A Language-Based Tuning Mechanism for Task and Pipeline Parallelism

Frank Otto, Christoph A. Schaefer, Matthias Dempe, and Walter F. Tichy

Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
{otto,cschaefer,dempe,tichy}@ipd.uka.de

Abstract. Current multicore computers differ in many hardware aspects. Tuning parallel applications is indispensable to achieve best performance on a particular hardware platform. Auto-tuners represent a promising approach to systematically optimize a program's tuning parameters, such as the number of threads, the size of data partitions, or the number of pipeline stages. However, auto-tuners require several tuning runs to find optimal values for all parameters. In addition, a program optimized for execution on one machine usually has to be re-tuned on other machines.

Our approach tackles this problem by introducing a language-based tuning mechanism. The key idea is the inference of essential tuning parameters from high-level parallel language constructs. Instead of identifying and adjusting tuning parameters manually, we exploit the compiler's context knowledge about the program's parallel structure to configure the tuning parameters at runtime. Consequently, our approach significantly reduces the need for platform-specific tuning runs.

We implemented the approach as an integral part of XJava, a Java language extension to express task and pipeline parallelism. Several benchmark programs executed on different hardware platforms demonstrate the effectiveness of our approach. On average, our mechanism sets over 90% of the relevant tuning parameters automatically and achieves 93% of the optimal performance.

1 Introduction

In the multicore era, performance gains for applications of all kind will come from parallelism. The prevalent thread model forces programmers to think on low abstraction levels. As a consequence, writing multithreaded code that offers satisfying performance is not straight-forward. New programming models have been proposed for simplifying parallel programming and improving portability. Interestingly, the high-level constructs can be used for automatic performance tuning. Libraries, in contrast, do not normally provide semantic information about parallel programming patterns.

Case studies have shown that parallel applications typically employ different types of parallelism on different levels of granularity [13]. Performance depends on various parameters such as the number of threads, the number of pipeline

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 328–340, 2010.

stages, or load balancing strategies. Usually, these parameters have to be defined and set explicitly by the programmer. Finding a good parameter configuration parameters is far from easy due to large parameter search spaces. Auto-tuners provide a systematic way to find an optimal parameter configuration. However, as the best configuration strongly depends on the target platform, a program normally has to be re-tuned after porting to another machine.

In this paper, we introduce a mechanism to automatically infer and configure five essential tuning parameters from high-level parallel language constructs. Our approach exploits explicit information about task and pipeline parallelism and uses tuning heuristics to set appropriate parameter values at runtime. From the programmer's perspective, a considerable number of tuning parameters becomes invisible. That is, the need for feedback-directed auto-tuning processes on different target platforms is drastically reduced.

We implemented our approach as part of the previously introduced language *XJava* [11,12]. XJava extends Java with language constructs for high-level parallel programming and allows the direct expression of task and pipeline parallelism. An XJava program compiles to Java code instrumented with tuning parameters and context information about its parallel structure. The XJava runtime system exploits the context information and platform properties to set tuning parameters.

We evaluated our approach for a set of seven benchmark programs. Our approach sets over 90% of the relevant tuning parameters automatically, achieving 93% of the optimum performance on three different platforms.

2 The XJava Language

XJava extends Java by adding tasks and parallel statements. For a quick overview, the simplified grammar extension in BNF style is shown in Figure 1. We basically extend the existing production rules for method declarations (rule 1) and statements (rule 7). New keywords are `work` and `push`, new operators are `=>` and `|||`. Semantics are described next.

2.1 Language

Tasks. *Tasks* are conceptually related to filters in stream languages. Basically, a task is an extension of a method. Unlike methods, a task defines a concurrently executable activity that expects a stream of input data and produces a stream of output data. The types of data elements within the input and output stream are defined by the task's input and output type. These types can also be `void` in order to specify that there is no input or output. For example, the code

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

declares a public task `encode` with input and output type `String`. The `work` block defines what to do for each incoming element and can be thought of as a

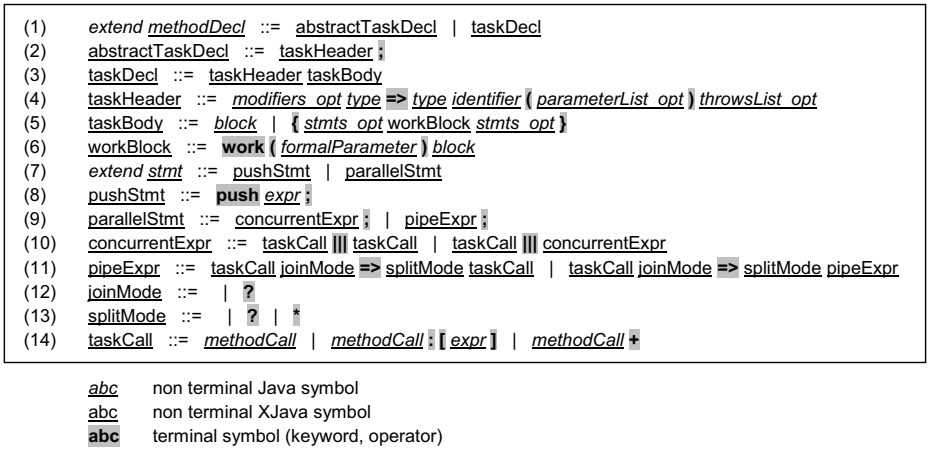

Fig. 1. The grammar extension of XJava

loop. A task body contains either exactly one or no work block (rule 6). A `push` statement inside a task body puts an element into the output stream. In the example, these elements are `String` objects encrypted by the method `encrypt` and the parameter `key`.

Parallel statements. Tasks are called like methods; parallelism is generated by combining task calls with operators to compose *parallel statements* (rule 9). Basically, these statements can be used both outside and inside a task body; the latter case introduces nested parallelism. Parallel statements allow for easily expressing many different types of parallelism, such as linear and non-linear pipelines, master/worker configurations, data parallelism, and recursive parallelism.

(1) Combining tasks with the "`=>`" operator introduces pipeline parallelism. In addition to the task `encrypt` above, we assume two more tasks `read` and `write` for reading and writing to a file. Then, the *pipeline statement*

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

creates a pipeline that encodes the content of the file `fin` and writes results to the file `fout`.
(2) Combining tasks with the "`|||`" operator introduces task parallelism. Assuming a task `compress`, the *concurrent statement*

```
compress(f1) ||| compress(f2);
```

compresses two files `f1` and `f2` concurrently.

By default, a task is executed by one thread. Optionally, a task call can be marked with a "`+`" operator to make it replicable. A replicable task can be executed by more than one thread, which is useful to reduce bottleneck effects in pipelines.

For example, the task `encode` in the pipeline example above might be the slowest stage. Using the expression `encode(key)+` instead of `encode(key)` can increase throughput since we allow more threads to execute that critical stage. The number of replicates is determined at runtime and thus does not need to be specified by the programmer. If the programmer wants to create a concrete number of task instances at once, say 4, he can use the expression `encode(key):[4]`.

2.2 Compiler and Runtime System

The XJava compiler transforms XJava to optimized and instrumented Java code, which is then translated into bytecode. The translated program consists of logical code units that are passed to the XJava runtime system *XJavaRT*. XJavaRT is the place where parallelism happens. It is designed as a library employing executor threads and built-in scheduling mechanisms.

3 Tuning Challenges

A common reason for poor performance of parallel applications is poor adaption of parallel code to the underlying hardware platform. With the parallelization of an application, a large number of performance-relevant tuning parameters arise, e.g. how many threads are used for a particular calculation, how to set the size of data partitions, how many stages a pipeline requires, or how to accomplish load balancing for worker threads.

Manual tuning is tedious, costly, and due to the large number of possible parameter configurations often hopeless. To automate the optimization process, search-based automatic performance tuning (auto-tuning) [23,1,20,22] is a promising approach. Auto-tuning represents a feedback-directed process consisting of several steps: choice of parameter configuration, program execution, performance monitoring, and generation of a new configuration based on search algorithms such as hill climbing or simulated annealing. Experiments with real-world parallel applications have shown that using appropriate tuning techniques, a significant performance gain can be achieved on top of "plausible" configurations chosen by the programmer [13,18].

However, as the diversity of application areas for parallelism has grown and the available parallel platforms differ in many respects (e.g. in number or type of cores, cache architecture, available memory, or operating system), the number of targets to optimize for is large. Optimizations made for a certain machine may cause a slowdown on another machine. Thus, a program optimized for a particular hardware platform usually has to be re-tuned on other platforms.

For illustration, let's think of a parallel program with only one tuning parameter t that adjusts the number of concurrent threads. While the best configuration for t on a 4-core-machine is probably a value close to 4, this configuration might be suboptimal for a machine with 16 cores. From the auto-tuner's perspective, t represents a set of values to choose from. If the tuner knew the purpose of t, it would be able to configure t directly in relation to the number of cores providing significantly improved performance.

To tackle the problem of optimization portability, recent approaches propose the use of *tuning heuristics* to exploit information about purpose and impact of tuning parameters [17]. This context information helps configuring parameters implicitly without enumerating and testing their entire value range.

4 Language-Based Tuning Mechanism

We propose an approach that exploits tuning-relevant context information from XJava's high-level parallel language constructs (cf. Section 2). Relevant tuning parameters are automatically inferred and implicitly set by the runtime system (XJavaRT). Therefore, porting an XJava application to another machine requires less re-tuning, in several cases no re-tuning at all. Figure 2 illustrates the concept of our approach (b) in contrast to feedback-directed auto-tuning (a).

Our work focuses on task and pipeline parallelism; both forms of parallelism are widely used. Task parallelism refers to tasks whose computations are independent from each other. Pipeline parallelism refers to tasks with input-output dependencies, i.e. the output of one task serves as the input of the next task.

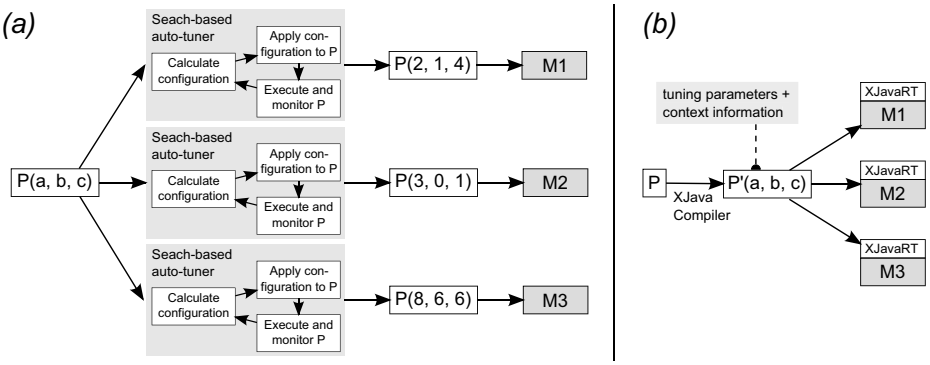

Fig. 2. Adapting a parallel program P to different target platforms $M1, M2, M3$. (a) A search-based auto-tuner requires the explicit declaration of tuning parameters a, b, c. The auto-tuner needs to perform several feedback-directed tuning runs on each platform to find the best configuration. (b) In our approach, we use compiler knowledge to automatically infer relevant tuning parameters and context information about the program's parallel structure. The parameters are set by the runtime system XJavaRT, which uses tuning heuristics that depend on the characteristics of the target platform.

First, we describe essential types of tuning parameters for these forms of parallelism (Section 4.1). Then, we show how the XJava compiler infers tuning parameters and context information from code (Sections 4.2 and 4.3). Finally, we describe heuristics to set the tuning parameters (Section 4.4).

4.1 Tuning Parameters

Tuning parameters represent program variables that may influence performance. In our work, we distinguish between *explicit* and *implicit* tuning parameters. The first have to be specified and configured by the programmer, the latter are invisible to the programmer and set automatically. In the following we describe essential types of tuning parameters for task and pipeline parallelism [13,17].

Thread count (TC). The total number of threads executing an application strongly influences its performance. To underestimate the number will limit speedup, to overestimate the number might slow down the program due to synchronization overhead and memory consumption.

Load balancing strategy (LB). The load balancing strategy determines how to distribute workload to execution threads or CPU cores. Load balancing can be done statically, e.g. in a round-robin style, or dynamically, e.g. in a first-come-first-serve fashion or combined with work stealing.

Cut-off depth (CO). Parallel applications typically employ parallelism on different levels. Low-level parallelism can have a negative impact on the performance, if the synchronization and memory costs are higher than the additional speedup of concurrent execution. In other words, there is a level CO where parallelism is not worthwhile and a serial execution of the code is preferable.

Stage replicates (SR). The throughput and speedup achieved by a pipeline is limited by its slowest stage. If this stage is stateless, it can be replicated in order to be executed by more than one thread. The parameter SR denotes the number of replicates.

Stage fusion (SF). From the programmer's perspective, the conceptual layout of a pipeline usually consist of n stages $s_1, ..., s_n$. However, mapping each stage s_i to one thread may not be the best configuration. Instead, fusing some stages could reduce bottleneck effects. Stage fusion represents functional composition of stages and is similar to the concept of filter fusion [14].

Data size (DS). Parallel programs often process a large amount of data that needs to be decomposed into smaller partitions. The data partition size typcially affects the program's performance.

The applications considered here expose up to 14 parameters that need to be tuned (cf. Section 5). Note that one application can contain several parameters of the same type.

The following sections show how our approach automatically infers and sets these parameters, except DS. As the most appropriate size of data partitions depends on the type of application, we leave this issue to the programmer or further tuning. The XJava programmer must define separate tasks for decomposing and merging data.

4.2 Inferring Tuning Parameters from XJava Code

The XJava compiler generates Java code and adds tuning parameters. Task parallel statements are instrumented with the parameter *cut-off depth* (CO).

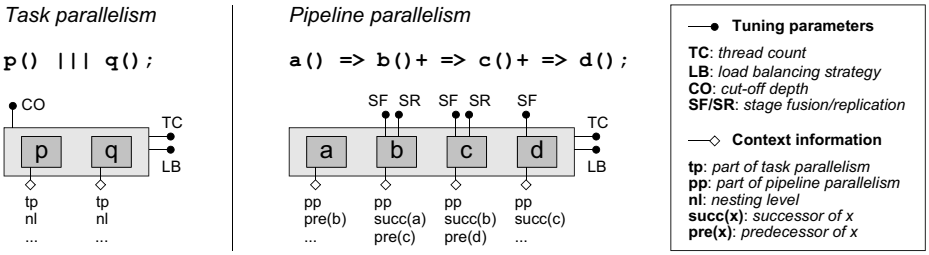

Fig. 3. Inferring tuning parameters and context information from XJava code

When compiling a pipeline statement consisting of n stages $s_1, ..., s_n$, the stages $s_2, ...s_n$ are instrumented with the boolean tuning parameter *stage fusion* (SF), indicating whether that stage should be fused with the previous one. In addition, the parameter *stage replicates* (SR) is added to each stage declared as replicable.

Figure 3 illustrates the parameter inference for a task parallel statement and a pipeline. A task parallel statement `p() ||| q()` is instrumented with the parameter CO. Depending on its value, that statement executes either concurrently or sequentially, if the cut-off depth is reached. A pipeline `a() => b()+ => c()+ => d()` compiles to a set of four task instances a, b, c and d. Since b and c are replicable, a tuning parameter SR is added to them. In addition, b, c and d get a boolean parameter SF defining whether to fuse that stage with the previous one. The parameters TC and LB for the overall number of threads and the load balancing strategy affect both task parallel statement and pipelines.

In Section 4.4, we describe the heuristics used to set the parameters.

4.3 Inferring Context Information

Beside inferring tuning parameters, the XJava compiler exploits context information about the program's parallel structure. The compiler makes this knowledge available at runtime to set tuning parameters appropriately.

The context information of a task call includes several aspects: (1) purpose of the task (pipeline stage or a part of task-parallel section), (2) input and output dependences, (3) periodic or non-periodic task, (4) level of parallelism, and (5) current workload of the task. Aspects 1-3 can be inferred at compile time, aspects 4 and 5 at runtime. However, XJavaRT has access to all information. Figure 3 sketches potential context information for tasks.

4.4 Tuning Heuristics

Thread count (TC). XJavaRT provides a global thread pool to control the total number of threads and to monitor the numbers of running and idle threads at any time. XJavaRT knows the number n of a machine's CPU cores and

therefore uses the heuristic $TC = \lceil n \cdot \alpha \rceil$ for some $\alpha \geq 1$. We use $\alpha = 1.5$ as a predefined value.

Load balancing (LB). XJavaRT employs different load balancing strategies depending on the corresponding context information. For recursive task parallelism, such as divide and conquer algorithms, XJavaRT applies a work stealing mechanism based on the Java fork/join framework [8]. For pipelines, XJavaRT prefers stages with higher workloads to execute, thus implementing a dynamic load balancing strategy.

Cut-off depth (CO). XJavaRT dynamically determines the cut-off depth for task parallel expressions to decide whether to execute a task parallel statement concurrently or in sequential order. Since XJavaRT keeps track of the number of idle executor threads, it applies the heuristic $CO = \infty$ if idle threads exist, and $CO = l$ otherwise (where l is the nested level of the task parallel expression). In other words, tasks are executed sequentially if there are no executor threads left.

Stage replicates (SR). When a replicable task is called, XJavaRT creates $SR = i$ replicates of the task, where i denotes the number of idle executor threads. If there are no idle threads, i.e. all CPU cores are busy, no replicates will be created. XJavaRT uses a priority queue putting tasks with lower work load (i.e. few data items waiting at their input port) at the end. This mechanism does not always achieve optimal results, but seems effective in practice, as our results show.

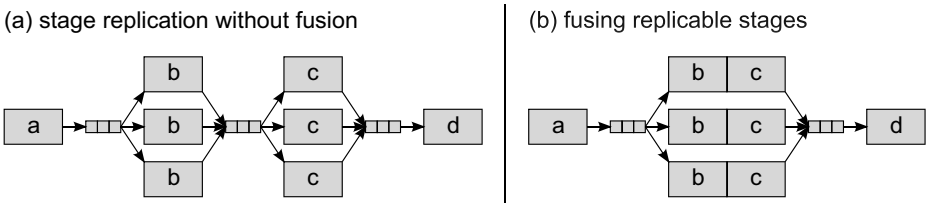

Fig. 4. Stage fusion for a pipeline `a() => b()+ => c()+ => d()`. (a) Stage replication without fusion introduces overhead for splitting and joining data items. (b) Stage fusion prior to replication removes some of this overhead.

Stage fusion (SF). In a pipeline consisting of several stages, combining two or more stages into a single stage can increase performance, as the overhead for split-join operations is reduced. Therefore, XJava fuses consecutive replicable tasks within a pipeline expression to create a single replicable task. Figure 4 illustrates this mechanism for a pipeline `a() => b()+ => c()+ => d()`.

5 Experimental Results

We evaluate our approach using a set of seven benchmarks that cover a wide range of parallel applications, including algorithmic problems such as sorting

or matrix multiplication, as well as the real-world applications for raytracing, video processing and cryptography. The applications use task, data or pipeline parallelism. We measure two metrics:

Implicit tuning parameters. We count the number of automatically handled tuning parameters as a metric for simplification of the optimization process. If more tuning parameters are automated, fewer optimizations have to be performed manually.

Performance. For each application, we compared a sequential version to an XJava version and measured the speedups *heur* and *best*:

- *heur:* Speedups of the XJava programs using our heuristic-based approach. These programs did not require any manual adjustments.
- *best:* Speedups achieved for the best parameter configuration found by an auto-tuner performing an exhaustive search.

The speedups over the sequential versions were measured on three different parallel platforms: (1) an Intel Quadcore Q6600 with 2.40 GHz, 4 GB RAM and Windows 7 Professional 64 Bit; (2) a Dual Intel Xeon Quadcore E5320 1.86 GHz, 8 GB RAM and Ubuntu Linux 7.10; (3) a Sun Niagara T2 with 8 cores (each capable of 8 threads), 1.2 GHz, 16 GB RAM and Solaris 10.

5.1 Benchmarked Applications

MSort and *QSort* implement the recursive mergesort and quicksort algorithms to sort a randomly generated array with approximately 33.5 million integer values.

Matrix multiplies two matrices based on a master-worker configuration, where the master divides the final matrix into areas and assigns them to workers. *MBrot* computes the mandelbrot set for a given resolution and a maximum number of 1000 iterations. *LRay* is a lightweight raytracer entirely written in Java. *MBrot* and *LRay* both use the master-worker pattern by letting the master divide the image into multiple blocks, which are then computed concurrently by workers.

The applications *Video* and *Crypto* use pipeline parallelism. *Video* is used to combine multiple frames into a slideshow, while performing several filters such as scaling and sharpening on each of the video-frames. The resulting pipeline contains eight stages, five of which are data parallel and can be replicated. *Crypto* applies multiple encryption algorithms from the `javax.crypto` package to a 60 MB text file that is split into 5 KB blocks. The pipeline has seven stages; each stage except those for input and output are replicable.

5.2 Results

Implicit tuning parameters. Depending on the parallelization strategy, the programs expose different tuning parameters. Figure 5 shows the numbers of explicit and implicit parameters for each application. Explicit parameters are declared and set in the program code. Implicit parameters do not appear in the code, they are automatically inferred by the compiler and set using our approach.

Benchmark		Tuning parameters			
	Parallelization	Explicit	Implicit (Automated)	Total	Saving
Qsort	divide & conquer	-	3 (TC, LB, CO)	3	100%
Msort	divide & conquer	-	3 (TC, LB, CO)	3	100%
Matrix	master/worker	1 (DS)	2 (LB, TC)	3	67%
Lray	master/worker	1 (DS)	2 (LB, TC)	3	67%
Mbrot	master/worker	1 (DS)	2 (LB, TC)	3	67%
Crypto	pipeline	1 (DS)	13 (TC, LB, 5*SR, 6*SF)	14	93%
Video	pipeline	-	14 (TC, LB, 5*SR, 7*SF)	14	100%
[average]		0.57	5.57	6.14	91%

TC: thread count LB: load balancing strategy CO: cut-off depth
SR: stage replication SF: stage fusion DS: data size

Fig. 5. Explicit and implicit tuning parameters for the benchmarked applications. On average, our approach infers and sets 91% of the parameters automatically.

On average, the number of explicit tuning parameters is reduced by 91%, ranging from 67% to 100%.

Our mechanism automatically infers and sets all parameters except the data size (DS). For *Matrix*, these are the sizes of the parts of the matrix to be computed by a worker; for *MBrot* and *LRay*, these are the sizes of the image blocks calculated concurrently. In *Crypto* the granularity is determined by the size of the data blocks that are sent through the pipeline. As *Video* decomposes the video data frame by frame, there is no need for an explicit tuning parameter to control the data size.

	Qsort	Msort	Matrix	Lray	Mbrot	Crypto	Video
Q6600	6192	10001	14241	14218	5800	21074	8412
2x E5320	7516	11955	9767	11896	6609	25681	10190
Niagara T2	32437	56476	72461	57658	29083	23505	42940

Fig. 6. Execution times (milliseconds) of the sequential benchmark programs

Performance. Figure 6 lists the execution times of the sequential benchmark programs. Figure 7 shows the speedups for the corresponding XJava versions on the three parallel platforms. Using our approach, the XJava programs achieve an average speedup of about 3.5 on the Q6600 quadcore, 5.0 on the E5320 dual-quadcore, and 17.5 on the Niagara T2.

The automatic replication of XJava tasks achieves good utilization of the available cores in the master-worker and pipeline applications, although the round-robin distribution of items leads to a suboptimal load balancing in the replicated stages. The blocks in *Crypto* are of equal size, leading to an even workload. The frames in *Video* have different dimensions, resulting in slightly lower speedups.

To examine the quality of our heuristics, we used a script-based auto-tuner performing an exhaustive search to find the best parameter configuration. We

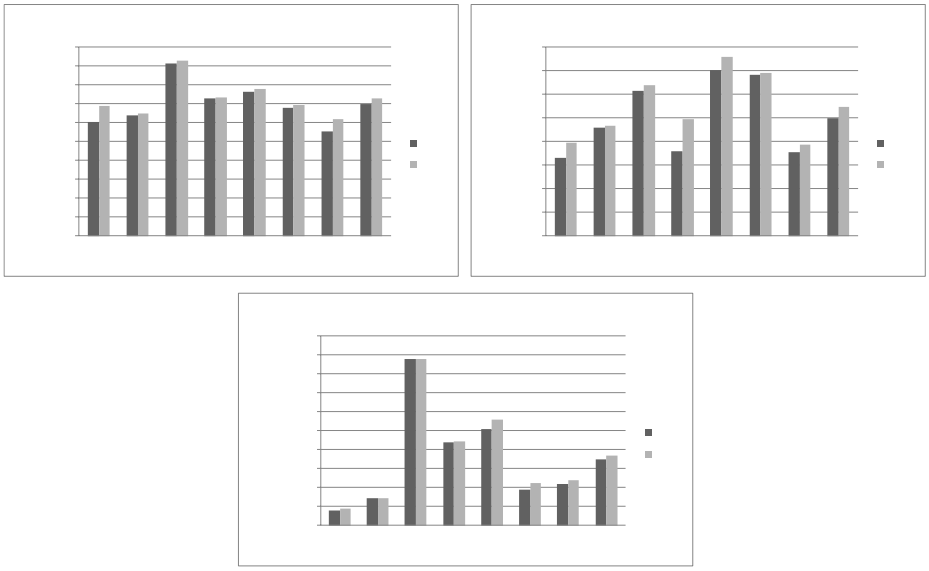

Fig. 7. Performance of our heuristic-based approach (*heur*) in comparison to the *best* configuration found by a search-based auto-tuner

observed the largest performance difference for *QSort* and for *LRay* on the E5320 machine. In general, *QSort* benefits from further increasing the cutoff threshold, as more tasks allow better load balancing with workstealing. For *LRay*, reducing the number of workers by one increases the speedup from 3.4 to 5 - we attribute this behavior to poor cache usage or other memory bottlenecks when using too many threads.

In all other cases, the search-based auto-tuner achieved only minor additional speedups compared to our language-based tuning mechanism. In total, the mean error rate of the *heuristic-based* configurations to the *best* configurations are 9% on the E5320 dual-quadcore, 7% on the Niagara T2, and 4% on the Q6600 quadcore. That is, our approach achieves 93% of the optimal performance.

6 Related Work

Auto-tuning has been investigated mainly in the area of numerical software and high-performance computing. Therefore, many approaches (such as *ATLAS* [23], *FFTW* [5], or *FIBER* [7]) focus on tuning particular types of algorithms rather than entire parallel applications. Datta et al. [4] address auto-tuning and optimization strategies for stencil computations on multicore architectures.

MATE [10] uses a model-based approach to dynamically optimize distributed master/worker applications. MATE predicts the performance of these programs. However, optimizing other types of parallel patterns requires the creation of new analytic models. MATE does not target multicore systems.

Atune [17,19] introduces tuning heuristics to improve search-based auto-tuning of parallel architectures. However, Atune needs a separate configuration language and an offline auto-tuner.

Stream languages such as StreamIt [21,6] provide explicit syntax for data, task and pipeline parallelism. Optimizations are done at compile time for a given machine; dynamic adjustments are typically not addressed.

Libraries such as java.util.concurrent [9] or TBB [16] provide constructs for high-level parallelism, but do not exploit context information and still require explicit tuning. Languages such as Chapel [2], Cilk [15] and X10 [3] focus on task and data parallelism but not on explicit pipelining and do not support tuning parameter inference.

7 Conclusion

Tuning parallel applications is essential to achieve best performance on a particular platform. In this paper, we presented a language-based tuning mechanism for basically any kind of application employing task and pipeline parallelism. Our approach automatically infers tuning parameters and corresponding context information from high-level parallel language constructs. Using appropriate heuristics, tuning parameters are set at runtime. We implemented our technique as part of the XJava compiler and runtime system.

We evaluated our approach for seven benchmark programs covering different types of parallelism. Our tuning mechanism infers and sets over 90% of the relevant tuning parameters automatically. The average performance achieves 93% of the actual optimum, drastically reducing the need for further tuning. If further search-based tuning is still required, our approach provides a good starting point.

Future work will address the support of further tuning parameters (such as data size), the refinement of tuning heuristics, and the integration of a feedback-driven online auto-tuner.

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report, University of California, Berkeley (2006)
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., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In: Proc. OOPSLA 2005. ACM, New York (2005)
4. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In: Proc. Supercomputing Conference (2008)

5. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT. In: Proc. ICASSP, vol. 3 (May 1998)
6. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In: Proc. ASPLOS-XII. ACM, New York (2006)
7. Katagiri, T., Kise, K., Honda, H., Yuba, T.: FIBER: A Generalized Framework for Auto-tuning Software. In: Proc. International Symposium on HPC (2003)
8. Lea, D.: A Java fork/join Framework. In: Proc. Java Grande 2000. ACM, New York (2000)
9. Lea, D.: The java.util.concurrent Synchronizer Framework. Sci. Comput. Program 58(3) (2005)
10. Morajko, A., Margalef, T., Luque, E.: Design and Implementation of a Dynamic Tuning Environment. Parallel and Distributed Computing 67(4) (2007)
11. Otto, F., Pankratius, V., Tichy, W.F.: High-level Multicore Programming With XJava. In: Comp. ICSE 2009, New Ideas And Emerging Results. ACM, New York (2009)
12. Otto, F., Pankratius, V., Tichy, W.F.: XJava: Exploiting Parallelism with Object-Oriented Stream Programming. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009 Parallel Processing. LNCS, vol. 5704, pp. 875–886. Springer, Heidelberg (2009)
13. Pankratius, V., Schaefer, C.A., Jannesari, A., Tichy, W.F.: Software Engineering for Multicore Systems: an Experience Report. In: Proc. IWMSE 2008. ACM, New York (2008)
14. Proebsting, T.A., Watterson, S.A.: Filter Fusion. In: Proc. Symposium on Principles of Programming Languages (1996)
15. Randall, K.: Cilk: Efficient Multithreaded Computing. PhD Thesis. Dep. EECS, MIT (1998)
16. Reinders, J.: Intel Threading Building Blocks. O'Reilly Media, Inc., Sebastopol (2007)
17. Schaefer, C.A.: Reducing Search Space of Auto-Tuners Using Parallel Patterns. In: Proc. IWMSE 2009. ACM, New York (2009)
18. Schaefer, C.A., Pankratius, V., Tichy, W.F.: Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009 Parallel Processing. LNCS, vol. 5704, pp. 9–20. Springer, Heidelberg (2009)
19. Schaefer, C.A., Pankratius, V., Tichy, W.F.: Engineering Parallel Applications with Tunable Architectures. In: Proc. ICSE. ACM, New York (2010)
20. Tapus, C., Chung, I., Hollingsworth, J.K.: Active Harmony: Towards Automated Performance Tuning. In: Proc. Supercomputing Conference (2002)
21. 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)
22. Werner-Kytola, O., Tichy, W.F.: Self-tuning Parallelism. In: Williams, R., Afsarmanesh, H., Bubak, M., Hertzberger, B. (eds.) HPCN-Europe 2000. LNCS, vol. 1823, p. 300. Springer, Heidelberg (2000)
23. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS Project. Journal of Parallel Computing 27 (2001)