Engineering Parallel Applications with Tunable Architectures

Proc. Int'l. Conf. on Software Engineering 2010, Vol 1, 405-414.

Christoph A. Schaefer
Karlsruhe Institute of Technology, IPD
Am Fasanengarten 5
76131 Karlsruhe, Germany
cschaefer@ipd.uka.de

Victor Pankratius
Karlsruhe Institute of Technology, IPD
Am Fasanengarten 5
76131 Karlsruhe, Germany
pankratius@ipd.uka.de

Walter F. Tichy
Karlsruhe Institute of Technology, IPD
Am Fasanengarten 5
76131 Karlsruhe, Germany
tichy@ipd.uka.de

ABSTRACT

Current multicore computers differ in many hardware characteristics. Software developers thus hand-tune their parallel programs for a specific platform to achieve the best performance; this is tedious and leads to non-portable code. Although the software architecture also requires adaptation to achieve best performance, it is rarely modified because of the additional implementation effort. The Tunable Architectures approach proposed in this paper automates the architecture adaptation of parallel programs and uses an auto-tuner to find the best-performing software architecture for a particular machine. We introduce a new architecture description language based on parallel patterns and a framework to express architecture variants in a generic way. Several case studies demonstrate significant performance improvements due to architecture tuning and show the applicability of our approach to industrial applications. Software developers are exposed to less parallel programming complexity, thus making the approach attractive for experts as well as inexperienced parallel programmers.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Performance, Design, Languages

1. INTRODUCTION

Performance tuning of parallel applications is a challenging task on today's multicore computers because they differ in a variety of ways (e.g., in the number of cores per chip, memory bandwidth, cache architecture, or employed operating systems). Being under pressure to deliver the best parallel program performance, many software developers are forced to hand-tune their programs for certain platforms. This approach is tedious, costly, and may lead to non-portable code that has to be re-tuned on new machines [29, 30, 27, 16, 18]. Recent studies [14, 15, 17] have shown that recompiling programs with appropriate instruction-level optimizations does not always lead to acceptable performance on new platforms, and that the parallel program architecture also needs adaptation. In addition, some situations require reordering of nested parallel components. Yet programmers are hesitant to do such invasive changes on new machines due to the required implementation effort.

We propose Tunable Architectures to address these problems for shared-memory parallel programs on multicore platforms. In principle, developers make the performance-relevant parts of a program architecture configurable and prepare the program for experiments. The experiments are done by an auto-tuner – this is an external program that systematically tests a program with different architecture variants on a particular machine to find the best-performing variant. Auto-tuning is a feedback-directed process consisting of several steps: choice of architecture configuration, program execution, monitoring of execution time, and generation of a new configuration based on optimization strategies such as hill climbing or simulated annealing. In this process, the software architecture converges to the best-performing architecture on a given target platform.

To realize Tunable Architectures, we propose a novel architecture description language that has operators to express parallel programming patterns [12] such as pipelines, producer-consumer, or fork-join. Developers start with atomic software components that contain executable code. The operators are used to compose a parallel program out of atomic software components. By default, all patterns have an associated set of performance parameters (e.g., number of producers for the producer-consumer pattern), and an auto-tuner is supposed to choose appropriate values out of a user-defined value range. The language also allows specifying one or more architecture variants that compose a parallel program in different ways; these are the variants that are tested by an auto-tuner.

Tunable Architectures advance earlier work [22, 4, 24, 11, 13, 10] in two important ways: they provide the technical means to express performance-relevant architecture variations needed specifically for parallel programs, and they automate the architecture optimization process for parallel programs on contemporary shared-memory multicore machines. Our case studies with realistic C# programs show

ICSE '10, May 2-8 2010, Cape Town, South Africa

that the approach works and that the language is expressive enough to implement a wide range of parallel programs. We also achieved significant performance improvements while keeping code portable. At the same time, developers had less exposure to the complexity of parallel programming and performance tuning, which makes the approach attractive not only for experts, but also for less experienced parallel programmers.

The paper is organized as follows: Section 2 motivates why we need tunable architectures for multicore parallel programs. Section 3 introduces our Tunable Architecture Description Language (TADL); an application of the language is illustrated by examples in Section 4. In Section 5, we describe the implementation concepts for configurable parallel programs and present a full-fledged TADL compiler that creates executable multicore applications. In Section 6, four case studies, including an industrial application, evaluate our approach in different contexts. Section 7 discusses related work, and we present our conclusions in Section 8.

2. WHY WE NEED TUNABLE ARCHITECTURES

Developing parallel applications is difficult when software engineers need to pay attention to low-level concurrency details, such as explicit thread management or synchronization. It is well-known that this way of parallel programming is error-prone. Although it might appear that programmers are able to achieve better performance through a fine-grained control of parallelization, programmers are overwhelmed in large programs by the amount of performance data and by the details of thread interleaving or locking protocols. Complex parallel programs are difficult to tune because code changes may have unforeseen non-local effects on correctness and performance. In addition, predicting overheads for parallel computations is difficult, which forces many programmers to manually experiment with code changes and observe their performance impact.

Many parallel programs not only have parallelization potential on low abstraction levels (e.g., at instruction level), but also at an architectural level. Such an architecture typically contains several layers of nested parallelism [14]. For illustration, let's think in a top-down fashion of a data analysis application processing data packets in a pipeline with several stages. Inside a stage, input packets of a particular stage could be processed in a master-worker fashion. Workers, in turn, could be replicated if they are stateless, and they could work on a disjoint partition of a packet. This example shows that we can have different types of parallelism, such as pipeline parallelism, task parallelism, and data parallelism in one single application. The example also shows that there are many opportunities to configure the program architecture in different ways to influence performance: How many pipeline stages are needed on the highest abstraction level? How many workers should be created? Which load-balancing strategy works best? How many disjoint data partitions lead to best performance?

Earlier studies [14, 15, 17] have shown that parallel program architecture can have a major impact on performance, and that the software architecture adaptation to a target platform can have a higher performance leverage than pure instruction-level optimizations. These adaptations, however, are rarely done because they require invasive code changes. Programmers are also hesitant to modify complex parallel code being afraid of introducing bugs.

Motivated by these observations, our Tunable Architectures approach makes parallel application architecture easily configurable, and exploits architecture-inherent parallelism. The developer implements the program's work items (that is, *what* to parallelize), and specifies in our architecture description language how these items are processed in parallel. Thread management and synchronization are done implicitly based on the operators used to compose a parallel program out of work items. We describe next the language constructs for Tunable Architectures in greater detail.

3. THE TUNABLE ARCHITECTURE DESCRIPTION LANGUAGE

The *Tunable Architecture Description Language (TADL)* specifies all architecture variants for a parallel program that should be tested by an auto-tuner. We developed a formal grammar specification and an ANTLR parser implementation [2], however, we omit the full presentation of the grammar because of its length. We use examples instead to explain the available constructs, syntax, and semantics. We also show how the architecture constructs are related to program code. We start with the key language constructs used to specify *atomic components* and *connectors*.

3.1 Atomic Components

An *atomic component* represents a piece of elementary sequential work (i.e., the component has no internal parallelism). Developers need to think of pieces of work in terms of atomic components rather than threads. An atomic component is implemented by a method. Although we used C# in our case studies, our concepts are also realizable with other languages (e.g, Java).

The method code associated to an atomic component manipulates individual data items; this eases the exploitation of potential data parallelism. If methods do not share states, it is easy for a run-time system to replicate their functionality and create copies working in parallel. Our language provides a keyword to mark such atomic components as replicable.

An atomic component in the architecture description language has an identifying name that also establishes a relation to a C# method providing the implementation. The definition of an atomic component has the prefix `AC_` , followed by the respective method name, and an optional `replicable` attribute:

```
AC_MyMethodName [replicable]
```

The method associated to an atomic component can contain arbitrary code. However, atomic components are intended to run concurrently. Therefore, it is the developer's responsibility to write correct code. In particular, the developer has to take care of potential side-effects. Shared data structures used by two or more atomic components should be synchronized to ensure the expected behavior. Our prototype implementation currently does not check code correctness or the proper use of data structures that are not part of the Tunable Architecture.

When an application is written from scratch, developers need to create all necessary atomic components first. We describe next how to glue them together using patterns.

3.2 Connectors

A Tunable Architecture has a tree structure with nodes of *connectors* and leaves of atomic components. In contrast to the definition of connectors in architecture description languages [13], where a connector handles the communication between two components, a TADL connector defines an entire processing strategy as well as the corresponding interactions for its child items.

The TADL keyword of a connector encloses a block of child items that can be either atomic components or connectors. There are five connector types:

Sequential Composition. The *Sequential Composition* connector provides sequential processing semantics for components. The language construct encloses one or more child items to be executed one after the other. The sequential composition describes parts of the program that are not executed concurrently. Sequential Composition requires all child items to have compatible input types and output types.

Tunable Alternative. The *Tunable Alternative* connector expresses an exclusive choice between two or more enclosed child items; the auto-tuner may pick any of them for a particular program execution. All child items have individual means for input and output.

Tunable Fork-Join. The *Tunable Fork-Join* connector introduces task parallelism; the sequential control flow is forked to execute all enclosed child items in parallel, and joined back to sequential after all child items are finished. The child items are not supposed to interact with each other, so there are no restrictions on input types and output types. Every child item is assumed to have individual means for input and output.

Tunable Producer-Consumer. The *Tunable Producer-Consumer* connector has exactly two child items: the *producer* and the *consumer*. It is used for synchronization, and the developer does not have to care about details (e.g., buffer creation, synchronized access, signalling), because they are handled by our prototype implementation. This connector has streaming semantics: the producer is designed to accept data from an enumerable data source. After processing a data element, the producer passes it on to the consumer. The semantics require that the consumer's input type must match the producer's output type.

Tunable Pipeline. The *Tunable Pipeline* connector introduces pipeline parallelism. It has two or more child items representing chained stages.

The connector also has streaming semantics: the first stage accepts data from a source with enumerable data elements, and the last stage passes all data elements to a sink. A stage that finished its work on a particular data element passes it on to the next stage.

We have carefully selected the aforementioned connector types to cover widely used parallelization strategies and program structures. With the Sequential Composition and Tunable Alternative connector, TADL supports the description of component sequences and exclusive component choices, respectively. The connectors for parallelization (Tunable Fork-Join, Tunable Producer-Consumer, Tunable Pipeline) and the replicable atomic components express data, task, and pipeline parallelism.

All connectors have a similar syntax, which we exemplify for a Tunable Pipeline connector with two stages. The first stage consists of an alternative, which means that an auto-tuner can try out a program with a pipeline with `AC_MyMethod1` in the first stage and `AC_MyMethod3` in the second stage, or with `AC_MyMethod2` in the first stage and `AC_MyMethod3` in the second stage:

```
TunablePipeline MyTunablePipeline {
    [source:AC_Source;sink:AC_Sink]
    TunableAlternative {
        AC_MyMethod1,
        AC_MyMethod2
    },
    AC_MyMethod3
}
```

3.2.1 Handling Input and Output

Every atomic component can be associated to an input component and an output component. An input component provides input data (e.g., from a data source or the command line); an output component takes data to process (e.g., stores it on disk, prints the results on screen, or modifies a data structure). Similar to atomic components, input and output components have an associated method with code to handle input or output, respectively.

Input and output components are defined and matched recursively for child items of a connector. For Sequential Composition, Tunable Alternative, and Tunable Fork-Join connectors, input and output components need to be defined for every child item. By contrast, the Tunable Producer-Consumer and the Tunable Pipeline connector have only one source and one sink.

Input and output components are defined for child items that are enclosed by one of the aforementioned connectors, and the relation between input and output components and corresponding method code is similar to atomic components. In addition, all method names of input and output components are listed in an array; the first method in this array is supposed to handle the input of the first child item, the second method for the second child item, and so on. If child item doesn't need inputs, its corresponding array position contains the `null` keyword.

As an example, consider a Tunable Fork-Join connector with two child items (`AC_MyMethod1` and `AC_MyMethod2`) that are executed in parallel. `AC_MyMethod1` has a component for both input and output, and `AC_MyMethod2` has a component to produce output, but needs no input.

```
TunableForkJoin
   [input:AC_InputMethodFor1,null;
    output:AC_OutMethodFor1,AC_OutMethodFor2] {
    AC_MyMethod1,
    AC_MyMethod2
}
```

For Tunable Producer-Consumer and Tunable Pipelines, I/O routines are not specified for every stage, but only once for the entire pattern, as shown in the previous example with the Tunable Pipeline.

4. EXAMPLES FOR TUNABLE ARCHITECTURES

As a proof of concept, we fully implemented several parallel C# applications that are all running on multicore computers. We discuss in greater detail the Tunable Architecture of two of these applications and sketch other applications in Section 6.

4.1 Parallel Video Processing

In a bottom-up approach, we implement a parallel video processing application that applies a sequence of filters on each bitmap image of a video stream. The implementation starts with methods for atomic components and input and output; i.e., one method for every filter, one method for loading the original video, and one method for processing a manipulated image. Listing 1 outlines the method signatures.

```
public IEnumerable<Bitmap> LoadVideo() {...}
public Bitmap Crop(Bitmap bmp) {...}
public Bitmap OilPaint(Bitmap bmp) {...}
public Bitmap Resize(Bitmap bmp) {...}
public Bitmap Sharpen(Bitmap bmp) {...}
public void ConsumeVideo(Bitmap bmp) {...}
```

Listing 1: Atomic component methods of the video processing application.

The four filter methods (i.e., `Crop()`, `OilPaint()`, `Resize()`, and `Sharpen()`) represent the associated implementation for all our atomic components, whereas `GetVideo()` and `Consume()` represent the associated implementation of an input and and output component (i.e., a source and a sink for a pipeline).

The components are assembled to a parallel program using the Tunable Architecture description shown in Listing 2. To process the algorithms in a pipelined fashion, we introduce a Tunable Pipeline connector with source and sink components handling input and output. The stages reference the respective filter method implementations. As all filters are stateless (that is, the processing of a particular bitmap does not depend on any previously processed bitmaps), we mark the child items of the Tunable Pipeline connector with the `replicable` keyword to potentially exploit data parallelism.

```
TunablePipeline MyVideoProcessing
    [source:AC_LoadVideo;
    sink:AC_ConsumeVideo] {
    AC_Crop[replicable],
    AC_OilPaint[replicable],
    AC_Resize[replicable],
    AC_Sharpen[replicable]
}
```

Listing 2: Architecture description of the parallel video processing application.

This example exploits data parallelism and pipeline parallelism. We discuss the performance results in Section 6.

4.2 Parallel Desktop Search

We implement a parallel desktop search engine in C# that crawls files on a local hard disk, creates an inverted index data structure that relates all document words to files on disk, and which allows user input to query all files containing specified words. The implementation starts with the methods shown in Listing 3:

```
public List<string> Crawl() {...}
public SearchResult StringSearch1
    (string path) {...}
public SearchResult StringSearch2
    (string path) {...}
public void UpdateIndex(SearchResult r) {...}
public void CreateIndexFile(Index i) {...}
public List<string> Query(string[] keywords) {...}
public string[] GetKeywords() {...}
public void ShowResults(List<string> results) {...}
```

Listing 3: Atomic Component methods of the parallel video processing application.

The methods implement the following component functionality:

- `AC_Crawl`: Crawls the folders and retrieves all file paths.
- `AC_StringSearch1`: Splits a text into words using a delimiter character.
- `AC_StringSearch2`: Implementation of the Knuth Morris Pratt (KMP) string search algorithm [5]. While the first string search algorithm performs a naive search with moderate overhead, the KMP algorithm employs a smarter search, but with slightly more overhead (e.g., storing word indices). As we don't know in advance which of the algorithms performs better on certain platforms, we let the auto-tuner decide.
- `AC_UpdateIndex`: Updates the file index data structure in memory.
- `AC_CreateIndexFile`: Stores index on disk.
- `AC_Query`: Performs a query to find all files that contain all keywords specified by the user.
- `AC_GetKeywords`: Retrieves a list of keywords from command line.
- `AC_ShowResults`: Prints the results of a query to the command window.

The Tunable Architecture shown in Listing 4 illustrates how to define two architecture variants for this application (`DesktopSearch1` and `DesktopSearch2`):

```
TunableAlternative DesktopSearchAlternatives {
    SequentialComposition DesktopSearch1
        [input:null,AC_GetKeywords;
        output:null,AC_ShowResults] {
        TunablePipeline
            [source:AC_Crawl;
            sink:AC_CreateIndexFile] {
            TunableAlternative {
                AC_StringSearch1[replicable],
                AC_StringSearch2[replicable]
            },
            AC_UpdateIndex[replicable]
        },
        AC_Query
    },
```

```
    SequentialComposition DesktopSearch2
        [input: null, AC_GetKeywords;
        output: null, AC_ShowResults] {
        TunableProducerConsumer
            [source: AC_Crawl;
            sink: AC_CreateIndexFile] {
            TunableAlternative {
                AC_StringSearch1[replicable],
                AC_StringSearch2[replicable]
            },
            AC_UpdateIndex[replicable]
        },
        AC_Query
    }
}
```

Listing 4: Architecture description of the parallel desktop search application.

To realize a parallel indexing process consisting of a string search algorithm and the index update method, we can either use a Tunable Pipeline or a Tunable Producer-Consumer connector. However, we do not know which strategy performs better on a particular hardware platform. According to the parallel pattern definitions in [12], the pipeline processes the data elements one by one (that is, each stage processes one item and then gets the next), whereas the producer-consumer strategy allows the consumer to fetch more than one item at a time. Consequently, a pipeline stage has less waiting time after processing a data element; however, the consumer can process a batch of data elements without synchronizing after each element.

Using a Tunable Alternative connector, we define two architectural variants and let the auto-tuner decide which one performs better. In both alternatives, we start with a Sequential Composition connector to ensure the indexing process is finished before the program accepts search queries.

The indexing process in the first alternative employs a Tunable Pipeline connector calling the `AC_Crawl` source to obtain files to be parsed, and the `AC_CreateIndexFile` sink to store processed words. There are two stages, one for searching and one for index updating with `AC_UpdateIndex`). The searching stage can have one of the two specified string search algorithms (i.e., `AC_StringSearch1` and `AC_StringSearch2`). We add `AC_Query` as the second child item to the Sequential Composition connector.

The second architecture alternative is similar to the first one, except that we define a Tunable Producer-Consumer connector. We use the same source and sink methods (i.e., `AC_GetKeywords` and `AC_ShowResults`).

This example illustrates data parallelism, pipeline parallelism, and the definition of performance-relevant architecture variants. The performance results are presented in Section 6. We remark that other architecture variants can be defined to do parallel indexing at different granularity levels (e.g., files, or chunks of words). This is important for parallel programs as a varying granularity of processing influences the program overheads and the performance on different machines.

Apart from the architecture declared above, we can think of other scenarios that provide even more flexibility when using the application. For example, we can use a Tunable Fork-Join connector instead of a Sequential Composition connector to allow queries to be performed while indexing is in progress. This can be useful if indexing takes a long time, but the user wants to perform searches as soon as possible on preliminary versions of the index. However, if querying and indexing run concurrently, additional synchronization is required that could lead to a slowdown of the indexing process. To instruct the auto-tuner to find the best variant, we would replace one of the Sequential Composition connectors with a Tunable Fork-Join connector. If the index data structure is synchronized, we could even use the same method implementation for `AC_Query`.

5. IMPLEMENTATION TECHNIQUES

The implementation of a parallel program with Tunable Architectures needs library and tool support. We present next the *Tunable Architecture Library*, the *TADL compiler*, and the *Automatic Architecture Tuner*. We also illustrate their usage in the software development process.

5.1 The Tunable Architecture Library

The *Tunable Architecture Library (TALib)* contains modules with implementations for every TADL connector; for example, the TALib contains the code to implement the Tunable Pipeline connector. In addition, each such module implements a set of predefined tuning parameters that influence performance. Table 1 shows all tuning parameters. The values of these parameters are chosen by an auto-tuner out of an associated set of values.

The TALib modules also handle exceptions that might have been thrown by atomic components. This means that a parallel pattern handles the exceptions of its enclosed child items. Each TALib module collects all exceptions thrown by its child items and re-throws them as one aggregated exception.

5.2 The TADL Compiler

The *TADL compiler* processes scripts written in the Tunable Architecture Description Language to generate source code. Compilation consists of two steps:

1. A preprocessor extracts all atomic component declarations from the TADL script. It uses reflection to establish the bindings between methods and their atomic component declarations.

2. The compiler analyzes the architecture description, translates it to an internal representation, checks type consistency for connectors, generates the source code, and integrates the code into a final executable program.

5.2.1 Tuning Wrappers

The TADL compiler generates code organized in *tuning wrappers*. A tuning wrapper is implemented as a class that initializes a particular TALib module and implements access methods to that module. The TADL compiler generates a tuning wrapper for each connector in the TADL script. For example, the compiler translates a construct defining a Tunable Pipeline connector into a tuning wrapper that handles the access to the TALib's Tunable Pipeline module.

A tuning wrapper also contains code to create an instance of the corresponding TALib module (e.g., an instance of a Tunable Pipeline connector). The wrapper has fields for the module's tuning parameters (e.g., a fork-join's number of worker threads) and methods to connect child items of a connector to its TALib module implementation. The wrapper also implements the methods for input and output data handling, and exposes a `Run()` method executing the wrapped

Connector	*TuningParameters*
Tunable Alternative	`Choice of Alternative` (defines which alternative is executed)
Tunable Fork/Join	`Num Worker Threads` (number of threads the component can use)
Tunable Producer/Consumer	`Buffer Size` (size of the central buffer between producer and consumer) `Batch Size` (number of data elements the consumer grabs at once)
Tunable Pipeline	`Stage Fusion` (for each supporting stage: enables or disables stage fusion)
Replication	`Num Instances` (number of replicated instances of the Atomic Component) `Load Balancing` (defines how the data elements are assigned to the instances) `Batch Size` (number of data elements an instance grabs at once)

Table 1: Predefined tuning parameters of TADL connectors.

TALib module (e.g., the Tunable Pipeline). The entry point for the execution of the Tunable Architecture is the `Run()` method of the tuning wrapper that implements the architecture's root component.

5.2.2 Tuning Instructions

Tuning wrappers just provide interfaces to the variables to tune. The auto-tuner, however, needs to be able to change variable values and get performance feedback. To achieve this we instrument the source code with *Atune-IL*, which is a `pragma`-based tuning language. We refer to our previously published work for details [18].

Atune-IL provides constructs to declare program variables as tuning parameters and to define measuring points within the program to return execution time feedback to the auto-tuner. In addition, Atune-IL offers block constructs to define the scope of tuning parameters; this is important for search space reduction.

Atune-IL provides a compact representation of several program variants. Before the auto-tuner starts, the Atune-IL statements are replaced by appropriate code fragments, such as variable assignments and calls to performance libraries. In a tuning wrapper, the TADL compiler instruments the variables declaring tuning parameters, sets measuring points at predefined positions to determine the wrapper's execution time, and encloses the wrapper class with a block statement.

5.3 The Automatic Architecture Tuner

The *Automatic Architecture Tuner* (auto-tuner) performs an automatic search-based optimization and uses Atune-IL instrumentations to steer the process. We adapted our auto-tuner from previously published work [17, 18] to support Tunable Architectures. Auto-tuning is a cyclical, feedback-directed process:

1. The tuner extracts the Atune-IL instrumentations from all tuning wrappers in the program and builds a data structure containing the required tuning information. In addition, the tuner builds up a tuple-based, multi-dimensional search space based on all parameters' value ranges. Each tuple represent a particular parameter configuration.

2. Starting from the current parameter configuration, we use empirical search algorithms to traverse the search space and find a new parameter configuration. The selection of parameter configurations is based on exchangeable search algorithms. Depending on the complexity of the search space, we employ adapted versions of hillclimbing, swarm optimization, and random sampling.

3. A new executable program is generated in which instrumentations are removed. Atune-IL placeholders are replaced by concrete values. Time measuring points are replaced by calls to an instrumentation run-time.

4. The tuner executes the new program variant and monitors it. The instrumentation run-time records, aggregates, and stores data from all measuring points.

5. The recorded monitoring results are gathered and prepared for further processing.

The tuning cycle (steps 2-5) is repeated until some predefined termination condition holds; this depends on the chosen search algorithm.

5.3.1 Search Space Reduction

One of the main problems of search-based auto-tuning is the explosion of the search space. In the worst case, an auto-tuner might try out the Cartesian product of all parameter domains, which grows exponentially in the number of parameters. If the search space is very large, even a smart search algorithm would require a long time to find a good parameter configuration.

To tackle this problem, the automatic architecture tuner takes advantage of the semantics of the TADL connectors. For search space partitioning, it uses the knowledge about Sequential Composition and Tunable Alternative connectors. Both ensure that their child items will never run concurrently. Thus, tuning parameters exposed in a particular child item or in its sub-tree will never interfere with parameters of other child items. From a tuning perspective, the sub trees of Sequential Composition and Tunable Architecture are independent. We call independent sub trees *tuning entities*, as the auto-tuner can optimize these sub trees separately.

Considering the tree structure of Tunable Architectures, the search space can be split into smaller parts. Figure 1 conceptually illustrates such a situation. If an architecture tree contains a Sequential Composition connector in a particular place, we can assign each of its sub-trees to a separate tuning entity. We thus obtain two significantly smaller parts of the search space. Instead of tuning the parameters of all components below the Sequential Composition connector together, the auto-tuner can optimize each tuning entity separately. This applies in a similar way for the Tunable Alternative connector.

For each of the remaining connectors, the auto-tuner applies a particular tuning heuristic that defines an optimization process for a connector. Using tuning heuristics, the

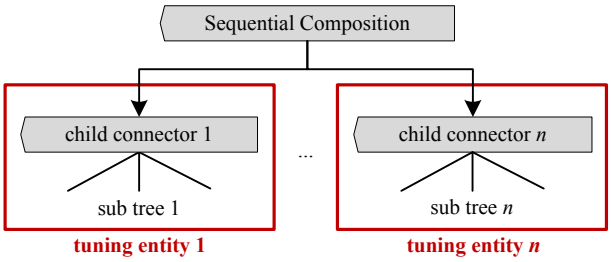


Figure 1: Exploiting tree structure of Tunable Architectures to create tuning entities.

auto-tuner exploits context knowledge about the connectors' parallelization strategies and performs a guided search. For example, let's think of a Tunable Pipeline connector with replicable child items. Instead of simply testing a subset of all parameter configurations, the auto-tuner tries to balance pipeline stages by increasing the number of threads for replicable child items with long executions times or by decreasing the number of threads for short-running child items. We refer to our earlier work for details [17].

5.4 How To Create An Auto-Tuned Parallel Application

The implementation process using Tunable Architecture is illustrated in Figure 2, along with the sequence of steps a developer has to do.

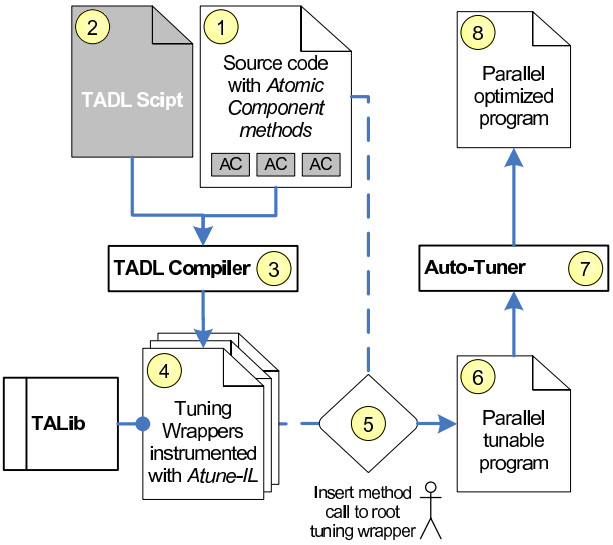


Figure 2: Entire process of creating a parallel auto-tuned application.

1. A developer programs the methods associated to atomic components (`AC`).
2. A Tunable Architecture is defined in a TADL script that specifies the composition of atomic components to a complete program.
3. The TADL compiler is invoked with 1 and 2 as inputs.
4. The TADL compiler produces code files containing tuning wrappers; the wrappers are instrumented with Atune-IL statements. Each wrapper handles the access to the corresponding TALib module.
5. The developer inserts in his or her program a call to the tuning wrapper that implements the architecture's root component. This call is typically inserted in the program's main method.
6. The completion of step 5 produces an executable, but not yet optimized parallel program. This program version represents an intermediate portable tuning template.
7. The automatic architecture tuner optimizes the parallel program for performance. It hooks on to the Atune-IL instrumentations that are interfaced in the tuning wrappers.
8. The completion of the previous step produces a parallel program optimized on a target platform. If the program is migrated to another platform, the developer repeats the process starting at step 6.

6. CASE STUDIES

To evaluate Tunable Architectures we conducted four case studies with parallel applications written in C#, one of which is a re-engineered industrial application. We discuss the context and experimental results next.

6.1 Parallel Applications

6.1.1 *Video Processing*

This application applies four different filters on each frame of an AVI (*Audio Video Interleave* format) video. It uses the Tunable Architecture described earlier by the TADL script in Listing 2. The TADL compiler generates the standard tuning parameters for the pipeline as well as for each of the replicable atomic components. In the script, just six lines define the entire architecture. The TADL compiler produces a configurable parallel program that is ready to execute.

As a performance benchmark, we used an AVI video consisting of 180 frames with a resolution of 800x600 pixels.

6.1.2 *Desktop Search*

This application indexes the words contained in a set of text files and allows queries on the index to return all files that contain all words in given list. The indexing performs most work and is the most relevant part for parallelization. The desktop search engine's Tunable Architecture is the one shown earlier in Listing 4.2; it also illustrates the use of alternatives. We defined two Tunable Alternative connectors to instruct the auto-tuner to test different variants of the architecture as well as of particular algorithms. This application context demonstrates the exchangeability of the connectors, and the definition of variants on each architecture level.

The performance benchmark for this case study consisted of 10,700 ASCII text files with file sizes ranging between 9KB and 613 KB.

6.1.3 *Biological Data Analysis*

Agilent's *MetaboliteID (MID)* [1], is a large commercial application (with more than 100.000 lines of code) for biological data analysis; we parallelized this sequential application using Tunable Architectures. MetaboliteID identifies metabolites in mass spectrograms, which is a key method for drug testing. Metabolism is the set of chemical reactions

taking place within cells of a living organism, and MetaboliteID compares mass spectrograms to identify the metabolites caused by a particular drug. The MetaboliteID application executes several algorithms in sequence (abbreviated *SC*, *EIC*, *ADC*, *UV*, *BTL*, *IPM*, *PCS*, *FMSC*, *FPM*, and *MFA*) that identify and extract the metabolite candidates.

Compared to the previous case studies, MID already existed in a sequential version. We show in this case study that Tunable Architectures can be used to re-engineer sequential programs to parallel programs, and tune the performance of the parallel program more easily.

Listing 5 illustrates the Tunable Architecture for MetaboliteID. We declared the methods implementing the aforementioned algorithms as atomic components. We then defined the parallel processing architecture that includes three Tunable Fork-Join connectors, two of which are nested, and the innermost fork-join executes two replicable atomic components in parallel. We had access to documentation and communicated with MetaboliteID's developers to ensure that the new order of algorithms still produces valid results.

In the compact description of our program architecture, we defined five parallel sections and exploited parallelism on three different application layers. Each of the tuning wrappers generated by the TADL compiler for the connectors were already instrumented with predefined tuning parameters; they were used by an auto-tuner to reduce the search space based on additional information about the nesting structure of the program (see Section 5.3.1) For example, the two Tunable Fork-Join connectors within the outer Sequential Composition connector are tuned separately because they don't influence each other.

```
SequentialComposition MID {
    AC_RunPreProcessing,
    TunableForkJoin {
        AC_RunSC,
        AC_RunEIC
    },
    TunableForkJoin {
        AC_RunADC,
        AC_RunUV,
        AC_RunMDF,
        AC_RunBTL_IPM,
        SequentialComposition {
            AC_RunPCS_FMSC,
            TunableForkJoin
                [input: AC_FPMInput, AC_MFAInput;
                output: AC_FPMOutput, AC_MFAOutput] {
                AC_RunFPM [replicable],
                AC_RunMFA [replicable]
            }
        }
    },
    AC_RunPostProcessing
}
```

Listing 5: Architecture description of the parallel version of MID.

A manual implementation of a parallel program with the same functionality as ours would have required more effort, in particular because nested parallel components require careful attention to thread management, synchronization, and locking protocols.

The input files for the performance benchmark were 1 GB in size and contained the data representing the mass spectrograms to compare.

6.1.4 Graph Rewriting

GrGen is the currently fastest sequential graph rewriting system [9]. In this case study, we simulated the biological gene expression process on the E.coli bacteria DNA [19]. The model of the DNA is represented as an input graph with more than 9 million nodes. This is the second study to re-engineer a sequential application for parallelism using Tunable Architectures.

GrGen has two performance bottlenecks that both provide potential for massive data parallelism. We declared the respective methods (`PromoterSeacrch` and `RnaPoly`) as replicable atomic components in our Tunable Architecture. Listing 6 shows the parallel GrGen's TADL script. The two atomic components must be executed one after another, so they are enclosed by a Sequential Composition connector. We also declared methods to handle input and output.

```
SequentialComposition GrGen
    [input: AC_PromoterInput, AC_RnaPolyInput;
    output: AC_PromoterOutput, AC_RnaPolyOutput] {
    AC_PromoterSearch [replicable],
    AC_RnaPoly [replicable]
}
```

Listing 6: Architecture description of the parallel version of GrGen.

The tunable parallel version of GrGen could be defined with just a few architecture script lines. The TADL compiler automatically generated tuning wrappers for the replicable atomic components; the compiler also generated predefined tuning parameters for the patterns as well as code instrumentations for feedback to the auto-tuner.

6.2 Experimental Results

We present performance results for each case study application that were obtained by our automatic architecture tuner. We then explain how programming effort was reduced and how Tunable Architectures helped reduce the potential of parallel programming errors.

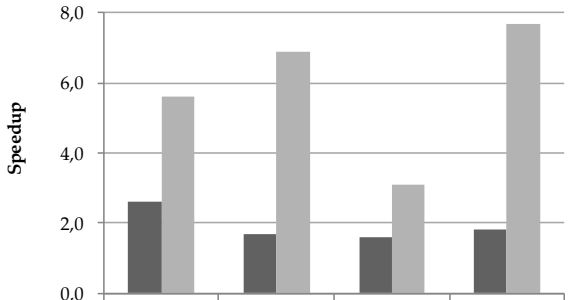
6.2.1 Performance

For performance evaluation, we are interested in two performance metrics:

- **Best obtained speedup.** This is the ratio between the execution time of the program's *optimized* parallel version and the execution time of the sequential version. It expresses how much faster the parallel program is compared to the sequential one.
- **Tuning Performance Gain (TPG).** This is the difference between the worst and the best speedup obtained by our auto-tuner. This metric indicates the impact of tuning the parallel program, that is, how much more the performance of the parallel program could be improved by tunable architectures and auto-tuning.

All experiments were done on an eight core machine, with two Intel Xeon E5320 QuadCore CPUs, clocked at 1.86 GHz, with 8 GB RAM, and running Windows Server 2003 32bit. Figure 3 shows the performance results for the worst and the best speedup (*Best Spd.* and *Worst Spd.*, resp.) and

summarizes the resulting TPG. The execution times of the sequential versions were 19 seconds for the video application, 14 hours and 35 minutes for the desktop search application, 85 seconds for MetaboliteID, and 45 seconds for GrGen.



	Video	DS	MID	GrGen
Worst Spd.	2,6	1,7	1,6	1,8
Best Spd.	5,6	6,9	3,1	7,7
TPG	215%	406%	194%	428%

Figure 3: Performance results of all case studies.

The results show that significant speedups could be indeed be achieved by architecture-level performance tuning, so it pays off for performance to use Tunable Architectures. For the desktop search application, the auto-tuner has finally chosen the best architecture based on the Tunable Producer-Consumer connector and the KMP algorithm. The tuning performance gain results show that auto-tuning is worthwhile. Based on our experience, a manual performance optimization leads to results somewhere within the range of the TPG, but rarely close to the maximum; this is because it is difficult to figure out intuitively what the best parameter configuration is.

6.2.2 *Programming Effort*

We compare the approach with Tunable Architectures with a manual approach without Tunable Architectures implementing the same functionality. We compare the programming effort and analyze the potential for parallel programming errors. In particular, we look at the following metrics:

- **Lines of code (LOC).** We use the LOC metric as a measure correlated to the implementation effort required to manually re-build the functionality we provide in our language, compiler, and library.
- **Number of synchronization primitives (syncs).** Thread synchronization is one of the most difficult tasks in parallel programming. We interpret the number of saved synchronization primitives as an indication for reduced error potential.
- **Number of tuning instrumentations.** Instrumentations are necessary for automatic performance tuning. The saved instrumentation statements in our approach indicate how much easier tuning becomes with Tunable Architectures.

To estimate the average number of LOC and synchronization primitives required for the manual implementation, we assume average values based on our experience from previous experiments: the implementation of a Tunable Pipeline connector requires 180 LOC / 10 syncs, a replicable component 120 LOC / 8 syncs, a Tunable Fork-Join connector 170 LOC / 10 syncs, a Tunable Producer-Consumer connector 150 LOC / 9 syncs, and a Tunable Alternative connector 15 LOC. We do not consider Sequential Composition connectors, as they contain neither parallelism nor tuning options.

Figure 4 lists the particular numbers for each of our case study applications.

		Video Processing	Desktop Search	MID	GrGen
LOC	Man. Impl.	300	495	290	120
	TA-based	3	3	3	3
	Reduction	**297** (99%)	**492** (99%)	**287** (99%)	**117** (98%)
Synchronizations[1)]	Man. Impl.	18	27	18	8
	TA-based	0	1	2	0
	Reduction	**18** (100%)	**26** (96%)	**16** (89%)	**8** (100%)
Tuning Instrumentation Statements	Man. Impl.	28	43	51	34
	TA-based	0	0	0	0
	Reduction	**28** (100%)	**43** (100%)	**51** (100%)	**34** (100%)

[1)] includes all synchronization primitives, such as *lock, notify, wait, join,* etc.

Figure 4: Reduction of LOC, synchronization, and instrumentation.

The comparison shows that our approach saves a large number of LOC; this helps reducing the overall application development effort. In addition, almost no manual thread synchronization and tuning instrumentations are necessary with Tunable Architectures; this reduces the potential for parallel programming errors.

Tunable Architectures do not overly inflate program code and introduce modest complexity in exchange for automation. This is supported by the class coupling metric [25] and the total LOC of the final application. With Tunable Architectures, the average number of class couplings for all our case study applications increases by 18%, the number of classes by 13%, and the total LOC by just 3%, compared to the original program version.

7. RELATED WORK

Architecture description languages (ADLs) have been explored for sequential programs [23, 3], and the recommendations of [22, 21] are also useful in our context. The ADLs discussed in [13], such as *C2* or *Darwin*, support the description of distributed systems, but do not offer means to describe adaptable parallelism within shared-memory programs.

Most related work addresses parallel programming for distributed-memory environments (e.g., the Web) [4] and pays little attention to parallel programming patterns or performance tuning for shared-memory computers; recently available multicore computers are different because they have a shared-memory architecture and a different programming model. This is similar for work in autonomous computing [6, 10], architecture adaptation [7, 8], self-management, and self-organization [28, 20], and self-optimization [6].

In the parallel programming domain, generative approaches such as CO^2P^3S [11, 26] are used to create program skeletons for distributed memory programs, which require man-

ual extensions by developers. There is no comparable support for auto-tuning on an architectural level.

8. CONCLUSION

As multicore computers have arrived on every desktop, many software engineers now have to switch from sequential application development to the more complex parallel application development. The design and performance optimization of parallel programs is more difficult than for sequential programs. Even experts have problems predicting the performance of complex parallel programs; as a result, experimentation is unavoidable in practice.

The tunable architectures approach proposed in this paper relieves developers from the burden of manual program adaptation and tuning. It provides a systematic approach that helps avoid common design and implementation pitfalls. Tunable architectures also simplify the exploitation of parallelism on an architecture level, rather than just on an instruction-level, which is a key leverage for performance. In addition, the quality of parallel code is improved: hard-coded optimizations are replaced by generic constructs, thus making multicore applications easier to port and re-tune on new platforms. Multicore processors are definitely here to stay, and we need approaches like this one to make multicore software development more accessible, for experts as well as for less experienced parallel programmers.

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 at the Karlsruhe Institute of Technology.

9. REFERENCES

[1] Agilent Technologies. *MassHunter MetaboliteID Software*, 2008. `http://www.chem.agilent.com`.

[2] antlr.org. *ANTLR Parser Generator*, 2009. `http://www.antlr.org`.

[3] P. Clements et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[4] P. C. Clements. A Survey of Architecture Description Languages. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[5] T. H. Cormen et al. *Introduction to Algorithms*. MIT Press, 2001.

[6] A. G. Ganek and T. A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1):5–18, 2003.

[7] D. Garlan et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, Oct. 2004.

[8] D. Garlan et al. *Software Architecture-Based Self-Adaptation*. Number ISBN 978-0-387-89827-8. Springer, 2009.

[9] R. Geißand J. Blomer. *GrGen.NET*. University of Karlsruhe, IPD Prof. Goos, 2008. `http://www.info.uni-karlsruhe.de/software/grgen/`.

[10] M. C. Huebscher and J. A. McCann. A Survey of Autonomic Computing–Degrees, Models, and Applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.

[11] S. MacDonald et al. Generative Design Patterns. In *Proceedings of the 17th International Conference on Automated Software Engineering*, pages 23–34, 2002.

[12] T. G. Mattson et al. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[13] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[14] V. Pankratius et al. Software Engineering For Multicore Systems: An Experience Report. In *Proceedings of the 1st ICSE Workshop on Multicore Software Engineering*, pages 53–60, New York, NY, USA, 2008. ACM.

[15] V. Pankratius et al. Parallelizing BZip2. A Case Study in Multicore Software Engineering. *IEEE Software*, 26(6):70–77, 2009.

[16] 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*, 36(2):183–196, 2006.

[17] C. A. Schaefer. Reducing Search Space of Auto-Tuners Using Parallel Patterns. In *Proceedings of the 2nd ICSE Workshop on Multicore Software Engineering*, pages 17–24, Washington, DC, USA, 2009. IEEE Computer Society.

[18] C. A. Schaefer et al. Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume 5704/2009 of *LNCS*, pages 9–20. Springer Berlin / Heidelberg, Jan. 2009.

[19] J. Schimmel et al. Gene Expression with General Purpose Graph Rewriting Systems. In *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques*, 2009.

[20] T. Seceleanu and D. Garlan. Developing Adaptive Systems with Synchronized Architectures. *Journal of Systems and Software*, 79(11):1514 – 1526, 2006. Software Cybernetics.

[21] M. Shaw and P. Clements. How Should Patterns Influence Architecture Description Languages? In *Working paper for DARPA EDCS community*, 1996.

[22] M. Shaw and D. Garlan. Characteristics of higher-level languages for software architectures. Technical report, Technical Report, CMU-CS-94-210, Carnegie Mellon University, Department of Computer Science, 1994.

[23] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[24] S. Siu et al. Design Patterns for Parallel Programming. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, page 230244, 1996.

[25] W. Stevens et al. Structured Design. *Classics in Software Engineering*, pages 205–232, 1979.

[26] K. Tan et al. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 38, pages 203–215, New York, NY, USA, 2003. ACM Press.

[27] C. Tapus et al. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the ACM/IEEE Supercomputing Conference*, Nov. 2002.

[28] M. Wermelinger. Towards a Chemical Model for Software Architecture Reconfiguration. *IEEE Proceedings Software*, 145(5):130–136, Oct 1998.

[29] O. Werner-Kytola and W. F. Tichy. Self-Tuning Parallelism. In *Proceedings of the 8th International Conference on High-Performance Computing and Networking*, pages 300–312, London, UK, 2000. Springer-Verlag.

[30] R. C. Whaley et al. Automated Empirical Optimizations of Software and the ATLAS Project. *Journal of Parallel Computing*, 27:3–35, Jan. 2001.