Parceive: Interactive Parallelization Based on Dynamic Analysis

Andreas Wilhelm*, Bharatkumar Sharma†, Ranajoy Malakar†, Tobias Schüle† and Michael Gerndt*
*Technische Unversität München
{wilhelma, gerndt}@in.tum.de
†Siemens AG
{bharatkumar.sharma, ranajoy.malakar, tobias.schuele}@siemens.com

***Abstract*—The advent of multicore processors poses an urgent need for tools to parallelize legacy software. Automatic parallelization methods are usually limited to the instruction level or to simple loops. However, identifying parallelism in industrial applications additionally requires deep program comprehension. To solve this problem, we propose Parceive, an interactive tool that aids parallelization of software at various granularity levels. Parceive uses dynamic binary instrumentation to trace programs written in C/C++. The collected data dependencies and profiling information are then analyzed to visualize potential parallelization candidates. This approach helps developers to comprehend the application and to parallelize their software. In this paper, we motivate our approach, illustrate the architecture of Parceive, and highlight optimizations to cope with industrial applications. A case study shows the usefulness of our approach.**

I. Introduction

Multicore processors are already mainstream, which poses a big challenge to the industry and the research community. To fully utilize the potential of such processors, software parallelism is indispensable. However, analyzing legacy code and implementing appropriate parallel architectures are time-consuming tasks that require substantial manual effort. Hence, most of the existing legacy software still operates sequentially and single-threaded. Besides the fact that parallel programming is fundamentally more difficult than strictly sequential programming, we believe the problem is largely due to a lack of appropriate tools.

Automatic parallelization, as implemented in many compilers, seemed to be the silver bullet for a long time. It allows programmers to keep their sequential programming models and, at the same time, benefit from hardware architectures with multiple cores. Unfortunately, the reality shows that even state-of-the-art compilers miss many opportunities for parallelism. To ensure well-behaved software, the integrated analyses often need to take the most conservative decisions. As a result, automatic parallelization is only applicable to small code blocks and simple loops without intricate dependencies. Even more critical, most limitations for compiler-based parallelization arise due to dependencies involving pointers. For languages like C/C++, it is known that accurate pointer analysis is undecidable [13]. To sum up, programmers are still required to parallelize their software manually.

Surprisingly, only few software tools are available that help developers with manual parallelization. Recently, Intel's Parallel Studio [7] and Pareon [17] have been introduced to support guided parallelization. As opposed to automatic approaches, they rely on dynamic analysis, which allows to check data dependencies at runtime using actual memory addresses. Consequently, these tools do not suffer from the problem caused by inaccurate pointer analysis. Unfortunately, dynamic analysis inherently has an input dependency problem, i.e., specific data dependencies may only be discovered using particular input sets. Thus, parallelism extracted from such tools can only be seen as potential parallelism since they cannot prove correctness for all inputs. Despite this limitation, we believe that the analysis results are invaluable for developers to comprehend and parallelize their software. What alleviates the mentioned problem is that program locations most relevant for parallelization require most of the runtime. As a result, such locations usually benefit from high accuracy and code coverage since the analysis executes them more frequently. Additionally, it is unlikely that relevant candidates for parallel execution are only affected by certain input sets.

In this paper, we propose Parceive, a tracing-based tool for interactive parallelization of sequential software. A typical workflow to bring parallelism into legacy code comprises four steps: (1) identification of candidates for parallel execution, (2) analyzing these candidates regarding runtime improvement and dependencies, (3) parallelization of the software, and (4) validation and optimization. The goal of Parceive is to support developers with the first three steps. For that purpose, our tool uses dynamic binary instrumentation to generate useful trace data for parallelism identification. Being less conservative than existing, compiler-based approaches allows us to focus both on fine-grained data accesses and the high-level architecture. Here is the difference to most existing tools, Parceive gives results that may be used as starting point for an extensive architecture refactoring to expose scalable solutions. However, the user is responsible for the correct parallelization.

The following section depicts the overall architecture of Parceive including components for trace generation, trace analysis, and visualization. In Section III, we show the usefulness of Parceive by a parallelization of BZip2. Finally, we discuss technical challenges and ways to extend Parceive.

II. Parceive

In this section, we present the architecture of Parceive and take a closer look at its components. Figure 1 shows an overview, containing realized components and some that are work in progress. For the sake of brevity, we focus on the existing techniques and defer discussions on the latter to

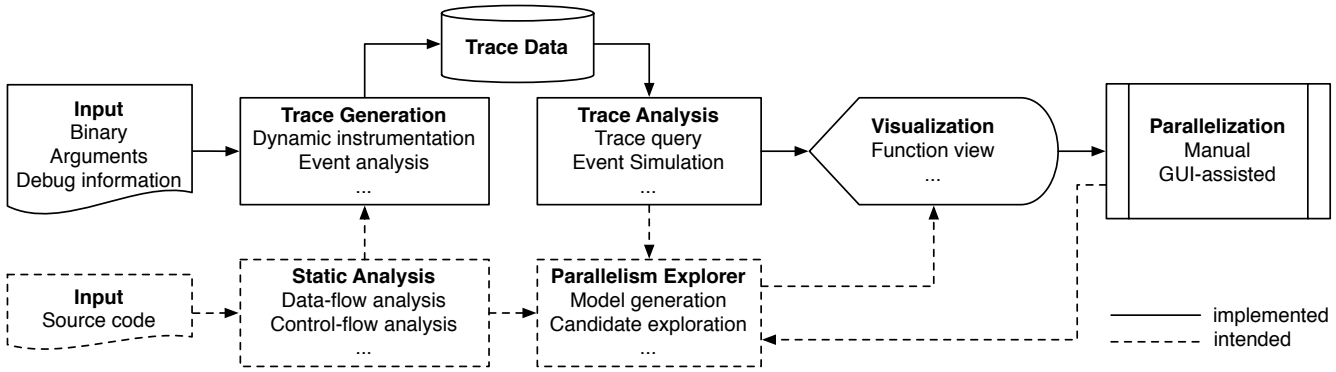

Fig. 1. Overview of Parceive

Section IV. Parceive takes an executable as input and dynamically instruments predefined instructions, e.g., function calls or memory accesses. The instrumentation inserts callbacks that are used to write trace data into a database at runtime. Based on this data, trace analysis identifies potential parallelism by determining hotspots and by exposing dependencies. Finally, the visualization component displays the analysis results, which can be used by programmers to comprehend program behavior and to parallelize their software.

A. Trace Generation

Producing dependable trace information for parallelization requires accurate reporting of function calls and corresponding profiling details for a given input. Sampling-based techniques, while being faster, are not rigorous enough to conclusively capture all crucial runtime events, like accesses to global memory and all function calls. A second option is instruction-level instrumentation of source code or intermediate code. Although this approach provides easy to use and, at the same time, sophisticated analysis facilities, it has two important drawbacks. Firstly, instruction-level instrumentation requires compilation, and thus, availability of the source code and the used static libraries. Secondly and more severely, adapting existing industrial software to conform to requirements of new build systems or compilers is error prone and may discourage usage of a parallelization tool in practice.

Parceive analyzes programs by utilizing dynamic binary instrumentation at the level of machine code during runtime. This approach is convenient for the users, as it does not require any time-consuming preparation (e.g., recompiling or relinking). Additionally, it gives full instrumentation coverage of user-mode code without requiring any source code. Several generic frameworks for dynamic binary analysis exist, such as Intel's Pin [10], Valgrind [11], or DynamoRIO [3]. We chose the Pin framework since it is efficient and supports high-level, easy-to-use instrumentation. Additionally, Pin is portable across several (Intel) architectures and is available for Android, Linux, OS X, and Windows [10], [11].

Our pintool injects analysis calls to write trace data into an SQLite database [12]. SQLite is a single-file based data storage, which enables precise and efficient queries for upcoming trace analyses. The used data model considers both runtime events (such as function calls or memory accesses) and meta-information (such as execution times or variable names). We incorporate the following instrumentation for data gathering:

- Call stack: function entries and exits are tracked to maintain a shadow call stack. For each call, we capture the respective call instructions, threads (in case of multi-threaded programs), and the spent execution time. Additionally, we extract function signatures and file descriptions from debug information.
- Memory accesses: analysis calls are injected to capture information about each memory access (e.g., memory type, memory address, access instruction). For stack variables, we utilize debug information to resolve variable names.
- Memory management: to handle heap memory, we instrument memory allocation and release function calls. The tracked locations are used during analysis to match data accesses using pointers.
- Threading: in case of concurrent software, Parceive needs to track calls of threading APIs, like PThread, to capture thread operations and synchronization.

Performance is one of the key requirements for our tool in order to enable good user experience. Since trace generation for industrial software means dealing with a large number of runtime events, optimizations at different levels are required. To minimize the overhead caused by the data base operations, we use bulk insertions and prepared SQL statements. Bulk insertions collect a group of write queries into one transaction instead of writing each query separately. Prepared statements optimize query creation and execution by generating foreign keys on the fly to maintain referential integrity. Moreover, the database queries are asynchronous; SQLite performs all write requests using a separate thread running in the background. As a result, our examples show that database operations require less than 3% of the overall analysis time.

Most of the runtime during trace generation is spent for resolving variable names for local and global variables. We managed to reduce this effort significantly by shifting the necessary analyses from execution time to instrumentation time (in case of pointer-free memory accesses). To further reduce

the time overhead, we apply binary instrumentation at coarse-grained image level (images are binaries or linked libraries that are mapped into the memory). Additionally, data structures are designed in a lock-free manner, and some data is stored in thread-local storage (TLS).

Even if the inserted analysis code solely calls null functions, the slowdown by dynamic analysis may be unacceptable in many industrial settings. Frequent memory accesses (e.g., those inside loops) or recursive function calls are only two possible causes for such slowdown. To solve this problem, we implemented a filter mechanism to control the instrumentation. More concretely, the user can specify a blacklist and a whitelist in XML-format determining parts that are excluded and included from instrumentation, respectively. For example, dynamic analysis of memory accesses may be problematic under excessive use of STL data structures. If developers are not interested in corresponding access methods, they can exclude them from the analysis by adding the namespace `std::*` to the blacklist. We managed to analyze most of our industrial test cases by using this filter mechanism.

B. Trace Analysis

After the execution of the user software has been finished and the database is filled with trace data, an analysis step investigates useful information for parallelization. We distinguish between analyses that can be performed solely by querying the database, and the ones that require a detailed simulation of events. The first category contains methods to obtain context-free information, like data dependencies between functions. For this analysis, Parceive queries the memory accesses of a function by abstracting from instances of function calls. Provided with this information, the user may easily detect data dependencies by memory locations that are accessed by multiple functions. Another example for a query-based analysis is function profiling. For every called function during execution, the accumulated inclusive-runtime (i.e., execution time including nested function calls) is computed. For upcoming versions, we already investigated other beneficial analyses, like counting specific function calls or accesses.

As mentioned previously, more intricate analyses require an investigation of the control and data flow, which can not be achieved by single queries to the trace database. Hence, we implemented a dynamic analysis framework to facilitate rapid development of dynamic trace analyses. It provides a clean API for communicating events to user analyses, where each event describes an operation of interest (e.g., function calls or memory access). An interpreter reads the trace data in chronological order and pushes user-selected event messages to registered receivers. The framework provides support for associating events with respective source locations. Additionally, shadow variables, locks, or threads are maintained to abstract from programming languages and threading models. This reduces writing analysis backends to the definition of required event-methods and alleviates tool development by a separation of concerns. Additionally, the framework allows such tools to be significantly smaller in code size.

For a proof of concept, we built an analysis backend for dynamic race condition checking. In the case of already concurrent software, this analysis checks for multiple threads accessing the same memory location (with at least one write access). Such data hazards often introduce serious and hard to find software defects. Our implementation uses either lockset-based access detection [14] or a hybrid approach that additionally incorporates vector-clocks [18]. The analysis defined event methods for memory accesses, acquiring and releasing locks, and thread handling. It requires less than 200 lines of C++ source code.

C. Visualization

The strength of user-guided parallelization heavily depends on an effective visualization of analysis results. Users want intuitive means to detect the existing parallelism potential of their software. Besides the representation of profiling information, like execution times and function calls, parallelism inhibiting dependencies need to be highlighted. In an optimal setting, the user may easily identify good candidates for a manual parallelization effort. Furthermore, parallelization often implies intricate and error-prone refactoring steps. We expect visualizations to foster program comprehension for providing detailed understanding of both fine-grained semantics and high-level relations.

Figure 2 shows Parceive's function view, the purpose of which is to highlight data dependencies across software routines. The main window depicts a graph containing the executed functions (rectangle nodes), accessed memory locations (round nodes), and the accesses (edges). We indicate the relative execution time of cumulated function calls by different colors, ranging from 0% (green) to 100% (red) of the overall runtime. Different colors of memory locations show the memory type (e.g., stack variables, heap variables, or static variables). Additionally, a bigger size of memory nodes reveals more data accesses (number of edges), which may point to high parallelization effort. We added some useful features to the user interface: a side pane that shows detailed profiling and call information, zooming and panning functionality of the graph, and an integrated code display to show memory accesses or allocation instructions in the user code. Section III exemplifies how the view may be used to parallelize existing software.

III. Case Study

We demonstrate the usefulness of Parceive by applying it to a single-threaded version of BZip2, a popular file compression algorithm. BZip2 compresses files by chopping them into fixed-size blocks that are processed separately. The overall algorithm can be depicted as follows:

while *not end of input stream* **do**
  read a block from the input stream
  compress the block
  update the CRC
  write the block to the output stream
end
write the final CRC to the output stream

In order to produce a valid output stream, BZip2 needs to obey three constraints: (1) The compressed blocks must be written in the same order as they are read from the input stream. (2) The bzip format requires each block to be aligned immediately after the last bit of its predecessor. (3) A cyclic redundancy

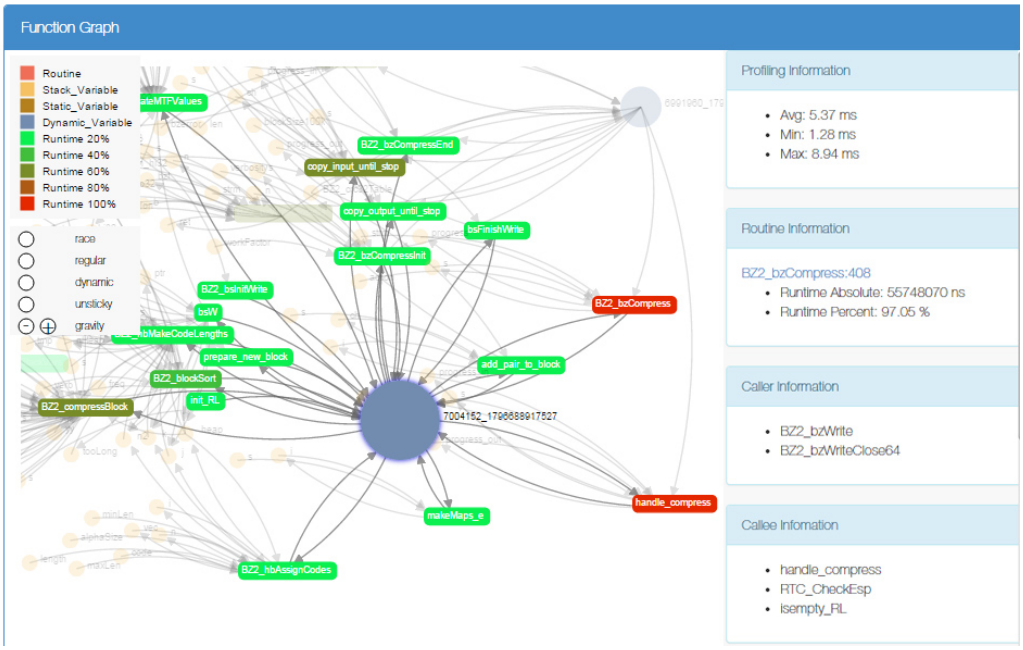

Fig. 2. Function view of Parceive showing functions (rectangles), memory locations (circles), and memory accesses (edges).

check (CRC) code is updated in every iteration. The resulting CRC is written to the output stream right after the final block. Note that every constraint may restrict the available parallelism since they require synchronization of subsequent blocks. We will show a pipeline-based solution consisting of sequential and parallel stages for implicit synchronization.

Three subsequent stages are used for compression. Every stage contains a reversible block transformation, and hence enables decompression of files by passing the stages in the opposite direction. For brevity, we only consider the compression operation, i.e., we use the following order of transformation stages.

1. Burrows-Wheeler transformation [4]: The idea of the Burrows-Wheeler transformation (BWt) is to reorder the characters of a block to simplify the following compression algorithm. The transformation increases the probability to find equal characters close to each other. Blocks containing such patterns are well-suited for local-adaptive compression like move-to-front and Huffmann coding. BWt applies N cyclic shifts (rotations) to a string S of N characters. Afterwards, the resulting strings of each rotation are lexicographically sorted and their last characters are extracted. The extracted substring L and I, the index of S in the sorted list, are forwarded to the next stage. Surprisingly, the transformation can be efficiently reversed given only (L, I).

2. Move-to-front coding [2]: In the second stage, a move-to-front (MTF) coding is performed on (L, I). Simply speaking, MTF returns for every character $L[i]$ an integer value $R[i]$ describing the relative distance to the preceding character $L[i-1]$. Consequently, a sequence of identical characters in L lead to a sequence of zeros in R, which is beneficial for Huffmann coding in the next stage. The reverse strategy can be applied to transform R to L.

3. Huffmann coding [6]: The last stage applies the Huffman compression to R, producing a prefix-free binary code (a set of codewords) with minimum expected codeword length. In contrast, decompression translates the stream of binary codes to individual byte values.

We used Parceive to analyze the compression of Shakespeare's Macbeth (in UTF-8 format, 100kB) with a sequential version of BZip2 [15]. The execution produced 502 megabytes of trace data and resulted in a slowdown factor of 812. Using the presented function view, we gained the following insights that are relevant for parallelization:

- As expected, most of the runtime is used for the block compression, where the traced 81 calls account for 94,8% of the overall execution time. A closer look at the function view and the source code reveals that the time for compression is indeed used for the BWt algorithm (88,7%), the MTF coding (3,8%), and the Huffmann coding (2,3%).
- The function calls for reading the input stream and writing to the output stream accumulate to 0,3% and 1,1% of the overall runtime, respectively. Like the functions for compression, `fread()` and `fwrite()` are called 81 times, i.e., once for every processed block.
- The found data dependencies highlight two crucial restrictions for parallelism. (1) Initially, BZip2 allocates a large data structure (`EState`) that stores the state for the compression process (e.g., a pointer to the currently processed block or the constantly updated CRC value). This structure is represented in Figure 2 by the selected (big) memory node. (2) The function view reveals two frequently accessed variables: `mode` and `state`. These variables create a nested state machine that controls stream processing. By a further investigation of the source code, we noticed that compression operations for different blocks are

independent. All shown dependencies arise due to the global data structure and the mentioned state machine.

Equipped with these insights, we were able to parallelize the file compression of BZip2. According to the trace information, `BZ2_blockSort` requires most of the runtime. Since this function can be easily replaced by appropriate parallel sorting algorithms, it seems to be a good candidate for parallelism. Although the stream operations and MTF coding are serial, this transformation would result in a parallel workload of 88,7%. However, Amdahl's law [1] tells us that the resulting software would not scale with more than 8 processor cores. Therefore, we chose a different parallelization strategy at a higher abstraction level. For the sake of brevity, we summarize only the most important steps of our solution:

- We use a three-stage pipeline which processes one block at a time (see Figure 3). The first stage reads a block from the input stream. The second stage compresses multiple blocks in parallel. The last stage aligns each block bit-precisely and writes it to the output stream. Note that the blocks must be written in the same order as they appear in the original file. For the implementation, we use EMB^2 [16], a parallel library that provides task handling and a pipeline framework. This framework allows us to specify the first and last stage as serial (in order) and the middle stage as parallel. This way, the library removes us from the burden of explicit thread management and synchronization for reading and writing blocks.

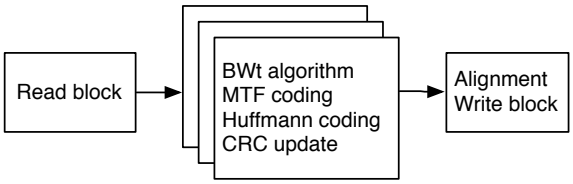

Fig. 3. Pipeline structure of parallel BZip2

- In the sequential version, a single instance of the data structure `EState` is used to process all blocks. Using Parceive, we identified those fields that essentially carry information between blocks, and independent ones. Fortunately, the latter are not accessed in the second (parallel) stage. This lets us resolve most of the data dependencies by separating the fields into two data structures, the first is allocated once and the second is allocated for every pipeline iteration.
- We resolved the remaining data dependencies by removing the nested state machine. This step was straightforward due to the task splitting of the pipeline framework and the dependencies shown in Parceive.

With an effort of one person-week, we managed to parallelize the file compression algorithm of BZip2. Besides the online documentation and the source code, Parceive turned out to be very helpful. It guided us to relevant data dependencies and improved our program comprehension. Figure III depicts the achieved speedup when compressing a 250MB sample file on our 48-core system (AMD Opteron, 1,9 GHz, SSD). Up to 16 cores, the parallel implementation achieves almost linear speedup, but does not scale beyond. This conforms with the maximum speedup of 19 according to Amdahl's law by considering the 94,8% runtime of the parallel pipeline stage. It also shows that the overhead of synchronization and task management in EMB^2 is negligible.

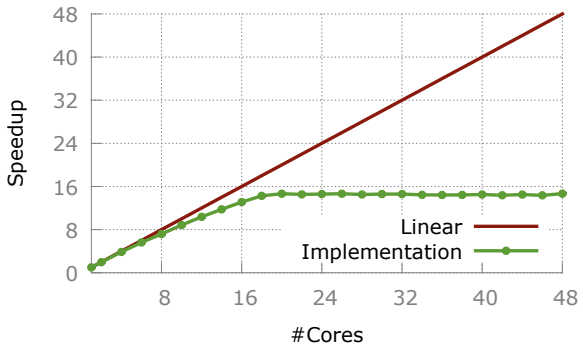

Fig. 4. Speedup for parallel version of BZip2.

IV. Future Work

We see many opportunities to enhance the effectiveness of Parceive in supporting developers with parallelization tasks. Obvious extensions are views that enable users to navigate through selected parts of the traced call graph. By incorporating profiling information, like execution times of functions, developers can easily identify hotspots in their applications. Another useful feature is to show the call stack for data accesses or memory allocations. This would allow to understand intricate reasons for specific data dependencies. Finally, scientific applications often spend significant runtime in loops using regular operations (e.g., for matrix multiplication). Although industrial software usually contains less regular accesses, we see potential for enhanced loop analysis. Tailored views may show respective data, such as dependencies between different loop iterations or the variance of execution times.

As mentioned in Section II, we intend to extend Parceive by additional components. If source code is available, static analysis can be used to get more accurate results. For example, such analyses may identify memory locations or whole functions that do not induce data dependencies. This information would be beneficial for trace generation, since it reduces the number of instructions that need to be instrumented. Another option is to utilize static analysis to find control dependencies, which is not always possible using dynamic techniques due to incomplete code coverage. To conclude, static analysis may allow Parceive to improve both its scope and accuracy of the results. However, to remain universally applicable, the source code needs to be treated as an optional input.

Figure 1 shows another intended component, the parallelism explorer. In contrast to the other analyses, this component considers potential candidates for parallelism that have not been directly traced. It rather identifies candidates for parallel execution that require (partly fundamental) refactoring of the software to expose them. For that purpose, the component first creates an execution model with abstractions from trace and static analysis. In a next step, this model is used to identify

potential candidates for parallelism. The identification may be either done manually by the user or (for simple patterns) automatically by the parallelism explorer. Finally, each found candidate needs to be validated and profiled by simulation to show the prospective performance impact and parallelism-inhibiting problems. For example, the user may select distinct stages of a pipeline pattern, which may exist in the application. Parceive checks this scenario and uses a tailored view to visualize useful information, like data dependencies between stages, the potential speedup, unbalanced stages, etc.

V. Related Work

Tools that aid programmers with parallelization have been developed in both academia and industry. In this section, we focus on approaches that utilize dynamic analyses to identify parallelism in real system executions. A well-established proprietary tool is the Intel Parallel Advisor, which is part of Intel's Parallel Studio suite [7]. The Advisor proposes the following three steps as a workflow: (1) the application gets profiled by Advisor to identify hotspots. (2) The user annotates parallelism candidates like loops or function calls in the source code. (3) By re-executing the software, Advisor inspects these candidates for potential speedup and occurring race conditions. Pareon from Vector Fabrics [17] follows a similar approach. Like Advisor, it mainly focuses on loop parallelism that may be detected by dynamic analysis. However, Pareon is specialized for the embedded domain.

Although these tools have a common goal with Parceive, namely guided parallelization, there are fundamental differences. Our tool focuses not only on low-level parallelism, but additionally tries to identify candidates at architecture level, such as parallel patterns (e.g., parallel pipelines). Also, Parceive fosters program comprehension by providing appropriate views. Instead of showing runtime-related information only, e.g., call graphs or speedup curves, Parceive contains views to depict structural relations of software. Finally, our tool does not require manual annotations, which lowers the burden for the user. Although this approach makes it necessary to instrument the whole program rather than annotated parts, it does not fundamentally affect the analysis performance. Our argument is that manual annotations for parallelism usually are applied to program hotspots, i.e., to most of the executed instructions. Thus, omitting annotations often makes only few difference in the number of traced events, but highly improves the usability.

From a research perspective, Embla [5] and Kremlin [8] are closely related to Parceive. However, these tools mainly focus on functional parallelism and critical-path-based detection of parallelism potential. Our goal is to cover a wider range, from parallelism detection to validation and optimization of parallelization scenarios. Thus, we incorporate dependence analysis, profile analysis, detection of parallelism-inhibiting problems (e.g., atomicity analysis), and especially appropriate views to show parallel candidates. An interesting project which tries to tackle this is the Prospector tool suite [9]. It contains a loop profiler, a data dependence profiler, a parallel speedup predictor, and a post analyzer. Like Parceive, it uses binary instrumentation to gather analysis data, and thus faces the same problems that arise with dynamic approaches, e.g., input dependence. However, Prospector lacks useful visualizations to show the runtime behavior and the software structure graphically for improving program comprehension.

VI. Conclusion

In this paper, we proposed Parceive, a trace-based tool for interactive software parallelization. The presented use case indicates that parallelism may exist at different abstraction levels, but identifying such locations is not trivial due to a lack of appropriate tools. We demonstrated how Parceive tries to fill this gap by graphically providing information about dependencies. Our approach uses dynamic binary instrumentation to generate trace data which are then analyzed and visualized. In order to deal with industrial applications, optimizations are necessary to reduce the amount of data. Although Parceive inherently faces the dynamic-input problem, we believe that the results are invaluable for programmers to comprehend and parallelize their software.

Acknowledgements

We thank Samya Bagchi, Krishna Prasad, and Daniel Hogrebe for their implementation efforts. Additionally, we gratefully acknowledge the support of Siemens.

References

[1] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the Spring Joint Computer Conference*, 1967.

[2] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *CGO*, 2003.

[4] M. Burrows and D. J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical report, Digital Systems Research Center, 1994.

[5] K. F. Faxén, K. Popov, L. Albertsson, and S. Janson. Embla-Data Dependence Profiling for Parallel Programming. In *CISIS*, 2008.

[6] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Resonance*, 11(2):91–99, 2006.

[7] Intel. Parallel Studio XE 2015. https://software.intel.com/en-us/intel-parallel-studio-xe, 2014. [Online; accessed Dec-2014].

[8] D. Jeon, S. Garcia, C. Louie, S. K. Venkata, and M. B. Taylor. Kremlin: Like Gprof, but for Parallelization. In *PPoPP*, 2011.

[9] M. Kim, H. Kim, and C.-K. Luk. Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming. In *HotPar*, 2010.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[11] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.

[12] Public. SQLite3. http://www.sqlite.org, 2014. [Online; accessed Dec-2014].

[13] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.*, 1994.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)*, 1997.

[15] J. Seward. BZip2 v1.0.6. http://www.bzip2.org, 2010. [Online; accessed Dec-2014].

[16] Siemens AG. EMB^2. https://github.com/siemens/embb, 2014. [Online; accessed Dec-2014].

[17] Vector Fabrics. Pareon. http://www.vectorfabrics.com/products, 2014. [Online; accessed Dec-2014].

[18] X. Xie, J. Xue, and J. Zhang. Acculock: Accurate and Efficient Detection of Data Races. *Software: Practice and Experience*, 2013.