XJava: Exploiting Parallelism with Object-Oriented Stream Programming

Frank Otto, Victor Pankratius, and Walter F. Tichy

University of Karlsruhe, 76131 Karlsruhe, Germany
{otto,pankratius,tichy}@ipd.uka.de

Abstract. This paper presents the XJava compiler for parallel programs. It exploits parallelism based on an object-oriented stream programming paradigm. XJava extends Java with new parallel constructs that do not expose programmers to low-level details of parallel programming on shared memory machines. Tasks define composable parallel activities, and new operators allow an easier expression of parallel patterns, such as pipelines, divide and conquer, or master/worker. We also present an automatic run-time mechanism that extends our previous work to automatically map tasks and parallel statements to threads.

We conducted several case studies with an open source desktop search application and a suite of benchmark programs. The results show that XJava reduces the opportunities to introduce synchronization errors. Compared to threaded Java, the amount of code could be reduced by up to 39%. The run-time mechanism helped reduce effort for performance tuning and achieved speedups up to 31.5 on an eight core machine.

1 Introduction

With multicore chips, software engineers are challenged to provide better performance by exploiting parallelism. Although well-established languages such as C++ or Java use the thread model for parallel execution, this model has turned out to be error-prone and difficult to handle for large programs. Programmers are forced to think on low abstraction levels and consider many details of synchronization. As a result, synchronization bugs, deadlocks, or data races are likely to occur. The growing complexity of multithreaded software makes it painful to locate such defects.

Several of our earlier case studies on parallel programming in various areas [10,11] show that best performance was achieved with a structured approach, considering different abstraction layers and patterns for parallelism. However, despite clear objectives and reasonable designs of the parallel programs, the implementation has always required a lot of effort.

XJava [9] is a programming language that extends Java with language constructs for parallelism, combining concepts from object-orientation and stream languages. Parallelism is exploited without requiring the programmer to explicitly define threads. The central construct in XJava is the task, which is syntactically similar to a method and which encapsulates an activity to be executed in

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 875–886, 2009.
© Springer-Verlag Berlin Heidelberg 2009

parallel. Tasks can be combined with special operators to obtain parallel statements that express master/worker or pipeline parallelism.

The XJava compiler basically performs a source-to-source transformation; XJava code is translated into native, instrumented Java code which is eventually translated into bytecode. The XJava compiler divides tasks and parallel statements into logical code units that are passed to a built-in scheduler. The scheduler transparently employs a fixed number of dedicated executor threads to execute these units. This approach intends to reduce the degree of non-determinism in execution behavior. The scheduling algorithms can easily be replaced or extended in the future to support real-time load balancing.

Our case studies show that XJava makes parallel programming simpler and more productive; there is less potential of inadvertently introducing synchronization bugs. To evaluate XJava's capabilities, we considered a desktop search engine and a selection of smaller benchmark programs. For each application, XJava code was compared in detail with equivalent threaded code as well as sequential code. Compared to threaded Java, XJava lead to code savings of up to 39% and required 87.5% – 100% less manual synchronization. We achieved speedups between 1.5 and 31.5 over sequential Java on an eight core machine.

In this paper, we briefly sketch the XJava extensions introduced in our previous work [9] and present the key details of the mechanisms in the compiler and runtime system. In addition, we present new results that demonstrate XJava's ability to implement parallel design patterns on several abstraction levels.

2 The XJava Language

XJava [9] extends Java with more intuitive constructs for general-purpose parallel programming. Several types of parallelism can be expressed in this language, although we only introduce two new keywords and two new operators.

XJava combines the principles of object-orientation with stream languages. XJava's design intends to make programs behave as programmer would intuitively expect. Thus, code understanding and maintainability should become easier as well. XJava makes parallelism accessible at higher abstraction levels and hides manual synchronization.

2.1 Tasks

Tasks are key constructs and are in fact special methods. Tasks encapsulate parallel activities that run within a thread, and can be combined to *parallel statements* (cf. Section 2.2). In contrast to methods, tasks have no return type, but typed input and output ports. A public task with input type `A` and output type `B` is declared as

```
public A => B t() { ... }
```

The input port receives a stream of data with elements of type `A`, and a stream of data elements of type `B` is sent to the output port. If a task has no input or

output, the respective port type is `void`. We distinguish between two kinds of tasks: *periodic tasks* and *non-periodic tasks*.

Periodic tasks define exactly one `work` block that is repeatedly executed for each stream element. For example, the code

```
String => String encode(Key key) {
  work(String s) { push encrypt(s, key); }
}
```

declares a periodic task that expects an input stream of `String` elements. The current element of the input stream is assigned to the local variable declared in the parenthesis after the `work` keyword. The work block – in this case, the encryption routine – is repeatedly executed for each element of the input stream. Each received element is encrypted with the key passed as an argument to the task. The `push` statement appends the encrypted element, i.e., a `String` object, to the output stream. The work block terminates when an end-of-stream token is received on the input port.

By contrast, **non-periodic tasks** do not have a work block, i.e., their code is executed only once. The body of a non-periodic task may contain a parallel statement for introducing nested parallelism.

2.2 Parallel Statements

Parallel statements are used to combine tasks to more complex parallel constructs. They consist of tasks that are concatenated by the operators "`=>`" and "`|||`".

The "`=>`" operator creates a *pipe statement*; it connects tasks on their input and output ports. This operator can be used to build pipelines of arbitrary lengths. For example, consider the tasks

```
void => String read(File fin) { /* ... */ }
String => void write(File fout) { /* ... */ }
```

`read` reads some file and turns its content into a stream of `String` objects; `write` accepts a stream of `String` objects and writes them to an output file. For given input and output files `fin` and `fout` and some key `key`, the statement

```
read(fin) => encode(key) => write(fout);
```

creates a pipeline for encoding a file and automatically exploits pipeline parallelism.

Concurrent statements are defined by the "`|||`" operator that concurrently executes tasks that are not allowed to be connected neither input nor output ports. For example, consider a task for simulating work based on randomly generated events:

```
void => void simulateW() { /* ... */ }
```

Suppose we want to run several simulations to collect data or retrieve statistical average values, we can use the following statements:

```
simulateW() ||| simulateW() ||| simulateW();   // 3 tasks
simluateW():i;                                 // i tasks
```

The first statement concurrently executes `simulateW` three times. Alternatively, the ":" operator can be used to define the number of parallel tasks dynamically.

3 The XJava Compiler

The XJava compiler extends the Polyglot compiler framework [8]. The XJava compiler checks if task declarations and parallel statements are valid and produces Java code that is instrumented with calls to the XJava scheduler (cf. Section 4).

3.1 Compiling Periodic Tasks

A periodic task defines exactly one `work` block; an arbitrary number of Java statements may precede or follow this block. Thus, the body of a periodic task can be divided into three parts that we call *BW* ("before work"), *W* ("work") and *AW* ("after work"). Consider a slightly modified form of the `encode` task from Section 2.1 with additional code before and after the work block:

```
public String => String encode(Key) {
  ... /* Java code */                           /* BW */
  work (String s) { push encrypt(s, key); }   /* W  */
  ... /* Java code */                           /* AW */
}
```

This task declaration is compiled to a wrapper class `EncodeWrapper`. Its superclass `PeriodicTask` is provided by the XJava runtime library; it is an abstract class defining the methods `beforeWork()`, `work(int)` and `afterWork()`. These methods are implemented in the wrapper class and contain the Java code before, in, and after the work block. The purpose of this separation is to divide a task into logical units that can be executed individually by the scheduler. This reduces the degree of non-determinism makes parallel execution easier to predict.

```
class EncodeWrapper extends PeriodicTask {
  Connector cin, cout;
  ...
  beforeWork() { ... /* Java code for BW */ }
  work(int n) {      /* repeatedly called by the scheduler */
    for (int i = 0; i < n; i++) {
      ... /* Java code for W */ }
  }
  afterWork() { ... /* Java code for AW */ }
}
```

`Connector` is a class for buffering elements that are exchanged between tasks and is part of XJava's runtime framework. `cin` and `cout` each contain buffers to receive or send elements. Whenever `encrypt` expects a new incoming element in the previous example, it calls the respective method of the buffer in `cin`; a `push` statement is mapped to a call to a similar method of the `cout` buffer. `work(int n)` provides a method to the scheduler to execute `n` iterations of the work block (cf. Section 4). The number `n` of iterations is determined by the scheduler. Local variables of each task are saved, i.e., each task instance has an internal state. When `work` has no more iterations to do and terminates, the `afterWork` method is called. In addition, the task will close its output stream and will be marked as finished.

3.2 Compiling Non-periodic Tasks

Since non-periodic tasks do not define a work block, their bodies cannot be separated. A non-periodic task `foo` is compiled to a wrapper class extending the class `NonPeriodicTask` in XJava's runtime framework. This class implements Java's `Runnable` interface; the `run` method contains Java code representing the body of the task:

```
class FooWrapper extends NonPeriodicTask {
  Connector cin, cout;
  ...
  run() { /* Java code for foo's body */ }
}
```

3.3 Compiling Parallel Statements

Task calls and parallel statements are managed by XJava's scheduler. Whenever a task is called, an instance of its corresponding class will be created and passed on to the scheduler. The scheduler decides when and how to execute it (cf. Section 4).

Parallel statements consist of a number of tasks connected by operators. The compiler checks in a parallel statement if input and output types of tasks match. Each task's input and output connectors are selected according to the operators involved. For example, consider the previous example of a pipe statement for encoding a file:

```
read(fin) => encode(key) => write(fout);
```

This statement is compiled to

```
Connector c1 = new Connector(); Connector c2 = new Connector();
ReadWrapper r = new ReadWrapper(fin, c1);
EncodeWrapper c = new EncodeWrapper(f, c1, c2);
WriteWrapper w = new WriteWrapper(fout, c2);
Scheduler.add(r); Scheduler.add(c); Scheduler.add(w);
w.join();
```

For each wrapper class, `connectors` as well as the arguments of the corresponding task call are passed to the constructor. The task calls themselves are mapped to calls of `Scheduler.add(Task)`. After each parallel statement, the compiler injects a join call; this method is part of XJava's runtime library and ensure that statements following the parallel statement will only execute after all tasks are finished. For a pipe statement, only the last task of the statement needs to be joined as it also finishes last. For a concurrent statement, there is no time order, so each task of that statement needs to be joined.

4 Scheduling Mechanism

Section 3 described how task declarations, task calls and parallel statements are compiled. Figure 1 shows how tasks are passed to and executed by the scheduler. The scheduler was designed to avoid unbounded growth of thread numbers and provide efficient execution. Both periodic and non-periodic tasks are compiled to wrapper classes; task calls are mapped to new instances of these classes and passed on to the scheduler.

When a **periodic task** *PT* is called, its instance is added to the task queue. The scheduler employs a fixed number of executor threads that take task instances from the queue, execute some iterations of their work method, and enqueue the task instance back. By default, the number of executor threads is equal to the number of CPU cores that are available. The number of work iterations is fixed and a default value is globally defined. In the future, we plan to dynamically adjust this value based on the execution behavior and work load. Tasks that are currently waiting for input elements are moved to the end of the queue. This mechanism prioritizes tasks for execution depending on the amount of work to be done.

Non-periodic tasks are also passed on to the scheduler, but they are not suitable for partial execution by executor threads. Instead, the scheduler decides based on the number of currently active threads and available memory, whether

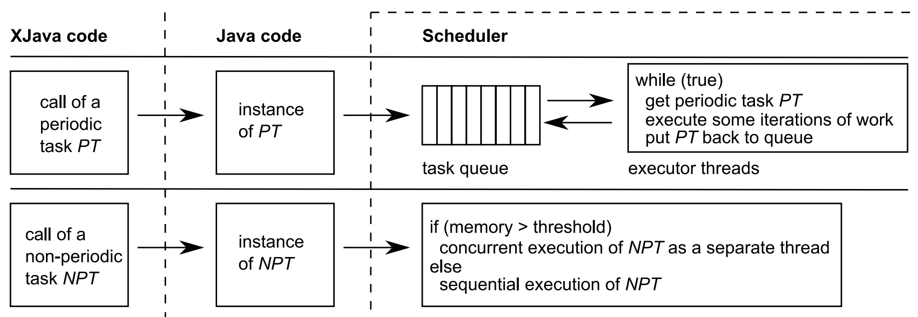

Fig. 1. Periodic and non-periodic tasks are compiled and passed on to XJava's scheduler. The scheduler provides executor threads for periodic tasks. Non-periodic tasks are executed either concurrently in separate threads or sequentially.

to execute a particular task sequentially by directly calling its run method, or to execute it in a separate thread.

5 Implementing Parallel Design Patterns

Parallel design patterns as described by Mattson et al. [7] provide useful principles for developing parallel software. However, the gap between patterns and program code has to be bridged by the programmer. XJava's language constructs help bridge that gap; the input-output-mechanism of tasks and their composability allow for expressing several types of parallelism such as pipelines, divide and conquer parallelism, or master/worker. As pipelines were already illustrated earlier, we now provide examples for divide and conquer and master/worker parallelism.

5.1 Divide and Conquer Algorithms

Divide and conquer algorithms are easy to express in XJava, as the parallel code is similar to the sequential version. For example, the following code sketches a sequential merge sort algorithm:

```
void mergesort(int from, int to) {
  if (to - from > threshold) {
    int x = from + (to - from)/2;
    mergesort(from, x);
    mergesort(x + 1, to);
    merge(from, x + 1, to);
  } else sort(from, to);
}
```

Parallelizing this algorithm in XJava is simple. (1) We need to make the method `mergesort` a `void => void` task. (2) Instead of sequentially calling `mergesort` two times in the recursion step, we combine these calls to a concurrent statement using the "`|||`" operator:

```
void => void mergesort(int from, int to) {         // (1)
  if (to - from > threshold) {
    int x = from + (to - from)/2;
    mergesort(from, x) ||| mergesort(x + 1, to);   // (2)
    merge(from, x + 1, to);
  } else sort(from, to);
}
```

The programmer does not need to think about synchronization or the number of running threads. This example shows how efficiently divide-and-conquer-based parallelism can be expressed in XJava. More details and performance results of this algorithm are given in Section 6.

5.2 Master/Worker Configurations

The master/worker pattern is a powerful concept that is used in many parallel applications. In XJava, it can be created by parallel statements that use variants of the "`=>`" operator. For example, consider a file indexing algorithm. The task declaration

```
File => void index() { /* ... */ }
```

specifies a worker task expecting a stream of `File` objects in order to index them. The non-periodic task

```
File => void indexers(int i) {
  index():i;
}
```

encapsulates a set of `i` workers that run concurrently to make indexing parallel. The task

```
void => File visitFiles(File root) { /* ... */ }
```

can be interpreted as a master task that recursively visits all files in the directory specified by the `root` argument. For a `File` object `dir` that we want to index, we can easily create a master/worker configuration employing a certain number, say `n`, workers:

```
visitFiles(dir) => indexers(n);
```

By choosing the "`=>`" operator, the workers are fed in a round-robin style with elements from the stream. Alternatively, the "`=>?`" operator distributes elements on a first-come-first-serve basis; the "`=>*`" operator would broadcast each element to all workers. In the file indexing context, the "`=>?`" apparently makes most sense: the order in which files are indexed is not important. In addition, the worker's waiting times are reduced, which results in better performance.

6 Experimental Results

We evaluated the sequential (i.e., single-threaded), multi-threaded, and XJava versions of four benchmark programs to compare code structure and performance. The threaded versions were written before the XJava versions to make sure that programs do not just re-implement the XJava concepts. We chose the programs to cover data, task, and pipeline parallelism. All programs were tested on three different machines: (1) an Intel Quadcore Q6600 with 2.40 GHz, 8 GB RAM and Windows XP Professional; (2) two Intel Xeon Quadcores E5320 with 1.86 GHz (Dual-Quadcore), 8 GB RAM and Ubuntu Linux 7.10; (3) a Sun Niagara 2 with 8 cores at 1.2 GHz capable of executing 4 threads, 16 GB RAM, and Solaris 10.

6.1 The Benchmark

JDesktopSearch (JDS) [5] is an open source desktop search application written entirely in Java. We use it as an example of a realistic, major application. It consists of about 3,400 lines of code (LOC) plus several additional libraries. The file indexing part is already multithreaded, but it can also be executed with just one thread. We re-engineered this part from Java to XJava; the relevant code had 394 LOC. We split the indexing methods into tasks to implement a master/worker approach as described in Section 5.2. Conceptually, the master task recursively walks through the root directory that has to be indexed, pushing the contained indexable files to the workers. We tested JDS for a 242 MB sample directory containing different file types and file sizes with a total of 12,223 files in 3,567 folders.

Additionally, we considered a selection of three smaller programs that are standard examples for parallelization. *Mandelbrot* computes mandelbrot sets based on a given resolution and a maximum number of iterations. *Matrix* multiplies matrices of type `double` that are randomly generated. *MergeSort* sorts an array of 3 million randomly generated integer values. It is a representative of divide and conquer algorithms; the XJava version implements the code already sketched in Section 5.1.

6.2 Results

Code. Table 2 shows code metrics for the sequential, threaded and XJava versions of the benchmark programs. In addition, the table presents total and relative improvements achieved by XJava over threaded Java. Outstanding benefits of XJava over threaded Java are that XJava saves code and significantly reduces the need for manual synchronization. For MergeSort, the sizes of the sequential and XJava programs are the same; and compared to the threaded MergeSort, XJava saved 39% of code. In XJava, fewer classes are required; the number of attributes is lower since attributes for synchronization or status notifications are no longer needed. We counted the occurences of `synchronized`, `wait`, `join`, `notify`, `notifyAll`, `sleep`, and `java.util.concurrent`. For the desktop search application, the number of synchronization constructs was reduced from 8 to 1; the other XJava benchmark programs do not require any manual synchronization at all. This effect comes from XJava's high abstraction level and its implicit synchronization mechanisms. Accordingly, the number of try-catch blocks is reduced for all programs since exceptions caused by concurrent modifications or interrupted threads do not need to be considered anymore. Our XJava programs used a few more methods (including tasks) than threaded Java, which lead to marginally lower average nested block depths.

Performance. We measured the performance of all benchmark programs' sequential, threaded and XJava versions; results are shown in Figure 3. Overall, XJava can indeed compete with threaded Java as execution times are on the same level. Only for divide and conquer parallelism as in the MergeSort program,

		JDS	%	Mandelbrot	%	Matrix	%	Mergesort	%
LOC	seq			86		51		72	
	par	394		116		90		118	
	xjava	362		99		61		72	
	improvement	32	8,1%	17	14,7%	29	32,2%	46	39,0%
classes	seq			1		1		1	
	par	2		2		3		2	
	xjava	1		2		1		1	
	improvement	1	50,0%	0	0,0%	2	66,7%	1	50,0%
attributes	seq			6		4		2	
	par	11		13		7		4	
	xjava	8		13		4		2	
	improvement	3	27,3%	0	0,0%	3	42,9%	2	50,0%
methods	seq			5		2		4	
	par	28		6		3		6	
	xjava (methods)	26		4		1		3	
	xjava (tasks)	3		3		3		1	
	improvement	-1	-3,6%	-1	-16,7%	-1	-33,3%	2	33,3%
synchronization*	seq			0		0		0	
	par	8		1		1		1	
	xjava	1		0		0		0	
	improvement	7	87,5%	1	100,0%	1	100,0%	1	100,0%
exceptions (try-catch, throw)	seq			1		0		0	
	par	34		3		3		1	
	xjava	32		1		0		0	
	improvement	2	5,9%	2	66,7%	3	100,0%	1	100,0%
nested block depth (average)	seq			2,33		2,75		2,20	
	par	1,85		2,43		3,00		2,38	
	xjava	1,84		2,00		2,17		2,20	
	improvement	0,01	0,5%	0,43	17,7%	0,83	27,8%	0,18	7,4%
nested block depth (standard deviation)	seq			0,94		1,09		0,75	
	par	1,48		0,90		1,27		0,99	
	xjava	1,47		1,12		0,90		0,75	
	improvement	0,00	0,3%	-0,21	-23,7%	0,37	29,0%	0,24	24,6%
nested block depth (maximum)	seq			3,00		4,00		3,00	
	par	8,00		3,00		5,00		4,00	
	xjava	8,00		4,00		3,00		3,00	
	improvement	0,00	0,0%	-1,00	-33,3%	2,00	40,0%	1,00	25,0%

* occurrences of Strings "synchronized", "wait", "join", "notify"/"notifyAll", "sleep", "java.util.concurrent"

Fig. 2. Code metrics of the benchmark programs. In terms of code sizes and used synchronization constructs, XJava shows significant improvements over threaded Java.

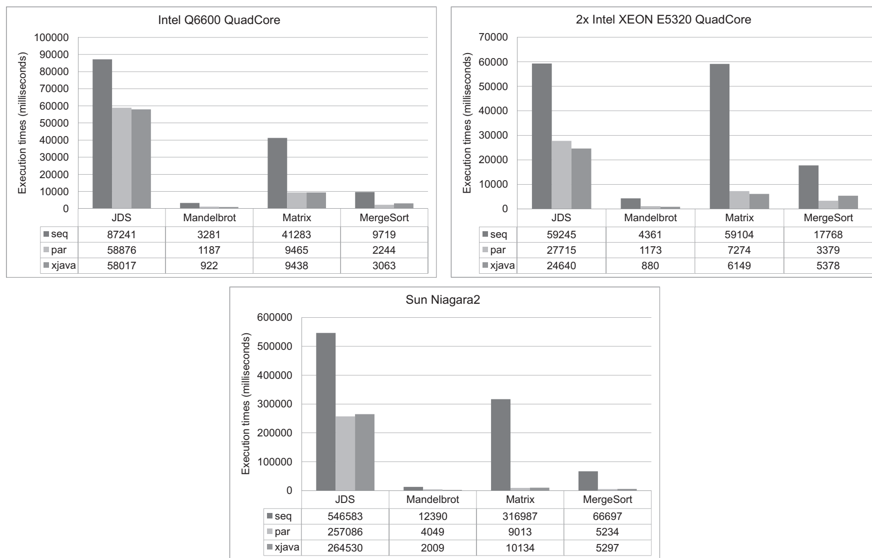

Fig. 3. Performance results of the sequential (*seq*), threaded (*par*) and XJava (*xjava*) versions of the benchmark programs

XJava tends to be a bit slower on all machines; on the Dual-Quadcore machine, the threaded version is even 1.6 times faster than the XJava version. The reason is most likely some overhead in the scheduler when executing non-periodic tasks. The Matrix program achieved an almost linear speedup on all machines, both for threaded Java and XJava. The speedups of the desktop search application were similar for Java and XJava; the maximum of only 2.1 can be explained with the memory bandwidth and bus bottlenecks.

Summary. Compared to threaded Java, the benchmark programs show that XJava simplifies parallel programming by reducing the amount of code, especially the need for manual synchronization and exception handling. We achieved speedups that can compete with the performance of threaded Java, although there is still potential for optimizing the scheduling mechanism.

7 Related Work

XJava is inspired by stream-oriented languages [14]. Stream programs consist of filters that are connected to form a stream graph. An input data stream of arbitrary length flows through that graph and is processed by the filters. Stream languages such as StreamIt have been demonstrated to efficiently exploit data, task, and pipeline parallelism for applications from the signal processing and graphics domain [15,4]. These languages were not designed for programming general-purpose applications; they use rather simple data structures and do not support object-orientation.

Chapel [2], Fortress [12] and X10 [3] are object-oriented languages for parallel programming. However, all of them are designed for explicit multithreading and require considerable manual synchronization. Their programming models focus on aspects of data and task parallelism; streams, master/slave or pipeline parallelism are not addressed explicitly.

As an alternative to new languages or language extensions, several libraries were designed to simplify parallel programming. Those libraries usually provide thread pools, data structures such as futures, locking mechanisms and synchronization constructs such as barriers. Intel's Threading Building Blocks [13] and Boost [1] are C++ libraries; the `java.util.concurrent` package [6] offers classes and interfaces for concurrent programming in Java. In contrast to native language constructs, the semantic information of library constructs is less powerful to enable more advanced optimizations or debugging – both essential in parallel programming.

8 Conclusion

XJava extends Java by providing tasks as native language constructs, which can be combined to parallel statements. In addition, XJava simplifies the implementation of parallel programming patterns and moves a significant part of low-level synchronization behind the scenes. Focusing in this paper on XJava's compiler

and scheduling mechanism, we benchmarked four XJava programs on three different multicore machines. Each XJava program was compared to equivalent threaded and sequential versions in terms of performance and code structure. XJava's approach is indeed applicable on programs of different complexity. We achieved speedups between 1.5 and 31.5 over sequential Java and, compared to threaded Java, code savings up to 39%.

Future work will include more case studies and experiments with applications from different domains. Also, more research is needed to evaluate different scheduling strategies in the context of XJava. A special focus will be on real-time tuning of scheduling parameters.

Acknowledgments. We thank the University of Karlsruhe and the Excellence Initiative for their support.

References

1. Boost C++ Libraries, `http://www.boost.org/`
2. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel Programmability and the Chapel Language. Int. J. High Perform. Comput. Appl. 21(3) (August 2007)
3. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Proc. OOPSLA 2005. ACM Press, New York (2005)
4. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proc. ASPLOS-XII. ACM Press, New York (2006)
5. JDesktopSearch, `http://sourceforge.net/projects/jdesktopsearch`
6. Lea, D.: The java.util.concurrent synchronizer framework. Sci. Comput. Program. 58(3) (2005)
7. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for parallel programming. Addison-Wesley, Boston (2005)
8. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
9. Otto, F., Pankratius, V., Tichy, W.F.: High-level Multicore Programming with XJava. In: ICSE 2009, New Ideas And Emerging Results. ACM Press, New York (2009)
10. Pankratius, V., Jannesari, A., Tichy, W.F.: Parallelizing BZip2. A Case Study in Multicore Software Engineering. Accepted for IEEE Software (September 2008)
11. Pankratius, V., Schaefer, C., Jannesari, A., Tichy, W.F.: Software engineering for multicore systems: an experience report. In: Proc. IWMSE 2008. ACM Press, New York (2008)
12. Project Fortress, `http://projectfortress.sun.com/`
13. Reinders, J.: Intel Threading Building Blocks. O'Reilly Media, Inc., Sebastopol (2007)
14. Stephens, R.: A Survey of Stream Processing. Acta Informatica 34(7) (1997)
15. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, p. 179. Springer, Heidelberg (2002)