Empirical study of partial evaluation of matrix and string algorithms

Ilya Balashov
Saint Petersburg State University
7/9 Universitetskaya nab.,
St. Petersburg, 199034 Russia
i.balashov@2017.spbu.ru

Semyon Grigorev
Saint Petersburg State University
7/9 Universitetskaya nab.,
St. Petersburg, 199034 Russia
s.v.grigoriev@spbu.ru

Daniil Berezun
Saint Petersburg State University
7/9 Universitetskaya nab.,
St. Petersburg, 199034 Russia
d.berezun@2009.spbu.ru

***Abstract*—This paper describes the empirical study on the partial evaluation technique applied to execution time optimization of general matrix and string algorithms. We used AnyDSL framework for partial evaluation during the experiments. Execution time of non-optimized code is compared with the results of partially evaluated code and execution times of commonly used tools. Insides on existing work and our plans are provided. The selection of algorithms, datasets, and experiment structure was clarified. Experiments have demonstrated speedup (up to 3 orders in some cases) of residual programs resulting after partial evaluation using AnyDSL tool.**

***Index Terms*—Partial evaluation, compilers, program optimization, specialization, automatic program transformation, linear algebra-based algorithms**

I. Introduction

Recent years have seen a significant increase in the sizes and complexity of programs in different areas of software engineering. Huge programs or libraries often contain some core code on which significant parts of the program depend, so this code needs to be highly optimized. However, creating a code with sufficiently low complexity for satisfying performance requirements is often an outstanding and time-consuming problem for an ordinary software engineer. A possible way of ensuring sufficient performance or such a code while keeping the development process comfortable for a programmer is the usage of automatic optimization tools and techniques, operating with program sources.

For instance, the so-called *partial evaluation* (or specialization) [1] technique is being actively used over the last years as a way to optimize program execution time automatically, using data known statically. A special tool named *partial evaluator* (or specializer) analyzes data (for example, function parameters) which was provided ahead of evaluation time, and applies several program optimization techniques to the code based on the structure of this data.

Existing results in the area of applied usage of partial evaluation for automatic code optimization include image processing [2], bioinformatics [3] and ray tracing algorithms [4].

In this paper, we applied partial evaluation technique to a code of several algorithms usually utilized as core algorithms in areas connected with linear algebra and string processing. Experiments on partial evaluation of some matrix and string algorithms with AnyDSL [2] framework and evaluating the suitability of the approach for application in industrial libraries and tools are provided. For each algorithm we described the datasets used and justified theoretical reasons why the algorithm could be successfully partially evaluated using AnyDSL tool. A compact overview of current research in the area and an inside on our future plans was also provided.

As a result, it is showed that partial evaluation with AnyDSL could successfully (from 10% and up to 1000 times) improve code performance in the general cases.

II. Background

A. Partial evaluation

Let's suppose:

- P is a program, which takes values a_n $[n = 1..m]$ as an input
- mix is a program which is defined as $mix\ [P, a_1] = P_a$
- $[\![P]\!][a_1, a_2, .., a_m] = [\![P_a]\!][a_2, .., a_m]$

Then the transformation of P and a_1 to P_a using mix is called *partial evaluation* [1]. Program mix is called *partial evaluator*. In other words, partial evaluation is a technique for evaluating parts of the program ahead of compilation with the usage of static input data.

A classic example of partial evaluation is the evaluation of power function. Code with linear complexity from Listing 1 could be partially evaluated using the knowledge of static power. So, assuming $n = 5$ code with constant complexity on Listing 2 could be received.

```
fn power(x, n):
   if x == 1:
      x
   else:
      x * power(x, n - 1)
```

Listing 1. Power function before evaluation

```
fn power5(x):
   x*x*x*x*x
```

Listing 2. Power function partially evaluation using n = 5

Despite partial evaluation is initially being used by Ershov [5], Jones [1] and other scientist in their work for compiler generation via Futamura projections [6], it could also be used for program optimization. For instance, a partial evaluator can

employ static data to unfold loops and conditional operators, propagate constants, etc [1].

However partial evaluation is a powerful method of program optimization, it is inherent in several difficulties. Firstly, a partial evaluator could inflate source code size heavily because of transformation such as loop unfolding and static data substitution. Therefore, evaluation results (code structure, bottlenecks, etc.) formal assessment becomes a non-trivial problem very often. To solve this issue in some degree, modern tools like AnyDSL [2] tends to translate evaluated code into some intermediate representation which is often much easier to understand and analyze. Secondly, divergent program partial evaluation with the application of average-quality tool may lead to the evaluation process divergence [1]. So, the programmer has to be very careful while using this technique for optimization purposes. Finally, partial evaluation imposes serious requirements on the programmer qualification: a deep understanding of the evaluation process is highly required. To solve the issue modern tools are introducing simplified language constructions, such as a special partial evaluation wrapper [2], attribute-driven evaluation [7] and many other various and creative methods.

B. Matrix algorithms

Algorithms on matrices (matrix-matrix, matrix-vector multiplication, tensor product, etc.) are very common in programs connected with linear algebra and linear algebra packages (BLAS).

For example, it is widely known that many graph algorithms could be explained in the language of matrices [8], [9]. Linear algebra allows constructing algorithms like Breadth-First Search or Shortest Path Search with the exploitation of basic linear algebra operations: matrix multiplication, Kronecker product and some other algorithms. Therefore, if it was possible to speed up different matrix multiplication algorithms, it should be possible to speed up a large class of algorithms.

One of the possible basic sets of linear algebra algorithms and operations for graph algorithm construction is named *GraphBLAS* standard [9], [10]. For our experiments we chose *matrix-matrix multiplication*, and *Kronecker (tensor) product* algorithms [11], which are considered as one of the core algorithms in GraphBLAS standard.

The results of matrix algorithms benchmarking are compared with the execution time of these algorithms implemented with SuiteSparse GraphBLAS library [10], which is usually considered as the state-of-art implementation of GraphBLAS.

Before the experiments we predicted in theory that both of these algorithms could be successfully (without at least noticeable lose in performance) partially evaluated due to linear structure of their code and a relatively small number of conditional jumps in the proper implementation.

C. Algorithms on strings

Algorithms on strings are employed in different like regular expression handling or bioinformatics [12]. One of the most common algorithms in this area are *pattern matching* (substring search) and *regular expression* (automaton) matching [11], so we chose these algorithms for our experiments.

Also, a pattern matching algorithm is often utilized in so-called KMP-Test [1]. It shows how effectively the partial evaluator could optimize trivial substring search algorithm measured in a degree of efficiency approximation to Knuth-Morris-Pratt algorithm execution time. Therefore, partial evaluation of this algorithm could give us the essential data for the analysis.

We used adjacency matrices as regular expression automata representation since the matrix-based regular expression matching algorithm's code is in a more linear form, so we predict it should be evaluated better.

Existing results [1] shows that both of these algorithms could be successfully partially evaluated and AnyDSL tool is considered by us as a "good enough" tool to show this theoretically good result in practice.

III. Algorithms implementation

All algorithms were implemented using AnyDSL Impala domain-specific language [2] for partial evaluation. AnyDSL framework was chosen due to its Impala DSL with comparatively simple Rust-like syntax and relatively available documentation. Algorithm code is represented as computation kernels, which is further linked with Google Benchmark-based [13] benchmarking code. Each algorithm was implemented in Impala twice: with partial evaluation language constructions and without them (therefore, with no partial evaluation).

Also, every algorithm was implemented with an alternative tool or framework that is usually used in practice for algorithm implementation in the corresponding area. In details, the following programs were used:

- SuiteSparse GraphBLAS [10] — for graph algorithms in the terms of linear algebra
- Grep and eGrep — for algorithms on strings and regular expressions

All the code is placed on GitHub:

https://github.com/ibalashov24/spec_experiments

IV. Experimental design

In this section, our experimental design for partial evaluation of selected algorithms using AnyDSL framework is described.

A. Experimental setup

Configuration of the experimental stand was:

- Intel Core i5-7440HQ (4x3.8GHz) CPU
- 16Gb RAM
- Ubuntu 20.04

Tools' versions were fixed on the following commits from their official repositories:

- Google Benchmark [13] — commit dated 22 December 2020
- AnyDSL [2] — commit dated 8 December 2020

- SuiteSparse GraphBLAS [10] — commit dated 14 July 2020

Default (e)Grep from Ubuntu 20.04 was employed.

We used SuiteSparse matrix collection [15] (and mostly the subset of it named Harwell-Boeing matrix collection [14]) because it contains a reasonably diverse set of matrices. COO (COOrdinate list) sparse matrix format was used.

In details, we got (both for matrix-matrix product and Kronecker product) sparse matrices presented in Table I.

	size	nonzero	symmetry, %	values
bcsstk16	4884	147631	100	real
fs_183_1	183	1069	41.8	real
can_256	256	2916	100	binary
eye3	3	3	100	binary
2blocks	4	8	100	binary
cover	8	12	16.67	binary
mycielskian3	6	5	0	binary
trec5	8	12	0	real

TABLE I
MATRICES USED IN THE PARTIAL EVALUATION EXPERIMENTS WITH ITS PARAMETERS

The First 3 of these matrices have a relatively big size, so they were used on the left side of the product (matrix-matrix and Kronecker) operator. Others were employed as static data during partial evaluation to prevent output code ramification and performance overhead creation since they are characterized by a relatively small size. Also, *bcsstk16* and *eye3* represents boundary cases, where the elements are concentrated near the main diagonal. To sum up, the selected matrices and their combinations represent a diverse class of matrices with different sizes, value types, symmetry percentages and boundary cases. So, the results presented in the current paper could be assumed as accurate for a much larger number of data.

For string algorithms, we used random strings and traffic dumps as sources and random strings or Latin words as patterns in order to show the results in a near-average case. Regular expressions (finite automata) were converted to COO sparse representation with our modification of Re2dfa tool [16]. Our benchmark used "short" strings with the length smaller than 200 characters as static data to avoid generated code ramification and overhead creation.

AnyDSL partial evaluation tool was executed in JIT-mode [2], which allows to perform partial evaluation at the run time.

B. Research questions

To evaluate our approach, we design experiments to address the following research questions:

Q1: Does partial evaluated benefits string and matrix-based graph algorithms performance (execution time) comparing to their basic versions?

Q2: In which degree partially evaluated algorithms code performance gets closer to their state-of-art implementations?

C. Result metrics

To evaluate the performance of partially evaluated code, we adopt the following widely used metrics for application performance:

- **Execution time** is computed by the Google Benchmark tool and measured in nanoseconds. For each algorithm, the tool gives three numbers: time spent in real life, time spent on CPU, and iteration number. We took *time spent in real life* to consider all hardware delays (for example, memory access delays).
  The smaller execution time is better.
- **Measure error** is computed by the Google Benchmark tool and measured in percents.
  Numbers smaller than 0.01% are considered a good result which guarantees a relatively small threat to validity.

V. RESULTS

This section presents our experimental results by addressing the research questions.

A. Does partial evaluation with AnyDSL benefits string and matrix-based graph algorithms performance comparing to their basic versions?

As seen from Tables II and III and Figure 1, partial evaluation gives significant, up to several times, speed up on test cases involving *2blocks*, *cover*, *mycielskian3* and *trec5* as right multiplicator. It may be explained with relatively distributed structure of these matrices non-zero elements, that allows partial evaluator to effectively perform optimizations like loop unfolding and constant propagation.
In contrast, non-zero elements of *eye3* matrix are concentrated near the main diagonal of the matrix which leads to relatively small execution time benefit (or absent) of partial evaluation — loop unfolding does not discard any empty iterations. We can also notice that the fact *bcsstk16* matrix elements are concentrated near the main diagonal does not affect execution times, because the partial evaluator can not make use of this fact in the dynamic matrix.

For string algorithms, we may observe a much more noticeable execution time increase after a partial evaluation than in graph algorithms. As could be seen from Table IV and Table V, the speed up lays between 10 and 100 times depending on the test. The reason for such a significant increase is that the most of iterations in classic substring search and pattern matching algorithms [11] with matrix input is not empty, like in previously discussed algorithms on sparse matrices graphs. Also, in substring search algorithm evaluation is simplified by the fact that the data is being iterated successively.
Moreover, there is an absent of non-logical operations with both source and pattern data as operands in these algorithms, so the partial evaluator can apply constant propagation optimization heavily due to trivial data separation.

The results show that in general partial evaluation with AnyDSL benefits string and matrix-based graph algorithms execution time comparing to their basic versions.

Time,ns (Spec No Spec SuiteSparse)	× *eye3*	× *2blocks*	× *cover*	× *mycielskian3*	× *trec5*
bcsstk16 ×	93608 121855 2270	133434 157850 7064	364772 4842889 8559	171085 2129094 511	308535 5226893 505
fs_183_1 ×	7796 6752 2553	20187 42353 12310	6928 38250 9796	1358 15194 506	6078 42493 507
can_256 ×	1016 1177 2259	5106 38221 6549	20339 66987 9409	2561 23105 503	9548 62668 506

TABLE II
EXECUTION TIME COMPARISON OF MATRIX MULTIPLICATION ALGORITHM NON-SPECIALIZED CODE (NO SPEC), SPECIALIZED CODE IN ANYDSL IMPALA (SPEC), CODE IMPLEMENTED WITH MODEL TOOL (SUITESPARSE)

Time,ns (Spec No Spec SuiteSparse)	⊗ *eye3*	⊗ *2blocks*	⊗ *cover*	⊗ *mycielskian3*	⊗ *trec5*
bcsstk16 ⊗	140628 140744 901878	276222 3032308 2145104	433397 4307538 4420688	276433 1967189 2958016	481805 4571625 1440326
fs_183_1 ⊗	916 934 25833	2186 21272 45159	3046 31732 88847	1838 14533 35109	3146 34356 47912
can_256 ⊗	1159 1069 35162	2772 30841 60600	4512 45731 130084	2736 22079 43479	4576 49512 61500

TABLE III
EXECUTION TIME COMPARISON OF KRONECKER (TENSOR) PRODUCT ALGORITHM NON-SPECIALIZED CODE (NO SPEC), SPECIALIZED CODE IN ANYDSL IMPALA (SPEC), CODE IMPLEMENTED WITH MODEL TOOL (SUITESPARSE)

Time, ns	Big source 1	Big source 2	Small pattern
No spec	33129202	28983335	2413
Spec	11757516	11686216	922
Grep (approx.)	24000000	50000000	1000

TABLE IV
EXECUTION TIME COMPARISON OF PATTERN MATCHING NON-SPECIALIZED CODE (NO SPEC), SPECIALIZED CODE IN ANYDSL IMPALA (SPEC)
AND APPROXIMATE TIME (THE TOOL OUTPUT IS IN INTEGER MS) FOR CODE IMPLEMENTED WITH MODEL TOOL (GREP)

B. In which degree partially evaluated algorithms code performance gets closer to their state-of-art implementations?

Tables II, III, IV and V show the time (in nanoseconds) of execution of matrix-based graph and string algorithms respectively.

For the string algorithms, we can see that partially evaluated code outperforms Grep (for pattern matching) and eGrep (for regular expressions matching) in several times (2 to 10000) on each of the datasets. However AnyDSL beat (e)Grep in both pattern and regular expression matching problems, we could see that the latter gave by several orders of magnitude stronger results. According to our analysis, it could be the result of using COO representation for regular expression's transition graph in the experiment: linear structure of a COOrdinate list structure allows the partial evaluator to use more aggressive optimizations such as vectorization or easier loop unfolding.

For graph algorithms in a matrix form (matrix multiplication and Kronecker product), we may observe that partially evaluated algorithms' code underperforms code of the same algorithms implemented with SuiteSparse GraphBLAS in 10 times in average. It could be considered a good result, since

Time, ns	Email (weak)	Email	Credit card
No Spec	1273112850	1951322141	1824
Spec	1623176	2223887	23.2
eGrep (approx.)	118945000000	174746000000	69000000

TABLE V
EXECUTION TIME COMPARISON OF REGULAR EXPRESSION (AUTOMATA) SEARCH NON-SPECIALIZED CODE (NO SPEC), SPECIALIZED CODE IN ANYDSL IMPALA (SPEC) AND APPROXIMATE TIME (THE TOOL OUTPUT IS IN INTEGER MS) FOR CODE IMPLEMENTED WITH MODEL TOOL (EGREP)

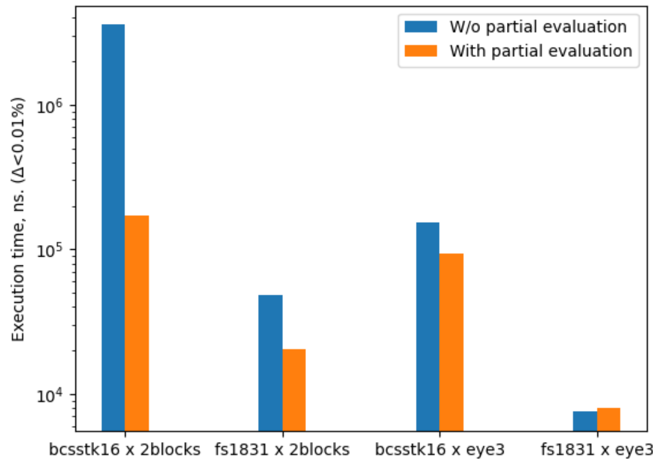

Fig. 1. Comparison of matrix-matrix multiplication algorithm execution times before and after partial evaluation for some matrices

non-partially evaluated code loses 2 orders in half of cases.

To sum up, for the selected string algorithms partially evaluated code outperforms their industrial implementations by execution time in a high degree; for the selected graph algorithms in matrix form partially evaluated code lags behind their state-of-art implementation by a factor of 10 (which could be considered as a good result for a semi-automatic optimization).

C. Conclusion

As a result, partial evaluation of several matrix and string algorithms usually used as core algorithms in different programs or libraries shows relatively good results. So, we could conclude that partial evaluation (at least, with AnyDSL framework) could be successfully applied as a helper technique for a programmer, who intends to automatically optimize algorithmic code in high-loaded systems.

VI. THREATS TO VALIDITY

A. Subject selection bias

In our research, we use only AnyDSL framework for the experiments. Other partial evaluation tools may give slightly different results due to more or less aggressive optimizations or different evaluation techniques.

B. Used datasets

Despite trying to run experimental code on both versatile and special datasets, we admit that partially evaluated code could give slightly different measures on some other special degenerate matrix sets.

VII. RELATED WORK

Partial evaluation of linear algebra (especially matrix algorithms) was studied before in several papers.

Firstly, it is measured [17] that partial evaluation of matrix convolution and pattern matching algorithms using AnyDSL framework and CUDA reduces execution times up to 8 times on most datasets.

Secondly, partial evaluation was applied for Viterbi algorithm optimization [18]. It was discovered that the evaluated version of the code outperforms the non-evaluated one by 1.5 times in some cases.

Moreover, AnyDSL team performed research [4] on the application of partial evaluation for ray tracing purposes in their library named Rodent. It was measured that partial evaluation makes an improvement in execution time of around 25% on selected datasets.

Also, several other partial evaluators could be used instead of AnyDSL for matrix and string algorithms code optimization. For instance, it could be LLVM.mix, which was successfully applied for database query optimization [19], or C-mix [1].

VIII. FUTURE WORK

In short term, we are planning to set up the experiments on more complex algorithms: shortest path algorithm and breadth-first search.

Also, there is an interesting task to translate our experiments' code into a new AnyDSL frontend language named Artic [20]. It allows parametric polymorphism, so it should be possible to implement semirings support, which is essential for algorithms on graphs in matrix representation.

REFERENCES

[1] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[2] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "Anydsl: A partial evaluation framework for programming high-performance libraries," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[3] A. Müller, B. Schmidt, A. Hildebrandt, R. Membarth, R. Leißa, M. Kruse, and S. Hack, “Anyseq: a high performance sequence alignment library based on partial evaluation,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 1030–1040.
[4] A. Pérard-Gayot, R. Membarth, R. Leißa, S. Hack, and P. Slusallek, “Rodent: generating renderers without writing a generator,” *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.
[5] A. P. Ershov, “Mixed computation: Potential applications and problems for study,” *Theoretical Computer Science*, vol. 18, no. 1, pp. 41–67, 1982.
[6] Y. Futamura, “Partial computation of programs,” in *RIMS Symposia on Software Science and Engineering*. Springer, 1983, pp. 1–35.
[7] E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher, “Runtime specialization of postgresql query executor,” in *Perspectives of System Informatics*, A. K. Petrenko and A. Voronkov, Eds. Cham: Springer International Publishing, 2018, pp. 375–386.
[8] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.
[9] T. A. Davis, “Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.
[10] J. E. Moreira, M. Kumar, and W. P. Horn, “Implementing the graphblas c api,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 298–309.
[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
[12] S. Rajesh, S. Prathima, and L. Reddy, “Unusual pattern detection in dna database using kmp algorithm,” *International Journal of Computer Applications*, vol. 1, no. 22, pp. 1–7, 2010.
[13] Google. Google benchmark. Accessed on 28 March 2021. [Online]. Available: https://github.com/google/benchmark
[14] I. S. Duff, R. G. Grimes, and J. G. Lewis, “Users’ guide for the harwell-boeing sparse matrix collection (release i),” 1992.
[15] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.
[16] ”re2dfa: convertor from regular expressions to graphs”. Accessed on 28 March 2021. [Online]. Available: https://github.com/ibalashov24/re2dfa
[17] A. Tyurin, D. Berezun, and S. Grigorev, “Optimizing gpu programs by partial evaluation,” in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 431–432.
[18] I. Tyulyandin, D. Berezun, and S. Grigorev, “Viterbi algorithm specialization using linear algebra,” *Accepted on SEIM21, to be appear in official proceedings*, 2021.
[19] E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher, “Runtime specialization of postgresql query executor,” in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2017, pp. 375–386.
[20] A. Team. Anydsl artic. Accessed on 28 March 2021. [Online]. Available: https://github.com/AnyDSL/artic