2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing

# A Region-Based Approach to Pipeline Parallelism in Java Programs on Multicores

Yang Wang
Shenzhen Institutes of Advanced Technology
Chinese Academy of Sciences, China
yang.wang1@siat.ac.cn

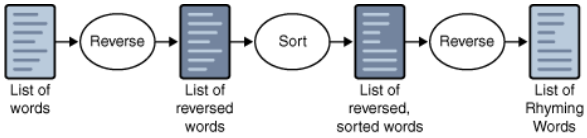
Kenneth B. Kent
CAS—Atlantic, Faculty of Computer Science
University of New Brunswick, Fredericton, Canada
ken@unb.ca

***Abstract*—As multicore architectures dominate mainstream computing platforms, migrating legacy applications into their parallel representation becomes a viable approach to reaping the benefits of multicore computing. In this paper we present a dataflow analysis tool that assists programmers to exploit the coarse-grained pipeline parallelism in stream-like Java applications on multicores. With this tool, programmers can partition a source Java program into a set of regions, which as pipeline stages, are connected via data channels to execute on multicores. To this end, we propose a simple yet effective framework that leverages JVMTI (JVM Tool Interface) and Javaagent techniques to track the data communication patterns among different regions, whereby a stream graph of the program is constructed. The graph is further used by the framework and programmers to re-factor the Java application into a pipelined program so that the potential of the multicores can be fully utilized. This procedure can be repeated in several rounds to progressively improve the performance. By applying this tool to several selected benchmarks, we demonstrate the effectiveness of the approach in terms of the performance improvements of some stream-like Java applications.**

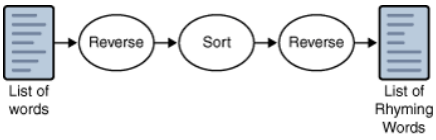
## I. Introduction

Multicore platforms have demonstrated their advantages over traditional computing platforms with respect to performance and cost benefits for many computational tasks. Therefore, migrating legacy applications into their parallel representation becomes a promising approach to fully exploiting the potentials of multicore computing. Although this approach is attractive, developing an effective one is still a great challenge.

There could be a variety of coarse-grained parallelisms in applications such as *task parallelism*, *data parallelism*, and *pipeline parallelism*, each having its own merits and demerits. Of these, pipeline parallelism is the basis for the stream programming paradigm, which is well-suited to multicore architectures. For example, consider a class that processes a collection of log files using a sequence of string manipulation utilities. It would be efficient if the output of one of these methods could be used as the input for another so that a series of method calls could be connected together to perform a higher-order function. An illustrative example, called *rhyming words* (RW), is shown in Figure 1, where one could reverse each word in a list, sort the words, and then reverse each word again to create a list of rhyming words. Without pipe streams,




(a) Without a pipeline, RW must store intermediate results




(b) With a pipeline, RW can send information directly from one stage to another

Fig. 1. An example to show the advantage of pipeline parallelism (quoted from [1])

the program would have to store the results somewhere (such as in a file or in memory) between each step, resulting in communication stall as shown in Figure 1(a). However, with pipe streams the output from one method could be immediately piped into the next without the stall between stages (Figure 1(b)).

We argue that for some legacy Java programs, their source code can be partitioned into different segments (i.e., *regions*), and these segments are not always dependent in terms of dataflow dependency, rather, they are independent and thus could be executed in parallel. By leveraging this parallelism, the performance of stream-based Java programs can be improved by converting it to a coarse-grained pipelined program running on a multicore. Although the benefits are obvious, the challenges are also enormous, given the fact that many Java legacy applications were designed and implemented without considering leveraging the pipeline parallelism. In particular, we need to address the following problems for pipelining a Java program:

1) What is the process pattern of the Java program on an input file? Does it exhibit any characteristics of pipeline processing?
2) How do we define the stages of the pipeline if the processing can be expressed as a pipeline?
3) How do we synchronize the pipeline between stages over data communication channels?

2377-5750/17 $31.00 © 2017 IEEE
DOI 10.1109/PDP.2017.69

4) How do we smooth the pipeline by load balancing the workloads in each stage?

To address these problems and verify the arguments, in this research we propose a dynamic dataflow analysis tool that could assist programmers to exploit the coarse-grained pipeline parallelism in stream-based Java applications on multicores. The programmers could utilize this tool to partition a source Java program into a set of regions, organized as a pipeline graph via data channels to run on multicores. Again, a region here is informally defined as a segment of source code that can run independently from other parts of the program. We achieve this goal by presenting a simple, yet effective framework that takes advantages of JVMTI (JVM Tool Interface) [2] and Javaagent [3] techniques to detect the data communication patterns between different code regions of the program. The resulting communication patterns are represented as a *stream graph* which is abstracted as a weighted *directed acyclic graph* (DAG), where its weighted nodes denote the code regions associated with the workload (as a percentage) and its weighted edges specify the data traffic flowing along the communication channels. The graph is then used by both the framework and programmers to re-factor the Java application into a pipelined program so that the potentials of the multicores can be fully exploited. This procedure can be repeated through several rounds to progressively improve the performance. To the best of our knowledge, it is the first work to facilitate the exploitation of the coarse-grained pipeline parallelism of legacy Java applications in a semi-automatic way to fully exert the unique advantages of multicores.

The paper proceeds as follows. We overview related work in Section II, and describe a motivational example as well as the basic idea of our approach in Section III. We present the proposed approach and the framework in Section IV. After that, we make some case studies to evaluate the approach in Section V and conclude with some future work in the last section.

## II. Related Work

Deployment of stream programs on multicores for efficient execution has been intensively studied for a decade as multicores were becoming the mainstream computing platforms. Gordon *et al.* [4] exploit diverse coarse-grained parallelisms, including task, data, and pipeline in stream programs for multicore platforms. Wang *et al.* [5] achieve a similar goal by using a machine learning based approach to partition streaming parallelism. Recently, Farhad *et al.* [6] propose a profile-guided approach to deployment of stream programs on multicores. Although these studies are interesting in terms of the adopted methodologies and the obtained results, and also bear some similarities in spirit to our idea in this paper, they are oriented to a stream language like *StreamIt* [7], rather than Java. To the best of our knowledge, there are few studies on exploiting the parallelism in stream-based core Java applications for efficiency on multicores [8], [9].

Spring *et al.* [8] study this problem by proposing *StreamFlex* that enables Java to combine stream processing code with traditional object-oriented components for high-throughput and low-latency applications. To achieve the goal, StreamFlex extends the core Java, and at the same time, improves the JVM with real-time capabilities, transactional memory and type-safe region-based allocation. Although StreamFlex is a rich and expressive language that can be implemented efficiently, it cannot satisfy our requirements to handle legacy Java programs without modifying the JVM.

XJava is an extension of Java with new parallel constructs to facilitate the exploitation of parallelism based on an object-oriented stream programming paradigm on shared memory machines [10]. Otto *et al.* [11] introduce an XJava-based tuning mechanism, which can leverage the compiler's context knowledge about the program's parallel structure to configure the tuning parameters (e.g., the number of pipeline stages) at runtime, instead of identifying and adjusting tuning parameters manually. Unlike XJava, our target is core Java programs, and parameter tuning in our framework is semi-automatic as we do not have a parallel structure to figure out for configuration.

A core Java-based investigation is conducted by Sun and Zhang [9] where two new approaches are proposed to automatically parallelize Java programs at runtime. The approaches, which rely on run-time trace information, dynamically recompile Java bytecodes that can be executed in parallel. Our idea is similar to this one by nature, but has a clear distinction that exploiting pipeline parallelism in stream-based Java programs is our interest, rather than the loop and trace parallelizations.

A highly related work to ours is by Thies *et al.* [12] where the same problem in a set of stream-based C programs is targeted by a practical approach to exploit the coarse-grained pipeline parallelism. Although their techniques show some value in practice for certain C programs, they cannot be directly applied to our case since our target is the pipeline parallelism in legacy Java programs. The language features are totally different, so are the frameworks to achieve the goals. Similarly, Lazarescu *et al.* [13], introduce an interactive trace-based analysis toolset for manual parallelization of C programs.

Recently released Java 8 introduces a new abstract layer, called *stream*, to transfer data in such a way that it can be processed as steady and continuous stream. Our approach bears some similarities with stream concept, but the streamlined region-level operations are more powerful and more easily structured than the stream operations in Java 8.

## III. A Motivational Example

In this section, we first use Memory Analyzer (MA) [14] as a motivational example to show how the process pattern of a legacy Java program could be detected and analyzed for potentially exploiting the pipeline parallelism, and then introduce the basic idea to achieve this task.

### A. Memory Analyzer

Memory Analyzer is an open source software tool in Java used to process and analyze the heapdump files to detect

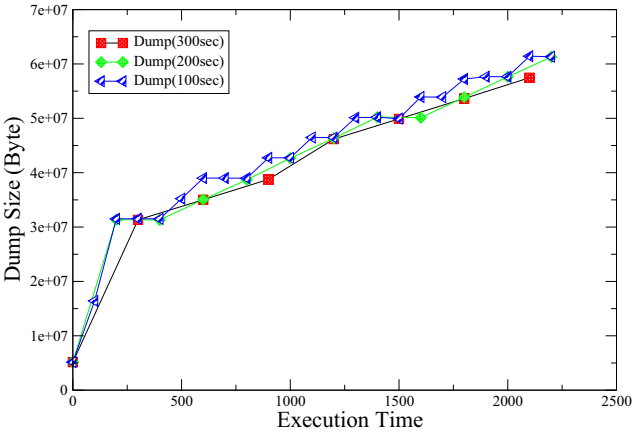



Fig. 2. Sizes of the generated portable heap dump files at different dumping rates during the execution.

any performance/memory anomalies in Java applications. To obtain the process patterns whereby the program is pipelined, we first observe the control-flow of MA when it is used to process a Java heapdump file. To this end, we run SpecJBB2005 benchmark on an IBM J9 virtual machine which is supported by Ubuntu 12.04 with hardware configuration: `3392.183` MHz processor, with a total of 9 hardware threads activated, each with a `8192KB` cache. During the computation, the IBM J9 Java virtual machine is periodically triggered (by external event "`kill -3 $pid`") to generate the dump files in *Portable Heap Dump (phd)* format. Figure 2 shows the sizes of the generated phd files at different dumping rates (`300s`, `200s` and `100s`) in the execution.

From this figure, one can easily see how the heap size is monotonically increased as the computation proceeds. We randomly selected a phd file and made the following observations from the execution of the benchmark,

1) Processing DTFJ image from phd file:
   a) found `8,158,376` identifiers, `8,157,118` objects, `1,258` classes
   b) `1,318` unreferenced objects marked as extra GC roots
   c) using conservative garbage collection root support with `2,577` roots
2) Removing unreachable objects
3) Purging dead objects from DTFJ image
4) Calculating dominator tree

The process has four stages. The first stage is to process the DTFJ image from the phd file. It takes `1,719ms` to get the DTFJ image and `19,565ms` to parse the DTFJ image, where `8,158,376` identifiers, `8,157,118` objects, and `1,258` classes are found, and another `1,318` unreferenced objects are marked as extra GC roots in addition to the supported `2,577` roots using the conservative garbage collection. Stage `2, 3` and `4` take another `15,015ms` out of the total `36,229ms` to process the whole file. Although the workloads are not well-balanced between stages, the observations still strongly evidence that MA can be transformed into a stream-like application as the flow of the execution at stage level is acyclic even inside each stage.

### B. Basic Idea

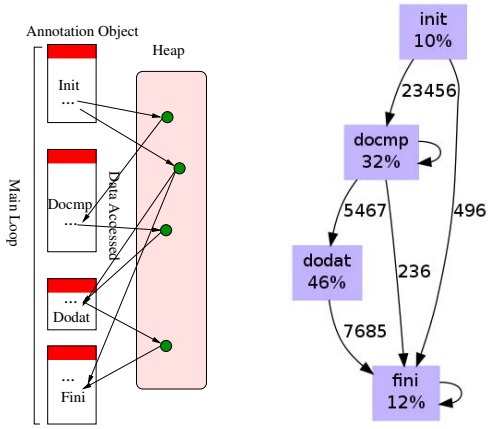
Instead of resorting to any compiler technology, we manually partition the target Java program at the source-code level into a sequence of coarse-grained pipeline stages that are connected by data channels. Our key insight is built on the assumption that the control-flow of the Java program can be modeled as interactions between a collection of source-code segments without cyclic controls (i.e., pipeline DAG), and furthermore, the data communication across the pipeline stages is stable throughout the life-time of the program. With this assumption, we can undertake a dynamic analysis of the code for region partitioning by observing the dataflow communication patterns between different regions. Such dataflow information is further used to construct a *stream graph*, a weighted DAG, where the weighted nodes represent the regions with computational workloads and the weighted edges denote the communication weights from source regions to target regions. The stream graph is in turn exploited to optimize the region partitions by combining or splitting the partitioned regions to balance the computational and communicational loads in the subsequent iterations until the final partitioned regions are satisfactory, according to some performance criteria.

To implement this idea, we first equip programmers with a set of *annotation classes*, which are empty classes implementing the `Phasiable` interface, one for each region. The annotation classes are used by the programmers to create respective *annotation objects* at appropriate locations to specify the boundaries of the regions. We then leverage the tool to record all communication across those boundaries between each pair of the regions during a training run.

After the regions and their inter-communication patterns are determined, we then isolate each region as a *pipeline stage* by encapsulating its computational logic into single or multiple threads (depending on the parallelism of the computation as well as the available resources) and allow the stages to communicate with each other over a data-channel object. In other words, for a pair of communicating stages, one stage writes into the channel and the other reads from it based on the producer/consumer pattern. As a result, synchronization is required on the data channel object. Finally, the pipelined program is recompiled to execute on the multicores, which not only pipeline the different iterations of the program, but also allow independent regions to run in parallel.

## IV. Analysis Tool Framework

Our approach is to develop a generic and pragmatic approach to exploiting coarse-grained pipeline parallelism in stream-based Java programs for efficiency on multicores. To have portability and applicability, we have extra requirements for the approach with respect to JVM modifications and efforts in Java program transformations.




(a) Partition into different regions with the annotation objects

(b) Corresponding stream graph

Fig. 3. Data communication detection and its constructed stream graph

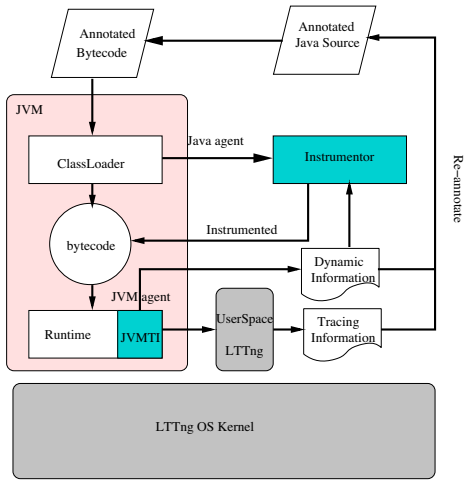



Fig. 4. Analysis tool framework for region-based parallelisms in Java programs

### A. Annotation Class

To detect dataflow communication patterns, we first allow programmers to instrument the Java source code with annotation objects at appropriate sites to partition the program into a sequence of regions. Each annotation object represents a segment of code, defined as a region. As such, the data accesses between any pair of regions can be represented by the relationships between the corresponding annotation objects. All these relationships together construct a stream graph as we described above.

To this end, we define an annotation class `Region` for each region as follows

```
public interface Phasiable {}
public class Region implements Phasiable {
}.
```

The analysis tool then leverages JVM's *lazy class loading* to recognize the regions together with the execution time in the training run. An illustrative example of this idea is shown in Figure 3 where the annotation object for each partitioned region, as well as the generated stream graph, are depicted.

### B. Restrictions on Regions

Based on our idea, regions can be defined in different granularities in terms of Java language structure, each could expose different parallelisms, and exhibit different shapes of the stream graph. Currently, we impose some restrictions on the placements of the annotation objects for easily reasoning about the structure of the graph. In particular, we force all annotation objects to appear within the main loop body of the *main* entry method, but are not within a nested loop, a nested control flow or as a part of a method of another class. These restrictions can be worked around by re-factoring the program using techniques such as *loop distribution* and *function inlining*. Moreover, programmers can evaluate the quality of the partition by figuring out how tight two communicational regions are and what classes are needed to pipeline between them. A region becomes a pipeline stage after its computational logic is implemented by a single or multiple threads, and the communication between any pair of regions is established via a data channel.

### C. JVMTI Tracing Agent

To detect the data communication pattern and construct the stream graph, we need to monitor each data (field) access bytecode instruction (e.g., `putfield` and `getfield`) at runtime and also record to which region it belongs. To avoid becoming overwhelmed by JVM implementation details, our approach is to leverage JVMTI to implement a tracing agent to gather the required information at runtime without instrumenting the Java source code. JVMTI is a native programming interface to the JVM to provide tools with both a way to inspect the JVM state and to control application execution in the JVM. It allows the agent to be implemented as a dynamic shared library to inspect the state and to control the execution of applications running in the JVM. On the other hand, the JVMTI agents are not governed by the JVM rules and are thus not affected by the JVM internals such as the garbage collection or runtime error handling. As such, the JVMTI option, we think, is the most convenient way for our purpose although there are other techniques to achieve the same functionality. Figure 4 shows the framework, which includes tracing and analyzer sub-systems.

To construct the stream graph, the JVMTI agent needs to accomplish two key tasks. First, it can identify/recognize the traced objects in the life time (e.g., create event) to investigate their information. Unlike C/C++, due to the impact of garbage collection in Java, the address of an object is not a fixed value, and thus cannot be used to identify the object. Similarly, the hashcode of an object may not be unique either. So we have to find a way to solve this problem. Second, the agent can record the field access&modification events to setup

data dependencies between regions, especially, the accesses or modifications to the local variables. Of course, there are also other factors influencing the stream graph such as native code and reflection. But in this paper, we only focus on the first two key tasks, and leave others as future work.

*a) Field Access Tracking:* When a class is loaded by the `ClassLoader`, the JVMTI agent can intercept this event and trigger its corresponding callback functions to process the event. In particular, when an annotation class is loaded, a new region is opened or the current active region is updated, which is recorded by the agent; the bytecodes following the opened/active region share the same region number until another annotation object opens up/activates a new region.

In particular, to monitor all the field visits we set a watch on the fields for each class after it has been prepared. In particular, we need to catch the event `JVMTI_EVENT_CLASS_PREPARE` and in its callback do the following:

```
(*jvmti)->GetClassFields(jvmti, klass,
                    &field_num, &fieldIDs);
int i;
for(i=0; i < field_num; i++)
{
   (*jvmti)->SetFieldModificationWatch(
               jvmti, klass, fieldIDs[i]);
   (*jvmti)->SetFieldAccessWatch(jvmti,
                      klass, fieldIDs[i]);
}
```

*b) Local Variable Tracking:* To monitor all the field visits of a particular object, the agent sets a watch in its callback on the fields for each class after it has been prepared (i.e., `JVMTI_EVENT_CLASS_PREPARE` event). However, monitoring the visits to local variables is relatively difficult as the JVM Agent is not informed when the local variables are visited. To address this issue, the agent first uses the index of local variables in the *frame structure* to track and specify the variable in the method, and cache the bytecode body of the *main* method when the class is prepared, then it iterates over all the local (array) variable `xload` and `xstore` instructions (e.g., `dload` and `dstore`) and sets breakpoints on them, and finally, catches the breakpoint events to retrieve back the instruction information for building up the producer-consumer relationships.

The JVMTI agent is the core of this framework, which can be triggered by any pre-defined event of the Java program to perform the specified operations for a fine-grained runtime trace. We are particularly interested in collecting various load/store instructions with respect to object fields and local variables, and object creation instructions as aforementioned. Within the agent, the trace is written into a buffer in a format of "opcode #region" as dynamic trace information. The region number refers to the hosting region or target region depending on the operation code. For example, if the operation code is related to a write, the #region is the current region, which is also marked as the most recent region for this updated field. Otherwise, if the operation code is read related, the #region will refer to the region that most recently defines the field (object) being read.

*c) Stream Graph Construction:* The dynamic trace information is further used by the JVMTI agent to create the stream graph. To this end, the agent maintains a table that indicates, for each accessed field (object), the number of the region (if any) that last wrote to that field. On encountering a store operation (e.g., `putfield`), the agent records which region is writing the field. Likewise, on every load operation, the agent does a table lookup to determine the region that produced the value being consumed by the load (e.g., `getfield`). Eventually, the unique producer-consumer relationship is established in a list that is output at the end of the program, along with the stream graph. During this process, more information can be gleaned and annotated on the graph. For example, nodes can be labeled with the amount of work they perform (in percentage), while edges are labeled with the number of communicated bytes.

Finally, the stream graph is used as a guideline for programmers to adjust the regions as well as the program itself (e.g., collapsing strongly connected regions), and the whole process is repeated until a satisfactory pipelined program is available. At that time, we can de-instrument the annotation objects and consider a pipelined program.

### D. Data Channels

The data channel is the primary challenge in our design. In general, it can be achieved either by explicitly modifying the code to implement a buffer shared by producer and consumer or by implicitly wrapping the existing code in a virtual environment that performs the buffering automatically [12]. The first approach requires a deep understanding of the program in order to infer all of the variables and object references that need to be remapped to a new location (e.g., a data channel object) for communication between pipeline stages. Such an approach seems largely intractable. The second approach avoids the complexities of the first one. We advocate to take advantage of the Java *pipe stream* to channel the output from one thread into the input of another. The principal reason to use the pipe is to keep each thread as simple as possible as the producer thread can simply send its results to a stream and forget about them, and the consumer reads the data from the stream, without having to care where the data comes from. By using pipes, we can connect multiple threads with each other without worrying about *thread synchronization*. Note that piped streams are only appropriate if the communication between the threads is at a low level. In other situations, we can use *queues*. The producing thread inserts objects into the queue, and the consuming thread removes them.

To this end, we leverage *Javassist* [15] to design a *Javaagent*[1] (i.e., *Instrumentor* in Figure. 4) that encapsulates the computational logic in each region into a single thread (in our current implementation). Javassist is a class library for editing

[1]Javaagent is a software component that provides instrumentation capabilities to an application.

bytecodes in Java. It can not only enable Java programs to define a new class at runtime and to modify a class file when it is loaded into the JVM, but also provide two levels of APIs that allow users, with source-level API, to specify inserted bytecodes in the form of source code (Javassist compiles it on the fly), and in contrast, with bytecode-level API, to directly edit a class file as other editors.

Our approach is to use the Javaagent to instrument the code whereby the stream graph is exploited to identify the stages as well as related classes that need to implement the communication between stages via pipes. The pipes for those identified communication stages are set up by leveraging *anonymous inner class* for each stage build-up.

To this end, the Javaagent creates pipe objects at the initialization phase of the program for each pair of communication regions and replaces the related data access instructions (bytecode known from detecting the data communication behavior) with those writing to or reading from the corresponding pipe objects. The regions become pipeline stages and the whole transformed program can be re-compiled to execute on a multicore. We will see from a case study that this procedure and operations are not always sufficient to pipeline any non-trivial Java applications in an efficient way. Therefore, programmer assistance facilitated by the framework is always needed.

### E. Load Balancing

Applications parallelized using the pipeline model are very sensitive to load balancing between stages since imbalanced stages would introduce bubbles into the pipeline, which dramatically degrades the performance. Therefore, the programmer must either evenly distribute the workload among the stages or leverage system mechanisms to shift resources from idle stages to the busiest stages at runtime. Although we have already partitioned the workload as balanced as possible at the pre-compilation phase, it is still possible that the dynamic behavior of the execution is not consistent with our expectation. This is because the number of threads in each stage has not been determined and the data channels (i.e., Java pipes) have not yet been set up. To address the load balancing issue at runtime, we use dynamic scheduling to share the load among the different stages. This general strategy can be achieved by either *overscription* or *work stealing*. Here, the overscription refers to the technique that allows the application to create more threads than available cores and rely on scheduling in the JVM, or even in the underlying OS, to keep all the cores busy (we suggest to set the number of threads to be 3-4 times the number of available cores). In contrast, the work stealing can keep one active thread per core and assign several work units to each thread; whenever a thread completes its assigned work, it can query other threads for additional work.

### F. Bridging Java Applications and LTTng

In addition to those aforementioned methods, we also integrate the Linux Trace Toolkit: next generation (LTTng) into the framework. LTTng is a state-of-the-art tracing toolset for C/C++ programs tracing at both user space and kernel space. The tracing information is very useful to assist the programmers optimize the annotation object instrumentation. Currently, we have integrated this toolset into the IBM J9 JVM for monitoring the user and kernel spaces [16]. However, in order to trace the Java programs, we have to extend the framework, where the *JVMTI agent* could be leveraged, by setting one or more LTTng tracepoints in corresponding callback functions for each traced event (the LTTng component in Figure 4). The hard part is getting the values of fields and method parameters, because in the current design only signatures (not values) are available from the JVMTI environments. The instrumented JVMTI Agent not only allows the user to avoid manual instrumentation of Java programs, but also enables LTTng to selectively trace Java programs based on interested JVM events. More important, with the integrated LTTng, the framework can detect related kernel-level tracing information, which could compensate the user-level JVM LTTng tracing to further help the programmers facilitate the partition of the Java programs for pipeline parallelism.

## V. Case Studies

In this section, we conduct two case studies to illustrate how useful the proposed framework is with respect to its functionality and performance benefits to stream-like Java legacy applications on multicores.

The experiments were conducted under Ubuntu 12.04 running on a hardware configuration of `8GB` memory and a `3392.183` MHz quadcore processor, with a total of `8` threads, each with `8192KB` cache. The preliminary results demonstrate that the proposed framework is a promising approach to exploit the coarse-grained pipeline parallelism in stream-based Java programs for efficiency on multicores.

### A. Functionality

We use *Budget Distribution Solver* (BDS) [17] as an example to illustrate how a Java program can be partitioned into different regions which are connected into a general DAG-shaped pipeline. BDS is a simulator developed for budget-driven MapReduce job scheduling in the cloud. The structure of BDS is simple; it contains a main loop to schedule a queue of MapReduce jobs according to a budget-centric metric. Each job thus experiences several stages that are executed one after another. Therefore, a developer does not need to refactor the source code to enable our approach. Rather, we can simply create the annotation objects in appropriate sites to partition the Java code into different regions and thereby to generate the stream graph. Figure 5 shows a recognized stream graph that is obtained by scheduling `100` MapReduce jobs described in [17]. The workloads in each region are well balanced, and the weight shown on each arc illustrates the number of data dependencies from source node to target node. From this figure, we can observe that the framework can well recognize the region in the program. More specifically, `Region 1` and `Region 2` are executed in a sequential mode while `Region 3` and `Region 4` are independent, and they can be executed

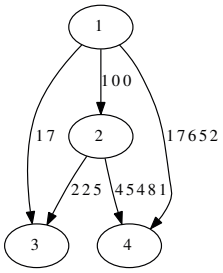



Fig. 5. A recognized stream graph of BDS [17].

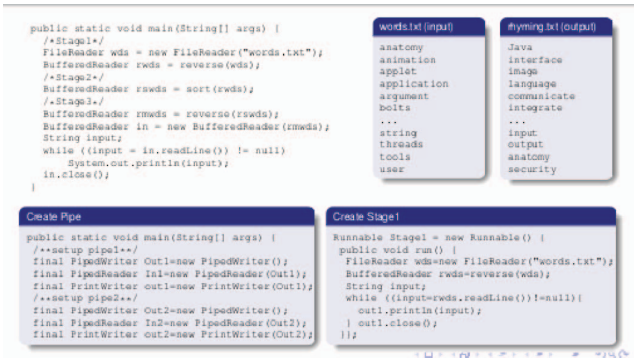



Fig. 6. Case Study: Pipelined RW Java application (1)

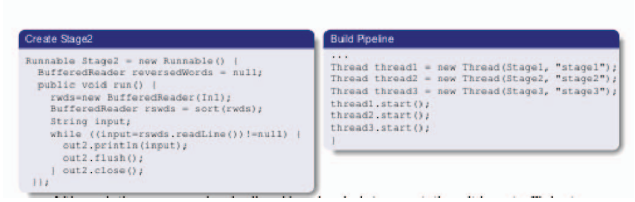



Fig. 7. Case Study: Pipelined RW Java application (2)

in parallel. This observation validates our argument that the segments of Java code (i.e., regions) are not always dependent, rather, they are independent in certain cases, and thus could be executed in parallel for high performance.

Note that this graph was not generated in one round of the workflow defined by the framework. Instead, it was obtained by multiple rounds of the workflow, progressively refining with the involvements of the programmers who may rely on the tracing and other useful information for the goal. For example, we integrated the LTTng Tracing Toolset into the framework, which can gather a wide range of tracing information at both user level and kernel level to facilitate the programmers well defining each region. The performance overhead in this process is not a serious issue as it only occurs in the initial region-recognization phase. Instead, the overall performance changes due to the code transformation is usually a pragmatic concern of the programmers, which are our topic in the next experiment.

### B. Performance Benefits

We now evaluate the performance benefits of this framework to stream-like Java programs. To this end, we use the RWL benchmark, which creates a rhyming word dictionary as shown in Figure 1, as an example to show how the transformed Java program would look using our tool framework, and then evaluate its performance improvement relative to the non-pipelined counterpart. We selected this benchmark as its process pattern represents many log-file based processing (e.g., top-k) in reality.

Figures 6 and 7 show the pipelined RW Java program at the source-code level after the transformation. All statements related to the pipeline definitions and the thread creations have been instrumented to construct the pipelined program according to the stream graph. In fact, with the source-level API of Javassist, the Instrumentor can specify inserted bytecodes in source code form. Note that for a Java program with a more general stream graph like BDS [17], we have a similar code structure, but with more complicated dependency relationships among the regions. As a consequence, the pure dataflow-driven execution, as performed in the pipeline program, may not be effective in the general case. Therefore, an efficient region-based scheduling algorithm for the general stream graph is highly desired.

To evaluate the performance benefits, we obtain a set of log files in CVS format from DeviantART [18], which specify the all deviations (*deviations.cvs*), the categories of deviations (*categories.cvs*), the user comments on deviations (*comments.cvs*), the daily best deviations (*daily_deviations.cvs*), the user favorite deviations (*favour-ites.cvs*) and all the viewed deviations (*views.cvs*). The performance improvements of the region-based pipeline RWL program on the set of daily log files from DeviantART [18] are shown in Figure 8.

From this figure, the performance of the pipelined program with different input datasets is consistently better than that of the non-pipelined version. This observation is not difficult to

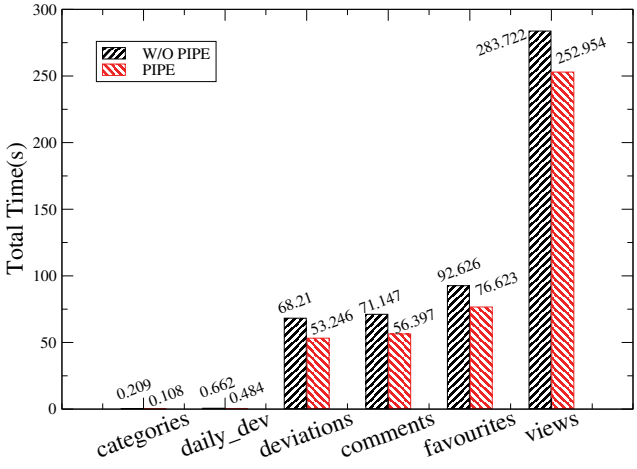



Fig. 8. Performance comparison between using and not-using pipeline parallelism for the RW Java application on the log files in Table I.

TABLE I
A SET OF LOG FILES FROM DEVIANTART [18]

| Name(CSV) | Record | Size (MB) | Imprv(%) |
|---|---|---|---|
| categories | 2,297 | 0.084 | 48.3 |
| daily_devs | 71,493 | 1.470 | 26.9 |
| deviations | 8,995,225 | 228.000 | 21.9 |
| comments | 9,387,924 | 249.000 | 20.7 |
| favourites | 12,070,853 | 321.000 | 17.3 |
| views.csv | 37,161,022 | 814.000 | 10.8 |

understand as the pipelined program removes the communication stall between the stages.

Since we run the same program on different input datasets, it is easy to understand the impact of the inputs on the overall performance. To this end, we compare the performance improvements, together with the numbers of records (Record) and the sizes of the files (Size) in Table I. As the Record and Size dramatically increase, say `10,000` times from `0.084M` to `814M`), the performance improvements slowly drop from `48.3%` to `10.8%`, only slowing down by a factor of `5`, which fully demonstrates that our approach is effective to exploit the pipeline parallelism in Java programs for high performance on multicores.

## VI. CONCLUSION

Although the framework is promising, based on our current evaluation, there is still some room left to be exploited for further improvements:

1) *Removing Annotation Objects:* By analyzing the bytecode based on the concept of *basic block*, the Javaagent could transparently partition the program at those points that allow each region to contain a set of basic blocks (i.e., a *super block*). With this improvement, the involvement of the programmer to partition the program can be significantly reduced.
2) *Algorithms to Optimize the Stream Graph:* We need algorithms to optimize the stream graph to balance the computation and communication by merging and splitting the region nodes, instead of each time asking programmers to manually complete the task.
3) *Efficient region-based DAG Scheduling:* We have noted that not all code regions are executed sequentially, some regions can execute concurrently. Based on this observation, we need to have a more efficient scheduler to run the DAG on a multicore platform. Since most contemporary JVMs delegate the thread scheduling to the native OS, how to achieve this scheduling may involve a cooperation between the JVM and OS.

Additionally, we plan to compare our improvements with hand-tuned parallel versions of the studied applications.

## ACKNOWLEDGMENT

The authors would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program and the Science and Technology Planning Project of Guangdong Province (No.2015B010129011 and 2016A030313183). Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project and the Centre for Advanced Studies—Atlantic for access to the resources for conducting our research. We also thank Stephen MacKay for his careful proofreading and editing the paper to improve its quality.

## REFERENCES

[1] "How to use pipe streams," 2014, http://enos.itcollege.ee/~jpoial/docs/tutorial/essential/io/pipedstreams.html.
[2] "JVMTI," 2014, http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html.
[3] "Javaagent," 2014, http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/instrument/package-summary.html.
[4] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 151–162, Oct. 2006.
[5] Z. Wang and M. F. O'Boyle, "Partitioning streaming parallelism for multi-cores: A machine learning based approach," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 307–318.
[6] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz, "Profile-guided deployment of stream programs on multicores," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES '12, 2012, pp. 79–88.
[7] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02, 2002, pp. 179–196.
[8] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: High-throughput stream programming in java," *SIGPLAN Not.*, vol. 42, no. 10, pp. 211–228, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1297105.1297043
[9] Y. Sun and W. Zhang, "On-line trace based automatic parallelization of java programs on multicore platforms," in *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on*, Feb 2011, pp. 35–43.
[10] F. Otto, V. Pankratius, and W. Tichy, "Xjava: Exploiting parallelism with object-oriented stream programming," in *Euro-Par 2009 Parallel Processing*, ser. Lecture Notes in Computer Science, H. Sips, D. Epema, and H.-X. Lin, Eds. Springer Berlin Heidelberg, 2009, vol. 5704, pp. 875–886.
[11] F. Otto, C. Schaefer, M. Dempe, and W. Tichy, "A language-based tuning mechanism for task and pipeline parallelism," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. DAmbra, M. Guarracino, and D. Talia, Eds. Springer Berlin Heidelberg, 2010, vol. 6272, pp. 328–340.
[12] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40, 2007, pp. 356–369.
[13] M. T. Lazarescu and L. Lavagno, "Interactive trace-based analysis toolset for manual parallelization of c programs," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, pp. 13:1–13:20, Jan. 2015.
[14] M. Analyzer, 2013, http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/.
[15] "Javassist," 2014, http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/.
[16] Y. Wang, K. B. Kent, and G. Johnson, "Improving j9 virtual machine with lttng for efficient and effective tracing," *Software: Practice and Experience*, pp. 973–987, 2015.
[17] Y. Wang and W. Shi, "Budget-driven scheduling algorithms for batches of mapreduce jobs in heterogeneous clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 3, pp. 306–319, Jul. 2014.
[18] "Deviantart," 2015, http://www.deviantart.com.