Pattern-based refactoring process of sequential source code

Korbinian Molitorisz
Karlsruhe Institute of Technology (KIT)
IPD, Chair Prof. Walter F. Tichy
Germany
molitorisz@kit.edu

***Abstract*— Software refactoring is a very well-studied subject, but with the huge gap between omnipresent multicore processors and the vast majority of software that has not been developed with multicore in mind, it gains new and important significance: At what locations should the sequential code be refactored? How should this be done for a given location? Is the code still correct? And finally: Does it execute faster?**

In this paper we present a refactoring concept to exploit parallelizable regions in legacy software. Our concept relies on the presence of recurring patterns and identifies potential regions, transforms them to parallel versions, tests them for correctness and tunes their parallel performance. We show early implementation results.

Keywords: Refactoring; Multicore; Architecture Patterns; Architecure Description Language; Sequential; Parallel

I. Introduction

Software engineers around the world face the big challenge of refactoring legacy software for multicore architectures. Whether it's a workstation, a laptop, a tablet or a cell phone: Hardware has shifted to multicore within only 10 years [1]. The era of free speedups from higher clock rates ended in 2002, as David Patterson, president of the ACM, already stated in his open letter in 2006. This imposes a new burden on every software engineer: How can legacy software properly be refactored? Multicore architectures might be ubiquitous but the knowledge how to use them is not.

We face a situation where literally billions of lines of legacy code have not been developed with multicore in mind. It is very unlikely that all sequential code artifacts will simply be developed from scratch because this would take ages and cost a fortune. So it is radically important to help software engineers refactor for the multicore era, as Hans Vandierendonck also states in [2].

This paper is structured as follows: Section II motivates the necessity of this research by naming typical parallel errors. In section III we present a refactoring concept that consists of three components and builds on the presence of recurring patterns. We show the current implementation status and evaluation in sections IV and VI. We conclude with current drawbacks. In [10] we already conducted an empirical study with software from different application domains with 131.000 lines of code showing that recurring patterns can indeed be identified and used in an automated transformation.

II. Motivation: Why we need refactoring support for existing source code

In this section we motivate our approach by defining the problem scope of a refactoring concept. It is no secret that developing parallel software is hard. Even more so, when it has to execute both correct and fast.

Up to this time the precondition that a program would always produce the same result under the same input was always true, as a sequential program would always execute the identical instruction sequence. Today in the multicore era this is not the case anymore: Each execution of a parallel program emits a different instruction sequence interleaving caused by the operating system scheduler. This makes data and control flow analysis highly relevant.

Additionally to that, parallel code implies a whole bunch of new program errors engineers have to deal with, like race conditions, deadlocks or atomicity violations. Refactoring is currently a manual task where software engineers invest a lot of time and have to be very skilled. One big challenge we are currently facing is to build tools that help to identify locations in sequential software where the control or data flow can safely be split up and producing a speedup gain. In the best case we can find out which hot spot won't put the program correctness at danger. In object-oriented environments where references and dynamic binding are heavily used this is far from trivial. Even more when we don't focus on small applications like matrix multiplication or array operations but general purpose applications. Until now, parallelization approaches operate on narrow use cases like matrices or loops without data dependencies [3, 4, 5, 6].

Parallel code throws this kind of determinism over board. Now each parallel execution contains its own control flow leading to an instruction interleaving. In fact, a parallel section has to be executed only often enough to evoke all possible instruction interleaving. This aspect also applies to data dependencies across the parallel instruction interleaving: If they don't share data, they can safely be executed in parallel. Otherwise the data has to be secured by measures of locks to avoid data races. However, deadlocks or atomicity violations can still occur. So obviously, control and data dependencies limit the parallel potential.

One example is given in Figure 1. The methods `print()` and `main()` both write to a global variable. In a sequential program the call made by `main()` only happens after `print()` has returned.

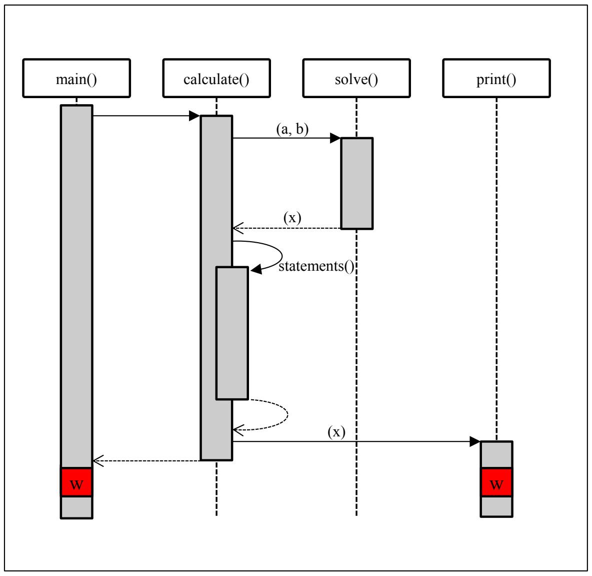

Figure 1: Data race example

When executed in parallel both calls might interfere and result in a race condition: The value in the global variable depends on which thread executed the statement last. In order to solve this issue it is sufficient to assure that the value change to the shared variable happens in an atomic instruction which can be realized using a lock. With this solution the parallel program executes the problematic instruction in serial order. But this raises the question whether the parallel execution still leads to a speedup, as a part of the parallel region will be executed sequentially. To conclude, parallel programs are error-prone and it is time-consuming to deal with the correctness of parallel programs.

III. REFACTORING CONCEPT

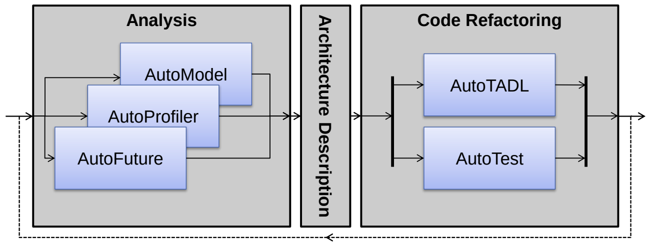

Figure 2: Pattern-based refactoring process

We present a parallelization concept for general purpose applications that identifies potential parallelism at different abstraction layers. We build on the premise that recurring patterns can be identified. As we already showed in the empirical study in [10], this premise can be satisfied. In this paper we extend the focus to a more general refactoring concept.

It is commonly known that there are recurring structures in software. As N. Pettersson, W. Lowe and J. Nivre show in [7], patterns occur at different levels of a software system. In the last years there has been some effort to detect these implicit structures for several reasons like improve source code quality or parallelism [3, 4, 5, 6]. Unfortunately, this research has focused on rather small applications or single patterns. In our refactoring concept we want to be able to deal with different patterns for a single hot spot and therefore define three distinct phases. They are shown in Figure 2.

- **Analysis**: As [7] suggests, patterns occur on different levels of a software system and have to be dealt with individually. In our concept this is expressed through the analysis components *AutoModel*, *AutoProfiler* and *AutoFuture*: The identification at model level needs different analyses compared to the identification at the level of object instances and methods. *AutoFuture* is a concept that automatically transforms and executes methods using the data type Future. *AutoProfiler* is a concept that identifies patterns in sequential code and maps them onto parallel patterns. *AutoModel* applies the pattern detection concept to diagrams.
- **Architecture Description**: In order to be as flexible as possible for future parallel libraries or runtimes, we feel the need to explicitly describe the identified pattern at the respective level. The actual transformation process should be separated from this. For this we define an architectural description language that also embodies implicit runtime parameters identified by the analysis modules. An example is shown in Figure 3 and discussed in section IV.
- **Code Refactoring**: This phase consists of the two components *AutoTADL* and *AutoTest*, the first being the actual transformation of sequential to parallel code according to the specification given in the *TADL*-language and the creation of unit tests for the correctness and performance of parallel code [9].

IV. COMPONENT IMPLEMENTATION

In this section we present the implementation of the analysis module *AutoProfiler*. *AutoFuture* has previously been published in [10]. *TADL*, *AutoTest* and *AutoModel* depicted in the refactoring concept are currently under development and will be published in the future.

In contrast to *AutoFuture* that makes use of static analyses, *AutoProfiler* uses a dynamic analysis in combination with a post-mortem processing of the runtime data in order to minimize the runtime footprint of the analysis. It currently identifies the architectural pattern *Master/Worker*. We currently extend its analysis to also include data dependencies to identify the *Pipeline* pattern at object level. In the first execution of the dynamic analysis, *AutoProfiler* gathers the following key figures from program execution:

- Caller/Callee-graph
- Method invocation count
- Method runtime share

With these numbers *AutoProfiler* creates a runtime graph to distinguish methods that rather delegate their work to corresponding child methods in the call hierarchy from

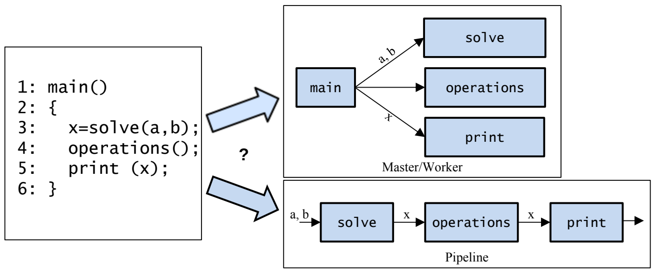
```
1: main()
2: {
3:   x=solve(a,b);
4:   operations();
5:   print (x);
6: }
```

Figure 3: Refactoring for parallelism

methods that actually consume this runtime share. Together with the invocation count it tries to identify master methods, that delegate and worker methods that receive work orders.

Additionally it distinguishes between time that is exclusively spent inside a method body (without time shares of called methods, exclTS) and the total amount of time spent inside a method body (inclusive time share, inclTS). One question is, at what hierarchy level to identify master and worker methods, as the main function is always a worker method in this sense. Our results show that it is useful to calculate the ratio of both time shares.

An example of the analysis in *AutoProfiler* is given in Figure 3. We see three methods with a data dependency between line 3 and 5 caused by `x`. Looking at this piece of code, it is unclear if the *Pipeline* pattern might be better suited than *Master/Worker*. If there was no data dependency between lines 3 and 5, `main()` could be seen as a master in a *Master/Worker* pattern. With the dependency on `x` it could still be refactored as *Master/Worker* with lines 3 and 4 as worker nodes and an implicit barrier before line 5 in order to join the parallel control flow back. In a situation where the runtime distribution between these worker methods is very imbalanced, one of the parallel workers would have to wait relatively long which leads to a poor parallelization gain.

Instead, a *Pipeline* would be more suited in this situation if we knew that the calling method `main()` was executed more than just once. A *Pipeline* only leads to a performance gain, when all stages constantly process items. It might be the case that the middle stage `operations()` is the longest running stage and this stage can be split in distinct parallel tasks internally. In our concept this is a runtime parameter *AutoProfiler* could identify and emit to a tuning tool [8].

V. Evaluation: Pattern AP_1: Master/Worker

We conducted an empirical study to evaluate the intuitive approach with the following characteristics and iteratively adjusted the boundaries for master and worker methods to the given values:

• M_1: High inclTS	(> 70%)
• M_2: Low exclTS	(< 5%)
• M_3: Low call number	
• W_1: High inclTS	(> 55%)
• W_2: High exclTS	(> 55%)
• W_3: High call number	

We used the *Parallel Programming Samples* benchmark that consists of 26 sequential and manually parallelized tools [11]. Before, we manually classified the benchmark and identified 13 hot spots in 8 tools that made use of 3 different parallel patterns (5 x *Master/Worker*, 7 x *Pipeline*, 1 x *Wave Front*). We evaluated *AutoProfiler* by running the respective sequential versions. *AutoProfiler* identified a total of 23 hot spots, 9 of which were true positive results. All occurrences of the *Master/Worker* pattern (5 of 5) were identified correctly. *AutoProfiler*'s precision rate is 39.1% and its recall rate is 69.2%.

AutoProfiler identified 10 additional hot spots that had not been found in the previous manual parallelization. We revised them and found that all of them were calls into the class library of the .NET-runtime. 7 of them were GUI updates and 3 were calls to indexer or properties. Concerning their architecture, GUI updates can generally be seen as a use case for *AutoFuture*, but as they are handled directly by the runtime library, an automatic parallelization is currently not possible. If we accounted these 7 use cases as true positives for *AutoProfiler*, its precision rate would increase to 69.6% and its recall rate to 80%.

In a next step we want to extend the dynamic analysis to also gather data dependencies as this is necessary to identify the *Pipeline* pattern. With this extension *AutoProfiler* would be able to distinguish between task and data parallelism and helps to solve the problem of ambiguity shown in Figure 3.

VI. Caveats and Criticism

Our concept is partly implemented and still work in progress. In this section we want to emphasize the open issues of our concept and implementation.

First of all the process is not yet fully implemented but reflects our current picture to the best of our knowledge. We see the urgency to aid in the typical phases of identifying and suggesting relevant hot spots. By automating most of the work we will need to show the relevance of our process by evaluating the correctness of our results together with the achieved speedup. For *AutoFuture* this could already be shown in [10]. In this paper we presented results that indicate the soundness of the basic analysis used in *AutoProfiler*. We feel motivated by the results that certain patterns exist across application domains and can be found and transformed to a parallel version automatically. As for any automated approach, we might produce false-positive and false-negative results, so we need to discuss the aspects correctness and speedup in more detail.

Our concept dictates that each parallel suggestion is being tested by triggering a race detector in order to preserve correctness. This cannot prove the absence of parallel errors like data races in general, but if a parallel error occurs under the given input, it will be found. Also it can be shown that the program is free of parallel errors under the given input. In order to extend this mechanism to a more generalized statement on correctness, the tests would have to be run with different inputs. So far we have not worked on input coverage.

If the tool chain produces incorrect output we currently try to fix it automatically by error pattern detection using *AutoTest* which is still unpublished. If this is not possible, the software engineer is given the information, what variable produces the race. So at the end the engineer has to engage in parallel refactoring but we believe that with our process a lot of recurring patterns can indeed be identified and refactored automatically.

The analysis modules try to look for very precise patterns. If the source code varies, it might not be possible to identify the pattern, although it is there. Here, we need more research on the boundaries of our analysis modules, but the problem stays: When specifying a pattern at some point there is a cut-off. Situations, where code is very close to the cut-off simply cannot be identified.

AutoFuture currently operates as static analysis and our results encourage us to extend it to also encounter runtime information. A proper points-to analysis has not yet been done automatically, as this is not possible statically.

One essential drawback is that our concept heavily relies on the presence of recurring patterns. The better code is written the more we might gain. This also accounts for the contrary: If the sequential code is bad and without any best-practice object-oriented guidelines, then the effort to identify patterns might lead to no result. We suggest applying traditional refactoring to improve the object-oriented code quality first and then run the pattern-based parallelization.

A general drawback is that any tool chain always bears the problem of correctness and speedup. Both goals are urgent for the acceptance of a tool chain implementing the concept. We see the necessity to produce correct output as more important than a high speedup and we believe that with our test-based correctness approach we can achieve a high acceptance rate.

The results we expect with our concept are not super linear speedups. Our focus is any speedup that can be obtained for free. Our results with AutoFuture shows, that speedups up to 3.34 on an Intel Core 2 Quad machine can be obtained without any knowledge about parallel programming which is quite respectable for a fully-automatic process.

VII. Conclusion

In this paper we presented a parallel refactoring process that tries to identify existing architectures and patterns in sequential code. We motivated the necessity and urgency of this research, as the gap between the omnipresence of multicore processors and the lack of knowledge, time and tool support is real.

In our concept we separate between the analysis and refactoring phase and introduce an explicit architecture language for the following reasons: The presence of patterns is independent of their technical implementation. When explicitly describing architectures it is easily adapted to parallel libraries in the future. Also it is possible to identify patterns at different levels of a software system. We currently introduce three different analysis modules but in the future there might even be more levels and in the proposed concept additional modules can easily be integrated. With the number of processor cores steadily growing a parallelization scheme should be able to adapt to changing runtime conditions. The architecture language we use enables specifying tuning parameters, such as replicable pipeline stages or alternative implementations of parallel algorithms.

Acknowledgements. We thank Siemens Corporate Technologies for their financial support. We also appreciate the support of the excellence initiative at the Karlsruhe Institute of Technology.

VIII. References

[1] R. R. Schaller, *Moore's law: past, present and future*, IEEE Spectrum, vol. 34, pp. 52–59, 1997

[2] H. Vandierendonck, T. Mens, *Averting the Next Software Crisis*, Journal Computer, vol. 44, pp. 88–90, 2011

[3] G. Tournavitis, Z. Wang, B. Franke, M. F. P. O'Boyle, *Towards a holistic approach to auto-parallelization*, Proceedings of the 2009 Conference on Programming Languages Design and Implementation, pp. 177–187, 2009

[4] G. Tournavitis, B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information", Proceedings of the 19th International Conference on Parallel architectures and compilation techniques, pp. 377–388, 2010

[5] C. Hammacher, K. Streit, S. Hack, A. Zeller, *Profiling Java programs for parallelism*, Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering, pp. 49–55, 2009

[6] S. Rul, H. Vandierendonck, K. De Bosschere, *Function level parallelism driven by data dependencies*, SIGARCH Computer Architecture News, vol. 35, pp. 55–62, 2007

[7] N. Pettersson, W. Lowe, J. Nivre, *Evaluation of Accuracy in Design Pattern Occurrence Detection*, IEEE Transactions on Software Engineering, vol. 36, pp. 575–590, 2010

[8] C. A. Schaefer, V. Pankratius, W. F. Tichy, *Engineering parallel applications with tunable architectures*, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 405-414, 2010

[9] J. Schimmel, K. Molitorisz, W. Tichy, *Race detection by exhaustive unit test-based testing of parallel programs*, unpublished

[10] K. Molitorisz, J. Schimmel, F. Otto, *Automatic parallelization using AutoFutures*,International Conference on Multicore Software Engineering, Performance, and Tools, pp. 78–81, 2012

[11] Microsoft Developer Network, *Samples for parallel programming with the .NET Framework*, 2011, http://code.msdn.microsoft.com/windowsdesktop/Samples-for-Parallel-b4b76364

[12] M. Baskaran, N. Vydyanathan, U.K. Bondhugula, J. Ramanujam, A. Rountev, P. Sadayappan, *Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors*, Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 219–228, 2009

[13] J. Mak, K Faxén, S. Janson, A. Mycroft, *Estimating and exploiting potential parallelism by source-level dependence profiling*, Proceedings of the 16th international Euro-Par conference on Parallel processing, pp. 26–37, 2010

[14] J. A. Poovey, B. Railing, T. M. Conte, *Parallel pattern detection for architectural improvements*, Proceedings of the 3rd USENIX conference on Hot topic in parallelism, page 12, 2011

[15] B. Chan, T. S. Abdelrahman, *Run-time support for the automatic parallelization of Java programs*, Journal of Supercomputing, vol. 28, pp. 91–117, 2004