Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications

Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy

University of Karlsruhe (TH), Am Fasanengarten 5, 76131 Karlsruhe, Germany
{cschaefer, pankratius, tichy}@ipd.uka.de

Abstract. Auto-tuners automate the performance tuning of parallel applications. Three major drawbacks of current approaches are 1) they mainly focus on numerical software; 2) they typically do not attempt to reduce the large search space before search algorithms are applied; 3) the means to provide an auto-tuner with additional information to improve tuning are limited.

Our paper tackles these problems in a novel way by focusing on the interaction between an auto-tuner and a parallel application. In particular, we introduce Atune-IL, an instrumentation language that uses new types of code annotations to mark tuning parameters, blocks, permutation regions, and measuring points. Atune-IL allows a more accurate extraction of meta-information to help an auto-tuner prune the search space *before* employing search algorithms. In addition, Atune-IL's concepts target parallel applications in general, not just numerical programs. Atune-IL has been successfully evaluated in several case studies with parallel applications differing in size, programming language, and application domain; one case study employed a large commercial application with nested parallelism. On average, Atune-IL reduced search spaces by 78%. In two corner cases, 99% of the search space could be pruned.

1 Introduction

As multicore platforms become ubiquitous, many software applications have to be parallelized and tuned for performance. Manual tuning must be automated to cope with the diversity of application areas for parallelism and the variety of available platforms [1].

Search-based automatic performance tuning (auto-tuning) [2–4] is a promising systematic approach for parallel applications. In a repetitive cycle, an auto-tuner executes a parameterized application, monitors it, and modifies parameter values to find a configuration that yields the best performance. It is currently a challenge to specify meta-information for auto-tuning in an efficient and portable way. As search spaces can be large, this additional information can be used an auto-tuner for pruning before applying any search algorithms.

In this paper, we introduce Atune-IL, a general-purpose instrumentation language for auto-tuning. We describe the features and highlight Atune-IL's approach to enrich a program with annotations that allow a more accurate search space reduction. In several case studies we show how Atune-IL works with different parallel applications and evaluate its effectiveness regarding search space reduction. Finally, we compare Atune-IL in the context of related work.

2 Requirements for a Tuning Instrumentation Language

A tuning instrumentation language for parallel applications should be able to instrument performance relevant variables such that their values can be set by an auto-tuner. It should also allow the demarcation of permutable program statements and provide the means to define program variants; for tuning, these variants represent alternatives of the same program that differ in performance-relevant aspects. Monitoring support is required to give run-time feedback to an auto-tuner, so that it can calculate and try out a new parameter configuration. It is desirable that the tuning language is as general as possible and not closely tied to a particular application domain. We further show that is also important to design the instrumentation language in a way that helps cut down the search space of configurations at an early stage.

2.1 Reducing Search Space before Tuning

The search space an auto-tuner explores is defined by the cross-product of the domains of all parameters within the program. As the search space grows exponentially, even an auto-tuner using a smart search algorithm may need a long time to find the best – or at least a sufficiently good – parameter configuration. For example, consider the search space consisting of 24 million parameter configurations of our first case study in Section 5. If a sophisticated search algorithm tests only 1% of the configurations, it will still perform 240,000 tuning iterations.

We therefore propose reducing the search space *before* any search algorithm is applied. To do so, our instrumentation language is able to capture the structure of an application along with characteristics important for tuning, such as parallel sections and dependencies between tuning parameters.

Analyzing Parallel Sections. Within application code, we define two sections to be independent of each other if they cannot be executed concurrently in any of the application's execution paths. Nested sections always depend on the enclosing section in that their execution and performance can be influenced by parameters of the enclosing section.

For illustration, consider a hypothetical program as in Fig. 1 with two parallel sections. The sections are independent and cannot run concurrently. Section P_1 has three tuning parameters, $t_1, ..., t_3$, while section P_2 contains two tuning parameters, t_4 and t_5. An auto-tuner would have to search in the worst case the cross product of all parameters $dom(t_1) \times ... \times dom(t_5)$. However, if the two sections are known to be independent, the search space can be reduced to the considerably smaller cross products of each section's parameters $dom(t_1) \times ... \times dom(t_3)$, and $dom(t_4) \times dom(t_5)$, respectively. The sections can be modularly tuned one

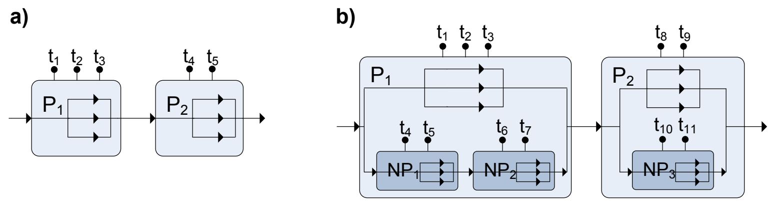

Fig. 1. a) Example of two independent sections with individual tuning parameters; b) example of two nested sections that come with additional parameters.

after another in separate tuning sessions. The parameters of the section that is currently not tuned are set to their default values.

Fig. 1 b) extends the previous example and shows P_1 and P_2, now with the nested sections NP_1, NP_2, and NP_3. To reduce search space in the case of nested structures, the following strategy seemed promising in our case studies. We perform several tuning sessions in which each lowest-level section is tuned together with all parent sections. In this example, this results in $dom(t_1) \times ... \times dom(t_5)$, $dom(t_1) \times dom(t_2) \times dom(t_3) \times dom(t_6) \times dom(t_7)$, and $dom(t_8) \times ... \times dom(t_{11})$. If a section is tuned in more than one tuning session (e.g., P_1), its optimal parameter values may differ depending on the nested section chosen before (NP_1 or NP_2). In this case, we tune the parameters of P_1 again and set the parameters of NP_1 and NP_2 to their best values so far. However, this is just an illustration that the information provided by our instrumentation language is useful for an auto-tuner; detailed algorithms are beyond the scope of this paper.

Considering Parameter Dependencies. A tuning parameter often depends on values of another parameter. That is, a particular parameter has to be tuned only if another parameter has some specific value. As an example, consider two parameters named *sortAlgo* and *depth*. The parameter *sortAlgo* selects either parallel merge sort or heap sort, while *depth* defines the depth of recursion within merge sort. Obviously, *depth* conditionally depends on *sortAlgo*. This information is important for an auto-tuner to avoid tuning unnecessary value combinations, as *depth* needs not be tuned for heap sort.

In summary, our instrumentation language makes the search space smaller so that even complex parallel programs can be auto-tuned.

3 Language Features of Atune-IL

This section introduces Atune-IL's features. We describe how to specify tuning parameters, statement permutation, blocks, and measuring points to meet the requirements introduced in the previous section. For a complete list of all language features refer to Table 1. Listing 1.1 shows a program instrumented with Atune-IL. The program basically searches strings in a text, stores them in an array and sorts it using parallel sorting algorithms. Finally, it counts the total characters the array contains.

Listing 1.1. C# program instrumented with Atune-IL

```
List<string> words = new List<string>(3);
void main() {
    #pragma atune startblock fillBlock
    #pragma atune gauge mySortExecTime

    string text = "Auto-tuning has nothing to do with tuning cars."

    #pragma atune startpermutation fillOrder
    #pragma atune nextelem
    words.Add(text.Find("cars"));
    #pragma atune nextelem
    words.Add(text.Find("do"));
    #pragma atune nextelem
    words.Add(text.Find("Auto-tuning"));
    #pragma atune endpermutation

    sortParallel(words);

    #pragma atune GAUGE mySortExecTime
    #pragma atune ENDBLOCK fillBlock

    countWords(words)
}

// Sorts string array
void sortParallel(List<string> words) {
    #pragma atune startblock sortBlock inside fillBlock

    IParallelSortingAlgorithm sortAlgo = new ParallelQuickSort();
    int depth = 1;
    #pragma atune setvar sortAlgo type generic
        values "new ParallelMergeSort(depth)","new ParallelQuickSort()"
        scale nominal
    #pragma atune setvar depth type int
        values 1-4 scale ordinal
        depends sortAlgo='new ParallelMergeSort(depth)'

    sortAlgo.Run(words);

    #pragma atune endblock sortBlock
}

// Counts total characters of string array
int countCharacters(List<string> words) {
    #pragma atune startblock countBlock
    #pragma atune gauge myCountExecTime
    int numThreads = 2;
    #pragma atune setvar numThreads type int
        values 2-8 scale ordinal

    int total = countParallel(words, numThreads);

    #pragma atune gauge myCountExecTime
    return total;
    #pragma atune endblock countBlock
}
```

We designed Atune-IL as a pragma-based language for shared-memory multicore platforms. Direct support for message parsing architectures (such as MPI) is not included. All Atune-IL statements are preceded by a language-dependent pragma directive (such as `#pragma` for C++ and C# or `/*@` for Java) followed by the `atune` prefix. This approach offers two benefits: 1) tuning information is separated from program code; 2) the instrumented program is executable even without auto-tuning, as pragmas and annotations are typically ignored by compilers.

Tuning Parameters. The `SETVAR` statement is used to mark a variable in the host language as tunable. Technically, it redefines an assignment of numeric or non-numeric parameters, and replaces the value by another one from a range specified as part of the `SETVAR` statement. Atune-IL generally assumes that the tuning parameter values are valid. The `generic` option allows the declaration of an arbitrary value to be assigned.

The `SETVAR` statement has a number of optional keywords to define additional specifications such as scale, weight, or context (see Table 1 for details).

An instrumented variable must be correctly declared in the host language and initialized with a default value. Atune-IL will modify this value at the point where the `SETVAR` instrumentation is located.

Parameter Dependencies. A conditional parameter dependency is defined by the optional `SETVAR` keyword `depends`. This causes a tunable parameter to be tuned only if the dependency condition evaluates to true. The condition may include more complex algebraic expressions.

Our program in Listing 1.1 contains the parameter `depth` that conditionally depends on parameter `sortAlgo`. The parameter `depth` is tuned only if merge sort is selected.

Permutation Regions. A set of host language statements can be marked to be permutable. The permutable statements are enclosed by a `STARTPERMUTATION` and `ENDPERMUTATION` statement, representing a permutation region. `NEXTELEM` delimits permutation elements; one permutation element may consist of several host language statements.

The example in Listing 1.1 shows a permutation region consisting of three permutation elements. Each element consists of a host language statement that adds an element to a list, so the whole region will generate a differently permuted list in different runs.

Measuring Points. Measuring points are inserted with the `GAUGE` statement, followed by a name to identify the measuring point. Atune-IL currently supports monitoring either execution times or memory consumption; the type of data to be collected is declared globally and uniformly for all measuring points. The developer is responsible for valid placements of measuring points.

The measuring points in Listing 1.1 are used to measure the execution time of two particular code segments (sorting and counting). For monitoring execution times, two consecutive measuring points with same name are interpreted as start and end time.

Blocks. Blocks are used to mark in a block-structured way the program sections that can be tuned independently (cf. Section 2.1). Such blocks run consecutively

Table 1. Atune-IL language features

Statement	*Description*
	Defining Tuning Parameters
`SETVAR <identifier>`	Specifies a tuning parameter.
`type [int|float|bool|string|generic]`	`type` specifies the parameter's type.
`values <value list>`	`values` specifies a list of numeric or non-numeric parameter values. Only feasible assignments are allowed.
`scale? [nominal|ordinal]`	*Optional*: Specifies whether parameter is ordinal or nominal scaled.
`default? <value>`	*Optional*: Specifies the parameters default value.
`context? [numthreads|lb|general]`	*Optional*: Specifies the parameter's context to provide additional information to the auto-tuner.
`weight? [0..10]`	*Optional*: Specifies the parameter's weight regarding the overall application performance.
`depends? <algebraic expr>`	*Optional*: Defines a conditional dependency to another parameter. Algebraic constraints are supported.
`inside? <block id>`	*Optional*: Assigns the parameter logically to specified application block.
	Defining Permutations Regions
`STARTPERMUTATION <identifier>`	Opens a permutation region containing an arbitrary number of target language statements. The order of the statements can be permuted by an auto-tuner.
`NEXTELEM`	Separates the code elements in a permutation region.
`ENDPERMUTATION`	Closes a permutation region. Corresponding `STARTPERMUTATION` and `ENDPERMUTATION` statements must be in the same compound statement of the host programming language.
	Defining Measuring Points
`GAUGE <identifier>`	Specifies a measuring point, e.g., to measure time.
	Defining Blocks
`STARTBLOCK <identifier>?`	Opens a block with an optional identifier.
`inside? [<block id>]`	Optional STARTBLOCK keyword: Nests the block logically inside specified parent block. A block can have only one parent block.
`ENDBLOCK`	Closes a block. Corresponding `STARTBLOCK` and `ENDBLOCK` statements must be in the same compound statement of the host programming language. A lexically nested block must be declared entirely within its parent block.

in any of the application's execution paths and their tuning parameters do not interfere with each other.

The code example in Listing 1.1 shows how to define blocks with Atune-IL. Basically, a block is enclosed by a `STARTBLOCK` and `ENDBLOCK` statement. Blocks have an optional name that can be referenced by other blocks.

Large parallel applications often contain nested parallel sections (cf. Section 2.1). To represent this structure, blocks support nesting as well – either lexically or logically. The latter requires the keyword `inside` after the `STARTBLOCK` definition to specify a parent block. A block can have one parent block, but an arbitrary number of child blocks.

Global parameters and measuring points are automatically bound to an implicit root block that wraps the entire program. Thus, each parameter and each measuring point belongs to a block (implicitly or explicitly).

In the example in Listing 1.1, `sortBlock` is logically nested within `fillBlock`, while `countBlock` is considered to be independent from the other blocks. According to our reduction concept mentioned in Section 2.1, the parameters `sortAlgo` and `depth` in `sortBlock` have to be tuned together with parameter `stringSearch` in `fillBlock`, as the order of the array elements influences sorting. This results in two separate search spaces that can be tuned one after another. These are: $dom(fillOrder) \times dom(sortAlgo) \times dom(depth)$, and $dom(numThreads)$ respectively. In addition, the dependency of $depth$ on $sortAlgo$ can be used to further prune the search space, because invalid combinations can be ignored.

It would be possible to obtain clues about independent program sections by code analysis. However, such an analysis may require additional program executions, or may deliver imprecise results. For these reasons, Atune-IL so far relies on explicit developer annotations.

4 Implementation of the Atune-IL Backend

We now discuss the principles of generating program variants. This is accomplished by the Atune-IL backend, consisting of the Atune-IL parser that preprocesses code to prune the search space, and a code generator. An auto-tuner connected to Atune-IL's backend can obtain the values of all tuning parameters as well as feedback information coming from measuring points. Then, a new program variant is generated where tuning variables have new values assigned by the auto-tuner.

4.1 Generating Program Variants

The Atune-IL backend deduces the application's block structure from `STARTBLOCK` and `ENDBLOCK` statements and analyzes dependencies of tuning parameters. It creates a data structure containing the corresponding meta-information and the current search space. If we connect an auto-tuner to the backend, the tuner can access this meta-information throughout the tuning process.

The generation of program variants is basically a source-to-source program transformation. Tuning parameter definitions are replaced by language-specific code (e.g., `SETVAR` statements are replaced by assignments). Measuring points introduced by `GAUGE` statement are replaced by calls to language-specific monitoring libraries. After the Atune-IL backend has generated a program variant, it compiles the code and returns the autotuner an executable program.

4.2 Templates and Libraries for Language-specific Code

The application of tuning parameter values as well as the calls to monitoring libraries require the generation of language-specific code. Atune-IL works with C#, C/C++, and Java, which are widely used general-purpose languages. For each language, there is a template file storing language specific code snippets, e.g., for variable assignment or calls to the monitoring library. Supporting a new host language and monitoring libraries thus becomes straightforward.

5 Experimental Results

In this section, we evaluate Atune-IL based on four case studies and applications.

MID. Our largest case study focuses on our parallelized version of Agilent's MetaboliteID (MID) [5, 1], a commercial application for biological data analysis. The program performs metabolite identification on mass spectrograms, which is a key method for testing new drugs. Metabolism is the set of chemical reactions taking place within cells of a living organism. MID compares mass spectrograms to identify the metabolites caused by a particular drug. The application executes a series of algorithms that identify and extract the metabolite candidates. The structure of MID provides nested parallelism on three levels that is exploited using pipeline, task, and data parallelism.

GrGen. GrGen is currently the fastest graph rewriting system [6]. For this case study, we parallelized GrGen and simulated the biological gene expression process on the E.coli DNA [7] as a benchmark. The model of the DNA results in an input graph representation consisting of more than 9 million graph elements. GrGen exploits task and data parallelism during search, partitioning, and rewriting of the graph.

BZip. The third case study deals with our parallelized version of the common BZip compression program [8]. BZip uses a combination of different techniques to compress data. The data is divided into fixed-sized blocks that are compressed independently. The blocks are processed by a pipeline of algorithms and stored in their original order in an output file. The size of the uncompressed input file we used for the experiments was 20 MB.

Sorting. The last case study focuses on parallel sorting. We implemented a program providing heap sort and parallel merge sort. To sort data items, the most appropriate algorithm can be chosen. Our sample array contained over 4 million elements to sort.

Table 2 lists the key characteristics of each application. The programs selected for our case studies try to cover different characteristics of several application types and domains. In addition, they reveal common parallelism structures that are interesting for tuning. For each program, input data is chosen in a way that represents the program's common usage in its application domain.

Table 2. Characteristics of the applications used in case studies

	MID	*GrGen*	*BZip*	*Sorting*
Host Language	C#	C#	C++	Java
Avg. Running Time (ms)	85,000	45,000	1,400	940
Approx. Size (LOC)[1]	150,000	100,000	5,500	500
Parallelism Types	Pipeline/ Task/Data	Task/ Data	Pipeline/ Task/Data	Task/ Data
# Identified Parallel Sections	6	3	2	2

As an additional proof of concept, we developed a sample auto-tuner that worked with Atune-IL's backend. The auto-tuner employs a common search algorithm, uses the search space information provided by Atune-IL, generates parameter configurations, starts the compiled program variants, and processes the performance results.

5.1 Results of the Case Studies

We instrumented the programs and let our sample auto-tuner iterate through the search space defined by Atune-IL. We performed all case studies on an Intel 8-Core machine[2].

Table 3 summarizes the results of all case studies. Although the programs may provide more tuning options, we have focused on the most promising parameters regarding their performance impact. The search space sizes result from the number of parameter configurations the auto-tuner has to check.

We now explain the parallel structure of the applications and how we used Atune-IL for instrumentation and search space reduction.

MID. MID has six nested parallel sections wrapped by Atune-IL blocks. The parent section represents a pipeline. Two of the pipeline stages have independent task parallel sections; one of them contains another data parallel section, while the other has two. In the task and data parallel sections we parameterized the number of threads and the choice of load balancing strategies. In addition, we introduced in the data parallel section parameters for block size and partition size; both depended on the load balancing parameters. Based on the nested

[1] LOC without comments or blank lines.

[2] 2x Intel Xeon E5320 QuadCore CPU, 1.86 GHz/Core, 8 GB RAM

Table 3. Experimental Results of Case Studies

	MID	*GrGen*	*BZip*	*Sorting*
Atune-IL Instrumentation Statements				
# Explicit Blocks	5	3	2	0
# Tuning Parameters	13	8	2	2
# Parameter Dependencies	3	3	0	1
# Measuring Points	2	3	1	1
Reduction of Search Space				
Search Space Size **w/o** Atune-IL	24,576,000	4,849,206	279	30
Search Space Size **with** Atune-IL	1,600	962	40	15
Reduction	99%	99%	85%	50%
Performance Results of Sample Auto-tuner				
Best Obtained Speed-up	3.1	7.7	4.7	3.5
Worst Obtained Speed-up	1.6	1.8	0.7	1.3
Tuning Performance Gain[3]	193%	427%	671%	269%

structure, Atune-IL's backend automatically created three separate, but smaller search spaces instead of a single large one. Considering the parameter dependencies as well, Atune-IL reduced the search space from initially 24,576,000 to 1,600 parameter configurations, that are manageable by common search algorithms. Our sample auto-tuner generates the necessary variants of the program. The best configuration obtains a speed-up of 3.1 compared to the sequential version. The moderate speed-up is caused by I/O operations, as the input data comes from hard disk and is too large for main memory.

GrGen. We wrapped three large parallel sections (search, partitioning, and rewriting of the graph) of GrGen by Atune-IL blocks. Due to Atune-IL's capability to handle multiple measuring points within a program, we added a measuring point to each block to allow fine-grained execution time feedback. In several places, we parameterized the number of threads, the number of graph partitions, and the choice of partitioning and load balancing strategies. In addition, we defined a permutation region containing 30 permutable statements specifying the processing order of search rules. Three parameters had conditional dependencies. As the parallel sections are not marked as nested, the Atune-IL backend considered them as independent and therefore reduced the search space from 4,849,206 to 962 parameter combinations. The best configuration found by the sample auto-tuner brought a speed-up of 7.7.

BZip. Two performance-relevant parameters of BZip are the block size of input data (value range set to $[100, \ldots, 900]$ bytes with step size of 100) and the number of threads for one particular algorithm in the processing pipeline

[3] The tuning performance gain represents the difference between the worst and the best configuration of the parallel program. We use it as an indicator for the impact of tuning.

(value range set to $[2, \ldots, 32]$ threads with step size of 1). As the block size does not influence the optimal number of threads and vice versa, we wrapped each parameter by a separate block. Thus, Atune-IL reduced the search space from $9 \cdot 31 = 279$ to $9 + 31 = 40$ parameter combinations. The best configuration achieved a speed-up of 4.7. Note that the worst configuration was even slower than the sequential version, which emphasizes the need for automated tuning.

Sorting. Our sorting application provides two tuning parameters, namely the choice of the sorting algorithm and the depth of the parallel merge sort that influences the degree of parallelism. As these parameters cannot be tuned separately, there is only one block. However, the parameter for the merge sort depth is relevant only if merge sort is selected. Therefore, we specified this as a conditional dependency, and the Atune-IL backend automatically cut the search space in half. The best configuration resulted in a speed-up of 3.5.

With our case studies we show 1) Atune-IL works with parallel applications differing in size, programming language, and application domain; 2) Atune-IL helps reduce the search space; 3) Atune-IL's constructs are adequate for expressing necessary tuning information within a wide range of parallel applications.

6 Related Work

Specialized languages for auto-tuning have been previously investigated mainly in approaches for numeric applications, such as ATLAS [9], FFTW [10], or FIBER [11]. Below, we mention the most relevant approaches.

XLanguage [12] uses annotations to direct a C or C++ pre-processor to perform certain code transformations. The language focuses on the compact representation of program variants and omits concepts for search space reduction.

POET [13] embeds program code segments in external scripts. This approach is flexible, but the development of large applications is difficult, as even small programs require large POET scripts. By contrast, Atune-IL requires only one line of code to specify a tuning parameter or a measuring point.

SPIRAL [14] focuses on digital signal processing in general. A mathematical problem is coded in a domain-specific language and tested for performance. It works for sequential code only.

7 Conclusion

The increasing diversity of multicore platforms will make auto-tuning indispensable. Atune-IL supports auto-tuning by providing an efficient way to define tuning information within the source code. Key contributions of Atune-IL are the extended concepts for search space reduction, the support for measuring points, as well as the ability to generate program variants. In addition, portability is improved, since platform-specific performance optimization can now be easily handed over to an arbitrary auto-tuner. Of course, Atune-IL is in an early stage and can be improved. We are currently working on an approach to generate

Atune-IL statements automatically for known types of coarse-grained parallel patterns.

Acknowledgments. We thank Agilent Technologies Inc. for providing the source code of MID as well as Agilent Technologies Foundation for financial support. We also appreciate the support of the excellence initiative in the University of Karlsruhe. Finally, we thank Thomas Karcher for his implementation of the Atune-IL parser [15].

References

1. Pankratius, V. et al.: Software Engineering For Multicore Systems: An Experience Report. In: Proceedings of 1st IWMSE. (May 2008) 53–60
2. Asanovic, K. et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, University of California, Berkeley (December 2006)
3. Tapus, C. et al.: Active Harmony: Towards Automated Performance Tuning. In: Proceedings of the Supercomputing Conference. (November 2002)
4. Werner-Kytölä, O., Tichy, W.F.: Self-Tuning Parallelism. In: Proceedings of the 8th International Conference on High-Performance Computing and Networking. (2000) 300–312
5. Agilent Technologies: MassHunter MetaboliteID Software. (2008) `http://www.chem.agilent.com`.
6. GeißR. et al.: GrGen.NET. University of Karlsruhe, IPD Prof. Goos. (2008) `http://www.info.uni-karlsruhe.de/software/grgen/`.
7. Schimmel, J. et al.: Gene Expression with General Purpose Graph Rewriting Systems. In: Proceedings of the 8th GT-VMT Workshop. (2009)
8. Pankratius, V. et al.: Parallelizing BZip2. A Case Study in Multicore Software Engineering. accepted September 2008 for IEEE Software (2009)
9. Whaley, R. C. et al.: Automated Empirical Optimizations of Software and the ATLAS Project. Journal of Parallel Computing **27** (January 2001) 3–35
10. Frigo, M., Johnson, S.: FFTW: An Adaptive Software Architecture for the FFT. In: Proceedings of the International Conference on Acoustics, Speech and Signal Processing. (May 1998) 1381–1384
11. Katagiri, T. et al.: FIBER: A Generalized Framework for Auto-tuning Software. In: Proceedings of the International Symposium on HPC. (2003) 146–159
12. Donadio, S. et al.: A Language for the Compact Representation of Multiple Program Versions. In: Proceedings of the 18th LCPC Workshop. (2006) 136–151
13. Yi, Q. et al.: POET: Parameterized Optimizations for Empirical Tuning. In: Proceedings of IPDPS. (March 2007) 1–8
14. Püschel, M. et al.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE **93** (February 2005) 232–275
15. Karcher, T.: Eine Annotationssprache zur Automatisierbaren Konfguration Paralleler Anwendungen. Master's thesis, University of Karlsruhe (August 2008)