Patty: A Pattern-based Parallelization Tool for the Multicore Age

Korbinian Molitorisz Tobias Müller Walter F. Tichy

Institute for Programming Structures and Data Organization
Karlsruhe Institute of Technology (KIT)
molitorisz@kit.edu, tobias.mueller@student.kit.edu, walter.tichy@kit.edu

Abstract

The free lunch of ever increasing clock frequencies is over. Performance-critical sequential software must be parallelized, and this is tedious, hard, buggy, knowledge-intensive, and time-consuming. In order to assist software engineers appropriately, parallelization tools need to consider detection, transformation, correctness, and performance all together.

This paper introduces a pattern-based process model that assists in all four parallelization tasks and hence facilitates transforming legacy software that had not been developed with multicore in mind. Our approach uses optimistic parallelization and generates a semantic model with static and dynamic information. With this information we detect parallelizable regions and runtime-relevant tuning parameters. The regions are then transformed to tunable parallel patterns. The process model covers the detection of parallelizable regions, the identification of appropriate parallelization strategies, and enhances traditional parallelization processes with correctness and performance validations. We implemented the pattern-based process model in Patty, a tool that actively assists engineers in the tedious and error-prone software parallelization tasks.

This paper also contains a user study that compares the effectivity of optimistic pattern-based parallelization as implemented in Patty to 1) a popular commercial parallelization tool and 2) pure manual parallelization. We demonstrate that our approach receives the best average scores from its users while delivering the best results within the least amount of time. In our user study Patty outperforms both control groups in subjective and objective measurements. Patty achieves parallel performance comparable to a skilled parallel software engineer within minutes rather than days of work. This makes our approach attractive for experts and inexperienced software engineers alike.

1. Why we need integrated parallelization

Modern multicore processors require parallel source code, but a large amount of performance-critical software is still sequential. From studies like [1] we know that the development overhead for parallel software is up to 2.4 times higher than for serial software. Even worse, the increased complexity has to be considered as lower bound for the effective overhead: The referenced study only accounts for the complexity that comes from using parallel libraries and language constructs, other vital aspects like identifying parallel potential, tuning parallel performance and parallel correctness are not covered. So, developing parallel software is hard, and we need to provide adequate means to assist engineers during these tasks.

Articles like [2] clearly point out the urgent need for a better tool support to avoid even the next software crisis, and several research activities like [3–8] and commercial tools like [9, 10] recently addressed this issue. We show that all current works exhibit certain deficits making them rather special solutions for specific problems, only applicable by software engineers familiar with the pitfalls of multicore software engineering, or impractical for general purpose applications. This paper has the following three main contributions:

1) It introduces a general process model for transforming sequential to parallel software. For widespread applicability our process model is geared to object-oriented code. It relies on the detection of commonly known parallel patterns via predefined *source patterns*. For each source pattern we insert an equivalent parallel pattern that contains defined *parameters with runtime impact*. We support a better program understanding by explicitly annotating each parallel pattern in the source code. These annotations are then transformed into parallel source code by instantiating a parallel runtime library. For performance aspects we generate a file with all tuning parameters and values. This file is processed by an auto tuning algorithm to determine the ideal parameter value configuration for a given target multicore platform. For correctness aspects we generate parallel unit tests and adapt a dynamic race detector to evoke exhaustive thread interleavings. As a result, our process actively assists all parallelization steps and generates parallel software that is automatically tunable and validatable.

2) This paper introduces Patty, a tool that implements pattern-based parallelization and integrates directly into a development environment. It uses static and dynamic analyses to generate a semantic model from sequential software and to derive the parallelization candidates and tuning parameters. Patty can either process source code or code annotations for parallelization, or it can process parallel source code for performance and correctness validation. Several other tools aim at assisting engineers during parallelization, but practically none of them achieved widespread applicability. In our view, this is due to three reasons:

- Lack of tool integration: Parallelization support is often not integrated into an IDE, so parallelization and software engineering tasks stay separate and too far away from each other.
- Lack of comprehensible input and visualization: Apart from timing constraints, engineers need assistance in parallelization because they have too little knowledge about multicore software

PMAM '15, February 7–11, 2015, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2712386.2712392

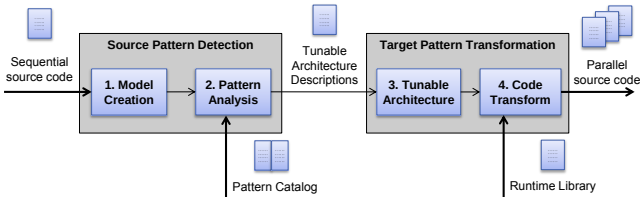

Figure 1. Process Model for Pattern-based Parallelization

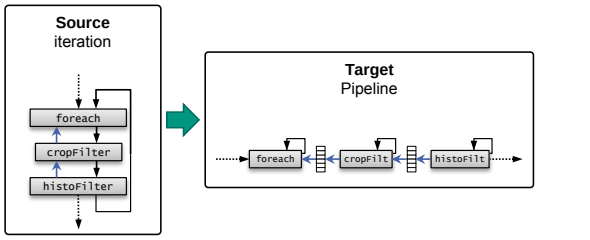

Figure 2. Source and Target Pattern for Pipelines

engineering in general or about the software to be parallelized. Black box approaches are transparent and hide their inner functioning, whereas it is vital to know what a process does to build up confidence and lead to tool acceptance.

- Lack of universality: Today, tools are either built for specific use cases that are by design rather rare in practice, or generalized in a way that they come down to pure runtime profilers. We see the necessity to deliver a holistic approach that covers all tasks in parallelization.

3) This paper presents a user study that researched the potential of tool-assisted parallelization. In theory, the community did a lot to reveal the stumbling blocks of multicore software engineering [11–13]. Recent works [14, 15] compare manual parallelization to implicit parallelization using parallel libraries, but to the best of our knowledge there is currently no user study that reveals actual needs and problems that engineers experience when parallelizing. We therefore conducted an empirical user study with engineers of different skill levels in software and multicore engineering and monitored their performance and actions.

This paper is structured as follows: In section 2 we present the pattern-based parallelization process. It detects parallelizable regions and maps them onto parallel target patterns. Apart from transforming software, it identifies tuning information and creates a tuning configuration and parallel unit tests. With pipeline, we present one common parallel pattern for data stream processing. Section 3 presents Patty, a tool that integrates with Microsoft Visual Studio as IDE and bridges the gap between software engineering and parallelization activities. In section 4 we introduce a user study with software engineers of different skill levels. This study clearly shows the pitfalls of manual parallelization and the benefits of tool-assisted parallelization. Comparing Patty to a commercial parallelization tool, the study shows that our approach returns better results with higher user satisfaction in less time.

2. Pattern-based Parallelization

Parallelization assistance needs to cut down complexity and remain comprehensible. According to the psychological studies in [16] recurring patterns that are clearly defined help people to understand complex matters. Applying this principle to multicore software engineering, T. Mattson introduces a process model and classification schema for parallel programming in [17, 18]. Several other works like [19] exist that also cover parallel design patterns to assist in parallel programming. All these works focus on giving hints on how to parallelize a given location. The questions how to reveal this location, how to optimize parallel performance, and how to assure parallel correctness is mostly not seen as an integral part of parallelization.

In subsection 2.1, we define the pattern-based parallelization process for software transformation depicted in figure 1. It focuses on the four relevant questions where to parallelize, how to parallelize, whether the result is correct, and under what conditions it executes faster on the target platform. Our current implementation contains the three parallel patterns master/worker, data-parallel loops and pipeline. Subsection 2.2 deals with the latter of which and presents source patterns, target patterns, and tuning parameters.

2.1 Process Model for Pattern-based Parallelization

As the diagram in 1 shows, our process model handles regular source code and gradually transforms it to parallel source code. Additionally to the creation of parallel source code, our process reveals runtime-relevant *tuning parameters* and correctness-related *parallel unit tests*.

This process is ongoing work. Early results have previously been published in [20–22]. At the moment of writing, the process model defines the following two phases:

- **Source Pattern Detection**: At first, we create a *semantic model* from the input source code. Therefore, we build the cross product from the control flow graph, the data dependencies, the call graph, and runtime information. It has to be mentioned that our process is geared to reveal a high amount of parallel potential, so we use optimistic parallelization analyses. As a side effect of optimistic parallelization, we are unable to give correctness guarantees for the resulting parallel version. Because of this, our process needs to take care of parallel errors. We will deal with this aspect later in this section.
  In a second step, we iterate over the semantic model in order to find source pattern instances. This step is based on a catalog of predefined pairs of sequential source and parallel target patterns. For each target pattern we specified equivalent sequential representations and formalized the mappings as pairs of source and target patterns. An example for the sequential version of a parallel pipeline is shown in figure 3 a). We will evolve this example during the course of thie paper.
  The semantic model contains runtime information. This enables us to reveal runtime-relevant parameters, so-called *tuning parameters*. Changing their values has implications on the runtime behavior of a parallel application, but not on its correct semantics. With tuning parameters, we extend traditional parallel patterns to so-called *tunable parallel patterns*. In order to be able to express tunable parallel patterns and generate an architecture description with tuning parameters we adapted the Tunable Architecture Description Language TADL [23] to our needs. Using TADL as interface, we draw a sharp boundary between the distinct tasks *detection* and *transformation*. This enables us to change, extend, or alter both steps independently. Figure 3 b) shows the resulting TADL annotation for the pipeline example.
- **Target Pattern Transformation**: We implemented TADL as a code annotation using preprocessor directives. This way we can add semantic information to plain source code that is visible to compilers capable of processing TADL, and that is transparent to incapable ones. Hence, the third step needs to process source code and identify TADL annotations and the architecture description. We insert the code annotations at the exact location where they have been found during pattern detection for the reason of program comprehensibility: We familiarize in-

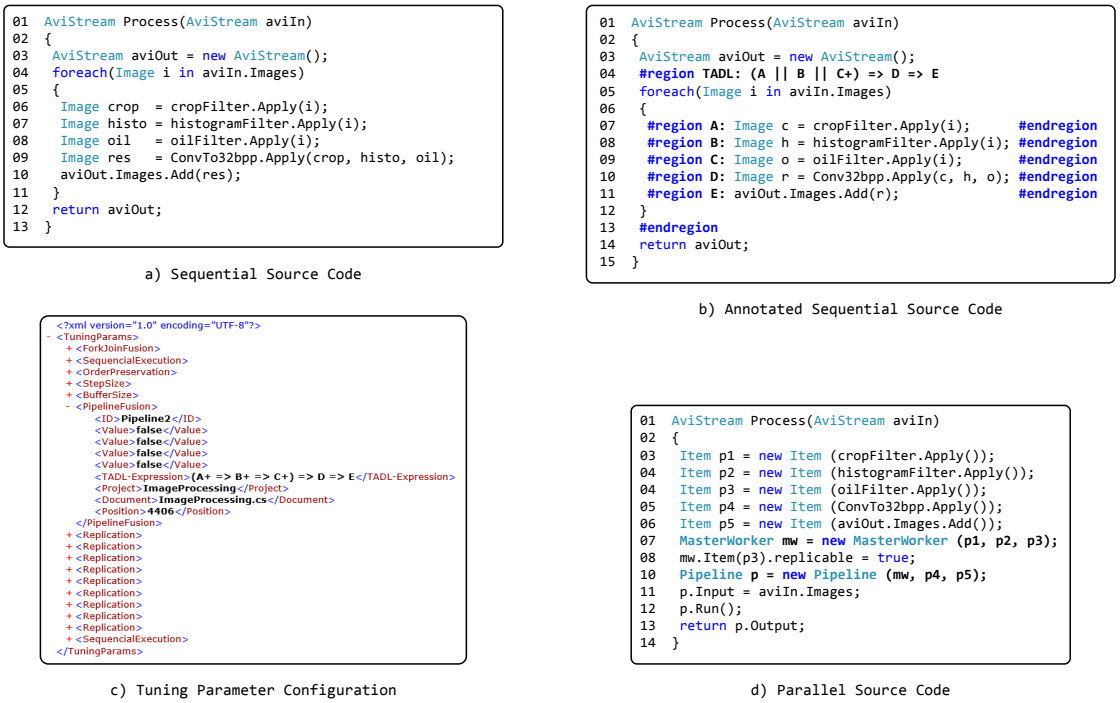

Figure 3. Phase Artifacts from Pattern-based Parallelization in Patty

experienced engineers with the notion of parallel patterns and enable experienced engineers to bypass automatic pattern detection and manually write code annotations like in OpenMP.
The last step produces parallel source code like the one shown in figure 3 d). For the purpose of standardization, we implemented a runtime library that contains data types for parallel patterns and that is capable of handling tuning parameters. If common parallel libraries like TBB, OpenMP or CILK++ can handle tuning parameters in the future, we can easily change our process to use them instead. As the process diagram points out, this last stage does not only produce parallel source code.
The tuning configuration file contains all identified tuning parameters, their current values and code location. Whenever the parallel application is executed, it initializes the parallel patterns with the specified values and executes as expected. An example tuning file is shown in figure 3 c). After program termination, all values in the configuration file can be changed, making the parallel applications automatically tunable on the target hardware without the need to recompile.
As we employ optimistic analyses, we cannot guarantee correct semantics in the parallelized version. To assist engineers in locating potential parallel errors like data races, we automatically generate parallel unit tests for each tunable parallel pattern. After this, we perform a path coverage analysis to generate a set of input data for each unit test. All unit tests are then executed on the dynamic data race detector CHESS [24]. For each parallel unit test and input data, CHESS computes and provokes all possible thread interleavings. As unit tests are rather small portions of a whole program, we can keep the search space for parallel errors also rather small which makes our approach to error detection very handy. As we previously showed in [22], we can locate parallel errors with a high detection accuracy at within several minutes.

2.2 Software Pipelining

The process model is currently implemented for the detection and transformation of master/worker, data-parallel loops, and pipelines. In this section, we will explain the algorithm for pipelines. As literature reveals, pipelines are heavily used in scheduling theory [25, 26]and stream-oriented applications such as signal, image, or video processing [18, 23, 27, 28]. Pipelines are characterized by distinct stages organized in a processing chain. A continuous flow of data stream elements gradually passes through the pipeline stages. Pipelines can either bind threads to stages or to data stream elements. We implement stage binding and use buffers to connect predecessor and successor stages. As Tournavitis and Franke show in [7], pipelines achieve the highest efficiency, when the execution times for all stage are evenly distributed, because this avoids idle stages and overfull buffers.

Stage binding pipelines imply the following preconditions for control and data dependencies: 1) A pipeline is defined on a continuous data flow. 2) The processing chain is fixed, so each stream element has to be processed in the identical order. 3) Data dependencies to former stages may not affect any other stream element. We consider locations in sequential source code as suitable for software pipelines that preserve these restrictions. As for all *target patterns* in our pattern catalog, we assembled *source patterns* as a set of dependency rules and runtime conditions for the *tuning parameters*. For pipelines, these rules are:

- **PL$_{PL}$: Pipeline logic**. As depicted, a stream typically flows continuously. We consider all sequential program loops in the source code as a first indication for pipelines. Each loop iteration executes the same statements and in the same sequence on different elements. Next, we process the loop header, increment and termination condition. This represents the generation of continuous stream elements. In a pipeline, this logic is not explicitly visible, but as it has to be retained, we transform the loop header to an implicit first stage called `StreamGenerator`.

Initially, we transform each statement in the loop body to a separate pipeline stage. Depending on data and control dependencies, stages are subsequently combined.

- **PL_{DD}: Data Dependencies**. Pipelines have a fixed processing order for all stream elements. For any given element, a single pipeline iteration and a single loop iteration must have the same semantic. Loop-interior data dependencies are preserved by the fixed processing order, so we do not have to consider them. Loop-carried data dependencies, such as from a statement s_i in iteration j to a statement s_k in a previous iteration, can change the semantics when executed in parallel. To solve this dependency and retain the correct semantics, we subsume s_i, s_k, and all statements in between in one pipeline stage.
- **PL_{CD}: Control Dependencies**. Statements like `break`, `return`, or `continue` affect the control flow, because they decide whether to execute the succeeding statements or where to branch to. A fixed processing order for all stream elements only permits control flow conditions that have no side effects on other stream elements. A conditional statement s_i in iteration i is therefore not allowed to affect the control flow in any other iteration.
- **PL_{DS}: Data stream**. We construct the pipeline data stream by analyzing read and write sets for all loop body statements. From this information we construct the data flow graph. It defines what data is being read and written in what pipeline stage. As a result, the pipeline logic will pass this data along the pipeline stages via buffers.
- **PL_{TP}: Tuning Parameters**. As we mentioned in section 2, our analysis captures static and dynamic information. This enables to derive tuning parameters for a given parallel architecture. We currently derive the following tuning parameters for pipelines:
  `StageReplication`: If the runtime distribution of all pipeline stages is highly imbalanced, the frequencies at which stages consume and produce elements are also highly imbalanced. This leads to a situation in which some stages wait and run idle, while others execute permanently and overfill their output buffers. We solve this by inserting hierarchical parallelism: We locate the stage with the highest runtime share and analyze its dependencies. If this stage has no side effects on other stages, we execute this stage in parallel to itself on the next available stream element. As we found out, this is a likely use case for stream-oriented applications. A stage replication value of two effectively doubles the frequency at which this stage is capable of receiving and producing elements.
  The optimal degree of parallelism depends on the runtime imbalance and on the number of available cores, so we define replicability as tuning parameter and postpone the determination of the optimal parameter value to the performance validation phase.
  `OrderPreservation`: When executing a replicated stage, the pipeline loses its order guarantee for stream elements. If the stream element e_{i+1} is processed faster than its predecessor e_i it will be added to the output buffer before e_i, resulting in a wrong data element order. Obviously it is undecidable whether an order violation compromises the correct semantics, so we generate this as a tuning parameter and test it during correctness testing. If `OrderPreservation` is set to true for a given replicated pipeline stage, the pipeline logic will restore the correct order before the stream elements are passed on to the next stage.
  `StageFusion`: Pipelines have a fixed stage processing order. If the runtime share of a pipeline stage is rather low, the thread and buffer management overhead will outweigh the advantage of parallel processing. In this case, it can be beneficial to execute neighboring stages within the same thread, because this saves the mentioned thread creation and buffer overhead. For each pair of adjacent pipeline stages, we generate the tuning parameter `StageFusion` that specifies whether to execute them within the same thread or not.
  `SequentialExecution`: Parallel software speeds up an application, because certain computations effectively happens at the same time. This comes at the cost of initializing parallel constructs and explicit synchronization. The longer executions can run in parallel, the higher are efficiency and speedup gain. Pipelines maximize parallel computations by runtime-balanced stages and long data streams. We deal with the first aspect by providing stage replication. For long data streams, pipelines as such are well-suited. In case of a data stream that is too short to compensate for the threading overhead, we provide a mechanism to execute the pipeline sequentially. With this parameter we ensure that pipeline execution never leads to a slowdown in comparison to the former sequential version.

3. Pattern-based Parallelization in Patty

In this subsection we introduce Patty, a tool that assists in different parallelization tasks. It is geared to the pattern-based process model shown in section 2. We implemented Patty as plugin for Microsoft Visual Studio and defined the following requirements:

- **R_1: Comprehensible parallelization**. The fundamental goal of Patty is to assist engineers in parallelization and automate several or all tasks. As we will see in section 6, it is crucial for the acceptance of any assistance tool that the engineer understands what the tool performs and that the results are reproducible. In order to keep software development and parallelization assistance close together, we decided to implement Patty on top of the widespread integrated development environment (IDE) Microsoft Visual Studio. We chose to provide both, a graphical representation of the process model and a graphical wizard for each phase. Both snippets are shown in figure 1 and figure 4 a). The process chart always highlights the current state of processing, its input and output data. The wizard is used to trigger the parallelization and for user input, such as the file path to the source code, or input data for the dynamic analysis.
  In contrast to some related work, our IDE integration helps to reflect the parallelization results back to the corresponding source code. We use color marks to draw overlays over the code annotations. With this notion, the engineer's attention is directly drawn to the detected parallel architecture. The code snippet in figure 4 b) shows the overlay. It shows a pipeline with two replicable stages. For the purpose of simplicity, we collapse the source code statements. At this stage, the parallelization process is in the interface state between detection and transformation: The target parallel architecture has been identified and is annotated to the source location, but has not yet been transformed to parallel software.
  As we have shown in section 2, we define correctness and performance validation as key elements in our parallelization process. With the parallel unit tests and the tuning configuration, we bridge the gap to data race detection and auto tuning. We relay the parallel software to these tools and integrate their results in the IDE. Figure 4 c) shows the snapshot of an auto tuning cycle: The auto tuner initializes the program with parameter values, executes it, measures and visualizes the runtime, and computes new parameter values. At the time of writing, we employ a basic tuning algorithm that explores the search space linearly in each dimension. For the future, we want to evaluate smarter algorithms like [29–31].

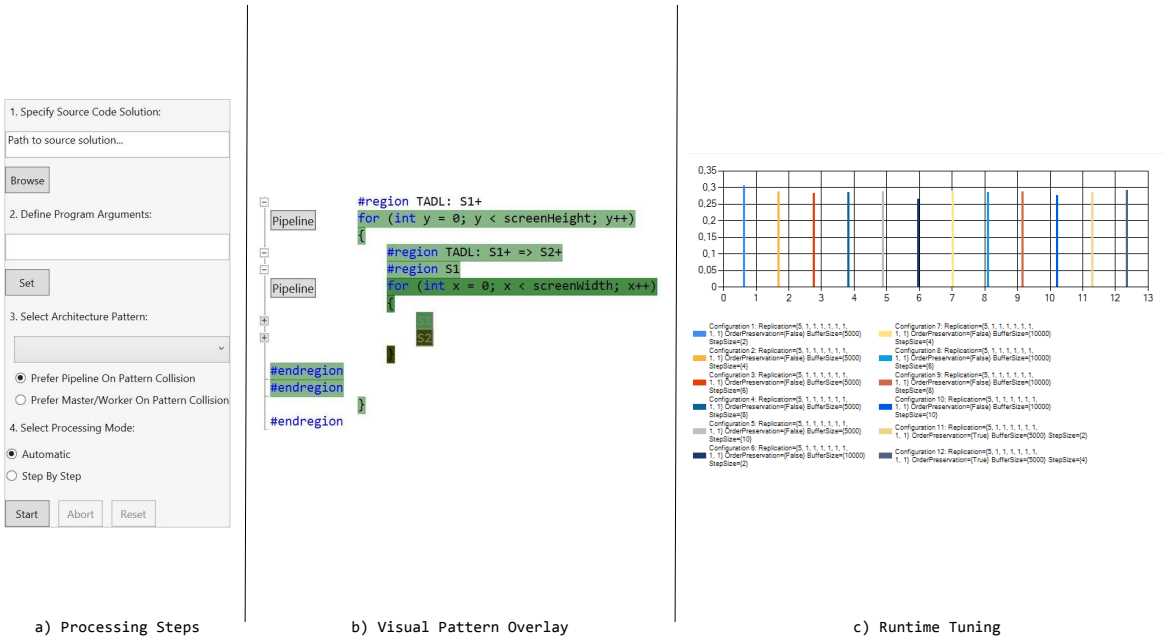

Figure 4. Pattern-based Parallelization in Patty

- **R_2: Adequate amount of information**. We see the necessity to visualize the phase artifacts after each step, and not only focus on the current processing step, as stated in R_1. Section 2 shows that each phase has distinct input and output artifacts: The analysis phase creates a semantic model from source code, the pattern matching phase derives parallelizable locations and produces architecture descriptions, the third phase processes the descriptions, and the code generation creates parallel architectures, parallel unit tests, and a tuning configuration. We make this information available to the engineer, if desired. Some of the artifacts are shown in figure 3. For each, we provide a visualization integrated into the IDE.
- **R_3: Flexible parallelization**. Currently, parallelization approaches dictate one way how to work with them. In our view, this does not take into account the variety of engineers' skill levels. We want to be able to support experienced and inexperienced software engineers alike. Patty therefore supports four different operation modes.
  1) Automatic parallelization: This operation mode requires no action from the user. The graphical process chart gradually progresses through the stages and indicates the current and previously completed stages. At the end, the parallel source code is available for compilation.
  2) Architecture-based parallel programming: More experienced engineers that already know where to parallelize can bypass the automatic detection. They can write parallel code by adding source code annotations in TADL syntax. The transformation step processes the TADL annotations and create the equivalent parallel architecture. This approach is comparable to compiler extensions like OpenMP. Here, engineers add annotations to sequential loops that are mapped on to parallel loops on compilation. In contrast to OpenMP, our approach as implemented in Patty automatically creates correctness and performance tests from a given TADL annotation.
  3) Library-based parallel programming: As we mentioned earlier, we provide a parallel library that contains data types for parallel architectures. Skilled engineers can bypass the TADL-based transformation step and explicitly develop parallel applications on a low abstraction level. Engineers can instantiate parallel data types and develop parallel code that is more flexible than in the other two operation modes. At the same time, explicit parallel programming causes the highest development overhead, because it does not offer automatic performance or correctness assistance. It enables parallel programming at the lowest level and is comparable to explicit parallel programming like PThreads. However, as we provide parallel data types and architectures, engineers do not have to deal with thread synchronization.
  4) Program validation: Parallel applications that have been developed using our approach consist of parallel unit tests and a tuning configuration. This enables an operation mode that addresses performance and correctness validation. For performance validation, the parallel application is repeatedly executed with different tuning parameter values. This enables to adjust the tuning configuration to the target multicore platform. For correctness validation, data race detection is repeatedly executed on the existing parallel unit tests and on the identified input data. This operation mode can be executed in the course of integration tests on and does not require source code insight from the engineer.

4. User Research Study for Patty

In theory we know that parallel software engineering is hard and studies like [1, 14, 15] indicate that implicit parallel program on an abstract level improves the development efficiency. To evaluate the real pitfalls and the benefits of an integrated pattern-based parallelization process, we performed a user study with software engineers. In this study, we gave answers to these question:

1. What are the problematic tasks in parallelization?
2. How can tools adequately assist in these tasks?
3. How much performance gain can be generated in an automatic approach and in what time?

In the remainder of this section we describe the experiment setup and present the results.

4.1 Experimental setup

The parallelization time for a project varies from hours to days of work, depending on the size of the project, the number of engineers, and their skill level. We wanted to gather participants from different skill levels and assemble a task that is manageable in reasonable time for all participants. We therefore decided to focus on detecting parallel potential. When transforming a location to parallel code, skilled engineers inseparably deal with aspects like correctness and performance, while inexperienced engineers deal with the basics of multicore programming. We will compare engineers' transformation performance in a future study.

For this study we collected ten participants with different experiences in general and multicore software engineering. We retrieved their skill level in both categories in interviews before we performed the actual study. From this score we composed three groups with an equal average experience level. Also we classified all group members from *inexperienced in software engineering*, *experienced in software engineering, but inexperienced in multicore engineering* to *experienced in multicore engineering*. Group 1 used Patty for their task, group 2 used the intel Parallel Studio, a prominent commercial tool that serves the same purpose as Patty. Group 3 was a control group that did not work with a parallelization tool. This group had to identify parallel potential using the standard tools available in Visual Studio.

We selected *RayTracing* as single benchmark program. The implementation consisted of 13 classes and 173 lines of code. We manually analyzed this program before to identify all locations that could profit from parallelization. The task for all three groups was: "`Find all source code locations that are appropriate candidates for parallel execution.`". Before the study, none of the participants knew the task and their group.

When the study began, we gave all participants the full source code in print. Each participant sat in front of a dedicated machine and had 15 minutes to get used to the working environment and familiarize with the source code. The maximum time to accomplish the given task was one hour. We informed all participants that their screen was recorded and later evaluated by us.

We prepared a questionnaire that was given to all participants after the study and interviewed them separately in case some aspects have not been covered by the questionnaire. We assembled the questions and answers in the standardized questionnaire format proposed in [32]. We created one questionnaire for the tool-based groups one and two, and one for the manual control group 3. The questions for groups 1 and 2 assessed the quality of the tool they used whereas we asked group 3 what features could improve their work, if they had to do this task again. After the study, we evaluated all questionnaires and screen recordings and assembled an evaluation with these three quality indicators:

1. Objective result: Identified source code locations. How many locations did the participants find? How many of these were correct?
2. Objective result: Total working time. How long do the participants need to find the first correct location? What is their total working time?

Indicator	Group 1: Patty	Group 2: intel
Clarity	2.00, 0.68	1.00, 1.75
Complexity	2.00, 1.42	0.75, 0.95
Perceivability	2.33, 0.83	1.00, 1.03
Learnability	2.33, 0.58	1.25, 1.59
Total Comprehensibility	2.17	1.00

Table 1. Comprehensibility: Average Values, Standard Deviation. [-3(worst) ; +3(best)].

Indicator	Group 1: Patty	Group 2: intel
Perceived tool support	2.00, 1.73	1.75, 0.96
Subjective satisfaction with result	0.67, 0.58	-0.25, 2.75
Overall assessment	2.25	1.40

Table 2. Subjective Tool Assistance: Average Values, Standard Deviation. [-3(worst) ; +3(best)].

3. Subjective experience: Questionnaire and interview. How do the participants evaluate the tool support? What assistance do the manual participants miss?

4.2 Study evaluation

- **Subjective Experience**: With the feedback questionnaire we assessed the subjective impressions the engineers had when they used the tools. For the manual control group, we determined what features were regarded suitable for parallelization and should therefore be integrated in a tool. We divided the subjective user experience into the following three aspects:
  Comprehensibility: We defined this aspect as a combination of the four indicators listed in table 1. We used a score from 0 to 7 in cross-value order. This means that on some questionnaire 0 was the best score and on others 7. For the evaluation, we normalized all values to a score from -3 (worst) to +3 (best). We can see that Patty receives better scores across all four comprehensibility indicators and an average score of 2.17 in comparison to 1.00 for intel's Parallel Studio. For all average values, the standard deviation is smaller for Patty, except for the indicator complexity. This states that the scores across all participants are closer together than for Parallel Studio which makes the results more reliable.
  Satisfaction: After the study ended, we assessed how satisfied the engineers were with range of parallelization support in the tool, and how satisfied they were with the results they achieved. The results are shown in table 2. Across all three indicators, Patty receives higher scores than intel's Parallel Studio. Concerning the subjective satisfaction, the intel group has a high standard deviation, so we looked into the results for this group. We found out that the participant with the highest skill in multicore engineering gave intel's Parallel Studio excellent scores. Then we compared his objective results to the ones achieved by the most skilled engineer in the Patty group. Our finding is that the subjective satisfaction with the intel tool is better than with Patty, but our tool produces better objective results.
  We were not able to conduct a blind study so that the intel group did not know that they used an intel tool. Because of this, we cannot exclude the possibility that the image of a brand has an influence on scoring. However, as satisfaction is a pure subjective metric, and the average score is still higher for Patty, we tend to relinquish this aspect. We conclude that Patty satisfied requirement R_1 from section 3. *Accuracy*: This aspects determines whether Patty displays an accurate amount of information to the engineer. We evaluated the questionnaires of the manual control group that assessed what tool support would help them in parallelization, if they had to do this task again.

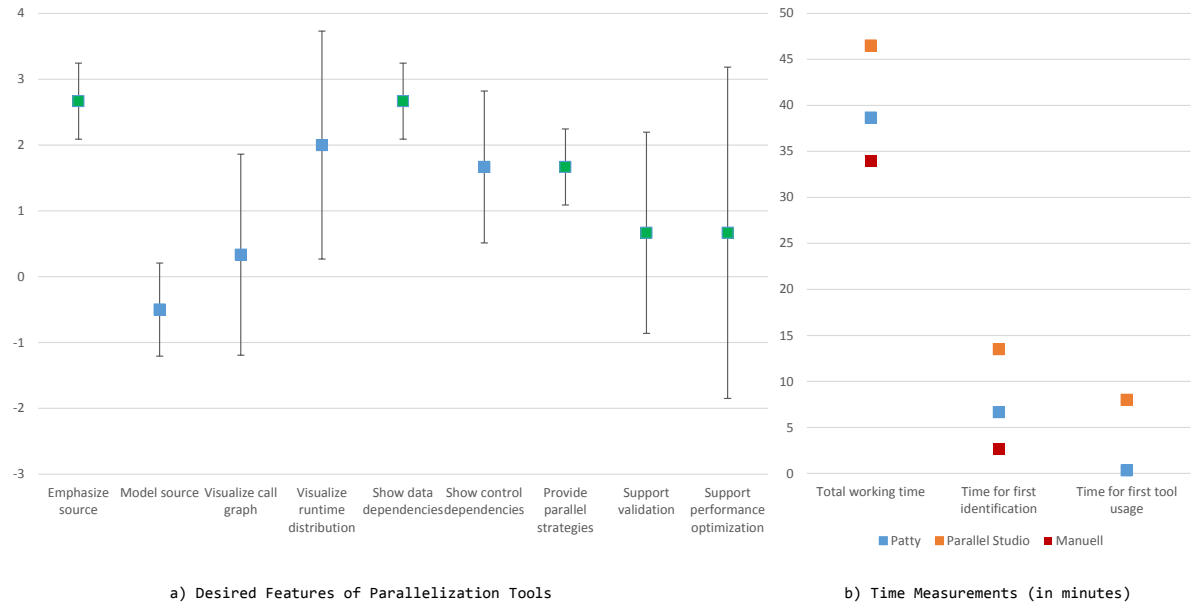

Figure 5. Study Evaluation Results

The results are given in figure 5 a). For the questionnaire we collected different tool features and let the manual control group decide, how helpful these feature would be to them. The figure plots the average values for all answers and their upper and lower quantiles. Green marks features, Patty is already capable of. We can see that Patty already provides three of the top five features, while intel's Parallel Studio provides only two features out of nine. Furthermore, it provides just one of the top five features (Visualize runtime distribution). We conclude that Patty meets requirement R_2.

- **Objective Results**:
  Intuitivity and *Efficiency*: In order to assess the quality of both tools, we manually evaluated all screen videos and measured the time until the participants used the tools as intended. The shorter it takes to start working with a tool implies a plausible UI design, so we took this measure as an indicator for *intuitivity*. As we see in 5 b), the Patty group immediately started parallelizing (Avg. 0.33 min). They identified their first code location with an average of 6.66 minutes. The intel group took more than twice as long (Avg. 13.5 minutes). The manual control group also identified their first code location in a short time, even faster than Patty (Avg. 2.66 minutes). After reviewing the videos we realized that almost all of the participants navigated through Visual Studio during the introductory phase and found the built-in profiling tool. When the study began, they directly executed it. For our benchmark program, the profiler reveals one code location with parallel potential. As we will see later, the manual control group was unable to detect the other locations hidden in the benchmark. Another finding is that intel has a fixed parallelization process that requires the engineers to know an annotation language. This language is used for estimating the parallel potential and helps to select a parallelization schema for a given location. In comparison to Patty we use TADL as optional annotation language. We use TADL for the annotation of tunable parallel architectures and not for performance estimation. As we mentioned in section 3, AP offers four different operation modes. During our study, only the experienced multicore engineer experimented with TADL without having known the language before. All other participants used the fully automatic parallelization. This shows that Patty meets requirement R_3. To summarize this aspect, Patty was more intuitive to use and led to faster first results for all participants in comparison to Parallel Studio.
  Effectivity: In order to assess the quality of the tool we need to know how many locations it identifies, how many it misses and in what time this is achieved. During study preparation, we manually identified three locations with parallel potential in the benchmark program. The manual control group identified the least amount of code locations (Avg. 2.0) and was the only group that produced false-positives. In all cases, this was due to the fact that data races were overlooked by the engineers. As figure 5 b) shows, the manual group also finished first (Avg. 34 minutes). In the questionnaires all of them were confident that they had found all locations with parallel potential. The intel group identified an average of 2.25 code locations within an average of 46.5 minutes. The group with the highest effectivity was Patty with an average of 3.0 across all participants and an average working time of 38.67 minutes.

5. Future work

With this study, we showed that our tool is capable of supporting pattern-based parallelization in an adequate way. The next step for us is to assess the detection accuracy of pattern-based parallelization as such. For this, we will focus on precision and recall. We are just in the process of conducting a study with a set of benchmark tools from different application domains with a total of 26,580 lines of code. We aim at parallelizing this code manually and evaluate

our approach against this manual parallelized version concerning both, the detection and the transformation quality.

We want to define the detection quality by the notion of *recall* and *precision* to show, how many of the relevant source patterns can be detected by Patty and how many relevant source patterns of the manual parallelization are included in the result set. Also, we want to quantify the runtime overhead by the dynamic analysis, so we will measure the *runtime* and *memory increase*. The transformation quality will be defined by the *performance improvement* of the parallel code in comparison to the original version.

Early results indicate that with pattern-based parallelization we achieve high values for precision and recall with a balanced F-score of approximately 70%. At the same time, early performance results indicate a parallel performance close to manual parallelization that is achieved within minutes and not days of work.

6. Related Work

This paper introduces a pattern-based parallelization process and implements the process in Patty, an automatic tool that installs on top of the common IDE Visual Studio. This section covers works from automatic parallelization and tool-assisted parallelization.

- **Automatic parallelization**: In the past decades, many analyses have been developed that can hardly be fully covered in this paper. For the purpose of this paper we examine some of the recent approaches to point out the most relevant concepts.

  All works on automatic parallelization either employ static source code analysis or dynamic runtime analysis [33]. Static analyses generally execute fast because they operate on source code. The downside is that they are pessimistic in their predictions. This leads to an overapproximation of the actual program dependencies and results in parallel code that might not even be executed. At the same time, parallel potential will mistakenly be sorted out due to the overapproximation.

  Dynamic analyses in contrast only gather dependencies that effectively occur at runtime. As they operate at runtime they generally have a huge performance and memory impact. The analysis of whole program executions is therefore unmanageable. Although the detection of parallel potential might be more precise and the parallel suggestions yield higher speedups, dynamic analyses also miss potential: As they only gather what is being executed under the given input data, they cannot detect parallel potential in code that is not executed during dynamic analysis.

  Recent research works like [34, 35] among others increasingly make use of a combination of both analysis techniques. J. Mak et al. try to detect task parallelism by executing single statements in threads [35]. Mak makes suggestions for the developer where the main thread should wait at a barrier. The parallelization has to be done by the software engineer. Many current works focus on loop and pipeline parallelism [5, 7, 8, 34]. While Rul identifies pipeline stages by analyzing control and data dependencies (like Patty), Tournavitis balances pipeline stages by runtime information. In contrast to all mentioned works we cover and differentiate both forms of parallelism. Another important aspect is that we detect parameters that influence the runtime behavior to enable the parallel code to adapt to current and future multicore hardware. The benefit of tuning parameters and their influence on software architectures has previously been shown in [27, 36].

  The detected patterns in our approach are internally represented by an architecture description and the code is transformed accordingly. Related works consider either implicit parallel programming as in OpenMP, CILK, TBB, or XJava [27]), or automatic parallelizing compilers like SUIF [37] or the Intel C++ compiler. The advantage of implicit parallel programming is that parallelism can be expressed comfortably and thread creation or synchronization overhead is hidden behind interfaces. This lowers the burden for software engineers to develop parallel software. Still, the detection of an adequate location and how to parallelize it properly is not answered. Refactoring sequential to parallel code is still left to the engineer. Compilers can achieve fully automatic parallelization but the parallel potential is limited because compilers formally prove the correctness of the parallel result. Many works on parallelizing compilers address automatic loop parallelization [38–43]. The basic observation is that most of the runtime is spent in program loops, so these program structures carry high parallel potential. As Hind reveals in [44], this potential is restricted to program loops with a predefined number of iterations and without any dependencies. In the presence of heap memory reference, compilers use pessimistic heuristics, and this permits only little performance gains. In scientific programming though, this mechanism is heavily used.

- **Tool-assisted parallelization**: The recent years brought forth several tools to assist in parallelization from both, the research and industry community. This section covers ParaGraph [45, 46], HTGviz [12], Prism by CriticalBlue [10] and the intel Parallel Studio [9].

  As its main feature, *ParaGraph* visualizes the control flow graph in order to reveal data dependencies. It is implemented as a plugin on top of Eclipse and uses the external source-to-source compiler Cetus [47] for the detection of control and data dependencies. ParaGraph uses a visualization comparable to traditional control flow graphs, but extends it in two ways: 1) ParaGraph displays basic source code blocks as nodes, but does not convert loop statements to conditional branches as in traditional control flow graphs. The authors claim that this increases the recognition value of the underlying source code. 2) ParaGraph adds control and data dependency edges to its graph representation. The weaknesses of ParaGraph compared to our approach are that ParaGraph does not distinguish between the different types of data dependencies (true, output, anti-dependency). Also, the graph serves as pure visualization and provides no further functionality for parallelization assistance.

  HTGviz also displays the control and data flow. It uses a static analysis to generate both. HTGviz is not integrated into a development environment but provides a graphical user interface. In contrast to ParaGraph, the graph in HTGviz can directly be manipulated, so that basic blocks can be grouped together and dependencies can be eliminated. It also detects hierarchies in nested loops. This enables a graphical collapse of nested blocks, and this improves code readability. HTGviz also visualizes the different types of data dependencies. One drawback is, that HTGviz does not reflect the changes back to the source code. This has to be done manually.

  Prism is an industrial tool by CriticalBlue which is used to identify parallel potential in sequential software. It does not make changes to the source code but provides hints to the user where to parallelize. Prism is able to perform symbolic execution of the user changes and calculates a speedup potential from the simulation. The number of computing cores can be specified for the simulation. In contrast to the first two tools, Prism executes the software to retrieve actual runtime information, and it produces a graphic representation of runtime shares and hot spot regions. Prism does not generate source code, so parallelization is a pure engineer task. For the purpose of validation,

Prism keeps track of the source code changes and refreshes the dependency graph, so that the engineer can identify potential data races.

Intel Parallel Studio is a tool chain that consists of three different tools. It integrates with Microsoft Visual Studio, provides a fixed parallelization process, and guides the engineer through three steps: 1) VTune Amplifier is a runtime profiler that reveals the code locations with the highest amount of runtime share. In comparison to all other tools in this section, it is the only one capable of handling sequential and parallel code. For parallel regions it shows how many threads executed them and how long the CPU was busy. VTune Amplifier displays the analysis results directly in the IDE, so it is easy to link the runtime information to the source code. VTune serves the purpose to reveal what locations carry the highest runtime, so it can be used to identify parallel potential. 2) Parallel Advisor assists in identifying the potential speedup for a given location. To achieve this, the engineer has to add annotations to the source code. Parallel Advisor provides an annotation language in which the engineer can define tasks. On this basis, Parallel Advisor computes the potential speedup gain. 3) Parallel Inspector is a dynamic detector for parallel errors. It executes a parallel program and identifies deadlocks and data races in the runtime trace.
To conclude, Parallel Studio is the most complex product in this section and offers a fine integration into an IDE. In contrast to Patty, Parallel Studio does not answer how to parallelize a code region. The range of the annotation language is currently still quite limited and it does not support the actual parallelization process. Parallel Studio embodies a general dynamic race detector, while Patty automatically add performance and correctness validation tests and integrate them into the parallelization process.

7. Conclusion

Even one decade after the dawn of the multicore age, parallelization remains hard, costly, knowledge-intensive and time-consuming, because tool support is still inadequate. Help is urgently needed, because multicore processors are here to stay.

In this paper we introduced a process model for software parallelization. This process model relies on the detection of tunable parallel architectures from sequential software. It uses optimistic analyses and elevates *correctness* and *performance testing* as central parallelization tasks, in addition to the traditional tasks *detecting* and *utilizing parallel potential*.

We instantiated this process model and implemented it in Patty, a plugin that integrates pattern-based parallelization into an IDE. With this measure we bridge the gap between parallelization tasks and general software engineering tasks. With Patty, parallelization becomes a regular task in any software engineering process.

At the same time, Patty addresses software engineers of different skill levels by providing flexible parallelization assistance. Patty offers five different operation modes, ranging from full automatism to explicit parallel programming. Within this spectrum, Patty provides a higher-level programming mode using architecture annotations and a lower-level programming mode using data types from a parallel runtime library. The last mode purely addresses the optimization of the tunable parallel architecture and deals with data race detection and performance optimization without source code insight.

To evaluate our approach we carried out a user study with software engineers of different skill levels. The three groups had to identify parallelizable regions in a given benchmark as a pure manual task, using intel's Parallel Studio, and Patty. Pattern-based parallelization as implemented in Patty receives better average user scores than Parallel Studio: Our tool is regarded to be structured more clearly, perceived more easily, easier to learn, provides a higher degree of flexibility, and is more intuitive to use (Patty: 2.17, Parallel Studio: 1.00). Concerning objective results, Patty outperforms Parallel Studio with respect to detection accuracy and time (Patty: 100% in 39 minutes, Parallel Studio: 75% in 47 minutes). This makes our approach a very efficient and effective solution to parallelization.

Another finding deals with the tool correspondence to parallelization tasks. From the manual control group we collected a list of features for a parallelization process. Comparing this feature list to Patty and Parallel Studio, we reveal that our tool provides five features out of nine, while Parallel Studio provides only two. From this we conclude that our approach assist in the right tasks and is well-suited for parallelization.

Pattern-based parallelization as implemented in Patty has the potential to serve as universal parallelization tool that is well-suited for engineers from different skill levels in an efficient and effective way. The free lunch might be over. But free snacks are obviously available.

Acknowledgment

The authors would like to thank André Wengert, Jochen Huck and Simon Wagner for their support in implementing Patty. We thank Siemens Corporate Technology for their financial support. We also appreciate the support of the Initiative for Excellence at the Karlsruhe Institute of Technology.

References

[1] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel programmer productivity: A case study of novice," in *Parallel Programmers, International Conference for High Performance Computing, Networking and Storage*, 2005. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.53

[2] H. Vandierendonck and T. Mens, "Averting the next software crisis," *Computer*, vol. 44, no. 4, pp. 88–90, Apr. 2011.

[3] C. Hammacher, A. Zeller, K. Streit, and S. Hack, "Profiling java programs for parallelism," in *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, ser. IWMSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 49–55. [Online]. Available: http://dx.doi.org/10.1109/IWMSE.2009.5071383

[4] Y. Liu, Z. Hu, and K. Matsuzaki, "Towards systematic parallel programming over mapreduce," in *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, ser. Euro-Par'11. Bordeaux, France: Springer-Verlag, 2011, pp. 39–50. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033408.2033414

[5] S. Rul, H. Vandierendonck, and K. De Bosschere, "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Comput.*, vol. 36, no. 9, pp. 531–551, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2010.05.006

[6] K. Streit, C. Hammacher, A. Zeller, and S. Hack, "Sambamba: runtime adaptive parallel execution," in *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, ser. ADAPT '13. Berlin, Germany: ACM, 2013, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/2484904.2484911

[7] G. Tournavitis and B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. Vienna, Austria: ACM, 2010, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854321

[8] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle, "Towards a holistic approach to auto-parallelization," in *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2009. [Online]. Available: http://homepages.inf.ed.ac.uk/gtournav/data/pldi121-tournavitis.pdf

[9] C. Intel, "Intel parallel studio," 2014. [Online]. Available: https://software.intel.com/en-us/intel-parallel-studio-xe

[10] C. CriticalBlue, "The prism technology platform," 2014. [Online]. Available: http://www.criticalblue.com/prism-technology.html

[11] D. Dobb's, "The state of parallel programming - the parallel programming landscape," 2012.

[12] U. Gleim and T. Schuele, *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C#*. Dpunkt, 2011.

[13] H. Sutter, "A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 16 – 23, 2005.

[14] F. Otto, C. A. Schaefer, M. Dempe, and W. F. Tichy, "A language-based tuning mechanism for task and pipeline parallelism," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. DAmbra, M. Guarracino, and D. Talia, Eds. Springer Berlin Heidelberg, Jan. 2010, no. 6272, pp. 328–340.

[15] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy, "Software engineering for multicore systems: An experience report," in *Proceedings of the 1st International Workshop on Multicore Software Engineering*, ser. IWMSE '08. New York, NY, USA: ACM, 2008, pp. 53–60. [Online]. Available: http://doi.acm.org/10.1145/1370082.1370096

[16] M. L. Youguo Pi, Wenzhi Liao and J. Lu, "Theory of cognitive pattern recognition," in *Theory of Cognitive Pattern Recognition*, P.-Y. Yin, Ed. I-Tech, 2008, p. 626.

[17] T. Mattson and M. Wrinn, "Parallel programming: Can we PLEASE get it right this time?" in *Proceedings of the 45th Annual Design Automation Conference*, ser. DAC '08. Anaheim, California: ACM, 2008, pp. 7–11. [Online]. Available: http://doi.acm.org/10.1145/1391469.1391474

[18] T. G. Mattson, B. A. Sanders, and B. K. Massingill, *Patterns for Parallel Programming*. Addison-Wesley Longman, 2004.

[19] V. Pankratius, A.-R. Adl-Tabatabei, and W. F. Tichy, *Fundamentals of Multicore Software Development*. CRC-Press, 2012.

[20] K. Molitorisz, J. Schimmel, and F. Otto, "Automatic parallelization using autofutures," in *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools*, ser. MSEPT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 78–81.

[21] K. Molitorisz, "Pattern-based refactoring process of sequential source code," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, ser. CSMR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 357–360. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2013.49

[22] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy, "Automatic generation of parallel unit tests," in *8th IEEE/ACM International Workshop on Automation of Software Test (AST 2013)*, May 2013.

[23] C. Schaefer, V. Pankratius, and W. Tichy, "Engineering parallel applications with tunable architectures," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, May 2010, pp. 405–414.

[24] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 267–280. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855760

[25] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012.

[26] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, pp. 8:1–8:45, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1880018.1880019

[27] F. Otto, V. Pankratius, and W. Tichy, "High-level multicore programming with XJava," in *31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009*, May 2009, pp. 319–322.

[28] D. J. Meder and W. F. Tichy, "Parallelizing an index generator for desktop search," in *Proceedings of the 2010 international conference on Computer Architecture*, ser. ISCA'10. Saint-Malo, France: Springer-Verlag, 2012, pp. 77–85.

[29] T. Karcher and V. Pankratius, "Run-time automatic performance tuning for multicore applications," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, Jan. 2011, no. 6852, pp. 3–14.

[30] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, Jan. 1965. [Online]. Available: http://comjnl.oxfordjournals.org/content/7/4/308

[31] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 533–549, May 1986. [Online]. Available: http://dx.doi.org/10.1016/0305-0548(86)90048-1

[32] B. Laugwitz, M. Schrepp, and T. Held, "Konstruktion eines fragebogens zur messung der user experience von softwareprodukten," in *Mensch und Computer 2006: Mensch und Computer im Strukturwandel*, A. M. Heinecke and H. Paul, Eds. Mnchen: Oldenbourg Verlag, 2006, pp. 125–134.

[33] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[34] M. Kim, H. Kim, and C.-K. Luk, "SD3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 535–546. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.49

[35] J. Mak, K.-F. Faxn, S. Janson, and A. Mycroft, "Estimating and exploiting potential parallelism by source-level dependence profiling," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, ser. EuroPar'10. Ischia, Italy: Springer-Verlag, 2010, pp. 26–37. [Online]. Available: http://portal.acm.org/citation.cfm?id=1887695.1887700

[36] C. A. Schaefer, *Automatische Performanzoptimierung Paralleler Architekturen: Neue Konzepte zur effizienten Enwicklung paralleler Programme*. Saarbrucken: Suedwestdeutscher Verlag fuer Hochschulschriften, 2010.

[37] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural parallelization analysis in SUIF," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, pp. 662–731, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1075382.1075385

[38] GCC, "Automatic parallelization in GCC," 2012. [Online]. Available: http://gcc.gnu.org/wiki/AutoParInGCC

[39] C. Intel, "Intel compilers," 2014. [Online]. Available: http://software.intel.com/en-us/intel-compilers/

[40] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting polyhedral loop transformations to work," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, L. Rauchwerger, Ed. Springer Berlin Heidelberg, Jan. 2004, no. 2958, pp. 209–225.

[41] M. E. Wolf, "Improving locality and parallelism in nested loops," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1992, UMI Order No. GAX93-02340.

[42] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 30–44. [Online]. Available: http://doi.acm.org/10.1145/113445.113449

[43] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. New York, NY, USA: ACM, 1991.

[44] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 54–61. [Online]. Available: http://doi.acm.org/10.1145/379605.379665

[45] I. Bluemke and J. Fugas, "C code parallelization with paragraph," in *2010 2nd International Conference on Information Technology (ICIT)*, Jun. 2010, pp. 163–166.

[46] ——, "A tool supporting c code parallelization," in *Innovations in Computing Sciences and Software Engineering*, T. Sobh and K. Elleithy, Eds. Springer Netherlands, Jan. 2010, pp. 259–264.

[47] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, Dec. 2009.