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

Christoph A. Schaefer, Victor Pankratius, Walter F. Tichy
Institute for Program Structures and Data Organization (IPD)
University of Karlsruhe, D-76128 Karlsruhe, Germany

Abstract

Automatic performance tuning (auto-tuning) has been used in parallel numerical applications for adapting performance-relevant parameters. We extend auto-tuning to general-purpose parallel applications on multicores.
This paper concentrates on Atune-IL, an instrumentation language for specifying a wide range of tunable parameters for a generic auto-tuner. Tunable parameters include the number of threads and other size parameters, but also choice of algorithms, numbers of pipeline stages, etc. A case study of Atune-IL's usage in a real-world application with 13 parameters and over 24 million possible value combinations is discussed. With Atune-IL, the search space was reduced to 1,600 combinations, and the lines of code needed for instrumentation were reduced from more than 700 to 25.

1 Introduction

As multicore platforms become ubiquitous, many software applications have to be parallelized and tuned for performance. In the past one could afford to optimize code by hand for certain parallel machines. Manual tuning must be automated in the multicore world with mass markets for parallel computers. The reasons are manifold: the user community has grown significantly, just as the diversity of application areas for parallelism. In addition, the available parallel platforms differ in many respects, e.g., in number or type of cores, number of simultaneously executing hardware threads, cache architecture, available memory, or employed operating system. Thus, the number of targets to optimize for has exploded. Even worse, optimizations made for a certain machine may cause a slowdown on another machine.

At the same time, multicore software has to remain portable and easy to maintain, which means that hardwired code optimizations must be avoided. Libraries with already tuned code bring only small improvements, as the focus of optimization is often narrowed down to specific problems or algorithms [11]. Moreover, libraries are highly platform-specific, and require interfaces to be agreed upon. To achieve good overall performance, there seems to be no way around adapting the whole software architecture of a parallel program to the target architecture.

Automatic performance tuning (auto-tuning) [5], [10], [19] is a promising systematic approach in which parallel programs are written in a generic and portable way, while their performance remains comparable to that of manual optimization.

In this paper, we focus on the problem how to connect an auto-tuner to a parallel application. We introduce *Atune-IL*, a general instrumentation language that is used throughout the development of a parallel program to define tunable parameters. Our tuning instrumentation language is based on language-independent *#pragma* annotations that are inserted into the code of an existing parallel application. Atune-IL has powerful features that go far beyond related work in numerics [5], [19], [14]. Our approach is aimed to improve the software engineering of general-purpose parallel applications; it provides constructs to specify tunable variables, add meta-information on nested parallelism (to allow optimization on several abstraction layers), and vary the program architecture. All presented features are fully functional and have been positively evaluated in the context of a large commercial application analyzing biological data on an eight-core machine. With our approach, we were able to reduce the code size required for instrumentation by 96%, and the auto-tuner's search space by 99%.

The paper is organized as follows. Section 2 provides essential background knowledge on auto-tuning general purpose parallel applications. Section 3 introduces Atune-IL, our tuning instrumentation language. Section 4 shows how program variants are generated automatically for tuning iterations. The mechanisms employed for performance feedback to the auto-tuner are sketched in section 5. Section 6 illustrates in an extensive case study how our approach was applied in the context of a real-world, parallel application, and discusses quantitative and qualitative improvements. Section 7 compares our approach to related work. Section 8 offers a conclusion.

2 Automatic Performance Tuning

Search-based auto-tuners have been proposed in the literature to deal with the complexity faced by compilers to produce parallel code [2], [5], [15], [16], [17]. Compiler optimizations are often based on static code analysis and are part of a compiler's internals. With the growing architectural variety of parallel systems, it is obvious that extending a compiler with optimization strategies for every platform becomes hardly feasible.

Technical Report 2009-2
Institute for Program Structures and Data Organization (IPD)
University of Karlsruhe (TH), Germany, Januar 2009

An auto-tuner is a library or an independent application used on top of existing compilers [10]. It dynamically executes a parameterized application several times, and explores the parameter search space systematically. On a given target platform, it tries to find a value configuration that yields the best performance. Auto-tuners work well for numeric optimizations such as parallel matrix computations, and are superior to humans especially when non-intuitive parameter configurations yield good performance results [2].

2.1 Tuning General-Purpose Applications

We designed and implemented *Atune*, an offline tuner that adjusts parameter values between two consecutive executions of a parallel program. We extended the auto-tuning principles known from numerics to work with general-purpose parallel programs. The associated process model is depicted in Figure 1.

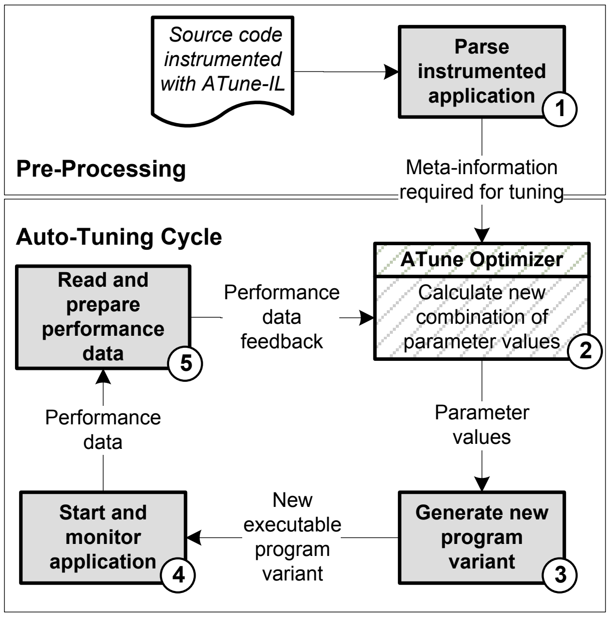

Source code instrumented with ATune-IL
Parse instrumented application
1
Pre-Processing
Meta-information required for tuning
Auto-Tuning Cycle
ATune Optimizer
Calculate new combination of parameter values
2
Read and prepare performance data
5
Performance data feedback
Performance data
Parameter values
Start and monitor application
4
New executable program variant
Generate new program variant
3

Figure 1: Atune's auto-tuning cycle

We assume that we have an existing parallel program written in a host language, which is instrumented with Atune-IL. The instrumentation language is used to mark tuning parameters in the code of a host language, to define value intervals for tuning, and to set monitoring probes (e.g., for execution time or memory consumption) at appropriate locations (cf. Section 3).

Atune's tuning cycle works as follows (cf. Figure 1):

(1) A pre-processor parses the instrumented program and builds up a data structure with tuning meta-information.

(2) The tuning meta-information is passed on to the *Atune optimizer*. As the internals of the optimizer are out of scope of this paper, we sketch only the principles here and refer to existing approaches [3], [15], [16], [17], [18] for details. The optimizer computes a tuple of values that represents a configuration of parameters. Atune basically moves along in an n-dimensional search space defined by the cross product of all parameter domains, i.e., $dom(p_1) \times dom(p_2) \times ... \times dom(p_n)$, to find a configuration $(val_1, ..., val_n)$ with $val_i \in dom(p_i)$ that yields the best performance. A simple search strategy is to systematically try out all combinations of parameter values. However, this frequently used technique is only feasible for small spaces. More sophisticated strategies therefore try to prune the search space based on different heuristics or previous tuning iterations [3], [15], [16], [17], [18]. In our approach, we designed Atune-IL in such a way that it helps Atune reduce the search space, using the developer's knowledge; most of the instrumentation constructs provide meta-information that can be exploited by Atune's optimizer.

(3) Atune weaves the computed parameter values back into the code of the parallel program. At the same time, all Atune-IL annotations and placeholders are removed, and measurement probes are replaced by calls to a performance monitoring library. The output of this stage is an executable variant of the original program (cf. Section 4). Note that this program corresponds to one whose tuning parameters would have been adjusted by hand.

(4) Next, Atune starts the program and monitors it. Data from all monitoring probes is recorded, summarized, and stored.

(5) The last step completes the feedback loop of the Auto-Tuning Cycle. The recorded monitoring results are transformed to a format usable by the Atune optimizer (cf. Section. 4.3).

The whole auto-tuning cycle (steps 2 - 5) is repeated until some predefined condition is met; this depends on the search strategy employed in step 2. It is therefore sensible to let Atune control the execution of all steps.

Atune-IL establishes the connection between Atune and the parallel application to tune. In the next section, we present the details of Atune-IL and show how programs are instrumented in the first step of the cycle.

3 The Tuning Instrumentation Language Atune-IL

This section introduces in a step-by-step fashion all features of our tuning instrumentation language. We start with a simple definition of tuning parameters, explain how to express parameter dependencies, and introduce the concept of tuning blocks that simplify tuning on several abstraction layers. Further on, we describe how to set monitoring probes. Thereafter, we discuss the assumptions, trade-offs, and design decisions behind Atune-IL.

3.1 Defining Tuning Parameters

In many situations, programmers want to change the values of a variable between subsequent tuning runs in order to observe the relative performance impact. Atune-IL helps automate this process with the *SETVAR* keyword; it is used to mark a variable in the host language as tunable and to define a set of values that the auto-tuner will try out. Like all Atune-IL statements, *SETVAR* is preceded by the *#pragma atune* prefix.

Defining Numeric Parameters

As an illustrative example, consider the code in Figure 2 that uses the variable *numThreads* to control the number of threads in a program. To let the auto-tuner vary this number, the programmer adds a *#pragma* annotation after the variable, followed by *SETVAR numThreads* to mark it as tunable. Using *TYPE int*, the domain of trial values is constrained to integers. The value range is defined by *VALUES 2-16 STEP 2*, implying that *numThreads* will be set to the values 2,4,… ,16.

```
public void SETVAR_Example1()
{
   int numThreads = 2;
   #pragma atune SETVAR numThreads
      TYPE int VALUES 2-16 STEP 2

   for (int i=1; i<=numThreads; i++)
   {
      Thread.Create(StartCalculation);
   }
   WaitAll();
}
```

Figure 2: Code example using the SETVAR statement to define a numeric tuning parameter

Defining Architectural Variants

A powerful feature of Atune-IL is that the *TYPE* of values in a *SETVAR* statement need not be numeric. Thus, architectural variants of a program can be defined as shown in Figure 3. Assuming that this program implements a sorting routine in a generic way, we can go to the point where the employed sorting algorithm is first instantiated and insert an annotation with *TYPE generic*; this allows us to include host language code for the creation of each algorithm instance. While the auto-tuner just sees two options that can be tried out in different tuning runs, it will actually try out two architectural variants of the program.

Architectural variants are useful for automating fallback mechanisms. For example, a parallel merge sort algorithm may work well in many cases, depending on the size of data and the characteristics of a multicore machine. However, for some borderline cases, a better performance may be achieved with a sequential sort that has less overhead than the parallel algorithm. Atune-IL is flexible to handle as many alternatives as necessary.

```
public void SETVAR_Example2()
{
   ISortAlgorithm sortAlgo = new QuickSort();
   #pragma atune SETVAR sortAlgo
      TYPE generic VALUES "new QuickSort()",
         "new ParallelMergeSort()"

   if (sortAlgo != null)
      sortAlgo.Run();
}
```

Figure 3: Code example using the SETVAR statement to define a non-numeric tuning parameter

Additional Support for the Optimization

The *SETVAR* keyword has additional options that were not mentioned in the previous examples. A value in the specified interval may be defined as the *START* value that is tried out first. This is useful when a variable that controls the number of threads should be tried out first with the number of available hardware threads.

A *WEIGHT* number may quantify the importance of the annotated variable for the overall optimization, and the *SCALE nominal* or *SCALE ordinal* keyword may inform Atune that this variable has nominal or ordinal scale. With this information, the optimizer may treat such variables in a different way.

3.2 Defining Parameter Dependencies

The *DEPENDS* keyword offers Atune additional meta-information that helps prune the search space. As an example, suppose that the parallel merge sort in Figure 4 has a parameter *depth* defining how far the input will be split up into partitions. This parameter could be varied in several runs to find out the best performance on a certain machine. As Atune's optimizer does not know that this parameter is only meant to work with merge sort, it would vary it for quick sort as well. Using *DEPENDS*, a developer can make his intention explicit and communicate to the optimizer to avoid unnecessary tuning iterations, thus reducing the search space.

```
public void DEPENDS_Example()
{
   ISortAlgorithm sortAlgo = new QuickSort();
   #pragma atune SETVAR sortAlgo
      TYPE generic VALUES "new QuickSort()",
         "new ParallelMergeSort()"

   int depth = 2;
   #pragma atune SETVAR depth
      TYPE int VALUES 2-8
      DEPENDS sortAlgo VALUES
         "new ParallelMergeSort()"

   if (sortAlgo != null)
      // Run() ignores depth if QuickSort is
      // selected
      sortAlgo.Run(depth);

}
```

Figure 4: Code example using the DEPENDS keyword to define a parameter dependency

3.3 Defining Tuning Blocks

Tuning blocks are used to mark parallel sections which may be tuned independently. Atune considers parallel sections enclosed in a tuning block to be independent if they run consecutively in any of the application's execution paths and their tuning parameters do not interfere with each other. Atune can exploit this information throughout the optimization process to reduce the search space.

For illustration, consider Figure 5. It shows the hypothetical execution paths of a parallel program, divided into two blocks that the developer knows to be independent (e.g., due to design decisions). Block one has three tuning parameters, $p_1, \dots, p_3$, while block two contains five tuning parameters, $p_4, \dots, p_8$.

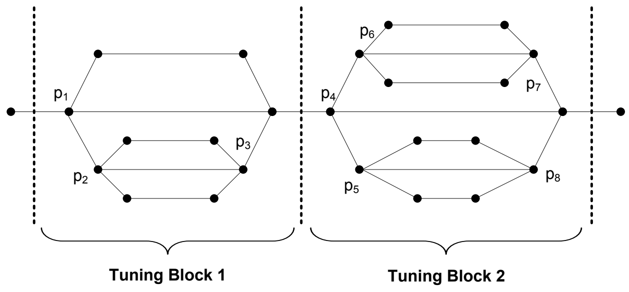


Figure 5: Concept of Tuning Blocks

Without the block instrumentations, Atune would try out in the worst case the cross product of all parameter domains: $dom(p_1) \times \dots \times dom(p_8)$. However, if the two blocks are known to be independent, the worst case for each block is reduced to the cross product of the respective parameter domains, i.e., $B_1 := dom(p_1) \times \dots \times dom(p_3)$ and $B_2 := dom(p_4) \times \dots \times dom(p_8)$, thus avoiding a large number of trials, namely $B_1 \times B_2$.

Figure 6 shows how to mark tuning blocks with Atune-IL. Basically, a tuning block is enclosed by a *STARTBLOCK* and *ENDBLOCK* statement. Tuning blocks may have a name, so that they can be referenced from other blocks.

```
public void TUNINGBLOCKS_Example()
{
   #pragma atune STARTBLOCK parallelSection

   // Here follows the code shown in
   // SETVAR_Example1() in Figure 2

   #pragma atune ENDBLOCK
}
```

Figure 6: Atune-IL statements to define a tuning block

It is of course technically possible to obtain clues about independent program sections by code analysis. However, such an analysis is complex, may require additional program executions, or may deliver imprecise results; this is why Atune-IL relies on explicit developer annotations.

Nested Structures

Tuning blocks can be lexically nested. A significant number of cross product operations can be saved when nested parallel sections are marked. When nested structures are detected, Atune starts the optimization in the tuning blocks at the innermost level and successively combines their parameter values with those in the directly enclosing blocks.

In situations where nested blocks cannot be expressed in the lexical scope of their enclosing blocks, the *INSIDE* keyword of the *STARTBLOCK* statement may be used to specify a logically nested structure, provided that the referenced blocks have a name. Figure 7 shows an example of a routine that is nested within the parallel section in Figure 6. Note that the code of this routine could be located in an entirely different file.

```
public void StartCalculation()
{
   #pragma atune STARTBLOCK nestedSection
      INSIDE parallelSection

   // Do the calculation in a nested parallel
   // section with own tuning parameters.

   #pragma atune ENDBLOCK
}
```

Figure 7: Defining a nested tuning block inside the parallel section shown in Figure 6

Atune internally creates a tree to handle the nested structure of tuning blocks. Therefore Atune automatically adds a root tuning block to its data structure enclosing the entire application. Tuning parameters specified outside a tuning block are logically assigned to the root tuning block.

3.4 Defining Monitoring Probes

Monitoring probes are inserted into the code by the *GAUGE* statement, followed by a name to identify the type of the probe. Currently, Atune supports probe types to monitor either execution times or memory consumption. The probe types are declared globally for all probes in a configuration file.

As an example, the probes in Figure 8 measure the execution time of a particular code segment. For probe types that measure execution times, two consecutive probes are interpreted as start time and end time, and the difference if computed automatically when the second probe is reached.

In case that memory consumption was specified in Figure 7 as the probes' type, the two statements would have been interpreted as two separate probes, both measuring memory usage at that point.

```
public void COMPLETE_Example()
{
   #pragma atune STARTBLOCK parallelSection

   #pragma atune GAUGE myExecTime

   int numThreads = 2;
   #pragma atune SETVAR numThreads
      TYPE int VALUES 2-16 STEP 2

   for (int i=1; i<=numThreads; i++)
   {
      Thread.Create(StartCalculation);
   }
   WaitAll();

   #pragma atune GAUGE myExecTime

   #pragma atune ENDBLOCK
}
```

Figure 8: Code example using the GAUGE statement to define monitoring points

3.5 Assumptions and Design Decisions

Atune-IL is designed to reduce the implementation effort for tuning instrumentation, and to help prune the search space for Atune so that fewer executions are required in the auto-tuning cycle. There are several assumptions about how Atune is employed; all of them were considered carefully in order to design a flexible language for the tuning of general-purpose parallel applications.

- Atune-IL was designed to be independent of the host programming language and the tuned application (for details, esp. on how we deal with library calls for probes, see section 4). As a trade-off, this flexibility requires the developer to take additional responsibilities in situations as described next.
- The Atune-IL parser does not check for coherence between the application's source code and its instrumentation statements. This would have required the implementation of a parser of every host language.
- Except for tuning, Atune-IL has no general control over the usage of the variables instrumented by the *SETVAR* statements.
- We assume that a tuning block is opened and closed within the same compound statement of the host programming language, such as a method or a loop. This applies as well for two consecutive *GAUGE* statements measuring execution times.
- A tuning block may contain an arbitrary number of *SETVAR* statements.
- For a given tuning block, we assume that no variable is accessed from outside the block.
- A variable that is instrumented with *SETVAR* must be correctly declared in the host language and initialized with a default value. Atune will modify this value at the point where the pragma instrumentation is located. The programmer must avoid any other write accesses to that variable that might interfere with the tuning process.
- Overhead in the monitoring library affects measurements. However, this overhead would also occur in an approach without auto-tuning.

In our opinion, we think that the aforementioned trade-offs are acceptable. In our case study (cf. section 6) these assumptions do not cause any serious problems in practice.

4 Generating Program Variants

We now discuss the principles of program generation used in step 3 of the auto-tuning cycle (cf. Figure 1). At this stage, Atune's optimizer has already determined a value for each tuning parameter, and the values need to be assigned to the corresponding variables in the source code of the parallel program.

4.1 General Principles

The *#pragma* statements described previously are categorized into three classes for which the variant generation process works differently. First, the *SETVAR* statement requires language-specific code insertions to set certain values for tunable variables. Second, statements with meta-information for the auto-tuner, such as *STARTBLOCK* or *ENDBLOCK*, are simply removed. Third, monitoring probes introduced by *GAUGE* are replaced by calls to language-specific monitoring libraries.

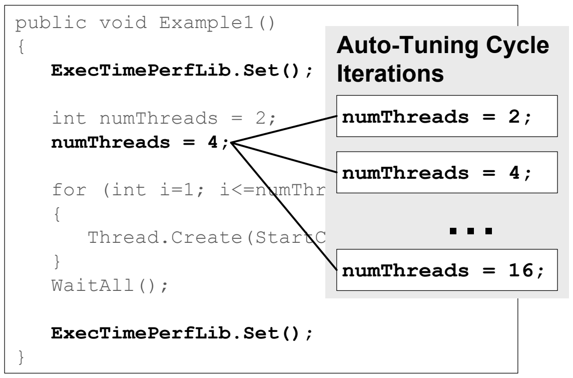
4.2 Templates and Libraries for Language-Specific Code

It may seem contradictory to require the generation of language-specific code and keep Atune-IL independent of the host language at the same time. We approached this problem by using standardized templates. For every Atune-IL construct (e.g., variable assignment with *SETVAR*), we store the corresponding code used in the host programming language in a template file. For implementation, we employed *StringTemplate* [12], [13] that also allowed us to capture the syntax of host language statements. As a proof of concept, we created such template files for several languages, including C#, Java, and Perl. New templates can easily be added by adding such files in a certain directory. The template to be used by Atune is defined in the central configuration file.

We defined a general interface to the monitoring library that provides functionality for measuring execution times and memory consumption. As a library implementation can only be used in programs written in the same language as the library itself, we created different implementations for various languages: Java, Perl, and C# (whose library is applicable to all programs based on the .NET Common Language Runtime). The interface is designed in such a way that developers may easily add implementations for other languages as well as extensions of probe types.

4.3 Tunable Variables and Monitoring Probes

As an example for the handling of tunable variables and monitoring probes, we illustrate a possible outcome of the generation process for C# in Figure 9; the generated variant is based on the code in Figure 8.



```
public void Example1()
{
   ExecTimePerfLib.Set();

   int numThreads = 2;
   numThreads = 4;

   for (int i=1; i<=numThr
   {
      Thread.Create(StartC
   }
   WaitAll();

   ExecTimePerfLib.Set();
}
```

Figure 9: Example for a generated variant based on the code in Figure 8

All *SETVAR* statements are replaced by a line of code that assigns a parameter value to the specified variable. For numeric parameters, a number is assigned; for non-numeric parameters of type "generic" the value is set to the specified string. In each of the auto-tuning iterations, a new program variant is generated by assigning the values obtained from Atune's optimizer.

For monitoring probes, *GAUGE* statements are replaced by appropriate library calls. As shown in Figure 9 for C#, the call is done via a static class name chosen according to the probe type and is followed by the method name `Set()`. This method contains the actual measurement functionality.

We omit the discussion of more subtle details of the generation process and refer to [6] for details.

5 Feedback of Performance Results

In step 4 of the auto-tuning cycle (cf. Figure 1) a generated program variant is executed monitored for performance. During runtime, the inserted calls to the monitoring library are used to record performance data.

At the end of the execution, all gathered values are written to a file. Atune reads the values from this file, aggregates them, and computes a new value for the overall objective function. The results are communicated to Atune's optimizer that uses them in the calculation of new parameter values.

The feedback of performance results completes the auto-tuning cycle.

6 Case Study

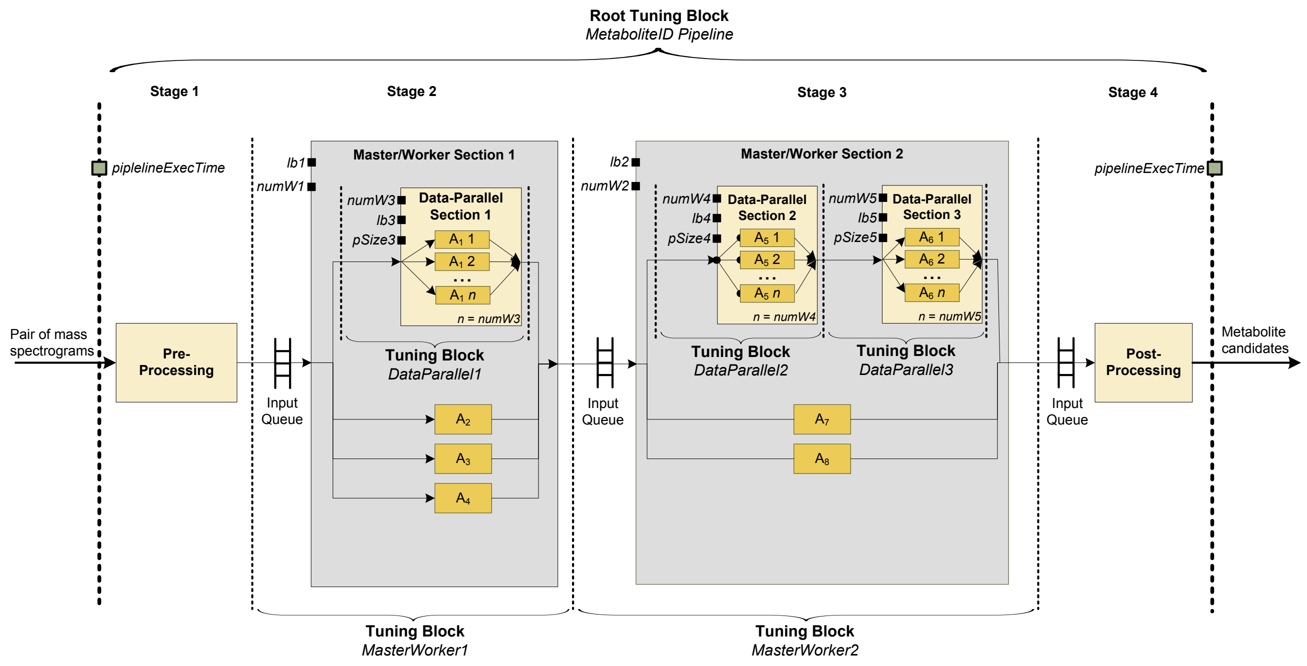
In this Section, we present a detailed case study on the instrumentation of a parallelized version of Agilent's MetaboliteID [1], a commercial analysis application for biological data. There were several reasons to choose this application:

- MetaboliteID is a large application (more than 100.000 lines of code in C#) containing potential parallelism at different levels of granularity.
- It is a commercial application providing a real-world scenario.
- The size and architecture of the application is similar to other large computation-intensive programs.

First, we parallelized MetaboliteID and identified tuning parameters that have an impact on the overall execution time of the program [10]. We then instrumented the application with Atune-IL to make it ready for tuning.

6.1 Biological Data Analysis

MetaboliteID performs so-called metabolite identification, a key method for testing new drugs. Metabolites are the intermediate products of metabolism. Metabolism is the set of chemical reactions taking place within cells of a living organism.



Tuning Parameters:
numW: Number of worker threads
lb: Choice of load balancing (LB) strategy
(*static*, *dynamic*, *work-stealing*)
pSize: Size of data partitions
(depends on lb*i*: parameter only
effective if static LB strategy selected)

■ Tuning Parameter
🟩 Monitoring Point
----- Border of Tuning Block
●—● Program Execution Path

Atune-IL statements used to define the tuning parameters and monitoring probes

- *lb1:* **#pragma atune SETVAR lb1 TYPE generic VALUES "LB.Dynamic";"LB.Static" DEFAULT "LB.Dynamic" SCALE nominal**
  (similar statements for *lb2, lb3, lb4, lb5*)
- *numW1*: **#pragma atune SETVAR numW1 TYPE int VALUES 1-4 STEP 1 DEFAULT 4 SCALE ordinal**
  (similar statements for *numW2, numW3, numW4, numW5*)
- *pSize3*: **#pragma atune SETVAR pSize3 TYPE float VALUES 0.1-0.5 STEP 0.1 DEFAULT 0.1 SCALE ordinal DEPENDS lb3 VALUES "LB.Static"**
  (similar statements for *pSize4, pSize5*)
- *pipelineExecTime*: **#pragma atune GAUGE execTime**

Figure 10: Parallel structure and tuning model of Agilent's MetaboliteID

The metabolite identification process is based upon the comparison of two body fluid samples. The first sample (control sample) is obtained before taking the drug. At certain times after the application of the drug, further samples (metabolite samples) are taken. Finally, mass spectrograms of all samples are prepared.

MetaboliteID compares each of the mass spectrograms of the metabolite samples with the control sample to identify the metabolites caused by the drug. The application executes a series of algorithms that identify and extract the metabolite candidates. This sequence is repeated for each metabolite sample.

6.2 Parallelizing MetaboliteID

We parallelized the application on different levels of abstraction to exploit available nested parallelism, as illustrated in Figure 10.

On the most coarse-grained level, we implemented a parallel pipeline to speed-up the processing of several pairs of mass spectrograms (control and metabolite sample).

Next, we turned to the individual pipeline stages. In principle, stage 1 reads the mass spectrograms, stages 2 and 3 are algorithm modules ($A_1 \ldots A_8$) carrying out the metabolite identification, and stage 4 aggregates the results. In stage 2 and 3, we had some of the algorithms work independently on disjoint parts of the mass spectrograms; for those algorithms, we were able to exploit task parallelism by using a Master/Worker pattern.

Algorithm modules were the lowest abstraction level that exploited parallelism. The internals of the algorithms were not modified, as we were focusing on coarse-grained application parallelization rather than on fine-granular algorithmic engineering. The algorithm modules A_1, A_5, and A_6 were enhanced to support data parallel execution. As they processed incoming fragments of mass spectrograms independently, we used for each module a data decomposition strategy that split up the input data into a number of partitions, and which created several parallel instances of the same module.

As shown in Figure 10, the data parallel section of module A_1 is nested in the master/worker section of stage 2, while the data parallel sections of A_5 and A_6 are nested in the Master/Worker section of stage 3. This complex structure of the parallel program required multi-level tuning.

6.3 Instrumenting the Parallel Program

After parallelizing MetaboliteID, we instrumented the application with Atune-IL statements to provide the necessary tuning meta-information for Atune.

We started with the definition of tuning blocks. Each of the parallel sections (e.g., each master/worker or data parallel section) was treated as a tuning block.

We continued with the specification of tuning parameters for each parallel section. We already identified in the earlier parallelization process the parameters that influenced the execution time of the application. Thereafter, we added the corresponding variables along with the functionality necessary to change the behavior of the application according to the variables' values.

For the master/worker sections, we defined the load balancing strategy (parameter *lb:* static or dynamic) and the number of worker threads (parameter *numW*: 2…16) as tunable parameters, and implemented a static and a dynamic load balancing strategy.

The data parallel sections have similar parameters for load balancing and the number of workers. In addition, they had a parameter to set the size of the data partition (parameter *pSize*) for the case when static load balancing was used. The parameter *pSize* had a depends-relationship to the parameter *lb*.

Finally we defined two monitoring probes to measure the execution time of the entire pipeline, i.e., the entire program.

6.4 Results

Implementation Effort

The listing in Figure 10 shows the Atune-IL statements we used to specify the required tuning meta-information. We defined five tuning blocks, 13 tuning parameters (three of them had a dependency) and two monitoring probes. Specifying all tuning meta-information using Atune-IL required 25 lines of instrumentation statements.

Without Atune-IL, one has to manually implement the tuning parameters, value ranges, as well as all other parameter information such as data type, scale, weight, or dependencies, as well as tuning blocks and monitoring libraries. In addition, the data structure for the tuning block structure and appropriate monitoring libraries must be created.

To compare the implementation effort with and without using Atune-IL, we created a separate program which encapsulated the logic to produce multiple variants of MetaboliteID based on tuning parameters. Apart from that, we added code directly into MetaboliteID. To get the same functionality as provided by the Atune-IL statements, the following implementation effort was necessary: the data structure for tuning blocks, tuning parameters, and monitoring probes requires 350 lines of code (LOC). The specification of a tuning block needs 8 LOC. The definition of each tuning parameter requires 10 LOC. In addition, 15 LOC are necessary to include a parameter in the tuning block data structure and to perform validations. A monitoring probe requires only one LOC, as we still used a function call. We also added functionality to measure the execution time, which takes 30 LOC.

Table 1: Comparison of implementation effort to add auto-tuning capabilities to MetaboliteID

	Atune-IL	**Manually implemented**
Data structure and validation logic	*included in Atune-IL*	350 LOC
Tuning blocks	$5 \cdot 2 = 10$ LOC	$5 \cdot 8 = 40$ LOC
Tuning parameters	13 LOC	$13 \cdot (10 + 15) = 325$ LOC
Monitoring probes	$2 \cdot 1$ LOC	$2 \cdot 1$ LOC
Monitoring functionality to measure exec. times	*included in Atune-IL*	30 LOC
Sum	**25 LOC**	**747 LOC**
	3.35%	**100%**

Table 1 summarizes the lines of code needed to add auto-tuning capabilities to MetaboliteID in the same way Atune-IL does. It shows that using Atune-IL the implementation effort is reduced by more than 96% !

Search Space Reduction

Using Atune-IL significantly reduced the search space for Atune's optimizer, thus saving tuning iterations.

We instrumented MetaboliteID with 13 tuning parameter definitions. Normally, the search space would have been the cross product of all parameter domains (24,576,000 parameter value combinations). Based on Atune-IL's tuning blocks, Atune could determine independent (nested) parallel sections, i.e. (nested) parallel sections running one after another in any of the application's execution paths and thus not interfering with each other.

Three independent parallel sections that could be tuned separately (cf. Figure 10):

- Tuning block *MasterWorker1* and the nested tuning block *DataParallel1* (640 parameter value combinations)
- Tuning block *MasterWorker2* and the nested tuning block *DataParallel2* (480 parameter value combinations)
- Tuning block *MasterWorker2* and the nested tuning block *DataParallel3* (480 parameter value combinations)

Thus, the search space consisted in the worst case was reduced to 640 + 480 + 480 = 1.600 combinations to be tried out. Compared to the original search space with 24,576,000 combinations, we had a reduction of more than 99%. In fact, 1,600 combinations were tried out by Atune.

Finally we tuned the instrumented version of MetaboliteID using Atune. The auto-tuner was able to generate the 1.600 necessary program variants that were all executed. Between the best and the worst parameter configuration, Atune determined a difference in execution time of approximately 45%. This result underlines that auto-tuning is helpful in a large parallel application such as ours.

7 Related Work

Search-based auto-tuning has been previously investigated in the area of numerical software and high-performance computing. Some approaches employ instrumentation languages developed specifically for this context.

The Fastest Fourier Transform in the West (FFTW) [5] uses generative programming techniques to generate a complete FFT application from scratch. In principle, the approach composes pre-defined blocks of code and tries out combinations until it finds the best result on a certain hardware platform.

The Automatically Tuned Linear Algebra Software (ATLAS) system [19] generates a platform-specific linear algebra library. Before the library is generated, the Automated Empirical Optimization of Software (AEOS) component executes micro benchmarks on a target platform and determines the hardware-specific parameters that yield the best performance. The optimization process is especially focused on memory characteristics such as latency or cache sizes.

XLanguage [4] uses a *#pragma* approach to direct a C or C++ pre-processor to perform certain code transformations. Contrary to the other related work, the optimization step is not part of the language. XLanguage provides useful constructs to generate loop unrollings explicitly in the high-level code, which is often applied to improve the performance of matrix multiplications. Although the language allows for various extensions, it lacks constructs that are required for tuning general-purpose parallel applications.

Parameterized Optimizing for Empirical Tuning (POET) [20] uses a language that embeds the segments of code that are used to generate an application directly into POET code. The code generation process is driven by transformation rules that are specified by the developer. This approach is flexible, but the software engineering of large applications is difficult. The syntax is verbose, so that even simple loop unrolling for numeric optimizations needs several dozens of lines of code.

SPIRAL [14] focuses on digital signal processing in general. A mathematical problem is coded in a so-called Signal Processing Language, a domain-specific language. Various platform-dependent versions are created and tested for performance. It works for sequential code only.

The Framework of Install-time, Before Execute-time and Run-time optimization (FIBER) [7] is a software framework that employs compiler directives and the script language ABClibscript to automate the optimization process. Similar to Atune-IL, FIBER can mark tunable variables and define values to be tried out. However, the entire approach focuses on numerics and was not designed for general-purpose parallel applications.

We summarize related work in Table 2 and compare each language with respect to several key characteristics. Atune-IL provides several capabilities in one single language. Note that contrary to other approaches, we separated the optimizer from the instrumentation language to gain more flexibility. Furthermore, our approach does not generate programs from scratch; it assumes that an already existing parallel program will be tuned.

Table 2: Comparison with existing approaches

	FFTW	ATLAS	XLanguage	POET	SPIRAL	FIBER	Atune-IL
Usable with any host programming language	-	-	-	✓	-	-	✓
Independent of application domain	-	-	✓	✓	-	-	✓
Monitoring support	-	-	-	-	-	-	✓
Support for nested parallelism	-	-	-	-	-	-	✓
#pragma-based approach	-	-	✓	-	-	-	✓
Program generation from scratch	✓	✓	-	-	✓	-	-
Numeric code optimizations included	✓	✓	-	-	✓	-	-

8 Conclusion

The increasing diversity of multicore platforms will make auto-tuning indispensable. Atune-IL connects a generic auto-tuner to general-purpose parallel applications. Portability is improved, as platform-specific performance optimization can now be easily sourced out to an auto-tuner. Additional key contributions of Atune-IL are the support for search space reduction, the ability to specify architectural variants, and the definition of different types of monitoring probes.

Of course, Atune-IL is in an early stage and can be improved in many ways. For example, the syntax for the definition of architectural variants can be adapted to work with pre-defined source code files. In addition, other types of monitoring probes could be added. Support for online-tuning during program execution is interesting as well. Various directions could be explored to integrate auto-tuners directly into compilers and extend programming languages by native constructs for tuning.

Acknowledgements

We thank Agilent Technologies Inc. for providing the source code of Metabolite ID as well as Agilent Technologies Foundation for financial support. We also appreciate the support of the excellence initiative in the University of Karlsruhe. Last but not least, we thank Thomas Karcher for his implementation of the Atune-IL parser.

References

[1] Agilent Technologies. MassHunter MetaboliteID software. http://chem.agilent.com/. Last accessed January 2009.

[2] K. Asanovic et al. "The landscape of parallel computing research: A view from Berkeley". Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[3] R. Chung, J.K. Hollingsworth, "Using Information from Prior Runs to Improve Automated Tuning Systems". *Proceedings of the ACM/IEEE SC2004 Conference*, pp. 30-38, Nov. 2004.

[4] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzarn, D. Padua, and K. Pingali, "A language for the compact representation of multiple program versions", in *18th International Workshop Languages and Compilers for Parallel Computing (LCPC)*, ser. LNCS, no. 4339, pp. 136–151, 2006

[5] M. Frigo and S. Johnson, "FFTW: An adaptive software architecture for the FFT," *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384 vol.3, 1998.

[6] T. Karcher. "Eine Annotationssprache zur automatisierbaren Konfguration paralleler Anwendungen", *Master's Thesis*, August 2008, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, Germany.

[7] T. Katagiri, K. Kise, H. Honda, and T. Yuba, "FIBER: A generalized framework for auto-tuning software", *High Performance Computing*, pp. 146–159, 2003.

[8] A. Morajko, E. César, T. Margalef, J. Sorribes, and E. Luque, "MATE: Dynamic performance tuning environment," *Euro-Par Parallel Processing,* vol. 3140, pp. 98–107, 2004.

[9] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque, "MATE: Monitoring, analysis and tuning environment for parallel/distributed applications", *Concurrency and Computation: Practice and Experience*, vol. 19, pp. 1517–1531, 2007.

[10] V. Pankratius, C. A. Schaefer, A. Jannesari, and W. F. Tichy. "Software engineering for multicore systems: an experience report". *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pp. 53-60, New York, NY, USA, 2008. ACM.

[11] V. Pankratius, A. Jannesari, W. F. Tichy. "Parallelizing BZip2. A case study in multicore software engineering". Accepted September 2009 for IEEE Software.

[12] T. Parr. "A Functional Language for Generating Structured Text". May 2004.

[13] T. Parr. The StringTemplate Homepage. http://www.stringtemplate.org/. Last accessed September 2008.

[14] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "SPIRAL: Code generation for dsp transforms", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[15] A. Qasem, K. Kennedy, and J. Mellor-Crummey, "Automatic tuning of whole applications using direct search and a performance-based transformation system", *The Journal of Supercomputing*, vol. 36, no. 2, pp. 183–196, 2006.

[16] V. Tabatabaee, A. Tiwari, and J. Hollingsworth, "Parallel parameter tuning for applications with performance variability", *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 57–57, 2005.

[17] C. Tapus, I.-H. Chung, and J. Hollingsworth, "Active Harmony: Towards automated performance tuning", *Supercomputing, 2002. Proceedings of the ACM/IEEE SC 2002 Conference*, pp. 44–44, 2002.

[18] O. Werner-Kytölä, W. F. Tichy, "Self-Tuning Parallelism", *Proceedings of the High Performance Computing and Networking Europe 2000*, Springer LNCS #1823, p. 300-312, 2000

[19] R. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project". *Parallel Computing*, 27(1-2), pp. 3-35, Jan. 2001

[20] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "Poet: Parameterized optimizations for empirical tuning", *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pp. 1–8, 2007.