Dynamic Scheduling of Irregular Stream Programs toward Many-Core Scalability

Changwoo Min, and Young Ik Eom

Abstract—The stream programming model has received much interest because it naturally exposes task, data, and pipeline parallelism. However, most prior work has focused on the 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 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 achieving scalability. Our dynamic load-balancing scheduler exploits producer-consumer relationships already expressed in the DANBI 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 result in many restrictions on supported program types, on the scope of dynamic scheduling, and on data ordering preservation. 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.

Index Terms—Stream programming, software pipelining, scheduling, load balancing, irregular programs

1 Introduction

The prevalence of multi-core processors has renewed interest in parallel programming models and runtimes, such as StreamIt [1], OpenCL [2], [3], [7] Cilk [4], TBB [5], and Galois [6]. 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 [6], [7], [8].

Stream programming models, such as StreamIt, have been extensively studied because they naturally expose 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], [9], [10], [11, [12], [13], [14], [15], [16], [17]. 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 $\times 86$ servers show a 1.5-4.3-fold difference in core-to-core memory bandwidth [18]. 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 Fig. 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 4 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 increases and thus significantly limits the scalability. On 40 cores, communication-intensive TDE [1] suffers from a larger load imbalance: the 85.3 percent of the execution time is spent to wait for the barrier synchronization, so TDE only achieves a speedup of 7.5 times. Fifield also reported similar results on an AMD IA64 NUMA system [12].

Although many approaches have been proposed to overcome the limitations of static scheduling, prior work on dynamic scheduling is insufficient because of restrictions on the supported types of stream programs [5], [12], [15], [19] or because dynamic scheduling is partially performed [10], [14], [16], or because the expressive power is limited by giving up the sequential semantics [7], [20]. Although Flexible Filters [19] and SKIR [12] propose dynamic

• *The authors are with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, Gyeonggi-do, Korea. E-mail: {multics69, yieom}@skku.edu.*

Manuscript received 12 Jan. 2014; revised 14 May 2014; accepted 15 May 2014. Date of publication 1 June 2014; date of current version 8 May 2015. Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2325833

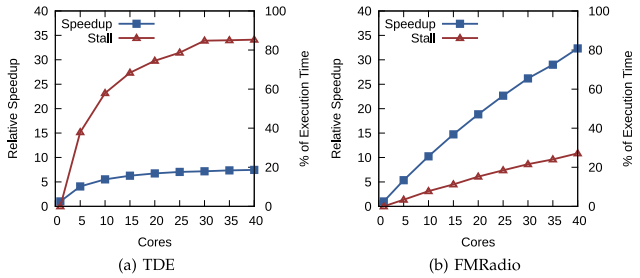


Fig. 1. Scalability of StreamIt programs.

scheduling for StreamIt programs based on a backpressure mechanism, they support only regular stream programs. GRAMPS [7], [20] and the flow graph feature in TBB [5] dynamically schedule irregular stream programs. However, GRAMPS does not guarantee data ordering between data parallel kernels, so 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 [21]. In distributed stream processing systems, Elastic Operator [16], Borealis [14], and ACES [15] adopt dynamic scheduling mechanisms, but, they have limitations. Elastic Operator [16] handles only the degree of data parallelism in a stateless component. Borealis [14] assumes all components are stateless. ACES [15] does not support cyclic pipelines.

In this paper, we introduce the DANBI programming model, which supports irregular stream programs, and its runtime design including dynamic scheduling mechanisms. We make 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 [7], [20], 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 many opportunities to schedule more efficiently. To this end, our scheduler dynamically performs load balancing based on the occupancy of the input and output queues, so 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 critical to achieve high scalability. Excessive data parallelism could result in the *thundering-herd problem*, in which 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, Probabilistic Speculative Scheduling (PSS) and Probabilistic Random Scheduling (PRS) (see Section 3.1). In contrast to prior work, 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.
- To achieve high scalability, frequently accessed data structures also need to be scalable. In the DANBI runtime, our ticket synchronization mechanism could be a performance bottleneck since competing threads need to check their tickets for ordering. Indeed, in cache-coherent many-core systems, frequent invalidation of the shared cacheline results in the performance collapse of an entire system [22], [23]. We present a scalable design of ticket synchronization, which minimizes the invalidation of shared cachelines.
- 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.

The remainder of this paper is organized as follows. Section 2 introduces the DANBI programming model, and Section 3 elaborates on the design of the DANBI runtime for many-core systems. Section 4 shows the extensive evaluation results. Related work is described in Section 5. Finally, in Section 6, we conclude the paper.

2 The DANBI Programming Model

The DANBI programming model extends state-of-the-art stream programming models [1], [7], [20] 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. As described in Table 1, 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. The computation kernel 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

TABLE 1
The DANBI Core API

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)`

one starting kernel. The input/output rates of the data queues can dynamically vary at runtime.

Data queue. The data queue 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], [9], [11], [12], 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 3.1.

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], [9], [11], [12], and no ordering [20], [24], [25]. 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.

To support ordering-dependent streaming applications, we introduce a *ticket synchronization* mechanism which

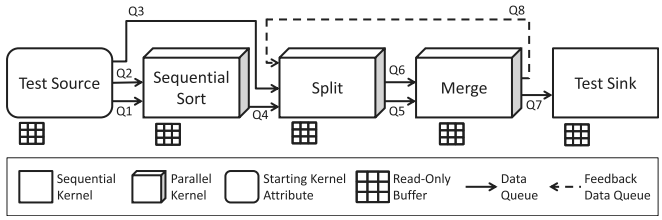


Fig. 2. A DANBI program: merge sort graph.

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 receives 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 1 to keep all relevant ticket numbers synchronized.

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

Fig. 2 presents merge sort, an example of a DANBI program graph, where the recursive concurrent merge operation is represented by a backward feedback data queue. `Sequential Sort` kernel sequentially sorts unsorted data from `Q1` in the unit of a chunk, the size of which is passed via `Q2`. A sorted chunk in `Q4` is further divided into two or more sub-chunks by `Split` kernel for parallel merge operation in `Merge` kernel. The size of each sub-chunk to be chopped is passed via `Q3`. As a result of `Split` kernel, chopped sub-chunks and their sizes are passed via `Q5` and `Q6`, respectively. `Merge` kernel merges two chunks in parallel by merging sub-chunks from the two chunks in parallel. If further merge operations are needed, the merged chunk is pushed to the `Split` kernel via `Q8`. Otherwise, the completely sorted data is pushed to `Test Sink` via `Q7`.

Fig. 3 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 (lines 16-18) and generates the average in an output queue (lines 16-18). The order of `push` operations is preserved in the order of `pop` operations by using the ticket synchronization. A ticket

```
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 }
```

Fig. 3. An example of a DANBI parallel kernel.

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 features provides a simple but powerful mechanism to describe irregular stream programs. For example, recursive algorithms such as parallel reduction can be expressed using ticket synchronization and feedback queues. Also, data-dependent control flows can be expressed by two or more ticket-synchronized input queues: one for control commands and the others for data.

3 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], [9], [11] cannot be used. Moreover, prior work on dynamic streaming is insufficient because there are many restrictions on supported program types [5], [12], [15], [19], on the scope of dynamic scheduling [14], [16], and on reserving data ordering [7], [20]. Our *dynamic load-balancing scheduling* mechanism does not rely on static work estimation, which could be inaccurate in modern 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, so 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 cacheline results in performance collapse of the entire system [22], [23]. Therefore, careful design of the contended data structures is essential. Especially when waiting for a commit order or ticket serving order, a naive approach, that repeatedly accesses a shared cacheline to check the order has significant overhead due to excessive coherence traffic. Instead, we design our data queue and ticket synchronization to check separated cachelines: a list-based queue is used to check the commit order similar to MCS list-based queuing lock [26], and an array accessed by a ticket number is used to check the ticket order similar in array-based queuing locks [27], [28]. Since all concurrent threads read and update the separated cacheline, we can minimize shared cacheline invalidation and improve scalability.

In the remainder of this section, we present our dynamic load balancing scheduling mechanism and the scalable design of the ticket synchronization. Due to space limitations, we do not present our queue algorithms in this paper.

3.1 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. Hereafter, we use the term *thread* for a user-level thread and *native thread* to explicitly indicate a native OS thread.

In the DANBI runtime, each native thread runs its own scheduler with no predetermined schedules. The DANBI scheduler decides the next runnable kernel and a thread to run the selected kernel. Scheduling decisions are made based on how related queues are filled, so 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 in the first case to decide the next runnable kernel (Section 3.1.2). For the second case, *Probabilistic Speculative Scheduling* and *Probabilistic Random Scheduling* are used to decide whether to keep executing the same kernel or switch to another (Sections 3.1.3 and 3.1.4). Since PSS uses the producer-consumer relationships, PSS is preferred to PRS. PSS and PRS make scheduling decisions with probabilities, so we switch to a new kernel based on the probabilities. 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 remainder of this section, we will elaborate on our scheduling mechanism in detail.

3.1.1 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.

3.1.2 Queue Event-Based Scheduling

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* (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. In this case, since a DANBI program cannot make progress until the scheduler find a proper thread in order, we express that the DANBI program is *stalled*.

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
1  if e is waiting then
2      k = rk
3      t = ready_queue[k].pop()
4      if t is null then t = rt
5      else ready_queue[k].push(rt)
6  else
7      if rk is a parallel kernel and there is no successful queue
       operation for the kernel then delete rt
8      else ready_queue[rk].push(rt)
9      if e is empty then k = producer kernel of q
10     else if e is full then k = consumer kernel of q
11     repeat
12         t = ready_queue[k].pop()
13         if t is null then
14             if k is a parallel kernel then
15                 t = spawn a new thread
16             else if k is a sequential kernel then
17                 if there is no running thread for k then
18                     t = spawn a new thread
19                 else
20                     k = randomly select a kernel in the
                       graph
21                     continue
22     until false
```

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 back-pressure mechanism [12], [19], [20], we extend it to incorporate waiting for a commit/ticket order and thread life-cycle management.

3.1.3 Probabilistic Speculative Scheduling

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 long 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*. 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, assume that pipelined parallel kernels 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 3.1.4).

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. $P^t_{i,i}$, the transition probability to itself, is the 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.

3.1.4 Probabilistic Random Scheduling

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), the waiting time for the commit or ticket order could increase significantly. To dynamically adapt to such circumstances, we use a *Probabilistic Random Scheduling* policy. If a thread waits too long for a commit or ticket order, we schedule a randomly selected kernel. The probability, P^r_i, of random scheduling for K_i is defined as follows:

$$P^r_i = \min\left(\frac{T_i}{C}, 1\right),$$

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^r_i. 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.

3.1.5 Terminating a DANBI Program

In procedural languages, a program is terminated when the program counter reaches the end. 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.

```
1  struct ticket_server_t {
2          int ncore;
3          int *tickets cacheline_aligned;
4  };
5  void init(ticket_server_t *ts, int ncore) {
6          ts→ncore = ncore;
7          ts→tickets = new int[ncore] cacheline_aligned;
8          ts→tickets[0] = 0;
9          for (int i = 1; i < ncore; ++i)
10                 ts→tickets[i] = i − ncore;
11 }
12 bool is_my_turn(ticket_server_t *ts, int issued_ticket) {
13         int index = issued_ticket % ts→ncore;
14         return ts→tickets[index] == issued_ticket;
15 }
16 void serve(ticket_server_t *ts, int issued_ticket) {
17         int next_index = (issued_ticket + 1) % ts→ncore;
18         ts→tickets[next_index] += ts→ncore;
19 }
```

Fig. 4. The pseudo-code of the ticket server.

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 [9], [29]. We argue that a program with inadequate queue sizes or uncontrolled feedback queues is an incorrect DANBI program.

3.2 Scalable Ticket Synchronization

Ticket synchronization operations are tightly integrated with data queue operations. When a queue is initialized, a ticket issuer and a ticket server for each endpoint of the queue are created according to ticket descriptions in Table 1. In reserve operations, ticket synchronization operations are called according to the ticket descriptions for enforcing the ordering of queue operations.

A ticket issuer is a counter which starts from zero and increments by one in `issue()` operations. Also, a ticket server is a counter which starts from zero and increments by one. In `is_my_turn()` operations, we test if the issued ticket passed by an argument matches to the ticket in a ticket server to decide whether the caller can be served or not. A thread which takes its turn performs a queue operation and then calls `serve()` to make the next thread served by increasing the ticket server's counter by one. A straightforward but inefficient design is to use a single counter variable for a ticket server. However, since all competing threads read the cacheline of the ticket server's counter, updating the counter invalidates all the shared cachelines. Such frequent invalidation of shared cachelines could result in contention meltdown in many-core systems [22], [23].

To avoid such contention meltdown, we design a ticket server in such a way that each competing thread access a separated cacheline in the `is_my_turn()` operation. In Fig. 4, we show the pseudo code of our scalable ticket server. At initialization, we first allocate an array of counters whose size is the number of cores (line 7). Each competing thread has a private counter in the array, which is determined by its issued ticket number (lines 13, 14). In the `serve()` operation, the private counter of the next ticket number is incremented by the number of cores for the next thread to be served (lines 17 and 18). We initialize the first

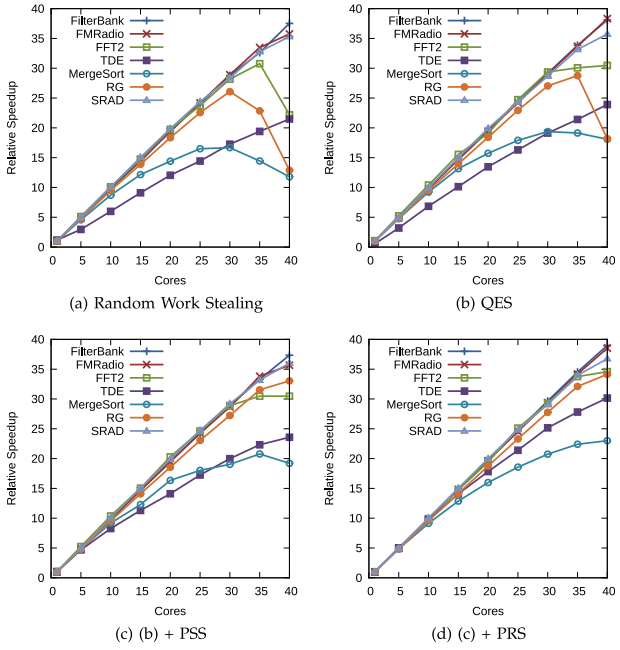


Fig. 5. Scalability of DANBI from 1 to 40 cores. Relative speedup is normalized to the single-core performance in Fig. 5d.

element to zero in order to serve the first issued ticket immediately, and initialize the rest so as to serve them after increasing the counters by the number of cores (lines 8-10). In this design, since the `is_my_turn()` operation accesses the privately owned cacheline and the `serve()` operation modifies the cacheline of the next thread, we can minimize the cacheline invalidation traffic and avoid the contention meltdown in a high level of concurrency. We will investigate the effectiveness of this design in Section 4.2.

4 Evaluation

In this section, we evaluate various aspects of the DANBI programming model and its runtime. We first describe our benchmark applications and discuss our experience on the DANBI programming model. Next, we evaluate the scheduling, footprint, and ticket synchronization of DANBI in terms of scalability. Finally, we compare DANBI to other parallel runtimes, StreamIt [1], OpenCL [3], and Cilk [4], in terms of scalability and performance sensitivity in a multiprogramming environment. To easily compare scalability, all relative speedup values in this section are normalized to the single-core performance of the DANBI runtime in Fig. 5d.

We performed all experiments on a four-socket system with a 10-core 2.0 GHz Intel Xeon E7-4850 (Westmere-EX) processor (40 cores in total). The system has 256 KB of per-core L2 caches and 24 MB per-processor L3 caches. Each processor forms a NUMA domain with 8 GB of 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.

4.1 Benchmark Suite

In order to broadly exercise the DANBI programming model and runtime, we developed seven benchmark applications from other parallel programming models. Table 2 shows the characteristics of the benchmark applications and Table 3 shows the input data sizes in our evaluation. All benchmarks have plenty of parallelism: for all applications, all kernels except `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 an image, one row or column of an image for RG and SRAD, and the number of elements for MergeSort. We will investigate how the queue size affects the performance in Section 4.3.

We ported two compute-intensive benchmarks from StreamIt benchmark suite [1], 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].[1] MergeSort recursively splits sorted arrays into two, and then merges the two concurrently [32]. As shown in Fig. 2, the recursive spawn-sync parallelism in Cilk can be successfully transformed into a DANBI program. Cilk functions synchronized by a barrier are naturally mapped to DANBI parallel kernels, and Cilk recursive functions can be represented 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, MergeSort is the most challenging application in our benchmark suite: there are data-dependent control flows in every kernel, workload of each kernel is highly biased (in our profiling, the `Merge` kernel takes the most

1. We intentionally did not compare with a mergesort implementation of the StreamIt package [31]. The StreamIt differently prallelizes the merge sort algorithm than DANBI and Clik do, so apple-to-apple performance comparison is not possible. In the StreamIt version, the recursion of the mergesort is flattened at the stream compile time because the StreamIt runtime cannot execute cyclic graphs. So, as the input size increases, stream graph and its compilation time also increase non-linearly. To sort 4,096 integers, it takes about 29 hours to generate a stream graph with 8,191 nodes.

TABLE 2
Benchmark Descriptions and Characteristics

Benchmark	Description	Origin	Kernel	Queue	Original LOC	DANBI LOC
FilterBank	Multirate signal processing filters	StreamIt	44	58	267	1239 (448)
FMRadio	FM Radio with equalizer	StreamIt	17	27	175	775 (312)
FFT2	64 elements FFT	StreamIt	4	3	181	537 (201)
TDE	Time delay equalizer for GMTI	StreamIt	8	7	749	976 (472)
MergeSort	Merge sort	Cilk	5	9	474	921 (654)
RG	Recursive Gaussian image filter	OpenCL	6	5	718	544 (304)
SRAD	Diffusion filter for ultrasonic image	OpenCL	6	6	2296	574 (304)

In parentheses, we present the LOCs without counting lines to connect queues and kernels.

of the execution time), and there are 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 [33] and the Rodinia suite [34], 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, porting an existing parallel program to a DANBI program is straightforward in many cases. As shown in Table 2, in terms of lines of code (LOC), the most tedious work is to connect queues and kernels. 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 (the `Split` kernel in Fig. 2). Though the additional rearrangement results in slower performance of DANBI with a smaller number of cores, the dynamic scheduling of DANBI eventually catches up the performance with a larger number of cores. We will discuss this in detail in Section 4.5.

4.2 Effectiveness of the Dynamic Load-Balancing Scheduling

We ran each application by varying the number of cores from 1 to 40. As we illustrated in Fig. 5, we ran each application in four different scheduling configurations to evaluate the effectiveness and scalability of the scheduling techniques. In all the configurations, we used the scalable ticket synchronization mechanism. The configuration in Fig. 5a uses random work stealing which does not use the producer-consumer relationships to make scheduling decisions. For comparison, our baseline scheduling configuration in Fig. 5b uses only the basic QES scheme. In addition to that, in Figs. 5c and 5d, we adopt PSS and PRS, respectively. For direct comparison of the graphs in Fig. 5, relative speedup is normalized to the single-core performance in Fig. 5d. For further analysis, Fig. 6 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 [35] which collects profiles based on sampling. Additionally, Fig. 7 illustrates how each scheduling mechanism behaves in the kernel scheduling of RG on 40 cores. The colors in Figs. 7b, 7c, 7d, and 7e correspond to the colors of kernels in Fig. 7a, except for black, which represents the stall cycle.

Random work stealing. As Fig. 5a shows, the scalability of random work stealing is very dependent on the characteristics of applications. The computation-intensive applications such as FilterBank and SRAD scale nearly linearly with up to 40 cores, whereas the performance of MergeSort and RG, the least compute-intensive applications, starts to degrade at 25 cores and 30 cores, respectively. That is because large fractions of time are expended upon stalling: 19.0 percent for MergeSort and 24.8 percent for RG, as shown in Figs. 6 and 7b. The increased stall increases fractions of the queue operation time and the scheduler time. As a result, only

TABLE 3
Benchmark Input Data Size and Running Time

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

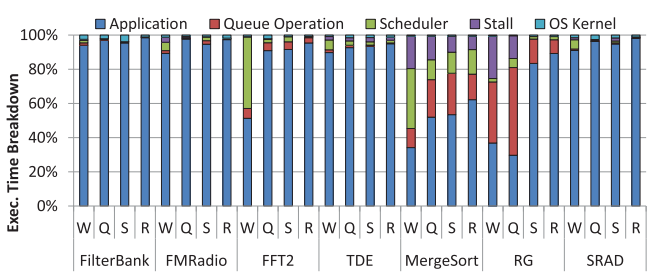


Fig. 6. Execution time breakdown of DANBI on 40 cores. **(W)**: Random Work Stealing, **(Q)**: QES, **(S)**: (Q) + PSS, and **(R)**: (S) + PRS.

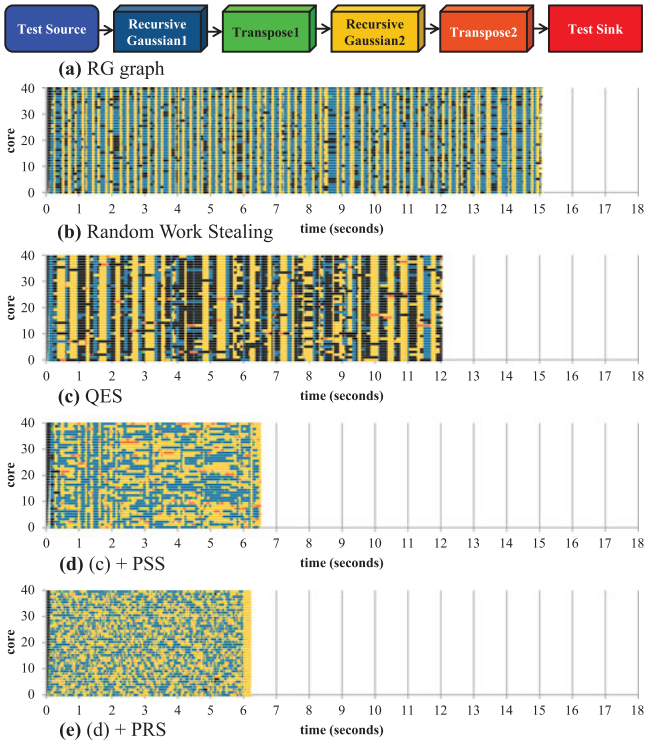


Fig. 7. Comparison of kernel scheduling for RG on 40 cores. The color in Figs. 7b, 7c, 7d, 7e represents that the kernel with the same color in Figs. 7a is running. For example, the yellow represents that RecursiveGaussian2 is running. The black color represents the stall cycle.

the small fractions are expended for the applications: 34.2 percent for MergeSort and 36.8 percent for RG. The 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 large overhead, especially in communication-intensive applications.

Queue event-based scheduling. By using the basic QES scheme, the mean speedup improves from 25.3 to 28.9 times. Moreover, MergeSort and RG scale up to 30 and 35 cores, respectively. As Figs. 5b and 6 show, QES significantly reduces the fraction of the stall: from 19.0 to 13.8 percent for MergeSort and from 24.8 to 13.3 percent for RG. Interestingly, the fractions of the queue are rather increased. As mentioned in Section 3.1.3), QES tends to maximize the degree of data parallelism as much as possible, and the high degree of data parallelism along with the ticket synchronization could result in the thundering-herd problem. In Fig. 7c, 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 Fig. 6 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 Fig. 7d, 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 percent for the queue operation, and from 13.3 to 0.03 percent

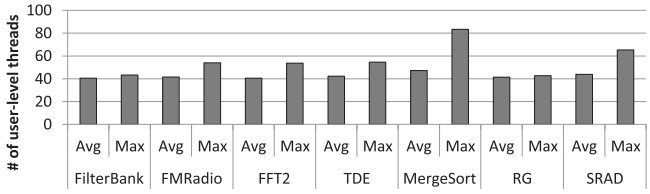


Fig. 8. Average and maximum number of user-level threads.

for the stall. In the case of MergeSort, the performance is marginally improved because there is little opportunity to exploit pipeline parallelism. The 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. Fig. 5d shows that PRS effectively mitigates the load imbalance. The 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 with shared cache and NUMA. Fig. 5d 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. The mean speedup with all the optimizations is 33.7 times. On average, 91 percent of the benchmark time is spent on the applications themselves. Table 3 shows the input data sizes and the absolute running times in milliseconds on 1 and 40 cores.

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. Particularly, 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 at mitigating fine-grain load imbalance. Our execution breakdown shows that the DANBI runtime imposes very little overhead: on average, 3.9 percent for queue operation, 2.8 percent for the scheduler, 1.5 percent for stall, and 1.1 percent for the OS kernel. As a result, our dynamic scheduling has little overhead to make scheduling decisions and outperforms the static scheduling as shown in Figs. 5 and 11.

4.3 Footprint and Performance Sensitivity of Queue Sizes

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. Also, the DANBI runtime dynamically creates and destroys user-level threads as needed. So the thread stack is a variable part in footprint. We evaluate the average and maximum number of threads on 40 cores. Since 40 native threads are running on 40 cores, the

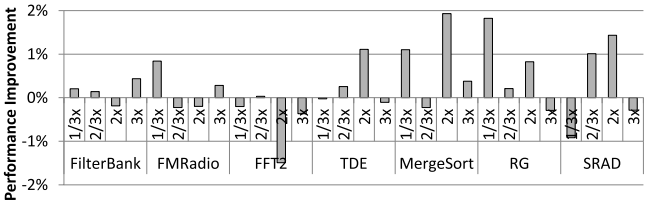


Fig. 9. Performance variation under different queue sizes.

minimum number of user-level threads is 40. As Fig. 8 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.

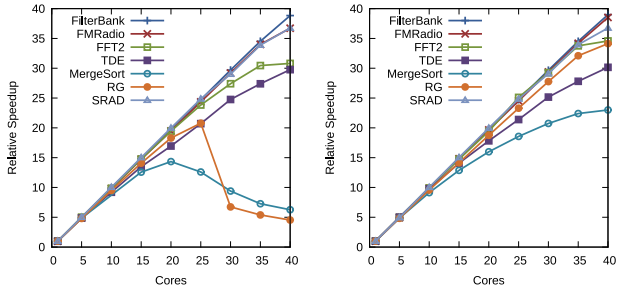
Our QES and PSS policies make scheduling decisions based on how much each queue is filled. Therefore, the data queue size could affect the scheduling decision. To evaluate how much the queue size affects performance on 40 cores, we vary the queue size to $1/3\times$, $2/3\times$, $2\times$, and $3\times$ of the queue size in Section 4.2. Fig. 9 shows that there are only marginal performance variations of below 2 percent. It shows that our scheduling mechanism can dynamically adapt to queue size by changing the degree of data and pipeline parallelism.

4.4 Effectiveness of the Scalable Ticket Synchronization

To verify how our scalable design of ticket synchronization is effective, we compare scalability between the naive ticket synchronization and the scalable one with all our scheduling policies enabled. Fig. 10 clearly shows that our scalable design is quite effective at avoiding performance collapse by reducing frequent shared cacheline invalidation. Replacing the scalable ticket synchronization to the naive one drops the mean speedup from 33.7 times to 26.2 times. Particularly, we observed significant performance degradations in RG, MergeSort, and FFT2, which are less computationally intensive and which mostly rely on data parallelism due to their short pipeline. In those applications, cycles per instruction (CPI) in the naive implementation is significantly increased due to the frequent shared cacheline invalidation: 0.65 to 0.73 for FFT2, 1.11 to 5.26 for MergeSort, and 1.3 to 10.0 for RG.

4.5 Comparison with Other Parallel Runtimes

In this section, we compare the performance and scalability of the DANBI runtime with the state-of-the-art parallel programming runtimes. For fair comparison, we modified the

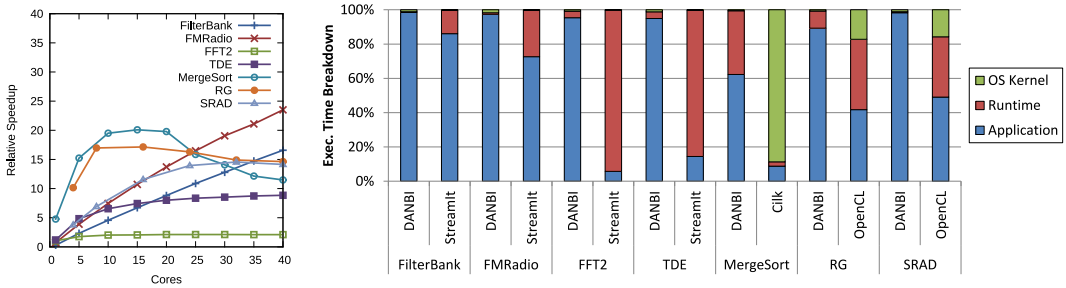


(a) Naive Implementation (b) Scalable Implementation

Fig. 10. Comparison of scalability between ticket synchronization mechanisms.

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. Fig. 11a shows the speedup of the other runtimes normalized to the DANBI single-core performance in Fig. 5d. Fig. 11b 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 Fig. 6. The breakdown of the other runtime is obtained by analyzing the collected profiles from Linux `perf record` command [35].

StreamIt. The original version of FilterBank, FMRadio, FFT2, and TDE ran on the latest StreamIt runtime obtained from the code repository [31]. We used StreamIt SMP backend [1], which is optimized for shared-memory multicore systems, with the highest optimization level (-O2). The StreamIt compiler generates statically scheduled multithreaded C code with barrier synchronization, and the generated C codes are compiled with GCC 4.6.3. The mean speedup of the four applications is 12.8 times. The performance of FFT2 starts to be saturated at 5 cores, and that of TDE is saturated at 15 cores. There is 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 Fig. 11b shows, a large portion of the execution time, 55 percent on average, is spent in the runtime which is the barrier synchronization overhead waiting for termination of all threads at each steady-state schedule. Fig. 1 shows that as the thread count increases, barrier synchronization overhead also rapidly increases, while scalability rapidly decreases. It reveals the



(a) Scalability (b) Execution time breakdown on 40 cores

Fig. 11. Scalability and execution time breakdown of other parallel runtimes. The relative speedup is normalized to the DANBI single-core performance in Fig. 5d and the execution time is classified into Application, Runtime, and OS Kernel. In DANBI, the runtime is the sum of the queue operation, scheduler, and stall time in Fig. 6.

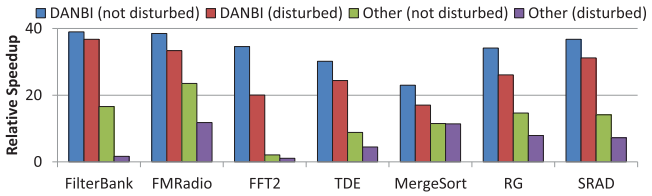


Fig. 12. Comparison of scalability on 40 cores while the disturbing thread is running.

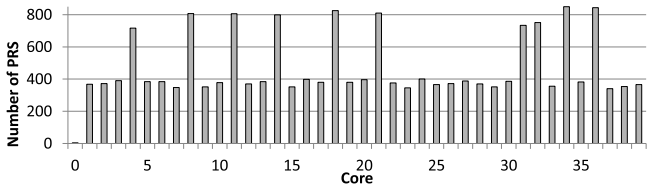


Fig. 13. The number of PRS taken in FilterBank. The disturbing thread was pinned at core 0.

limitations of static scheduling. As a result, while our dynamic scheduling has additional overhead to make scheduling decisions, it outperforms the static scheduling. For the same applications, the DANBI runtime achieves 35.6-fold mean speedup while spending only 2.3 percent 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 Fig. 11a, 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 the thread count increases: for 10, 20, 30, and 40 cores, the OS kernel takes 57.7, 72.8, 83.1, and 88.7 percent 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, to 1.1, 5.5, 18.1, 34.1, 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. With 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, with 23-fold speedup for DANBI and 11.5-fold speedup for Cilk.

OpenCL. RG and SRAD which originate from OpenCL 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 of the allowable BIOS configurations are 4, 8, 16, 24, 32, and 40 cores. Fig. 11a shows that the performance improvement of SRAD is saturated at 24 cores, and the performance of RG starts to degrade at 16 cores. As the core count increases, the fraction of runtime rapidly increases: in the case of SRAD, runtime takes 6.7, 21.1, and 38.3 percent of the execution time on 8, 24, and 40 cores, respectively. We found that more than 50 percent of the runtime was spent in the work stealing scheduler of TBB [5], which is an underlying framework of Intel OpenCL. On 40 cores, OpenCL versions of RG and SRAD achieve 14.6 and 14.1-fold speedups, respectively, while DANBI achieves a significantly higher speedup: 34.1 and 36.8-fold 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 with 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.

4.6 Performance Sensitivity in a Multiprogramming Environment

To more deeply understand the characteristics of the schedulers, we ran our benchmarks on 40 cores with a disturbing thread that infinitely performs arithmetic operations. The disturbing thread is pinned to an arbitrary core. Its CPU usage is limited to 90 percent by using `cpulimit` [39], which controls the CPU usage of a process by continuously sending it `SIGSTOP` and `SIGCONT` signals. The disturbing thread makes benchmark threads on the core slow down by 10 times. Such performance degradation on a core could seriously degrade the performance of a whole benchmark, because the overall throughput of a pipeline is limited by the slowest kernel [10], with 90 percent degradation in the worst case.

In Fig. 12, we compare the relative speedup of the DANBI and other parallel runtimes with and without the disturbing thread. The performance degradation of the DANBI runtime is significantly lower than that of other parallel runtimes: on average, 20 percent for DANBI and 50 percent for other runtimes. Our experimental results show that in the DANBI runtime, as a benchmark has more kernels, its performance degradation is lower. The performance degradation of FilterBank is the lowerst (5.7 percent), followed by FMRadio (13.4 percent). As discussed in Section 4.1, the two benchmarks have many kernels with complex pipelines. The correlation between the number of kerenels and its peformance degradation comes from two factors. First, since there is no fixed kernel-to-core mapping, the impact of the disturbance spreads over all kernels. Second, our dynamic load-balancing scheduling, especially PRS, adapts well in this circumstance. As Fig. 13 shows, the numbers of PRS taken at the disturbed core (core 0) is significantly lower than those of the others. That is because when a slow thread running on the disturbed core and non-disturbed fast threads are running for

the same kernel, the fast threads are likely to be scheduled to another kernel by PRS due to their longer waiting cycles. Therefore, in larger stream graphs with many kernels, we can prevent the slow thread from being a performance bottleneck by executing other kernels.

Among the parallel runtimes, StreamIt performs worst: the average performance degradation of FilterBank, FMRadio, FFT2, and TDE is 59.7 percent. The performances of FilterBank and FMRadio, which show the smallest performance degradation in the DANBI runtime, were degraded most, by 89.9 and 49.8 percent, respectively. It shows that the bulk synchronous execution of static schedules is brittle in performance variability. The performance degradation of OpenCL benchmarks, RG and SRAD, is 47.5 percent. Since OpenCL kernels are barrier synchronized, the load imbalance caused by the disturbed core significantly degrades the overall performance. Finally, it is interesting that Cilk shows only a negligible performance degradation of 0.8 percent. As discussed in Section 5, under a high level of concurrency, Cilk scheduler frequently calls `pthread_yield()`, so the OS scheduler takes 88.7 percent of the benchmark time to handle `pthread_yield()`. Since our disturbing thread can be almost processed by the OS time which was spent for `pthread_yield()` (i.e., it fails to disturb), there is only negligigle performance degradation.

5 Related Work

Data-flow oriented stream processing has received much interest in the context of both many-core systems and distributed systems. 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 model [40] 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], [9], [29]. However, as shown in Fig. 1, it significantly suffers from load imbalance when an application has data-dependent control flows or the architecture has performance variability. 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 [41] and FDP [10] dynamically adjust the number of threads for a stage. However, their scope of dynamic scheduling is limited to data parallelism. Flexible Filters [19] identifies bottleneck filters through profiling of the application, and accelerates the execution of the bottleneck filters by using the back-pressure mechanism. SKIR [12] proposed a dynamic scheduling mechanism based on work stealing with the back-pressure mechanism, but it failed to achieve scalability with more than 24 cores due to high communication cost and excessive dynamic scheduling overhead. Though these two works provide load balancing on stream programs, they support scheduling only on regular stream programs. GRAMPS [7], [20] 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 back-pressure mechanism. However, it does not support data ordering between parallel stages because of the 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 [21], and experimental environments are relatively limited to: an idealized simulator with no scheduling overhead [7] and a 12-core, 24-thread machine [20]. Also, the experimental results of GRAMPS [20] show that scalability of TDE and FFT2, which were commonly used in DANBI and GRAMPS, is worse that that of DANBI due to the higher runtime overhead. Though the flow graph feature in TBB [5] supports the dynamic scheduling of cyclic pipelines, it does not support `peek` operation.

Dynamic scheduling in distributed systems. Elastic Operator [16] 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 [14] provides sophisticated local, neighborhood, and global scheduling optimizations, but it assumes all components are stateless. ACES [15] 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 [4], [5], [42], [43]. 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.

6 Conclusion and Future Work

In this paper, we introduced the DANBI programming model and its runtime design including the dynamic scheduling mechanisms 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 DANBI 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, in the scope of dynamic scheduling, and on preserving data ordering. In addition, we proposed our scalable design of ticket synchronization which minimizes the invalidation of shared cachelines. Our experimental results show that DANBI achieves an almost linear speedup with 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.

Though the design goal of our current scheduling schemes is to minimize CPU stalls, we could additionally improve performance by taking care of data locality. In NUMA architectures, previous studies show that it is critical to minimize contention on memory controller and interconnection at the same time [44], [45]. In larger number of cores, optimization for memory accesses will be important to achieve high scalability. We believe that there are many opportunities to co-optimize scheduling policies and data placement policies. Perhaps the biggest limitation of DANBI is that its queue size is fixed. We expect that its extension to a conceptually unbounded queue is straightforward: we envision a list of array ring queues which automatically grow and shrink under certain occupancy thresholds. We expect that our localized memory access and array-based queue will enable DANBI's easy portability to other architectures, such as GPGPUs and Intel Xeon Phi, with various memory models. We will need additional architecture-specific optimizations, for example coalescing memory accesses in GPGPUs, to get the best results. Another interesting avenue of research would be optimization for energy consumption; in environments such that it is infeasible to accurately estimate energy consumption, dynamic scheduling approaches would help. Despite of the advantages of DANBI, in some environments, traditional static scheduling approaches would be better; static scheduling would suits better real-time applications, where performance predictability is important.

Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Plannig (2010-0020730). Young Ik Eom is the corresponding author of this paper.

References

[1] M. I. Gordon, "Compiler techniques for scalable performance of stream programs on multicore architectures," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2010.

[2] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi, "An OpenCL Framework for Heterogeneous Multicores with Local Memory," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 193–204.

[3] Khronos OpenCL Working Group. (2011). The OpenCL Specification Version 1.1 [Online]. Available: http://www.khronos.org/opencl

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 1998, pp. 212–223.

[5] Intel TBB. (2014). [Online]. Available: http://software.intel.com/en-us/intel-tbb/

[6] 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 *Proc. 32nd ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2011, pp. 12–25.

[7] 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.

[8] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the GPU," in *Proc. Conf. High Perform. Graph.*, 2010, pp. 29–37.

[9] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2008, pp. 114–124.

[10] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 147–156.

[11] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: Portable stream programming on graphics engines," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 381–392.

[12] J. M. Fifield, "Generating, optimizing, and scheduling a compiler level representation of stream parallelism," Ph.D. dissertation, Dept. Comput. Sci., Univ. Colorado, Boulder, CO, USA, 2011.

[13] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August, "Parallelism orchestration using DoPE: The degree of parallelism executive," in *Proc. 32nd ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2011, pp. 26–37.

[14] 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 *Proc. 2nd Biennial Conf. Innovative Data Syst. Res.*, 2005, pp. 277–289.

[15] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive Control of Extreme-scale Stream Processing Systems," in *Proc. IEEE 26th Int. Conf. Distrib. Comput. Syst.*, 2006, p. 71.

[16] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–12.

[17] A. Pop and A. Cohen, "OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs," *ACM Trans. Archit. Code Optimization*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013.

[18] G. Hager, G. Jost, R. Rabenseifner, and J. Treibig. (2012). Performance-oriented programming on multicore-based clusters with MPI, OpenMP, and hybrid MPI/OpenMP [Online]. Available: http://blogs.fau.de/hager/tutorials/isc12/

[19] R. L. Collins and L. P. Carloni, "Flexible filters: Load balancing through backpressure for stream programs," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, 2009, pp. 205–214.

[20] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2011, pp. 22–32.

[21] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 365–376.

[22] 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 *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 1–8.

[23] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proc. Linux Symp.*, 2012, pp. 119–130.

[24] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops*, 2010, pp. 170–177.

[25] Storm. (2014). [Online]. Available: http://storm-project.net/

[26] 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.

[27] 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.

[28] G. Granunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors," *Computer*, vol. 23, no. 6, pp. 60–69, Jun. 1990.

[29] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *Proc. 22nd ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 1–10.

[30] MIT CSAIL Supertech Research Group. The Cilk Project. (2014). [Online]. Available: http://supertech.csail.mit.edu/cilk/

[31] StreamIt SVN. (2014). [Online]. Available: https://svn.csail.mit.edu/streamit/

[32] S. G. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *IEEE Trans. Comput.*, vol. C-36, no. 11, pp. 1367–1369, Nov. 1987.
[33] NVidia. NVIDIA OpenCL SDK Code Samples. (2014). [Online]. Available: http://tinyurl.com/ayo8brs
[34] 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 *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[35] Perf: Linux profiling with performance counters. (2014). [Online]. Available: https://perf.wiki.kernel.org/
[36] Intel Cilk Plus. (2014). [Online]. Available: http://software.intel.com/en-us/intel-cilk-plus/
[37] Intel OpenCL SDK. (2014). [Online]. Available: http://tinyurl.com/adaaw7r
[38] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.
[39] CPU Usage Limiter for Linux. (2014). [Online]. Available: http://cpulimit.sourceforge.net/
[40] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
[41] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services," in *Proc. 18th ACM Symp. Operating Syst. Principles*, 2001, pp. 230–243.
[42] 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 *Proc. 20th Annu. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Language, Appl.*, 2005, pp. 519–538.
[43] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *Proc. 37th Int. Conf. Parallel Process.*, 2008, pp. 536–545.
[44] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for NUMA-aware contention management on multicore systems," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 557–558.
[45] Z. Majo and T. R. Gross, "Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead," in *Proc. Int. Symp. Memory Manage.*, 2011, pp. 11–20.

Changwoo Min received the BS and MS degrees in computer science from Soongsil University, Korea, in 1996 and 1998, respectively, and the PhD degree from the College of Information and Communication Engineering of Sungkyunkwan University, Korea, in 2014. From 1998 to 2005, he was a research engineer in Ubiquitous Computing Lab (UCL) of IBM Korea. Since 2005, he has been a research engineer at Samsung Electronics. His research interests include parallel and distributed systems, storage systems, and operating systems.

Young Ik Eom received the BS, MS, and PhD degrees from the Department of Computer Science and Statistics of Seoul National University, Korea, in 1983, 1985, and 1991, respectively. From 1986 to 1993, he was an associate professor at Dankook University in Korea. He was also a visiting scholar in the Department of Information and Computer Science at the University of California, Irvine from September 2000 to August 2001. Since 1993, he has been a professor at Sungkyunkwan University in Korea. His research interests include parallel and distributed systems, storage systems, virtualization, and cloud systems.