data analysts and programmers have a good understanding of how they intend to structure the data analysis task. Previous work, including our own [10], [11], has shown that there is benefit in providing a language and infrastructure where there is explicit decomposition of the development effort in terms of a *logical* task (i.e., how a data analysis application should be implemented in terms of fundamental building blocks), and a *physical* task (i.e., how the logical task should be mapped onto the physical resources). Cognitively, the logical task is much closer to analysts and developers as it is in their domain of expertise. On the other hand, the physical task requires deeper understanding of processor architectures, networking, and interactions of other system components. In most cases, only well seasoned systems developers can do it, and even they are only effective when dealing with reasonably small applications [10]. That means that automated help is required. Presenting an environment where logical and physical tasks

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

are separated has the advantage of exposing knobs that can be tweaked at compile-time. This approach can lend itself to making adjustments of the computational plumbing to better suit the runtime environment [10]. On the other hand, as we articulated earlier, even such an approach is inadequate to cope with variations that happen at *runtime*. This indicates that there is value in providing adaptive runtime techniques. In particular, techniques that are transparent to developers such that the logical/physical task separation is preserved.

A final motivating aspect is the trend of packing a large number of processing cores in modern chips. Clusters of workstations (COWs) currently deployed in most high-performance computing installations typically have from 2 to 32 cores per node. The trend in chip design is to ramp up the number of cores to even higher numbers [12]. Considering such trend, it is imperative that middleware geared towards different data analysis processing tasks be able to effectively make use of these large number of cores. This indicates that the idea of *computational elasticity* must be directly supported by middleware and compilers alike.

For these reasons, we propose a strategy that permits portions of the computation (described as operators) that are data parallel to transparently make use of additional cores. This elasticity must come in response to changes in an operator's workload or in response to changes in the availability of processing cycles, because additional applications may come online and compete for the same computational resources.

The main contributions of this work are as follows:

1) We formalize what an *elastic* operator is in the context of stream processing and describe how it fits in the SPADE language, the programming language employed by IBM's System S, a large-scale stream processing middleware. An important aspect here is preserving the isolation of a *logical* view versus a *physical* view of the application in the sense that developers need not change their operator code to transform them into elastic operators.
2) We propose an adaptive algorithm for assessing changes in the incoming workload as well as changes in the availability of computational resources, providing *quick* reaction to these changes at runtime. These reactions cause the operator to tweak the amount of resources it uses, quickly stabilizing at a new performance level.
3) We demonstrate through synthetic benchmarks the space where elasticity is most beneficial as well as provide an assessment of how it affects the application in terms of (minimal) additional overheads.
4) We demonstrate experimentally how a performance-critical operator, central to a large-scale radioastronomy application, can substantially profit from the dynamic adaptation at runtime.

2. System S and SPADE

System S [13], [14], [15] is a large-scale, distributed data stream processing middleware under development at the IBM T. J. Watson Research Center. It supports structured as well as unstructured data stream processing and can be scaled to a large number of compute nodes. The System S runtime can execute a large number of long-running jobs, which take the form of data-flow graphs. A data-flow graph consists of a set of Processing Elements (PEs) connected by streams, where each stream carries a series of tuples. The PEs are containers hosting *operators* that implement data stream analytics and are distributed over the compute nodes. The compute nodes are organized as a shared-nothing cluster of workstations (COW) or as a large supercomputer (e.g., Blue Gene). The PEs communicate with each other via their input and output ports, connected by streams. PEs can be explicitly connected using name-based static stream subscriptions that are resolved at compile-time or through expression-based subscriptions that are established dynamically at runtime, upon the availability of streams that match the subscription expression. Besides these fundamental functionalities, System S provides several other services, including fault tolerance, scheduling and placement optimization, distributed job management, storage services, and security.

SPADE [11] is a language and a compiler for creating distributed data stream processing applications to be deployed on System S. Concretely, SPADE offers: (1) a language for flexible composition of parallel and distributed data-flow graphs, which can be used directly by programmers or sit under task-specific higher-level programming tools and languages (such as the System S IDE, StreamSQL dialects, or framework-specific languages [16] for data processing paradigms such as map-reduce); (2) a toolkit of type-generic built-in stream processing operators, which include all basic stream-relational operators, as well as a number of *plumbing* operators (such as stream splitting, demultiplexing, etc.); (3) an extensible operator framework, which supports the addition of new type-generic and configurable operators (UBOPs) to the language, as well as new user-defined non-generic operators (UDOPs) used to wrap existing, possibly legacy analytics; (4) a broad range of edge adapters used to ingest data from outside sources and publish data to outside destinations, such as network sockets, databases, and file systems.

Programming Model: The SPADE language provides a stream-centric, operator-based programming model. The stream-centric design implies a programming language where an application writer can quickly translate the flows of data from a block diagram prototype into the application skeleton by simply listing the stream data flows. The second aspect, operator-based programming, is focused on design-

ing the application by reasoning about the smallest possible building blocks that are necessary to deliver the computation an application is supposed to perform. The SPADE operators are organized in terms of domain-specific toolkits (signal processing, data mining, etc). In most application domains, application engineers typically have a good understanding about the collection of operators they intend to use. For example, database engineers typically design their applications in terms of the operators provided by the stream relational algebra [1], [17]. Data analysis and data mining programmers tend to employ signal processing and stream mining operators [18], [19].

A Code Generation Approach: A key distinction between SPADE and other stream processing middleware is its emphasis on *code generation*. This distinction enables us to address the issues we raised in Section 1. Specifically, hardware architecture designs are changing rapidly. This changing hardware forms the basis of large parallel processing facilities. However, such diversity makes it challenging for software developers to write applications that extract the best performance out of their computational resources. A code generation framework addresses these challenges by adapting the logical application to the physical specifics of a computational platform through simple application recompilation.

Given an application specification in SPADE'S intermediate language, the SPADE compiler generates *specialized* application code based on the computation and communication capabilities of the runtime environment. Currently, this specialization includes (1) *code fusion*, the ability to translate an application logically defined in terms of operators into a set of processing elements (PEs) such that multiple operators may be placed inside a single PE and the streams between them are converted into function calls [7]; (2) *vectorization*, where operations expressed in terms of vectors (vectors are basic types in SPADE) are translated into code that employs SIMD instructions using SSE or AltiVec instruction sets [20]; (3) *profiling*, where performance data collection employs the different mechanisms provided by a particular CPU (i.e., hardware counters) for obtaining metrics, which are later used for optimizing the placement of operators on processing elements; and (4) *operator elasticity*, where operators declared as elastic are placed in a container that can dynamically adjust its processing level as a function of available cycles and incoming workload. The focus of this work is the last of these items, which involves runtime adaptation, in addition to the compile time optimizations already performed by SPADE. A detailed discussion of this idea is the topic of Section 3.

3. Design

Standard SPADE operators perform their computations sequentially. As tuples arrive, they are processed by a single flow of control. In this section, we describe a design that allows such operators to perform their computations in parallel, while also dynamically adapting their level of parallelism to obtain the best performance on a particular node.

This approach, which we refer to as *operator elasticity*, applies to operators that are amenable to data parallelism. In particular, operator elasticity works best with operators that are pure functions and maintain no state in between processing different tuples. Stateful operators that are thread-safe do work in this framework, but their scalability may be limited by the granularity of synchronization they use to maintain consistent state. We present an example of such an operator as an alternate design in Section 4.

Our current implementation also assumes that the results from tuples do not have to be submitted downstream in the same order that they arrive. The elastic operator approach can still be applied in applications that have ordering constraints, but the implementation requires a mechanism to track tuple ordering, introducing additional costs.

Architecture. As shown in Figure 1, we have introduced new threads and data structures into a SPADE operator to make it elastic.

- **Dispatch thread**: The dispatch thread is the focal point for managing the distribution of incoming work in the form of tuples carried by the operator's input streams. This thread replaces the main thread of execution originally used by SPADE. It is responsible for receiving tuples from upstream operators, dispatching them into a global work queue, and periodically throttling the thread level to achieve the best instantaneous performance. The dispatch thread can increase the degree of parallelism by creating new workers or waking up old ones, and it can decrease parallelism by putting workers to sleep.
- **Work queue**: This queue enables the reception of tuples to happen independently of their processing. The dispatch thread places tuples to be processed in this queue.
- **Worker threads**: A worker thread repeatedly pulls the next available tuple off of the global work queue and processes it. The operator is parallelized by having multiple worker threads, each processing their own tuples. Worker threads also check if they have been told to go to sleep by the dispatch thread.
- **Alarm thread**: The alarm thread periodically wakes up and tells the dispatch thread it is time to reevaluate the thread level. For the experiments in Section 5, this period is set to 1 second[1]. If this period is too short, then performing the calculation itself can impact performance. If it is too long, then the system cannot quickly

1. The default time slice for a process in the Linux kernel is 100 milliseconds. A period of 1 second allows worker threads to get 10 time slices before the dispatch thread evaluates their performance.

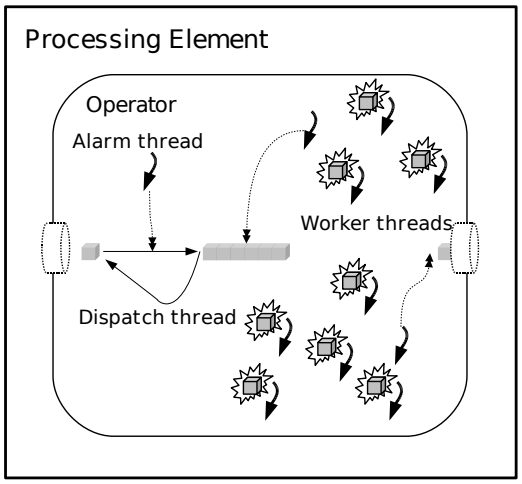


Figure 1: Elastic operator architecture.

adapt to changes. This work focuses on adjusting the level of parallelism in a single node. We assume that for time periods significantly longer or shorter than 1 second, the adjustment will be handled by other means.

There are two designs that are present in other similar systems that we explicitly decided against: private work queues and work stealing [21]. Private work queues alleviate contention on global data structures, which can reduce the overhead costs of parallelization. The main benefit of reducing this cost is that we can exploit a smaller granularity of parallelism. However, it also introduces the problem of load balancing. The classic solution to load balancing with private queues is to allow threads to steal work from other threads. The granularity of parallelism that we aim to exploit is coarse enough that the overhead from a global queue does not dominate. This was revealed by earlier studies we conducted. Ultimately, using a global queue allows us to get the best load distribution. Further, we are also in a system with a constant stream of data. If we assume that tuples from upstream operators arrive in steady fashion (as is the case in an oversubscribed system), then there would be no need to steal work even if there were private queues.

Rates are measured in tuples per second. We do not measure rates in bytes per second because there is a weak correlation between the size of a tuple and the time it takes to process it. If this assertion seems counter-intuitive, consider that the SPADE language and runtime system are meant to work with any arbitrary application that can be expressed using a streaming model. Consequently, small tuples can still require large computations. The best measure of performance in our computational model is the rate at which whole computations can be carried out, which is independent of the size of the messages that initiated the computations.

The tuples per second rate is calculated by maintaining a sliding window of timestamps for how many total number of tuples have entered the operator. We average the most and least recent entries for a rate that represents the recent past.

We chose tuples processed per second as our metric because in a streaming application which will run indefinitely, it is the best indication of total system performance. Locally observed rates can indicate what is going on in the rest of the system, as the rate at a single node is the result of more than just how fast that node can process its tuples. Local rates depend on how quickly upstream nodes can feed it tuples, and back-pressure from downstream nodes will prevent a particular node from submitting more tuples.

Elastic Operator Algorithm. When the dispatch thread reevaluates the thread level, the goal was to both aggressively seek out higher levels of parallelism, yet recognize when increased parallelism is detrimental. Further, we also want it to adjust parallelism at runtime based on measured performance, even if the workload of the application or load on the system changes. The algorithm presented in Figure 2 meets these requirements.

The variable n in Figure 2 represents the active thread level. Threads are created, suspended and woken up, but they are never destroyed. The active thread level is the number of threads that are actively processing tuples; it does not include suspended threads. Never destroying threads is a deliberate design decision. Thread creation is relatively expensive—it is the cost of a pthread_create(), which on a Linux system is the same as a process fork. Waking up a suspended thread is cheap; the dispatch thread signals a condition variable and the worker wakes up from blocking on that condition. Suspended threads also have a negligible impact on overall system performance, even if they are around for a long time. They do not impact system scheduling and the only resource they consume is their local call stack. These reasons lead us to conclude that suspending threads, rather than destroying them, was the best design. In Section 5, we present experiments which show the thread level changing frequently in response to both external system load and the influence of other elastic operators.

For each thread level, we record both the peak rate seen at that level (peak) and the last seen rate at that level (last). The peak rates are used to detect performance decline. Comparing peak rates across thread levels is used to detect the best thread level on a stable system with a steady workload. Comparing the current rate with the peak rate at that thread level is used to detect when the system is no longer stable, either due to a change in workload or a change in the system.

Initially, the algorithm continually increases the thread level as long as the peak performance increases, creating threads along the way (lines 28–30). Once a thread level is reached at which there was no significant improvement

```
int n = 0;
bool peaking = false;
ThreadList threads;
DoubleList peak;
DoubleList last;

bool less(double a, double b)
{
  return ((b - a) / a) >= TOLERANCE;
}

void set(DoubleList& p, int n, double c)
{
  // New peak.
  if (c > p[n]) {
    if (less(p[n], c)) {
      // Invalidate peaks.
      for (int i = n+1; i < p.size(); ++i) {
        p[i] = INFINITY;
      }
    }
    p[n] = c;
  }

  // Decay peak rate.
  else {
    p[n] -= p[n] * DECAY_RATE;
  }
}
```

```
1  double curr = calcAverageRate();
2  set(peak, n, curr);
3  last[n] = curr;
4
5  if (peaking) {
6    if (curr < last[n+1]) {
7      threads[n+1]->wakeup();
8      ++n;
9    }
10   peaking = false;
11 }
12
13 // Infer a busy system.
14 else if (less(curr, peak[n]) || (n && less(peak[n], peak[n-1]))) {
15   // Never suspend thread 0.
16   if (n != 0) {
17     threads[n]->suspend();
18     --n;
19     peaking = true;
20   }
21 }
22
23 // If we only have one active thread, or if peak performance at
24 // the level below us is significantly less than current level,
25 // then maybe we can still improve performance.
26 else if (!n || less(peak[n-1], peak[n])) {
27   // Probe higher.
28   if (n+1 == threads.size()) {
29     threads.create();
30     ++n;
31   }
32   // Only wakeup thread if it performed better.
33   else if (peak[n] < peak[n+1]) {
34     threads[n]->wakeup();
35     ++n;
36   }
37   // else, stable from above
38 }
39 // else, stable from below
```

Figure 2: The variables used to infer the state of the system and their initial conditions are show in the left column. The Elastic Operator algorithm is shown in the right column.

compared to the last thread level, there is *stability from below* (line 39). The assumption with this approach is that if there is a performance increase going from thread level $N-1$ to N, then there will probably be a performance increase with thread level $N+1$. The first time this is not true, we settle at that thread level. We consider this *stability from below* because we infer stability by only looking at the thread levels below the current one. If performance has not improved significantly from thread level $N-1$ to N, then it is likely that thread level $N+1$ will either harm performance or remain the same. In either case, the best thing to do is to stay at thread level N.

The other kind of stability, *stability from above*, is achieved when the peak rate from thread level $N-1$ indicates that the performance will probably improve if the thread level is increased (i.e., there exists an upward trend; line 26), but the peak rate from thread level $N+1$ indicates otherwise (line 33). This situation can occur because at some previous point, thread level $N+1$ was explored, but not chosen because it harmed performance.

If external load is placed on the system or if the workload itself changes, the measured rate may fall significantly below the peak observed rate (line 14). When this happens, the algorithm tries decreasing the thread level and asserts that it actually increases performance (lines 16–19). Note that even on a busy system, decreasing the thread level might harm performance. Our goal is to maintain the highest possible performance for the application, not necessarily be a fair consumer of system resources. This assumption means that we rely on fair scheduling at the kernel level. For this reason, we decrease the thread level, but enter a peaking state (line 19). We only settle on the lower thread level if, during the next thread throttling period, the measured rate is better than the higher thread level (lines 5–10). Note that in this case, we use a strict comparison (line 6) in order to prevent performance from gradually creeping downwards. Once pressure on the system is relieved, the algorithm will again seek out higher thread levels that yield better performance.

The external load placed on an elastic operator can actually come from another elastic operator. If multiple instances of this algorithm are running on the same node, they will compete for resources, even though they are part of the same application. However, since they are both greedily trying to maximize their performance—not system utilization—we

postulate that they will end up with a fair configuration that still benefits total application performance. We will experimentally show whether this claim holds in Section 5.

There is a fundamental conflict between maintaining stability and continually trying to improve performance by changing thread levels. Once stability is reached—either from above or from below—the thread level is not "set." The fitness of this decision is constantly reevaluated. While reaching stability is one of our goals, we still want the system to be nimble; it should react quickly to changes in the system and the workload. For this reason, the algorithm does two things to the observed peak rates: decay and invalidation (line 2).

If the current rate is larger than the peak rate for a thread level, then the current rate becomes the peak rate. However, if the current rate is less than the peak rate, we decay the peak rate by a small percentage. This technique allows us to converge on a stable thread level faster than always maintaining the peak. It also means the algorithm is less susceptible to anomalous effects; e.g., queueing artifacts during application initialization can result in inflated rates. Rate decay ensures that only recent history, not ancient history, is used for making decisions. However, if the current rate is significantly larger than the peak rate, then the algorithm assumes that there has been a fundamental change in either the workload or the system itself. In this case, we invalidate the peak rates for all thread levels above the current one, which spurs renewed probing into those levels. This technique facilitates adaptation and enables recovery from an overloaded system.

4. Case Study – Radio Imaging

To assess the effectiveness of the Elastic Operator algorithm, we will carry out experimental evaluation (see Section 5) using synthetic workloads as well as a radioastronomy application, being developed as part of the Australian Square Kilometre Array Pathfinder Project (ASKAP) [22]. Here we briefly describe the structure of a radio imaging application, developed in collaboration with ASKAP.

Synthesis radio imaging is the process of transforming radio data collected from an array of antennas (the interferometer) into a visible image. A key component in the radio imaging is the process of *gridding*, which transforms spatial frequency data into a regular grid on which Fast Fourier Transform (FFT) can be carried out. Gridding is the most computationally expensive component of the imaging process.

Considering 3-D coordinates (u, v, w) measuring the distances in wavelengths between two antennas and two directional angles to a point-object in the sky (l, m), the sky image $I(l, m)$ is related to the output of the interferometer $V(u, v, w)$, called visibility, by the following equation [23]:

$$V(u,v,w) = \int \frac{I(l,m)}{\sqrt{(1-l^2-m^2)}} G_w(l,m) e^{-2\pi i[ul+vm]} dldm \quad (1)$$

where kernel $G_w(l,m) = e^{-2i\pi[w(\sqrt{1-l^2-m^2}-1)]}$. Equation (1) is the Fourier transform of the product of $G_w(l,m)$ and $I(l,m)$, which is equivalent to the Fourier transform of $V(u,v,w=0)$.

The visibility is measured at a pre-defined location determined by the architecture of the antenna array, resulting into the sampled visibility:

$$V_s(u,v,w) = V(u,v,w)\Delta(u,v,w) \quad (2)$$

where $\Delta(u,v,w)$ is the M-points, 3-D sampling function at location u_j, v_j, w_j:

$$\Delta(u,v,w) = \sum_{j=1}^{M} \delta(u-u_j, v-v_j, w-w_j) \quad (3)$$

Gridding is done by convolving the sampled visibility with a function $g_w(u,v)$, which is the inverse Fourier transform of the kernel $G_w(l,m)$, where the kernel is sampled onto a regular grid:

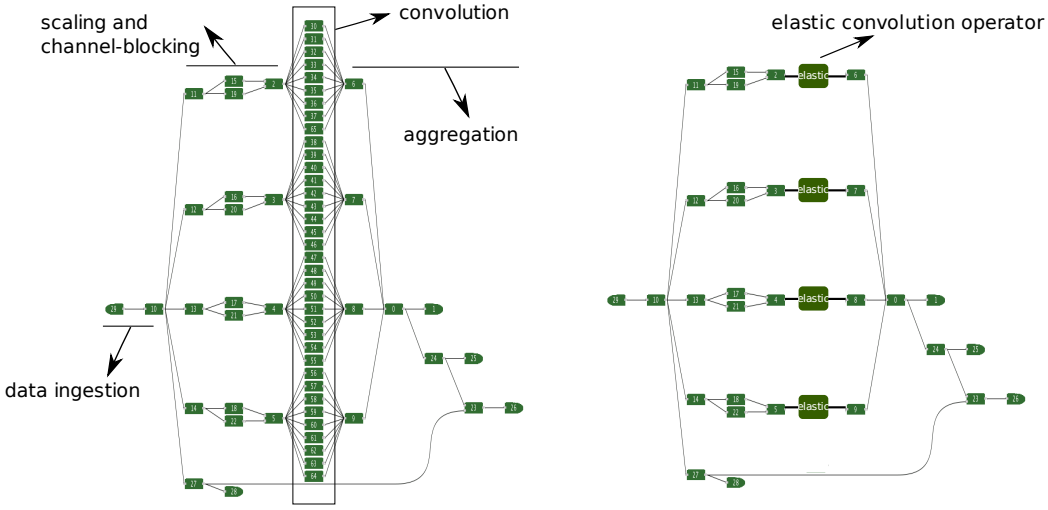
$$V_g(u,v,w) = [g_w(u,v) \otimes V_s(u,v,w=0)]\, \Omega(u,v) \quad (4)$$

Where $g_w(u,v)$ is the inverse Fourier transform of $G_w(l,m)$ and the $\Omega(u,v)$ is a Comb function (sum of regularly spaced Dirac functions). Note that the convolution function depends of the third coordinate w. In practice, a finite set of $g_w(u,v)$ is used by quantizing the w-axis into a finite number of planes. Also, note that the above equation transforms the 3-D data into a 2-D grid, thus creating the resulting image.

The application (seen in Figure 3), as implemented in SPADE, comprises the following:

Data ingestion: The visibility data is typically collected from the antennas after long hours of observations in the form of complex-valued $V(u,v,w)$ readings. For the purposes of performance evaluation, data ingestion is simulated in System S by two SPADE operators. The first operator generates the coordinates (u,v,w) as random values between 0 and 1 and a sequence number seq—the data is not read from a file, but generated at runtime. The second operator is a Functor[2] that generates the visibility (i.e., the output of the interferometer) as a complex number (two real-valued number) from their coordinates. The process generates a set of complex-valued samples with real-valued coordinates $(u,v,w)[i]$, where $i = 0...N-1$).

2. The Functor is a SPADE operator that can perform data transformations through algebraic operations as well as projections and data filtering.



(a) Reference implementation

(b) Elastic implementation

Figure 3: The Radio Imaging application

Scaling and channel-blocking: Scaling is the transformation of the coordinate $(u, v, w)[i]$ units from meter-distances into wavelengths within pre-selected frequencies of analysis. For a frequency f, the coordinates are converted into wavelengths by a Functor performing the following vectorial transformation:

$$(u_f, v_f) = (u, v) * f/CellSize; \quad w_f = f * w/wCellSize$$

where $CellSize$ and $wCellSize$ are pre-selected parameters that define the required granularity of analysis in the 3-D space. This scaling process is carried out for each frequency of interest (16384 in total), generating a 16384 dimensional data for each incoming tuple. Next, scaled data is grouped into block of frequencies and each block is processed independently.

Indexing: Within each block of frequencies, data is indexed for data distribution and partitioning. Two types of indexing are performed. Indexing to map data to sectors in the convolution matrix and indexing to the final grid. Indexing is carried out by two Functors running in parallel, and then joined back into single set of tuple using a Join operator[3].

Convolution: The data is convolved with the matrix using a Convolution user-defined operator (UDOP). The convolution matrix is divided into $L \times L$ sub-areas, where each sub-area is convolved separately. Each of these areas generate a local grid that holds the results of convolving the data with that specific matrix region.

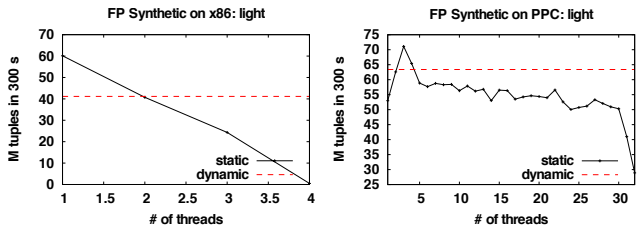
Aggregation: All local grids from each block of frequencies are summed back into a final image grid. This is done by two set of operators. The first set aggregates[4] the local grid corresponding the matrix sub-areas, resulting into a grid for that particular block of channels. Next another Aggregate operator sums up these local grids into a final grid, producing the final image.

Note that in Figure 3(a), it can be seen that we have multiple instances of the Convolution operator. This is another degree of freedom in parallelizing this application as it allow us to have concurrent instances of the same operator working on disjoint regions of the convolution matrix. As will be seen in Section 5, we implemented the Convolution operator (Figure 3(b)) as an elastic operator as its operations are commutative and associative and can be carried out in parallel, but we also employed multiple instance of that operator. We used this method as a means of partitioning the work and, hence, reducing contention in accessing the shared matrix data structures.

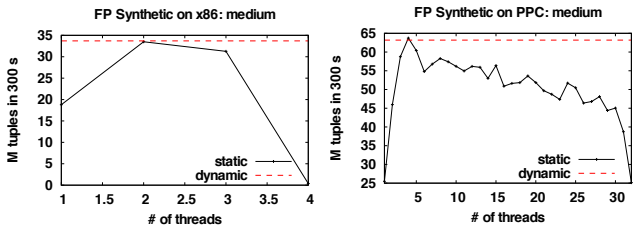
We also experimented with an alternate parallelization strategy. What limits parallelism in this application is that, in the end, one matrix must be the aggregate of all matrix operations. Our initial parallelization maintained one copy of the matrix and each operator updated that matrix using lock-free sychronization (specifically, wait-free compare and swaps). While this implementation was both correct and in parallel, it was in fact slower than the sequential version—the synchronization overhead was too great for the parallelism to overcome. The design depicted in Figure 3 and used in the experiments in Section 5 maintains separate, local copies of the matrix inside each operator. Synchronization is delayed

3. A Join operator performs a stream-based relational join, correlating tuples originated from two separate streams.

4. SPADE supplies an Aggregate operator that can be used for regular group-by aggregations.

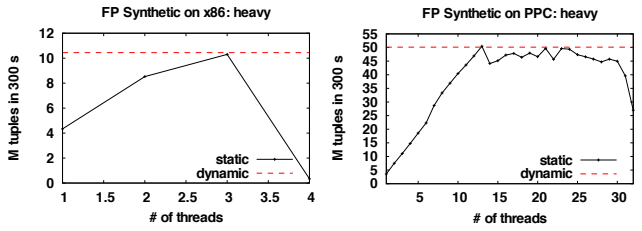


(a) Light workload: AMD (left-side), Power5+ (right-side)

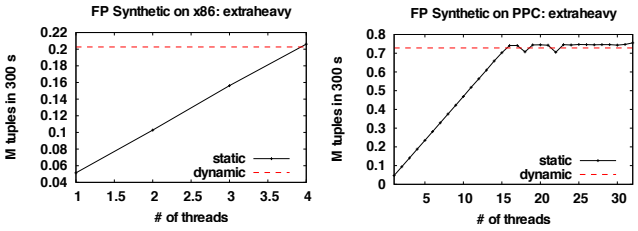


(b) Medium workload: AMD (left-side), Power5+ (right-side)

Figure 4: Algorithm stability with light and medium synthetic workloads



(a) Heavy workload: AMD (left-side), Power5+ (right-side)



(b) Extra heavy workload: AMD (left-side), Power5+ (right-side)

Figure 5: Algorithm stability with heavy and extra heavy synthetic workloads

and done in stages, allowing the overall execution time to benefit from the parallelism. As with other data processing domains, how an application is parallelized matters.

5. Results

We ran two sets of experiments to analyze the effectiveness of the Elastic Operator algorithm: experiments using a synthetic benchmark and experiments using a real application.

The experiments with the synthetic benchmark had two main purposes. First, to verify that the control algorithm was able to seek the best operating point irrespective of how much work that had to be carried out per tuple. Second, to assess whether the additional overhead imposed by the Elastic Operator algorithm was small, even when compared to a hand-tuned application. Moreover, using synthetic workloads, we also wanted to make sure that the algorithm would *quickly* adapt to changes in the availability of computational resources and *tweak* its behavior in light of changes in the availability of computing cycles.

The second batch of experiments was carried out using the Radio Imaging application, where we focused our analysis on the Convolution operator. Our primary aim was to demonstrate how effective elasticity is for a real application. Furthermore, we wanted to verify experimentally that even though multiple elastic operators would be independently seeking out their highest operating point (in this case, defined as operator throughput) that, globally, the application would settle on its highest throughput level.

We employed two machines in our tests. A quad-core machine running at 2.6 GHz (specifically, it has two dual-core AMD Opteron 2218 processors) and a 16-core Power5+ machine, where each core runs at 1.9 GHz with simultaneous multithreading (SMT) capabilities. In all cases, we were running RedHat Linux with a stock installation of System S and SPADE, including the compiler and code generation support for generating elastic operators.

Synthetic Study: Figures 4 and 5 summarize our results with the synthetic benchmarks on the AMD and Power5+ machines. In these experiments, we controlled the amount of computation by varying the number of floating point operations carried out on behalf of each incoming tuple. Specifically, we employed a *light* workload where 1 floating point operation was carried out per incoming tuple. We also used a *medium* workload—1000 floating point operations per tuple; a *heavy* workload—10,000 floating point operations per tuple; and an *extra heavy* workload—1 million floating point operations per tuple. These experiments also capture the amount of overhead the Elastic Operator algorithm incurs as it performs data copying, queue and thread management, among other bookkeeping operations required by the algorithm. This overhead is most apparent with the *light* and *medium* workloads, where the granularity of work is sometimes too small to overcome the synchronization and bookkeeping overhead. When this happens, adding more threads actually decreases throughput. The Elastic Operator algorithm detects this decrease in throughput and backs off the level of parallelism accordingly.

In Figures 4 and 5, we plot a straight line (labeled *dynamic*) with the throughput that was observed throughout the experiment when the Elastic Operator was employed.

It can be seen in Figure 4 that once the Elastic Operator

algorithm settles on its best and steady operating point, that it is at or very near the best static configuration. Again, we emphasize that picking the *right* static configuration requires manual tuning. We can also see that for *light* and *medium* workloads, not too many additional threads are required as the ratio of computation to the extra overhead imposed by the Elastic Operator algorithm has diminishing returns. Note that this is discovered quickly during probing. More interestingly, in Figure 5, when the ratio of computation per tuple is much higher, the benefits of elasticity become evident. We notice that the steady throughput enjoyed by the elastic implementation corresponds to the maximum throughput seen by the static implementation as it uses a number of threads close to the physical number of cores in the machine (close to 4 for the AMD machine and 16 for the Power5+ machine). For the Power5+ machine, we see that in those circumstances the SMT capability does not help as the thread level settles on 16, rather than 32. In both cases, using more than the number of physical cores present is not helpful because the operator does mostly floating point operations.

The Elastic Operator algorithm is fundamentally a simple control algorithm. We wanted to study how quickly and accurately the dynamic adjustments happen in reaction to changes in the availability of computational cycles from scarcity to abundance and other points in between. This new experiment simulates how a runtime system hosting multiple applications would behave. We wrote an application that can as, time passes, emulate spikes in utilization by pegging one or more *logical* cores (recall that the Power5+ machine has 32 of those). Figure 6 depicts timelines where we track both the effective throughput and multithreading level for the *heavy* workload configuration. Not surprisingly, Figure 6(a) shows an inverse correlation between spikes in external utilization and dips in throughput. But, more interestingly, we can clearly see in Figure 6(b) the probing up and down by the Elastic Operator algorithm and its tracking the external load curve.

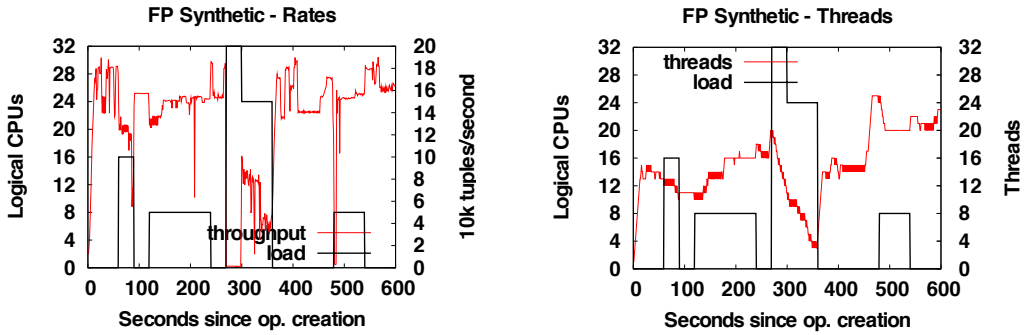
Radio Imaging Application: Our next set of experiments employed the Radio Imaging application described in Section 4, running on the Power5+ machine. Here our aim was to observe the performance of elastic operators in a real application. Furthermore, we wanted to visualize the effects of having a collection of elastic operators individually seeking out their best operating point at the same time. Figure 7(a) shows the throughput for the Elastic Operator in the Radio Imaging application. It settles on a throughput level that is within less than 10% of the best static configuration. We should point out that in Figure 7(a), we show the number of threads *per instance* of the Convolution operator on the x-axis, rather than the total number of threads for all operators.

We also observe that competition among independent copies of the Elastic Operator algorithm results in good overall system performance. We broke down the multithreading level and throughput per instance of the elastic Convolution operator—recall that we have 4 copies of the operator (see Figure 3(b)). While they are all sharing the same physical resources, they are independently pursuing their best operating point (Figure 7(b)). The reliance on the fairness of the Linux scheduler ensures that the operators eventually settle on doing about the same amount of work, steadying the overall throughput a short time into the experiment (see the *total* curve in Figure 7(b)). Likewise, about the same number of threads (Figure 7(c)) are steadly active on the behalf of each instance of the elastic operator.

Figures 6(b) and 7(c) show thread levels changing in real time. The thread level in Figure 6(b) changes in response to external system load. This experiment shows that both the Elastic Operator algorithm and the thread synchronization mechanisms can respond quickly to changes in the system. The eventual stability of the individual—but interacting—thread levels in each Elastic Operator in Figures 7(b) and 7(c) shows that independent operators, each trying to maximize their own performance in a closed system, will eventually settle on a configuration good for the entire system.

Finally, it should be noted that the system support for the Elastic Operator algorithm changes the default threading model used by the SPADE code generator. In the default model, each independent processing element starts a new thread and, independently, each new source edge adapter responsible for ingesting the external data into the streaming application also starts one. The processing of tuples is driven by that thread and the flow of execution is such that each tuple causes a depth-first traversal of the operator flowgraph. From the onset, we were surprised to observe that the elastic implementation running with a single thread in its static configuration produced a 2-fold improvement in speedup. This is seen as the *reference* curve in Figure 7(a). This occurrence is a beneficial side effect of decoupling operator data ingestion from data processing. Ultimately, with 4 elastic operators, we obtained a 5-factor throughput improvement over the original implementation.

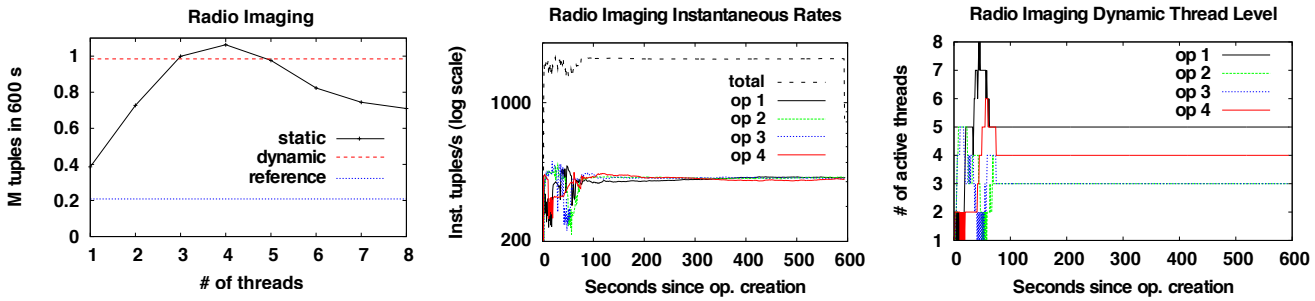
Lessons Learned and Critique: While we originally designed the operator abstraction in SPADE to be the smallest granular computational block in a streaming application, we have seen that decoupling the operator's internal processing from data ingestion is particularly beneficial for data parallel operations. We have seen that even for modestly large computations, elasticity is a win. But, more important is the fact that the Elastic Operator algorithm performs the dynamic tweaks automatically and transparently. The runtime performance optimization task, a potentially time-consuming undertaking and, typically, outside of the area of expertise of many application developers, would otherwise have to be



(a) External load and throughput correlation

(b) External load and multithreading level

Figure 6: Dynamic adjustments to external loads



(a) Static and dynamic results

(b) Per operator instance throughput

(c) Per operator instance thread level

Figure 7: Radio Imaging Application

done manually for every performance-hungry application. We have experimented with adding more information for making adaptation decisions, from longer histories to more control knobs. Ultimately, we verified that the simplicity of the current algorithm ensures low overhead and results in substantial throughput improvements in practice.

6. Related Work

Stream processing has been an active research area over the last few years. In the relational data processing world, frameworks such as STREAM [1], Borealis [2], StreamBase [3], TelegraphCQ [4], among others, have focused on providing stream processing middleware and, in some cases, declarative languages for writing applications. On the programming language side, approaches such as StreamIt [5] and the Aspen language [6] share some commonalities with SPADE–the most important being the philosophy of providing a high-level programming language, shielding users from the complexities of a distributed environment. Another similar approach is Pig Latin [16].

However, many distinctions exist when contrasting other works with our basic principles in terms of language and systems design. In addressing this point, we will focus the rest of the discussion on ideas for maximizing computational performance for streaming and non-streaming applications.

In terms of streaming and distributed middleware, we have observed three distinct approaches in the literature. First, there is the *deterministic scheduling* approach. In this case, operators declare *a priori* their communication patterns such that a steady state schedule can be computed at compile/planning time. This approach is used by, for example, StreamIt [5]. The limitation here is that in many cases, one cannot predict traffic rates beforehand nor can it be assumed that a single application will be running alone. Both issues are particularly true to the application domains targeted by System S [15]. Second, there is the *compile-time/multiple-copy* approach. In this case, experience in running/tuning an application in a particular platform provides enough information to an application developer to decide how many copies of a particular data-parallel segment should be spawned in the production environment. DataCutter [24], with its *transparent filter copies*, is an example of such an idea. It should be noted, however, that this is a common design pattern in parallel computing, in general—where it is known as a a master/slave configuration. The two approaches described so far rely on compile-time decisions, with minimal runtime adaptation support (e.g., DataCutter can dynamically assign work to the different transparent filter copies, but not change the number of filter copies at runtime).

This brings us to *runtime adaptation* techniques. An

example of these techniques is implemented in the System S Nanoscheduler [25]. In this case, the goal is to maximize the weighted throughput of an application by dynamically adjusting to traffic bursts. In our case, we aim at maximizing application throughput by ensuring that all elastic operators can seek their highest operating point possible, automatically, as seen in Section 3. In SPADE'S case, we rely on code generation to only include this capability if it is deemed useful, rather than embedding it in the communication substrate, as done in an earlier version of System S that employed the Nanoscheduler. And, this leads to the general paradigm of *auto-tuning*. An earlier approach that illustrates the auto-tuning idea was carried out in the Atlas project [26], which was aimed at automatic generation and optimization of linear algebra functions for a particular hardware architecture. Atlas is only one example of auto-tuning; the literature on compiler optimization has many more and we refer the reader to an example of exploiting this approach to optimizing the tiling of parallel *for* loops [27]. Finally, we point out that Hadoop [28], a map-reduce framework, is an example of runtime adaptation that is closer in spirit to our work. When spawning additional mapper or reducer tasks, it does so as the need arises. The main difference to our work is the fact that we are addressing stream processing rather than map-reduce batch jobs; where there is little *a priori* knowledge of the incoming workload.

7. Conclusions

In this work, we have demonstrated the idea of operator *elasticity* in the context of streaming middleware. The importance of employing runtime elasticity for streaming applications is that not only do traffic patterns vary, but resource availability fluctuates as these long-running applications perform their data analysis tasks while coexisting with other applications that share the same computational environment.

We have shown experimentally that the Elastic Operator algorithm generally finds the best operating level, has low overhead, and quickly adapts to changing conditions. More importantly, in situations where a collection of elastic operators is employed, our algorithm, in conjunction with the operating system's fairness policy, ensures that all operators run at the best efficiency level. This is important because it globally speeds up the whole application, under the assumption that the application's physical layout cannot be changed at runtime.

As we have stated, the aim for providing such capability in SPADE was to decrease the amount of guesswork and calibration time typical developers have to allocate as they deploy their applications on different platforms and runtime infrastructures. Recently, we have also looked at using code generation techniques for transparently specializing the code that is issued for an operator so that it can make use of SIMD instructions should they be available on the computational nodes hosting the applications [20].

In the near future, we plan to extend the concept of elasticity to go beyond a single multicore node, such that we can better leverage distributed resources. We also aim to address other kinds of parallelism common in streaming applications such as *task* parallelism and *pipelining*.

References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: The Stanford stream data manager," *IEEE Data Engineering Bulletin*, vol. 26, 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *Proceedings of Conference on Innovative Data Systems Research (CIDR 05)*, 2005.

[3] "StreamBase Systems," http://www.streambase.com.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Proceedings of Conference on Innovative Data Systems Research (CIDR 03)*, 2003.

[5] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the International Conference on Compiler Construction (CC 2002)*, April 2002.

[6] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and exploiting concurrency in networked applications with aspen," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2007)*, 2007.

[7] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu, "Scale-up strategies for processing high-rate data streams in System S," in *Proceedings of the International Conference on Data Engineering (ICDE 2009) – to appear*, 2009.

[8] D. A. Menascé and V. A. F. Almeida, *Capacity Planning For Web Performance*, 1998.

[9] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling*, 2000.

[10] B. Gedik, H. Andrade, and K.-L. Wu, "A code generation approach for optimizing high performance distributed data stream processing," in *Proceedings of the USENIX Annual Technical Conference (USENIX 2009) – submitted for publication*, 2009.

[11] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S declarative stream processing engine," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD 2008)*, 2008.

[12] P. Gepner and M. F. Kowalik, "Multi-core processors: New way to achieve high system performance," in *Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC 06)*, 2006, pp. 9–13.

[13] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: A distributed, scalable platform for data mining," in *Workshop on Data Mining Standards, Services and Platforms (DM-SSP 06)*, Philadelphia, PA, 2006.

[14] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and

C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the Stream Processing Core," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD 2006)*, 2006.

[15] K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang, "Challenges and experience in prototyping a multimodal stream analytic and monitoring application on System S," in *Proceedings of the International Conference on Very Large Data Bases Conference (VLDB 2007)*, 2007.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD 2008)*, 2008.

[17] J. D. Ullman, *Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[18] G. Hulten and P. Domingos, "VFML – a toolkit for mining high-speed time-changing data streams," http://www.cs.washington.edu/dm/vfml, October 2003.

[19] D. S. Turaga, O. Verscheure, J. Wong, L. Amini, G. Yocum, E. Begle, and B. Pfeifer, "Online FDC control limit tuning with yield prediction using incremental decision tree learning," in *Sematech AEC/APC, 2007*, 2007.

[20] H. Wang, H. Andrade, B. Gedik, and K.-L. Wu, "Using vectorization through code generation for high throughput stream processing applications," in *Proceedings of the ACM International Conference on Supercomputing (ICS 2009) – submitted for publication*, 2009.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1998)*, 1998, pp. 212–223.

[22] "The Australian Square Kilometre Array Pathfinder," http://www.atnf.csiro.au/projects/askap.

[23] T. Cornwell, K. Golap, and S. Bhatnagar, "W projection: A new algorithm for wide field imaging with radio synthesis arrays," in *Proceedings of the Astronomical Data Analysis Software and Systems XIV ASP Conference Series*, vol. 347, December 2005, p. 86.

[24] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, "DataCutter: Middleware for filtering very large scientific datasets on archival storage systems," in *Proceedings of the 8th Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, College Park, MD, Mar. 2000.

[25] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," Lisbon, Portugal, July 2006.

[26] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[27] L. Renganarayanan and S. Rajopadhye, "Positivity, posynomials and tile size selection," in *Proceedings of the ACM/IEEE SC Conference (SC 2008)*, Austin, TX, November 2008.

[28] "Hadoop," http://hadoop.apache.org.