DANBI: Dynamic Scheduling of Irregular Stream Programs for Many-Core Systems

Changwoo Min[†‡] and Young Ik Eom[†]
[†]Sungkyunkwan University, Korea
[‡]Samsung Electronics, Korea
{multics69, yieom}@skku.edu

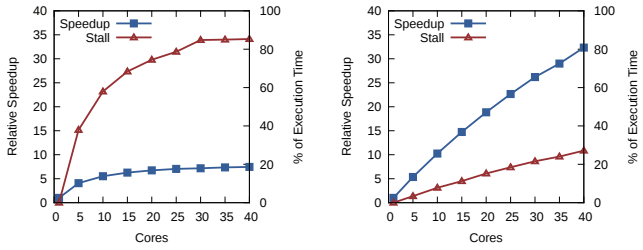
***Abstract*—The stream programming model has received a lot of interest because it naturally exposes task, data, and pipeline parallelism. However, most prior work has focused on static scheduling of regular stream programs. Therefore, irregular applications cannot be handled in static scheduling, and the load imbalance caused by static scheduling faces scalability limitations in many-core systems. In this paper, we introduce the DANBI[1] programming model which supports irregular stream programs and propose dynamic scheduling techniques. Scheduling irregular stream programs is very challenging and the load imbalance becomes a major hurdle to achieve scalability. Our dynamic load-balancing scheduler exploits producer-consumer relationships already expressed in the stream program to achieve scalability. Moreover, it effectively avoids the thundering-herd problem and dynamically adapts to load imbalance in a probabilistic manner. It surpasses prior static stream scheduling approaches which are vulnerable to load imbalance and also surpasses prior dynamic stream scheduling approaches which have many restrictions on supported program types, on the scope of dynamic scheduling, and on preserving data ordering. Our experimental results on a 40-core server show that DANBI achieves an almost linear scalability and outperforms state-of-the-art parallel runtimes by up to 2.8 times.**

Keywords*—*Stream Programming, Software Pipelining, Scheduling, Load Balancing, Irregular Programs

I. Introduction

The prevalence of multi-core processors has renewed interest in parallel programming models and runtimes, such as StreamIt [1], OpenCL [2] , Cilk [3], TBB [4], and Galois [5]. Also, the application types running on the processors have been expanded from regular applications such as scientific simulations to irregular applications such as computer graphics and big data analysis [5], [6], [7].

The stream programming model, such as StreamIt, has been extensively studied because it naturally exposes task, data, and pipeline parallelism [1]. In the stream paradigm, a program is modeled as a graph where computation kernels communicate through FIFO data queues. Since each computation kernel accesses only local input and output data queues, it can be effectively applied to various hardware architectures including shared memory multiprocessors, heterogeneous multiprocessors, GPGPUs, and distributed computing systems [1], [8], [9], [10], [11], [12], [13], [14]. Most previous work on stream programming models and runtimes has focused on the *static scheduling* of *regular stream programs* where the input/output rates of data queues are statically known at compile time. Since irregular programs with dynamic input/output rates and feedback loops cannot be expressed in that model, its applicability is significantly limited. Moreover, static scheduling exhibits serious limitations in performance scalability and portability to complex hardware architectures. In static scheduling, the compiler generates static schedules for each thread based on the work estimation of each kernel, and the runtime iteratively executes the pre-computed schedules with barrier synchronization. Therefore, the effectiveness of the static scheduling is basically determined by the accuracy of the performance estimation which is difficult or barely possible in many hardware architectures. For instance, even commodity x86 servers show 1.5 – 4.3 times difference in core-to-core memory bandwidth [15]. Furthermore, the load imbalance caused by an inaccurate work estimation or data-dependent control flow significantly deteriorates performance scalability as the core count increases. In Figure 1, we show the scalability of the StreamIt runtime, which is a state-of-the-art stream system using static scheduling. Two StreamIt programs were run on a 40-core Intel IA64 NUMA system. (See Section IV for a detailed description of the environment.) In theory, they should be perfectly scalable, because the compiler generates perfectly balanced schedules for each thread by its estimation and the programs do not have any data-dependent control flow. However, in reality, the stall caused by the load imbalance rapidly increases as the core count



(a) TDE (b) FMRadio

Fig. 1: Scalability of StreamIt programs on a 40-core system

[1]"DANBI" is a Korean word meaning "timely rain"; we picked it as the project name for its nature of dynamic resource scheduling.

increases and thus significantly limits the scalability. On 40 cores, communication-intensive TDE [1] suffers from a larger load imbalance: the 85.3% execution time is spent to wait for the barrier synchronization and thus TDE only achieves a speedup of 7.5 times. Fifield also reported similar results on an AMD IA64 NUMA system [11].

Although many approaches have been proposed to overcome the limitations of static scheduling, prior work on dynamic scheduling is insufficient because they have restrictions on the supported types of stream programs [4], [11], [13], [16], [17] or partially perform dynamic scheduling [12], [14] or limit the expressive power by giving up the sequential semantics [6], [18]. Although Flexible Filters [16] and SKIR [11] propose dynamic scheduling of StreamIt programs based on a backpressure mechanism, they support only regular stream programs. GRAMPS [6], [18], the flow graph feature in TBB [4], and the Bobox system [17] dynamically schedule irregular stream programs. However, GRAMPS does not guarantee data ordering between data parallel kernels, thus the expressive power of the data parallel kernels is limited and additional reordering overhead is imposed. Also, GRAMPS and the flow graph do not support `peek` operation, which is commonly used for sliding window computation [19]. The Bobox system does not support data parallel kernels. In distributed stream processing systems, Elastic Operator [14], Borealis [12], and ACES [13] adopt dynamic scheduling mechanisms. However, they have limitations: Elastic Operator [14] handles only the degree of data parallelism in a stateless component. Borealis [12] assumes all components are stateless. ACES [13] does not support cyclic pipelines.

In this paper, we introduce the DANBI programming model, which supports irregular stream programs, and propose its dynamic scheduling mechanism. This paper makes the following specific contributions:

- We introduce the DANBI parallel programming model which extends the stream programming model to support irregular stream programs. DANBI allows a cyclic graph with feedback queues and the dynamic input/output rates of data queues. In contrast to GRAMPS [6], [18], a DANBI program preserves its sequential semantics even under parallel execution. Data ordering across multiple queues in a kernel can be optionally enforced by our ticket synchronization mechanism. With the combination of the feedback queues and ticket synchronization, we can effectively describe complex irregular programs, such as recursive algorithms.
- Since the DANBI program graph contains all the producer-consumer relationships, there are a lot of opportunities to schedule more efficiently. To this end, our scheduler dynamically performs load balancing based on the occupancy of the input and output queues and thus naturally exploits the producer-consumer relationships. Although such a scheduling scheme could help to improve scalability, naive solutions will face limitations on scalability. We found that exploiting the proper degree of pipeline parallelism over data parallelism is critically important to achieve scalability in many-core systems. Excessive data parallelism could result in the *thundering-herd problem* – the gain from parallel execution can be overshadowed by the cost of communication and synchronization among contending threads. To effectively avoid the thundering-herd problem and dynamically adapt to load imbalance, we propose two probabilistic scheduling techniques, PSS and PRS. (See Section III-A.) In contrast to prior work [11], [12], [13], [14], [16], our scheduling mechanism can fully support irregular stream programs without any restrictions and can dynamically adjust task, data, and pipeline parallelism simultaneously. While our scheduling mechanism is developed for the DANBI programming model, the techniques can be applied to other stream programming models, such as StreamIt.
- We developed the DANBI benchmark suite with seven applications ported from StreamIt, Cilk, and OpenCL. Also, we evaluated the performance and scalability of the DANBI runtime and obtained an almost linearly scalable performance on a 40-core IA64 system. In comparison with other parallel runtimes, the DANBI runtime outperforms the state-of-the-art parallel runtimes up to 2.8 times on 40 cores.

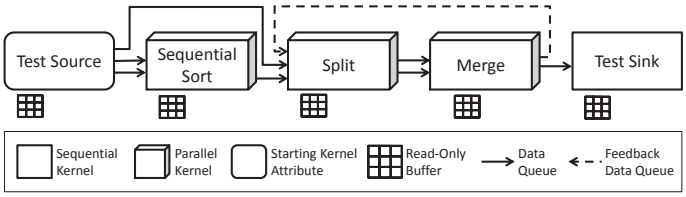


Fig. 2: A DANBI program: merge sort graph

Queue
`q_accessor* reserve_push(q, push_num, ticket_desc)`
`void commit_push(q_accessor)`
`q_accessor* reserve_peek_pop(q, peek_num, pop_num, ticket_desc)`
`void commit_peek_pop(q_accessor)`
`void* get_q_element(q_accessor, i)`
`void consume_ticket(ticket_desc)`
Read-only Buffer
`void* get_rob_element(rob, i)`

TABLE I: The DANBI Core API

The rest of this paper is organized as follows. Section II introduces the DANBI programming model and Section III elaborates on the design of the DANBI runtime for many-core systems. Section IV shows the extensive evaluation results. Related work is described in Section V. Finally, in Section VI, we conclude the paper.

II. The DANBI Programming Model

The DANBI programming model extends state-of-the-art stream programming models [1], [6], [18] to support irregular stream applications. A DANBI program is represented as a graph of independent computation kernels communicating through unidirectional data queues. The graph can be cyclic with feedback data queues. It is not necessary to know the

input/output rates of the data queues at the compile time. Figure 2 presents an example of a DANBI program graph, merge sort, where the recursive concurrent merge operation is represented by using a backward feedback data queue. As described in Table I, seven core APIs are provided for writing a computation kernel. In the rest of this section, we explain each element of the DANBI programming model in detail.

Computation Kernel: It is a user-defined function that operates on zero or more input queues, output queues, and read-only buffers. It is explicitly defined as a *sequential* or *parallel* kernel. A sequential kernel must run serially with a thread, whereas multiple threads can concurrently execute a parallel kernel for data parallelism. Therefore, all parallel kernels should be stateless. Additionally, if a kernel is annotated as a *starting kernel*, it is executed at the beginning. A DANBI program has at least one starting kernel. The input/output rates of the data queues can dynamically vary at runtime.

Data Queue: It is a unidirectional communication channel between kernels, which is modeled as an array based FIFO queue with `push`, `pop`, and `peek` operations. Concurrent producers and consumers for a parallel kernel can work on the same data queue. While the traditional stream models statically determine the degree of data parallelism by using split/join, which replicates data queues and kernels [1], [8], [9], [10], [11], [20], our concurrent data queue approach enables the DANBI runtime to dynamically determine the level of data parallelism by adjusting the number of running threads for a kernel. To work on multiple queue items at the same time with efficiency, a part of the data queue is first *reserved* for exclusive access, and then *committed* to notify when exclusive use ends. In the reserve-commit semantics, `peek` and `pop` operations are combined in one operation, `peek_pop`. From an application point of view, all operations are blocking ones. Reserve operations are blocked when there are not enough elements or rooms. In other words, when there are enough elements or rooms, concurrent reserve operations succeed regardless of whether a previous reserve operation is committed or not. Even under concurrent reserve operations, commit operations are totally ordered according to the reserve order. A commit operation is blocked when the previous reserve operation is not yet committed. Computation can be interleaved with reserve and commit operations. When a queue operation is blocked, the DANBI runtime schedules other threads. The details of our scheduler will be explained in Section III.

Ticket Synchronization: FIFO ordering on queues between sequential kernels is maintained by default. However, queues with parallel kernels are not automatically ordered, since a parallel kernel can execute out of order. Essentially, there are two approaches to deal with data ordering for parallel kernels: total ordering by using split/join [1], [8], [9], [10], [11], [20] and no ordering [18], [21], [22]. The former is not adequate for irregular workloads because the input/output rates should be statically known for a joiner to deterministically merge the split data queues. Since the latter does not preserve the sequential semantics, it limits the expressive power of the data parallel kernel and imposes additional sorting overhead at the last sequential kernel.

```
1  __parallel
2  void kernel(q **in_qs, q **out_qs, rob **robs) {
3    q *in_q = in_qs[0], *out_q = out_qs[0];
4    ticket_desc td = {.issuer=in_q, .server=out_q};
5    rob *size_rob = robs[0];
6    int N = *(int *)get_rob_element(size_rob, 0);
7    q_accessor *qa;
8    float avg = 0;
9
10   qa = reserve_peek_pop(in_q, N, 1, &td);
11   for (int i = 0; i < N; ++i)
12     avg += *(float *)get_q_element(qa, i);
13   avg /= N;
14   commit_peek_pop(qa);
15
16   qa = reserve_push(out_q, 1, &td);
17   *(float *)get_q_element(qa, 0) = avg;
18   commit_push(qa);
19 }
```

Listing 1: A parallel kernel in the DANBI programming model

To support ordering-dependent streaming applications, we introduce a *ticket synchronization* mechanism which enforces the ordering of the queue operations for a parallel kernel. The key idea is analogous to serving customers in a bank: a customer first gets a ticket from a ticket issuer, and then waits until the teller's serving ticket number matches the issued ticket number. In the DANBI programming model, a *ticket* is the number which represents the order. The *ticket issuer* issues a ticket whose value starts from zero and increases by one at each issuance. The *ticket server* manages a serving ticket number which starts from zero and increases by one after each service. It provides service only when the requester's ticket number is the same as the serving ticket number. Otherwise, the requester is blocked, and the DANBI runtime schedules other threads. Since the initial issuing ticket number and the initial serving ticket number are the same, the first ticket issued is served first. After the issuing and serving ticket numbers are incremented, the second ticket issued is served. Thus, the serving order is totally ordered by the issuing order. A data queue is optionally defined to issue or serve a ticket in the reserve operations. To serve a ticket, the source of the issued ticket is also described. Since an issued ticket can be consumed by multiple queues, we can enforce data ordering across multiple queues. When multiple queues are conditionally accessed with an issued ticket, the serving ticket number of the unaccessed queue needs to be increased by using `consume_ticket()` in Table I to keep all relevant ticket numbers synchronized.

Read-only Buffer: It is an array of pre-computed values, which can be read from a computation kernel and is accessible via the index.

Listing 1 is an example of a DANBI kernel code, which is defined as a parallel kernel (Line 1). The kernel calculates simple moving averages for N input elements (Line 10–14) and generates the average to an output queue (Line 16–18). The order of `push` operations is preserved in order of `pop` operations by using the ticket synchronization; a ticket is

issued when popping from the input queue, `in_q`, and it is served when pushing to the output queue, `out_q` (Line 4). The initialization code, which creates kernels, queues, and read-only buffers and connects them, is omitted due to space limitations.

In summary, compared to previous work, the DANBI programming model provides several important features: (a) supporting dynamic input/output rates, (b) supporting a cyclic graph with feedback data queues, (c) multiple fan-in and fan-out of data queues for a kernel, (d) supporting concurrent producers and consumers working on a data queue, (e) supporting `peek` operation and (f) optionally enforcing total ordering of data queue operations for a kernel. The combination of the above features provides a simple but powerful mechanism to describe irregular stream programs. For example, recursive algorithms such as parallel reduction can be expressed by using ticket synchronization and feedback queues.

III. The DANBI Runtime for Many-Core Systems

Even though the DANBI programming model is general and powerful, designing an efficient and scalable runtime for many-core systems is challenging. The key technical challenges are as follows:

- Since the DANBI programming model supports irregular applications with dynamic input/output rates and feedback loops, there is no statically determinable schedule. Thus, static scheduling approaches [1], [8], [9], [10] cannot be used. Moreover, prior work on dynamic streaming is insufficient because it has many restrictions on supported program types [4], [11], [13], [16], [17], on the scope of dynamic scheduling [12], [14], and on reserving data ordering [6], [18]. Our *dynamic load-balancing scheduling* mechanism does not rely on a static work estimation, which could be inaccurate in modern complex many-core architectures, or offline profiling. It makes scheduling decisions based on queue occupancy and dynamically adjusts the degree of data parallelism and pipeline parallelism to avoid the thundering-herd problem, and thus improves scalability.
- To achieve scalable speedup in many-core systems, most concurrently accessed data structures should also be scalable. In cache-coherent many-core systems, frequent invalidation of a shared cache-line results in performance collapse of the entire system [23], [24]. Therefore, careful design of the contended data structures is essential. Especially, when waiting for a commit order or ticket serving order, a naive approach, which repeatedly accesses a shared cache-line to check the order, has significant overhead due to excessive coherence traffic. Instead, we design our data queue and ticket synchronization to check separated cache-lines: a list-based queue is used to check the commit order similar to MCS list-based queuing lock [25] and an array accessed via a ticket number is used to check ticket order similar to array-based queuing locks [26], [27]. Since all concurrent threads read and update the separated cache-line, we can minimize shared cache-line invalidation and improve scalability.

Due to space limitations, we do not present our scalable data queue and ticket synchronization design in this paper, but focus on our dynamic load-balancing scheduling mechanism.

A. Dynamic Load-Balancing Scheduling

The DANBI runtime employs a user-level thread mechanism on top of pinned native threads to avoid expensive mode switching overhead [28]. Since context-switching between user-level threads occurs only at function call boundaries, i.e., queue operations, the number of spilled registers can be minimized. Hereafter, we use the term *thread* for a user-level thread and *native thread* to explicitly mention a native OS thread. In the DANBI runtime, each native thread runs its own scheduler with no predetermined schedules. The DANBI scheduler decides a next runnable kernel and a thread to run the selected kernel. Scheduling decisions are made based on how related queues are filled, and thus producer-consumer relationships in a stream graph are naturally exploited. We make scheduling decisions at two points: (1) when a queue operation is blocked with a queue event such as full, empty or waiting, and (2) when the thread execution of a parallel kernel is ended. *Queue Event-Based Scheduling (QES)* is used at the first case to decide the next runnable kernel (§ III-A2). For the second case, *Probabilistic Speculative Scheduling (PSS)* and *Probabilistic Random Scheduling (PRS)* are used to decide whether to keep executing the same kernel or switch to another (§ III-A3 and § III-A4). Since PSS uses the producer-consumer relationships, PSS is preferred to PRS. If both PSS and PRS are not taken, we keep executing the same kernel. When a thread is blocked, it is pushed to a *per-kernel ready queue*, and it is popped from the queue when re-scheduled. We implemented the ready queue as a concurrent FIFO queue to avoid starvation.

In the rest of this section, we will elaborate our scheduling mechanism in detail.

1) Determining the Initial Schedule: At the beginning of the DANBI runtime, each native thread selects one of the unchosen starting kernels. If there is no such kernel, non-starting parallel kernels are randomly selected. After that, each native thread spawns a new user-level thread for the corresponding kernel and transfers control to the user-level thread.

2) Queue Event-Based Scheduling (QES): A queue operation can be blocked, when a queue is *empty*, *full*, or *waiting for a commit or ticket order*. When blocked, our scheduler selects a next runnable kernel and a thread by using *Queue Event-Based Scheduling (QES)* (Algorithm 1). When an input queue is empty, it schedules the producer of the queue. Similarly, when an output queue is full, it schedules the consumer of the queue. When it is blocked, while waiting for a commit or a ticket order, another thread of the same kernel is scheduled.

Thread life-cycle management is incorporated into this process. The goal is to reduce the memory footprint by minimizing the number of threads. After selecting a kernel, it first pops a thread from the per-kernel ready queue. When the ready queue is empty, it spawns a thread only if creating another thread does not violate the concurrency constraint of the kernel. If it cannot spawn a new thread – i.e., a running thread for the

sequential kernel already exists, we randomly re-select another kernel. When the thread of a parallel kernel has no successful queue operation, we can safely delete the thread since it has no side effects.

QES performs dynamic load balancing when a data queue becomes full or empty with no predetermined schedules. Though it is similar to the backpressure mechanism [11], [16], [18], we extend it to incorporate waiting for a commit/ticket order and thread life-cycle management.

Algorithm 1: Queue Event-Based Scheduling

```
Input: current running kernel rk, current running thread rt,
       queue q, queue event e
Output: selected kernel k, selected thread t
if e is waiting then
    k = rk;
    t = ready_queue[k].pop();
    if t is null then t = rt;
    else ready_queue[k].push(rt);
else
    if rk is a parallel kernel and there is no successful queue
    operation for the kernel then delete rt;
    else ready_queue[rk].push(rt);
    if e is empty then k = producer kernel of q;
    else if e is full then k = consumer kernel of q;
    repeat
        t = ready_queue[k].pop();
        if t is null then
            if k is a parallel kernel then
                t = spawn a new thread;
            else if k is a sequential kernel then
                if there is no running thread for k then
                    t = spawn a new thread;
                else
                    k = randomly select a kernel in the graph;
                    continue;
    until false;
```

3) Probabilistic Speculative Scheduling (PSS): In QES, many threads for a kernel may make the same scheduling decision if they schedule at roughly the same time. As a result, QES tends to maximize the degree of data parallelism as far as the sizes of input/output queues are available. The high degree of data parallelism, however, could result in the thundering-herd problem, especially when queue operations are ordered by ticket synchronization. If we exploit pipeline parallelism more aggressively than data parallelism, we can avoid the thundering-herd problem and improve scalability by reducing the degree of data parallelism.

To exploit pipeline parallelism more aggressively, we introduce *Probabilistic Speculative Scheduling (PSS)*. At the end of the thread execution of a parallel kernel, we decide whether to continue running the same kernel or not. We probabilistically schedule another kernel before the corresponding queue becomes completely empty or full. For brevity, let's assume that pipelined parallel kernel K_{i-1}, K_i, and K_{i+1} are connected by queue Q_x and Q_{x+1}. Assuming an infinite number of threads are running for the three kernels, the transition probability between the kernels is determined by how much each queue is filled. Under this condition, incoming transition probabilities from K_{i-1} and K_{i+1} to K_i are defined as follows:

$$P_{i-1,i} = F_x$$
$$P_{i+1,i} = 1 - F_{x+1}$$

where $P_{m,n}$ is the transition probability from K_m to K_n, and F_x is the fill ratio of Q_x ranging from 0 to 1. In the same way, we can calculate the bidirectional outgoing transition probabilities of K_i as follows:

$$P_{i,i+1} = F_{x+1}$$
$$P_{i,i-1} = 1 - F_x$$

After balancing out the incoming and outgoing probabilities, the balanced transition probabilities for parallel kernel K_i are defined as follows:

$$P^b_{i,i-1} = \max(P_{i,i-1} - P_{i-1,i}, 0)$$
$$P^b_{i,i+1} = \max(P_{i,i+1} - P_{i+1,i}, 0)$$

where $P^b_{m,n}$ is the balanced transition probability from K_m to K_n. At the end of the thread execution of a parallel kernel, we arbitrarily determine the transition direction and take the transition with the probability of $P^b_{i,i-1}$ or $P^b_{i,i+1}$. If we decide to take the transition to another kernel, we select a thread in a similar way as in Algorithm 1. Otherwise, we try *Probabilistic Random Scheduling* described in Section III-A4. For a kernel with multiple input and output queues, we take the emptiest input queue and the fullest output queue.

Since we randomly choose the transition direction, the actual transition probabilities are as follows:

$$P^t_{i,i-1} = 0.5 \times P^b_{i,i-1}$$
$$P^t_{i,i+1} = 0.5 \times P^b_{i,i+1}$$
$$P^t_{i,i} = 1 - P^b_{i,i-1} - P^b_{i,i+1}$$

where $P^t_{m,n}$ is the transition probability from K_m to K_n taken by our scheduler. $P^t_{i,i}$, transition probability to itself, is a probability of no transition. In the steady state with no thread transition, $P^t_{i,i-1}$ and $P^t_{i,i+1}$ are 0, and $P^t_{i,i}$ is 1. Under this situation, F_x and F_{x+1} are 0.5. Therefore, PSS iteratively attempts to assign threads to kernels for filling all the data queues in a graph by half. Scheduling to fill queues in half actually means scheduling to perform *double buffering* which is widely used for overlapping communication and computation. It is arithmetically simple and does not require predetermined schedules.

4) Probabilistic Random Scheduling (PRS): PSS works well in most cases, but when the execution time of a kernel is significantly different due to data dependent control flow or fine-grain architecture variability such as shared cache miss, simultaneous multi-threading (SMT), or dynamic voltage and frequency scaling (DVFS), waiting time for the commit or ticket order could increase significantly. To dynamically adapt to such circumstances, we use a *Probabilistic Random Scheduling (PRS)* policy. If a thread waits too long for a commit

or ticket order, we schedule a randomly selected kernel. The probability of random scheduling for K_i, P_i^r, is defined as follows:

$$P_i^r = \min(T_i/C, 1)$$

where T_i is the number of consecutive waiting events for a thread, and C is a large constant greater than 1 (10,000 in our experiments). The probability increases linearly as the waiting count becomes larger. It is analogous to a bank customer who has waited too long in line and will likely switch to a different bank next time. However, only when not taking PSS, we decide whether to take PRS or not with probability P_i^r. If taking PRS, we randomly select a kernel and a thread in a similar way to Algorithm 1. This too is arithmetically simple and does not require predetermined schedules.

5) Terminating a DANBI program: In procedural languages, a program is terminated when the program counter reaches the end of the program. However, since the control flow of a DANBI application is sometimes determined by queue status, terminating a DANBI program is different from the methods used in procedural languages. When a starting kernel is terminated, it propagates a termination token through the output queues. A non-starting kernel is terminated when it receives the termination tokens from all input queues. When all starting and non-starting kernels are terminated, a DANBI program is finally terminated. However, there is no guarantee that the DANBI program will be terminated when the queue size is inadequate or the behavior of the feedback queue is uncontrolled. Even static scheduling mechanisms have difficulty guaranteeing deadlock freedom with feedback queues [8], [29]. We argue that a program with inadequate queue sizes or uncontrolled feedback queues is an incorrect DANBI program.

IV. Evaluation

We first describe our benchmark applications and how we created them. Next, we evaluate the scalability of the DANBI runtime and analyze how our dynamic load-balancing scheduling techniques contribute to achieving scalability. Also, we analyze the behavior of our thread life-cycle management scheme and the sensitivity of the performance to various queue sizes. Finally, we compare the scalability and the performance of the DANBI runtime with other parallel runtimes, StreamIt [1], OpenCL [2], and Cilk [3]. We performed all experiments on a 4-socket system with 10-core 2.0 GHz Intel Xeon E7-4850 (Westmere-EX) processor (40 cores in total). The system has 256 KB per-core L2 caches, 24 MB per-processor L3 caches. Each processor forms a NUMA domain with 8 GB local memory (32 GB in total). The processors communicate through a 6.4 GT/s QPI interconnect. The machine runs 64-bit Linux Kernel 3.2.0 with GCC 4.6.3.

A. Benchmark Suite

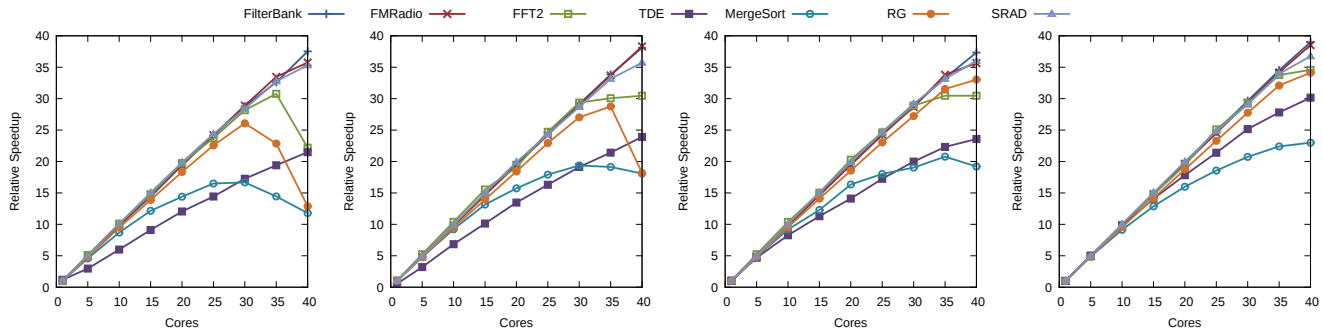
In order to broadly exercise the DANBI programming model and runtime, we developed seven benchmark applications from other parallel programming models. Table II shows the characteristics of the benchmark applications. All benchmarks have plenty of parallelism: for all applications, all kernels except the test source and test sink are parallel kernels and all data queue operations are ordered by ticket synchronization. We replaced file I/O operations in the original benchmarks with memory operations to limit the effect of the OS kernel. As a baseline for the evaluation, we set the size of each data queue to maximally exploit data parallelism (i.e. for all 40 threads to work on a queue). More specifically, when a producer of a queue, Q, generates maximum P-sized data at once and a consumer of Q consumes maximum C-sized data at once, the size of Q is determined as $max(P * T, C * T)$, where T is the number of native threads assigned for the DANBI runtime. P and C are naturally determinable by a problem itself: for example, they would be the size of a image, one row or column of a image for RG and SRAD and the number of elements for MergeSort. We will investigate how the queue size affects performance in Section IV-C.

Benchmark	Description	Origin	Kernel	Queue
FilterBank	Multirate signal processing filters	StreamIt	44	58
FMRadio	FM Radio with equalizer	StreamIt	17	27
FFT2	64 elements FFT	StreamIt	4	3
TDE	Time delay equalizer for GMTI	StreamIt	29	28
MergeSort	Merge sort	Cilk	5	9
RG	Recursive Gaussian image filter	OpenCL	6	5
SRAD	Diffusion filter for ultrasonic image	OpenCL	6	6

TABLE II: Benchmark descriptions and characteristics

From StreamIt benchmark suite [1], we ported two compute-intensive benchmarks, FilterBank and FMRadio, with complex pipelines, and another two communication-intensive benchmarks, FFT2 and TDE, with straight pipelines. The numbers of kernels are different from the original StreamIt benchmark suite, because the DANBI programming model does not have a splitter/joiner [1] and we manually fuse kernels with the same code. We could nearly mechanically port StreamIt applications to DANBI applications, since the DANBI programming model supports all the core functionalities of StreamIt including peeking and data ordering. The only manual work was to change the filter arguments in StreamIt to read-only buffers in the DANBI programming model.

To investigate how recursive algorithms can effectively be represented in the DANBI programming model, we ported a parallel merge sort from Cilk [30]. MergeSort recursively splits sorted arrays into two, and then merges the two concurrently [31]. As shown in Figure 2, the recursive spawn-sync parallelism in Cilk can be successfully transformed to a DANBI program. Cilk functions synchronized by a barrier are naturally mapped to DANBI parallel kernels, and Cilk recursive functions can be represented by using feedback queues and ticket synchronization in the DANBI programming model. Function arguments in Cilk can be represented as either a read-only buffer or a ticket synchronized data queue depending on whether they are changing in the middle of execution or not. In terms of scheduling, it is the most challenging application in our benchmark suite: data-dependent control flows in every kernel, highly biased workload among kernels (i.e., in our profiling, most benchmark time is spent in the Merge kernel),



(a) Random Work Stealing (b) QES (c) (b) + PSS (d) (c) + PRS

Fig. 3: Scalability of DANBI from 1 to 40 cores. Relative speedup is normalized to the single core performance in Figure 3d.

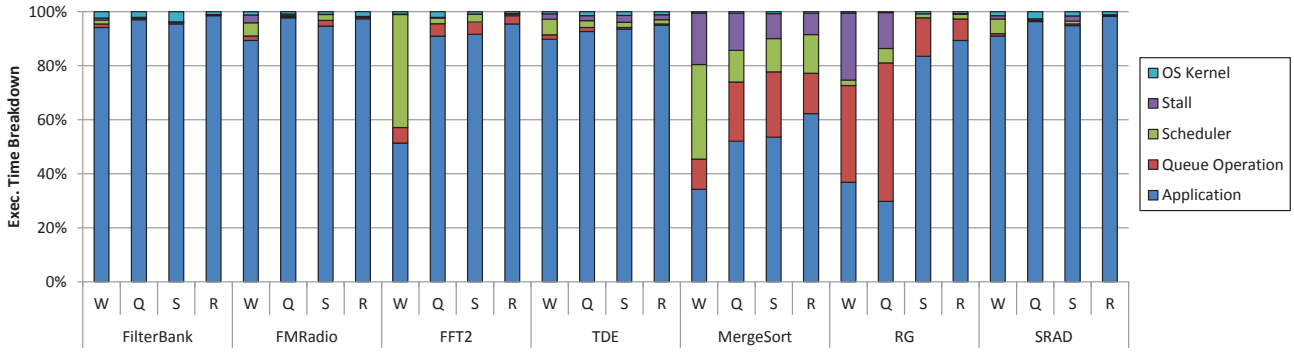
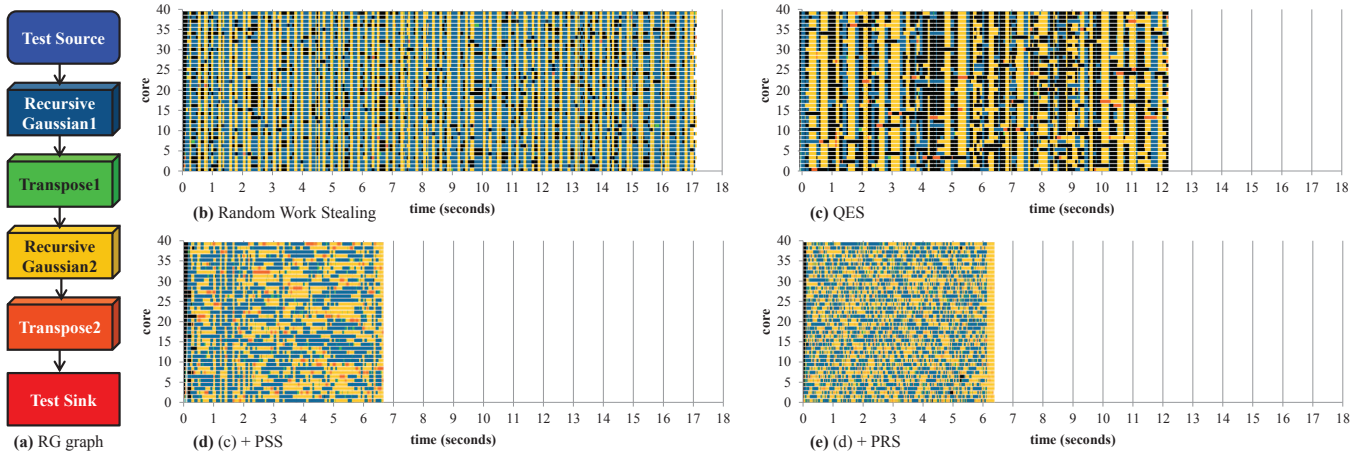


Fig. 4: Execution time breakdown of DANBI on 40 cores. **(W)**: Random Work Stealing, **(Q)**: QES, **(S)**: (Q) + PSS, and **(R)**: (S) + PRS



(a) RG graph **(b)** Random Work Stealing **(c)** QES **(d)** (c) + PSS **(e)** (d) + PRS

Fig. 5: Comparison of kernel scheduling for RG on 40 cores. The color in Figure 5b-e represents that the kernel with the same color in Figure 5a is running. For example, the yellow portions represent that RecursiveGaussian2 is running. The black color represents the stall cycle.

and few opportunities to exploit pipeline parallelism due to the short pipeline.

RG and SRAD are data-parallel image filter applications from OpenCL. They were ported from NVIDIA OpenCL SDK [32] and Rodinia suite [33], respectively. Porting OpenCL applications to DANBI applications takes similar effort to that of Cilk. Barrier synchronized OpenCL kernels are naturally mapped to DANBI parallel kernels.

As discussed earlier, the porting an existing parallel program to a DANBI program is straightforward in many cases. One of the most difficult cases is that an original program does not sequentially consume the input data. For example, in MergeSort, when merging two chunks into one larger chunk, the merge sort in Cilk, first, recursively divides the chunks into multiple sub-chunks and then merges pairs of the sub-chunks in parallel. In Cilk, the sub-chunks are represented via indexes in the original chunk. However, in streaming models including DANBI, data should be sequentially consumed from an input queue. Thus, the additional rearrangement of the pairs of the sub-chunks is needed to perform parallel merging as in Cilk. Though the additional rearrangement results in the slower performance of DANBI in a smaller number of cores, dynamic scheduling of DANBI eventually catches up the performance in a larger number of cores. We will discuss this in detail in Section IV-D.

B. Scalability of the DANBI Runtime

To evaluate the scalability of the DANBI runtime, we ran each application by varying the number of cores from 1 to 40. As we illustrated in Figure 3, we ran each application in four different scheduling configurations to evaluate the effectiveness and scalability of the scheduling techniques. The configuration in Figure 3a uses random work stealing which does not use the producer-consumer relationships to make scheduling decisions. For comparison, our baseline scheduling configuration in Figure 3b uses only the basic QES scheme. In addition to that, in Figure 3c and Figure 3d, we adopt PSS and PRS, respectively. For direct comparison of the graphs in Figure 3, relative speedup is normalized to the single core performance in Figure 3d. For further analysis, Figure 4 shows the execution time breakdown for each application when using all 40 cores. Each bar is spilt into five categories, showing the fraction of time spent in the application code, queue operation, scheduler, stall, and OS kernel. In the DANBI runtime, the stall means no work progress due to waiting for a commit or ticket ordering. The breakdown is obtained with a cycle-accurate low-overhead profiling code using a CPU time-stamp counter and Linux `perf record` command [34] which collects profiles based on sampling. Additionally, Figure 5 illustrates how each scheduling mechanism behaves in the kernel scheduling of RG on 40 cores. The colors in Figure 5b-e correspond to the colors of kernels in Figure 5a, except for black which represents the stall cycle.

Random Work Stealing: As Figure 3a shows, the scalability of random work stealing is very dependent on the characteristics of applications. The compute-intensive applications such as FilterBank and SRAD scale nearly linearly up to 40 cores, whereas MergeSort and RG, the least compute-intensive applications, start to degrade performance at 25 cores and 30 cores, respectively. That is because large fractions of time are expended on stalling, 19.0% for MergeSort and 24.8% for RG, as shown in Figure 4 and Figure 5b. The increased stall increases fractions of the queue operation time and the scheduler time. As a result, only the small fractions are expended for the applications, 34.2% for MergeSort and 36.8% for RG. Mean speedup over single core performance is 25.3 times. Our experimental results clearly show the limitations of random work stealing on stream parallelism: suboptimal scheduling decisions without using producer-consumer relationships incur a large overhead especially in communication-intensive applications.

Queue Event-Based Scheduling: By using the basic QES scheme, mean speedup improves from 25.3 times to 28.9 times. Moreover, MergeSort and RG scale up to 30 and 35 cores respectively. As Figure 3b and Figure 4 show, QES significantly reduces the fraction of the stall: from 19.0% to 13.8% for MergeSort and from 24.8% to 13.3% for RG. Interestingly, the fractions of the queue are rather increased. As we mentioned in Section III-A3, QES tends to maximize the degree of data parallelism as far as possible, and the high degree of data parallelism along with the ticket synchronization could result in the thundering-herd problem. In Figure 5c, most threads work for a kernel at the same time and it increases the contention of the data queues and the stall induced by ticket synchronization. The large fraction of time for the queue operation and the stall in Figure 4 confirms this.

Probabilistic Speculative Scheduling: PSS effectively avoids the thundering-herd problem by aggressively exploiting pipeline parallelism over data parallelism. In the PSS scheduling in Figure 5d, various kernels are executed at the same time, and there are very few stall cycles, as shown in black. As a result of this, the fractions of the queue operation and the stall in RG are significantly reduced: from 51% to 14% for the queue operation, and from 13.3% to 0.03% for the stall. In the case of MergeSort, performance is marginally improved because there is little opportunity to exploit pipeline parallelism. Mean speedup also improves to 30.8 times.

Probabilistic Random Scheduling: In MergeSort, the time taken for merging the sub-chunks heavily depends on the size of the sub-chunks. Therefore, higher data parallelism of the Merge kernel increases the chance of load imbalance, since the different sized sub-chunks are likely to be merged in parallel. Figure 3d shows that PRS effectively mitigates the load imbalance. MergeSort speedup on 40 cores improves from 19.2 times to 23.0 times. Also, the performance of the two communication-intensive benchmarks, FFT2 and TDE, are likely to be affected by fine-grain architecture variability such as shared cache and NUMA. Figure 3d shows that PRS effectively adapts to such circumstances and thus improves the scalability: from 30.5 times to 34.6 times for FFT2 and from 23.6 times to 30.2 times for TDE. Now, all applications scale up to 40 cores without saturation. Mean speedup with all the optimizations is 33.7 times. On average, 91% of the benchmark time is spent on the applications themselves.

Benchmark	Input data size (MB)	Running time (QES+PSS+PRS, msec)	
		1-core	40-core
FilterBank	98	288,504	7,399
FMRadio	977	226,783	5,887
FFT2	14,648	228,613	6,610
TDE	14,832	507,566	16,835
MergeSort	3,815	243,072	10,569
RG	3,000	220,890	6,471
SRAD	5,000	199,018	5,416

TABLE III: Benchmark input data size and running time in 1-core and 40-core

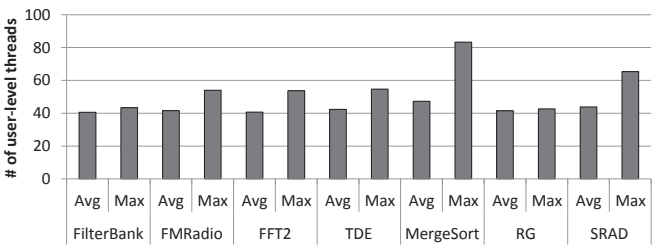


Fig. 6: Average and maximum number of user-level threads

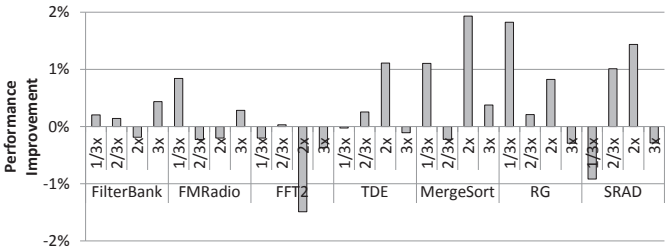


Fig. 7: Performance variation under different queue sizes

Table III shows the input data sizes and the absolute running times in milliseconds on 1-core and 40-core.

In summary, random work stealing, which is widely used, reveals the scalability limitations due to its blindness to the producer-consumer relationships. Our experimental results on QES and PSS show that exploiting the producer-consumer relationships for making scheduling decisions is critically important for achieving high scalability. Especially, PSS is quite effective to avoid the thundering-herd problem by scheduling speculatively before a data queue becomes completely full or empty. Finally, PRS is effective to mitigate fine-grain load-imbalance. Our execution breakdown shows that the DANBI runtime imposes very little overhead: on average 3.9% for queue operation, 2.8% for the scheduler, 1.5% for stall, and 1.1% for the OS kernel.

C. Footprint and Performance Sensitivity of Queue Size

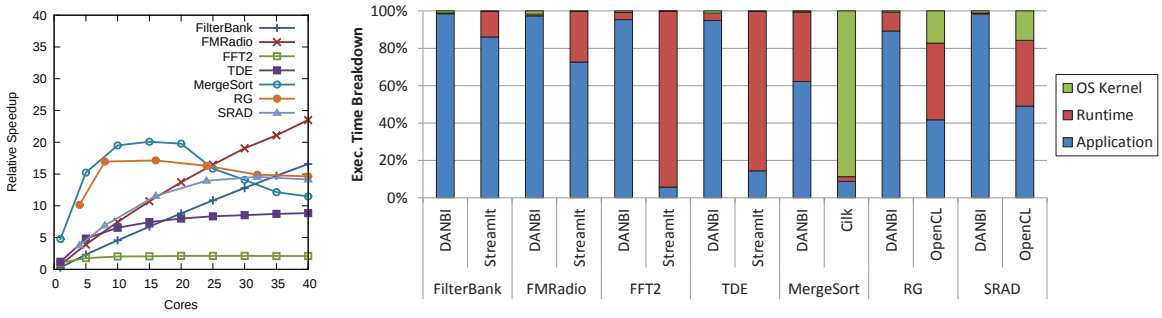
One of the interesting aspects in the DANBI runtime is the footprint and its relationship to performance. The footprint of a DANBI application is determined by the developer's settings for the data queue size. In addition, the DANBI runtime dynamically creates and destroys user-level threads as needed. Therefore, the thread stack is a variable part in the DANBI footprint. We evaluate the average and maximum number of threads on 40 cores. Since 40 native threads are running on 40 cores, the minimum number of user-level threads is 40. As Figure 6 shows, our thread life-cycle management mechanism incorporated with QES tightly manages the number of user-level threads: 43 threads on average and 83 threads at maximum.

Our QES and PSS policy make scheduling decisions based on how much each queue is filled. Therefore, the data queue size set by the developer could affect the scheduling decision. To evaluate how much the queue size affects performance on 40 cores, we vary the queue size to 1/3x, 2/3x, 2x, and 3x of the queue size in Section IV-B. Figure 7 shows that there are only marginal performance variations, below 2%. It shows that our scheduling mechanism can dynamically adapt to queue size by changing the degree of data and pipeline parallelism.

D. Comparison with Other Parallel Programming Runtimes

In this section, we compare performance and scalability of the DANBI runtime with the state-of-the-art parallel programming runtimes. For fair comparison, we modified the original benchmarks to perform I/O operations on memory rather than files, and ran the benchmarks on the latest available versions of the original runtimes. Figure 8a shows the speedup of the other runtimes normalized to the DANBI single core performance in Figure 3d. Figure 8b shows the execution time breakdown in three categories: application, parallel runtime, and OS kernel. In DANBI, the runtime means the sum of the queue operation, scheduler, and stall time in Figure 4. The breakdown of the other runtime is obtained by analyzing the collected profiles from Linux `perf record` command [34].

StreamIt: The original version of FilterBank, FMRadio, FFT2, and TDE ran on the latest StreamIt runtime obtained from the code repository [35]. We used StreamIt SMP backend [1], which is optimized for shared-memory multicore systems, with the highest optimization level (-O2). StreamIt compiler generates statically scheduled multi-threaded C codes with barrier synchronization, and the generated C codes are compiled with GCC 4.6.3. Mean speedup of the four applications is 12.8 times. Performance of FFT2 starts to be saturated at 5 cores, and that of TDE is saturated at 15 cores. There is the possibility that the static scheduling without runtime scheduling overhead could outperform our dynamic scheduling. However, since the performance of modern many-core systems are difficult to estimate, the suboptimal static schedules lead to the large stalls and the limited scalability. As Figure 8b shows, a large portion of the execution time, 55% on average, is spent in the runtime which is the barrier synchronization overhead waiting for termination of all threads at each steady state schedule. Figure 1 shows that as the thread count increases, barrier synchronization overhead also rapidly increases, while scalability rapidly decreases. It reveals the limitations of static scheduling. As a result, while our dynamic scheduling has additional overhead to make scheduling



(a) Scalability

(b) Execution time breakdown on 40 cores

Fig. 8: Scalability and execution time breakdown of other parallel runtimes. The relative speedup is normalized to the DANBI single core performance in Figure 3d and the execution time break time is classified into Application, Runtime, and OS Kernel. The runtime portion of DANBI is the sum of the queue operation, scheduler, and stall time in Figure 4.

decisions, it outperforms the static scheduling. For the same applications, the DANBI runtime achieves 35.6 times mean speedup while spending only 2.3% of the execution time for the runtime.

Cilk: We ran the original version of MergeSort from MIT [30] on the latest Intel Cilk Plus runtime [36]. Due to changes in Cilk keywords, we made minor modifications. In Figure 8a, the performance improvement is saturated at 10 cores and the performance begins to degrade at 20 cores. The fraction of the OS kernel in the execution time increases non-linearly as thread count increases: for 10, 20, 30, and 40 cores, the OS kernel takes 57.7%, 72.8%, 83.1%, and 88.7% of execution time, respectively. Contention on work stealing causes disproportional growth of OS kernel time, mostly in the OS scheduler, because Cilk scheduler calls `pthread_yield()` when it fails to acquire a lock on a victim's work queue. In our experiments, the yielding count soars as the thread count increases: 1.1 million, 5.5 million, 18.1 million, 34.1 million, and 49.6 million for 5, 10, 20, 30, and 40 cores, respectively. Simply replacing yielding with spinning does not help: spinning shows similar scalability because the overhead in the OS scheduler simply moves to the Cilk scheduler. It clearly shows the limitations of blind random stealing in Cilk which does not exploit the producer-consumer relationships. In a small number of cores, Cilk outperforms the DANBI runtime, because the DANBI version of MergeSort requires one additional memory copy in the Spilt kernel, which splits two sorted arrays into smaller chunks for a parallel merge. On 40 cores, DANBI significantly outperforms Cilk: 23 times speedup for DANBI and 11.5 times speedup for Cilk.

OpenCL: The applications originated from OpenCL, RG and SRAD, ran on the latest Intel OpenCL runtime [37]. Since the Intel OpenCL runtime does not provide the functionality to change the number of involved threads, we changed the BIOS configuration of our test machine to change the number of cores. All allowable BIOS configurations are 4, 8, 16, 24, 32, and 40 cores. Figure 8a shows that the performance improvement of SRAD is saturated at 24 cores and the performance of RG starts to degrade at 16 cores. As core count increases, the fraction of runtime rapidly increases: in the case of SRAD, runtime takes 6.7%, 21.1%, and 38.3% of execution time on 8, 24, and 40 cores, respectively. We found that more than 50% of the runtime was spent in the work stealing scheduler of TBB [4], which is an underlying framework of Intel OpenCL. On 40 cores, OpenCL versions of RG and SRAD achieve 14.6 and 14.1 times speedup, respectively, while DANBI achieves a significantly higher speedup: 34.1 and 36.8 times speedup with a significantly lower runtime overhead.

In summary, we found that achieving scalability in many-core systems – 40 cores in our experiment – is very challenging even in state-of-the-art parallel runtimes. StreamIt, Cilk, and OpenCL perform well up to approximately 15 cores, but they begin to struggle with more than 20 cores. Moreover, bulk-synchronous style execution [38] – barrier synchronization between the steady state schedule in StreamIt, synchronization on returning from recursion in Cilk, and synchronization between kernels in OpenCL – shows larger scalability limitations as the core count increases. In contrast, the dynamic load-balancing scheduling of the DANBI runtime enables nearly linear scalable performance speedup, at least up to 40 cores.

V. Related Work

Data-flow oriented stream processing has received a lot of interest in the context of both many-core systems and distributed systems. Here we present a selection of papers most related to our work.

Static Scheduling in Many-core Systems: StreamIt [1] is a representative stream programming model and runtime. It follows the synchronous data flow (SDF) model [39] which supports only regular applications with static input/output rates. Scheduling is generated offline by the compiler based on the estimation of the execution time and the communication requirement of each kernel [1], [8], [29]. However, as shown in Figure 1, it significantly suffers from load imbalance when an application has data-dependent control flows or the architecture has performance variability (e.g. cache miss, memory location, SMT, and DVFS). Moreover, since it iteratively executes the

steady state schedule in a bulk-synchronous way with barrier synchronization, the load imbalance tends to more severely affect scalability with a larger number of cores.

Dynamic Scheduling in Many-core Systems: SEDA [40] dynamically adjusts the number of threads for a stage and the number of events processed by each iteration. However, the scope of dynamic scheduling is limited to data parallelism. Flexible Filters [16] identifies bottleneck filters through profiling of the application, and accelerates the execution of the bottleneck filters by using the backpressure mechanism. SKIR [11] proposes a dynamic scheduling mechanism based on work stealing with the backpressure mechanism, but it fails to achieve scalability in more than 24 cores because of excessive work stealing overhead. Though these two works provide load balancing on stream programs, they support scheduling only on regular stream programs. GRAMPS [6], [18] is the first runtime that supports dynamic scheduling of irregular stream programs. It performs dynamic load balancing based on work stealing with a per-kernel work queue and a backpressure mechanism. However, it does not support data ordering between parallel stages because of significant queue manipulation overhead. It limits expressive power in parallel kernels and imposes additional overhead for reordering the data at the last sequential kernel. Moreover, GRAMPS does not support `peek` operation, which is commonly used for sliding window computation in many realistic stream applications [19], and their experimental environments are relatively limited to: an idealized simulator with no scheduling overheads [6] and a 12-core, 24-thread machine [18]. The flow graph feature in TBB [4] and the Bobox system [17] support dynamic scheduling of cyclic pipelines. However, the flow graph also does not support `peek` operation and the Bobox system does not support data parallel kernel.

Dynamic Scheduling in Distributed Systems: Elastic Operator [14] iteratively adjusts the level of data parallelism based on peek throughput and measured throughput. However, it is limited to dynamically changing the degree of data parallelism. Borealis [12] provides sophisticated local, neighborhood and global scheduling optimizations, but it assumes all components are stateless. ACES [13] first optimizes schedules offline, and then dynamically adapts the schedules based on buffer occupancy and processing rates. However, it does not support the cyclic pipeline. Moreover, it is based on the brittle assumption that processing rate is proportional to CPU utilization.

Topology Unaware Work Stealing: Work stealing is a widely used load balancing technique in many parallel runtimes due to its fine-grained nature [3], [4], [41], [42]. It performs well for programs with simple dependencies (e.g. fork-join), but it works poorly on complex pipelines as it does not exploit producer-consumer relationships.

VI. Conclusion and Future Work

In this paper, we introduced the DANBI programming model and proposed its dynamic scheduling mechanism for load balancing. The DANBI programming model extends stream programming models to support irregular programs. Our load-balancing scheduler exploits producer-consumer relationships in a stream program to generate scalable schedules. Moreover, we also presented two probabilistic scheduling policies which effectively avoid the thundering-herd problem and dynamically adapt to load imbalance. It surpasses prior static stream scheduling approaches, which are vulnerable to load imbalance, and prior dynamic stream scheduling approaches, which have many restrictions on supported program types, on the scope of dynamic scheduling, and on preserving data ordering. Our experimental results show that DANBI achieves an almost linear speedup up to 40 cores, while other state-of-the-art parallel runtimes begin to have difficulty at around 15 or 20 cores. On 40 cores, DANBI outperforms StreamIt by 2.8 times, Cilk by 2 times, and Intel OpenCL by 2.5 times.

There are many avenues for future work. The design goal of our current scheduling schemes is to minimize CPU stalls by using dynamic load balancing. However, by taking care of data locality such as NUMA and cache locality, we could additionally improve the performance. Next, we plan to extend the DANBI programming model and the runtime to heterogeneous computing environments, such as GPGPUs and Xeon Phi coprocessors. We expect that our localized memory access and array based queue will enable DANBI to be easily portable to other architectures with various memory models. Finally, we will extend our work to distributed computing environments. Our scheduling mechanism will be suitable even in distributed environments, since scheduling decisions are made in a distributed manner with no centralized control. To this end, we will extend our runtime to support conceptually unbounded data queues with a bounded memory footprint.

Acknowledgements

We thank the anonymous reviewers for their feedback and comments, which have substantially improved the content and presentation of this paper. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2012-0006423).

References

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

[2] Khronos OpenCL Working Group, "The OpenCL Specification Version 1.1," http://www.khronos.org/opencl, 2011.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98, 1998, pp. 212–223.

[4] Intel TBB, http://software.intel.com/en-us/intel-tbb/.

[5] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11, 2011, pp. 12–25.

[6] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, pp. 4:1–4:11, Feb. 2009.

[7] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the GPU," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10, 2010, pp. 29–37.

[8] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08, 2008, pp. 114–124.

[9] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09, 2009, pp. 214–223.

[10] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS XVI, 2011, pp. 381–392.

[11] J. M. Fifield, "Generating, Optimizing, and Scheduling a Compiler Level Representation of Stream Parallelism," Ph.D. dissertation, University of Colorado, 2011.

[12] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *CIDR*, 2005, pp. 277–289.

[13] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive Control of Extreme-scale Stream Processing Systems," in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, 2006, p. 71.

[14] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09, 2009, pp. 1 –12.

[15] G. Hager, G. Jost, R. Rabenseifner, and J. Treibig, "Performance-oriented programming on multicore-based Clusters with MPI, OpenMP, and hybrid MPI/OpenMP," http://blogs.fau.de/hager/tutorials/isc12/, 2012.

[16] R. L. Collins and L. P. Carloni, "Flexible filters: load balancing through backpressure for stream programs," in *Proceedings of the seventh ACM international conference on Embedded software*, ser. EMSOFT '09, 2009, pp. 205–214.

[17] D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral, "The bobox project parallelization framework and server for data processing," *Charles University in Prague, Technical Report*, vol. 1, p. 2011, 2011.

[18] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic Fine-Grain Scheduling of Pipeline Parallelism," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11, 2011, pp. 22–32.

[19] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10, 2010, pp. 365–376.

[20] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12, 2012, pp. 53–64.

[21] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, 2010, pp. 170–177.

[22] "Storm," http://storm-project.net/.

[23] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, 2010, pp. 1–8.

[24] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proceedings of the Linux Symposium*, 2012.

[25] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[26] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[27] G. Granunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors," *Computer*, vol. 23, no. 6, pp. 60–69, Jun. 1990.

[28] L. Soares and M. Stumm, "FlexSC: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, 2010, pp. 1–8.

[29] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10, 2010, pp. 1–10.

[30] MIT CSAIL Supertech Research Group, "The Cilk Project," http://supertech.csail.mit.edu/cilk/.

[31] S. G. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *IEEE Trans. Comput.*, vol. 36, no. 11, pp. 1367–1369, Nov. 1987.

[32] NVidia, "NVIDIA OpenCL SDK Code Samples," http://tinyurl.com/ayo8brs.

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54.

[34] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/.

[35] "StreamIt SVN," https://svn.csail.mit.edu/streamit/, Accessed 1 Nov. 2012.

[36] Intel Cilk Plus, http://software.intel.com/en-us/intel-cilk-plus/.

[37] Intel OpenCL SDK, http://software.intel.com/en-us/vcsource/tools/opencl-sdk/.

[38] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[39] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, Sept. 1987.

[40] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01, 2001, pp. 230–243.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05, 2005, pp. 519–538.

[42] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, 2008, pp. 536–545.