Using Performance Event Profiles to Deduce an Execution Model of MATLAB with Just-In-Time Compilation

Patryk Kiepas[1][0000−0003−4008−5822], Corinne Ancourt[1][0000−0002−3310−7651], Claude Tadonki[1][0000−0003−1194−6400], and Jarosław Koźlak[2][0000−0002−4744−9433]

[1] MINES ParisTech/PSL University, Paris (FRANCE)
{patryk.kiepas, corinne.ancourt, claude.tadonki}@mines-paristech.fr,
[2] AGH University of Science and Technology, Kraków (POLAND)
kozlak@agh.edu.pl

Abstract. The knowledge about how an interpreter executes programs allows writing faster code and creating powerful source-to-source compilers. However, many languages and environments either lack a specification of the execution semantics, or the semantics frequently changes with new releases. In this article, we present (1) *performance event profiles* with *execution regions*, inventive use of event-based sampling on processors where we correlate profiles of many performance events to find regions with desired properties. Furthermore, we use the *performance event profiles* to deduce (2) a static, tree-based *execution model* of MATLAB with the Just-In-Time (JIT) compilation. The environment of MATLAB is closed-source, which makes it a perfect testbed for our approach. With the new model and better understanding of how MATLAB executes programs, we propose a new code transformation, (3) *repacking of array slices*, which can increase the amount of JIT-compiled code in MATLAB programs.

Keywords: Hardware performance counters · Event-based sampling · Performance event profile · MATLAB · Just-In-Time compilation · Execution model · Repacking of array slices.

1 Introduction

Listing 1.1 depicts how MATLAB programmers express element-wise computations as loops or array operations (with or without array slicing). All three codes perform the same computation producing the same result. However, they use a different type of floating-point operations, execute different number and kind of CPU instructions, and, more importantly, have vastly different performance.

Listing 1.1: Three versions of the *striad* kernel (the Schönauer Vector Triad). In MATLAB, it is common to express loops as array operations (loop vectorisation).

```
% Scalar loop code (loop)
for k = 1:N
  a(k) = b(k) + c(k) .* d(k);
end
```

```
% Vector code with array slicing (vec)
a(1:N) = b(1:N) + c(1:N) .* d(1:N);
% Vector code on whole arrays (vec_01)
a = b + c .* d;
```

To find out about what kind of instruction codes from Listing 1.1 execute, we could use dynamic binary instrumentation with tools, e.g. DynamoRIO [2] or Intel PIN [11]. However, MATLAB is a closed-source environment with no linking symbols available. This closed nature of MATLAB makes even the task of matching machine code to a specific language component (e.g. garbage collector, JIT compiler, external libraries) hard. Moreover, the machine code of the MATLAB environment is mixed up with the dynamically generated code created by the JIT compiler.

Instead, we could analyse how codes from Listing 1.1 execute using popular metrics and performance model. For example: the Roofline Model by Williams et al. [20] which graphically shows if the code is memory or compute-bound; the Top-Down μ-architecture Analysis Method (TMAM) by Yasin [21] which indicates which part of the processor pipeline is the execution bottleneck with a detailed 4-level hierarchy of 33 metrics; or to use metrics like cache miss ratio or cycles per instructions (CPI) [10]. However, at the core, metrics and models mentioned above do not consider how the program execution changes over time, which is the key to successful analysis of MATLAB programs.

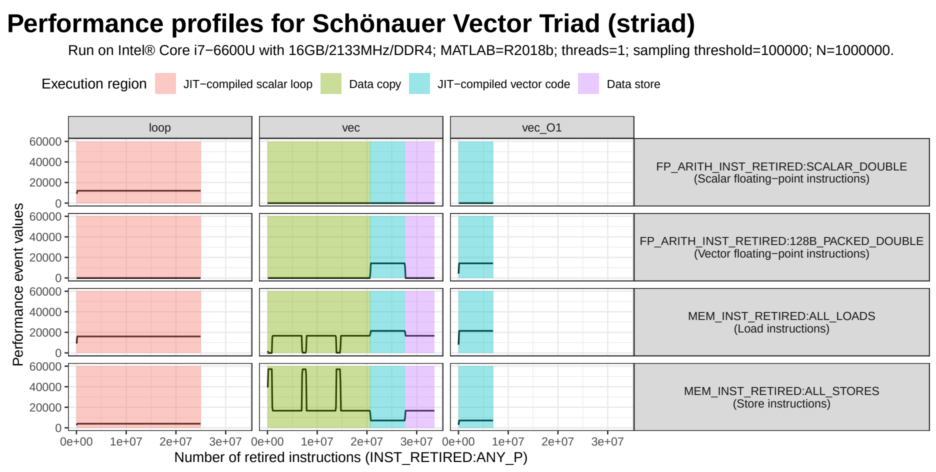

Fig. 1: Performance profiles of three versions of the *striad* kernel: loop computation (`loop`); vectorised operations with array slicing (`vec`); vectorised operations on all data (`vec_01`). The figure depicts four distinctive regions: JIT-compiled scalar loop, data copy, JIT-compiled vector code, and data store.

Figure 1 shows how the program execution of the three codes changes as more instructions on the processor retire. The graph depicts several *performance event profiles* built with the Event-Based Profiling (EBS) [16] of four performance events: scalar and vector floating-point operations, load and store instructions (from top to bottom). By aligning the profiles together, we find *execution regions* with particular properties, e.g. data copy or computation. Moreover, we can draw several observations from Figure 1: (1) loops perform scalar arithmetic instructions; (2) array operations perform vector arithmetic instructions (e.g. with Intel SSE, AVX extensions); (3) array slicing requires a copy of the data; (4) loop and vector codes without array slices are perfectly regular.
Throughout this paper, we use the two concepts of *performance event profiles* and *execution regions* to discover rules of program execution in the MATLAB environment. We analyse built-in and user-defined functions, common arithmetic operators, array referencing and slicing, and the impact of the JIT compilation. Finally, we build a simple execution model of MATLAB programs from which we can derive promising code transformations.
In the paper, we make the following contributions:

– We describe *performance event profiles* built using hardware event-based sampling on processors (Section 2). The profiles consist of values of several performance events sampled over time. With a notion of *execution regions* in the profiles, we can find parts of programs with desired properties, e.g. data copy or floating-point computation.
– We briefly present our open-source tool `mPAPI`[3], which gives access to hardware performance counters from the MATLAB/Octave programs (Section 2). The tool uses the `PAPI` library [18] and supports two measurements modes: *counting* and *sampling*.
– Using *performance event profiles*, we deduce and build an *execution model* for MATLAB expressions in the presence of JIT compilation (Section 3). The proposed model takes source-code of expression and predicts how MATLAB executes it. Furthermore, the model uses easy-to-follow graphical tree structure, called *(minimal) instruction tree*, obtained directly from an abstract syntax tree (AST) form of this expression.
– Finally, with the knowledge obtained from the execution model, we propose a new code transformation, *repacking of array slices*, which increases the amount of MATLAB instructions that the JIT compiler executes together (Section 4).

Complementary materials including more experiment results and the implementation of the model are available online[4].

2 Performance Event Profiles

Performance event profiles describe the change in performance events measured over time. Fortunately, hardware performance counters in modern CPUs work

[3] https://github.com/quepas/mPAPI
[4] https://github.com/quepas/lcpc19-execution-model

in two modes: (1) *counting mode* where the values of a performance event accumulate in a single register; and more importantly (2) *sampling mode* where a program interruption occurs and a measurement is taken every time a sampling event used as "time" reaches a specified *sampling threshold.* Therefore, the capabilities of building performance profiles are already built-in inside modern CPUs with mechanisms of *counter overflow*, Event-Based Sampling (EBS), and recently Precise Event-Based Sampling (PEBS) on Intel processors [16,4].

2.1 Concept Definitions

Performance Event Profile. We define a performance profile as $P = (T, M)$ where $T = (t_1, t_2, \dots, t_l)$ and $M = (m_1, m_2, \dots, m_l)$ are sequences of the sampling event and measurement values such that $t_i, m_i \in \mathbb{N}$, and the sampling event values are strictly increasing $\forall i < j : t_i < t_j$. Both T and M have the same length l, and the length of the profile is denoted by $|P| = l$. Moreover, T and M have their *event domain* $\mathcal{E}$, which states what performance event the sequence represents. The universe of event domain $\mathcal{E}$ contains any performance event available on the CPU. For example, a sequence M of *L1 cache misses* measurements has the event domain as follows $\mathcal{E}(M) =$ `L1D:REPLACEMENT`. The event domain of the performance profile $\mathcal{E}(P) = (\mathcal{E}(T), \mathcal{E}(M))$ with sequences T and M creates a full description of the profile. Event domains specify the profile content, where M holds the measurements and T stores values of the sampling event.

Profiles Group. Building only one performance profile at a time would result in poor resource management, as the rest of hardware performance registers are unused. Therefore, it is beneficial to measure several performance profiles at once, creating a *group* $G = \{P_1, P_2, ..., P_g\}$ of g profiles. In a machine with N hardware performance registers, we can create only $g = N - 1$ profiles, because the last register measures the sampling event. Moreover, profiles in a group are *aligned* because they are measured together; thus, we can simplify the group definition to $G = (T, \{M_1, M_2, \dots, M_g\})$, where the profiles and their measurements share the sampling event from T. As in the case of performance profiles, groups have their event domain $\mathcal{E}(G) = (\mathcal{E}(T), \{\mathcal{E}(M_1), \mathcal{E}(M_2), \dots, \mathcal{E}(M_g)\})$.

Execution Region. A section of the program execution with particular properties is an *execution region* expressed as a binary predicate $\varphi : T \to \{0, 1\}$. The predicate marks the desired region in the domain of the sampling event. Thus, the predicate indicates when the region starts and ends during the program execution. Moreover, the data copy region indicates either the occurrence of array slicing or at least where program execution has properties similar to the data copy. In the construction of the predicate φ, we use only aligned measurements coming from the same profiles group. Otherwise, ambiguities can appear, e.g. when two performance profiles P_1 and P_2 have different length $|P_1| \neq |P_2|$. Nevertheless, it is possible to use performance profiles from two or more groups after aligning them using, e.g. dynamic time warping (DTW) [7].

2.2 Performance Profiles with mPAPI

Apart from manually programming hardware performance counters, several libraries give easy access to both modes of measurement: counting and sampling. In our work, we have focused on the `PAPI` [18], because it is a comprehensive, open-source, up-to-date, actively maintained library with a C API which makes it possible to integrate with MATLAB through C MEX API[5]. In order to access performance event directly from MATLAB source-code and to mitigate the possible imprecisions and overheads of measuring hardware performance counters [19,22], we have built `mPAPI`, an open-source tool for MATLAB. `mPAPI` supports counting and sampling modes of collecting performance events, along with multiplexed and per-thread measurements.

3 Execution Model for JIT Compilation in MATLAB

Without a Just-In-Time (JIT) compiler, MATLAB would be an interpreter which executes (interprets) instructions step by step. The interpreter fetches, decodes, and executes each instruction in isolation without any knowledge about future instructions. However, with the JIT compiler, MATLAB can defer execution of the instruction as long as possible (in the APL interpreter, this concept is known as *drag-along* [1]). This delay often creates new optimisation opportunities for the JIT compiler, because the compiler carries information about future instructions. The opportunity leads to better instruction scheduling, register allocation and code optimisations, especially involving the execution of two or more instructions which can execute together. In this section, we prepare a model describing when the JIT compiler optimises code regions consisting of many instructions.

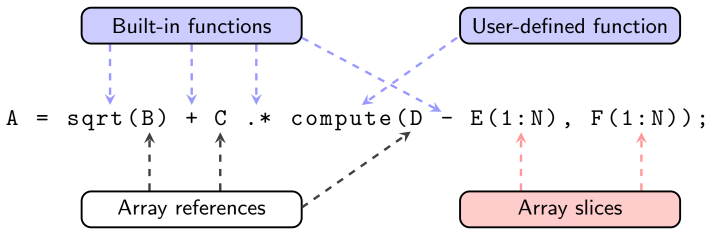

Fig. 2: Common components of expressions with array operations in MATLAB.

Scope of the Model. We focus on expressions consisting of four components depicted in Figure 2: (1) built-in functions; (2) user-defined functions; (3) array references; and (4) array slices. Our model aims at expressing these components and their order of execution.

[5] https://www.mathworks.com/help/matlab/call-mex-files-1.html

Instruction Block. Instead of compiling only single instructions, the JIT compiler can compile whole blocks of instructions, thus, benefiting from new optimisation opportunities. We define the *instruction block* Γ as a program segment containing one or more instructions $\Gamma.inst = \{\gamma_1, \gamma_2, \ldots\}$. Each instruction γ_i indicates a call to the MATLAB built-in or user-defined function. Array slicing is expressed separately, because it always performs the same operation, a data copy. Furthermore, a single block can contain multiple calls to the same function (thus, $\Gamma.inst$ is a multiset).
All instructions from the block are scheduled together for the execution by the JIT compiler. Therefore, we say they are *JIT-compiled*, which means, the JIT compiler is responsible for fetching, decoding, optimising, and scheduling them together for the execution. However, we are not concerned about their order, because in many cases, the original execution order would be hard to deduce and highly dependent on the compiler, compilation heuristics, and the target machine.

Detecting Instruction Blocks. The result of the JIT compilation of an instruction block is a machine code stored in the instruction cache on the processor. Before the execution, the processor fetches and decodes this machine code into micro-operations (μops), executable by the processor. The fetch-decode phase results in several observable performance events on the processor, such as instruction cache miss (`L2_RQSTS:CODE_RD_MISS`) and hit (`L2_RQSTS:ALL_CODE_RD`). Therefore, an activity of the performance events related to the fetch-decode phase could indicate the beginning and end of the instruction block execution.

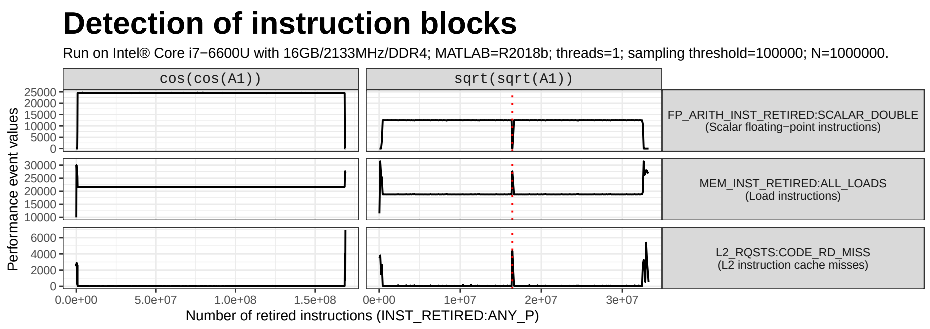

Fig. 3: Code examples generating one and two instruction blocks. In this example, `cos` is a combinable function.

Figure 3 presents how the activity of the L2 instruction cache misses indicates boundaries of instruction blocks. The first code `cos(cos(A1))` is JIT-compiled and executed as one instruction block because we observe the activity of the cache misses only at the beginning and end of the computation. The second code `sqrt(sqrt(A1))` is also JIT-compiled; however, the code is divided into two instruction blocks marked by the spike of cache misses in the middle of the computation. The existence of two instruction blocks indicates here that the `sqrt` function is not *combinable* with itself and requires one instruction block for each call. In the next paragraph, we use the concept of *combinable function* to find which functions can coexist with others in the same instruction block. The execution of fewer instruction blocks resulting from the use of combinable functions can improve program efficiency.

Combinable Functions. Instruction blocks can potentially contain any combination of functions which make it infeasible to test them all. Therefore, we propose to use three simple detection patterns in Table 1, to assess if a given function is *combinable*, meaning, it can coexist with other functions in the same instruction block. Every pattern in Table 1 is a composition of at least two functions because only then we can observe if the JIT compiler creates two instruction blocks for one expression and if the function is combinable.

Table 1: Three detection patterns used for finding dynamic (JIT) compilation of MATLAB functions. The presented patterns use built-in functions, `cos` and `atan2`.

Detection pattern	**Unary**	**Binary**
Self-composition	`cos(cos(A1))`	`atan2(atan2(A1, A2), A3)`
Composition `plus:+`	`cos(A1)+cos(A2)`	`atan2(A1, A2)+atan2(A3, A4)`
Arguments `plus:+`	`cos(A1+A2)`	`atan2(A1+A2, A3+A4)`

Table 1 presents code patterns which we use to detect if given built-in functions (unary and binary) can be a part of a multi-instruction block. The first pattern is a self-composition `f(f(A))`, which is the simplest way for one function to create a complex expression. The next two patterns compose function `f` with addition operator +/`plus`, which is JIT-compiled (we have verified this using the same approach as described). The second pattern `f(A)+f(B)` tests if the `plus` composes well with functions `f` as its arguments. Finally, the third pattern `f(A+B)` validates if the function `f` executes with a complex expression as its argument inside a single instruction block.
Using these three patterns from Table 1, we have executed various built-in functions to find out several *combinable functions*: `abs`, `ceil`, `cos`, `exp`, `floor`, `ldivide=.\`, `minus=-`, `plus=+`, `rdivide=./`, `round`, `sin`, `tan`, `times=.*`, `transpose`, and others. The list of non-*combinable functions* includes `acos`, `cumsum`, `diff`, `fft`, `log`, `mtimes=*`, `power`, `prod`, `sqrt`, `sum`, and much more. Furthermore, although not shown in the paper, the user-defined functions are never *combinable*, and they evaluate just like non-combinable built-in functions do.

3.1 The Minimal Instruction Tree Model

The knowledge about (non-)combinable functions and the content of particular instruction blocks lack the information about the execution order of these blocks. Furthermore, because instruction blocks do not express array slicing, we would also like to track when the slicing appears. For this task, this section introduces instruction trees.

Instruction Tree. Similar to the abstract syntax tree (AST), the *instruction tree* represents instructions and usages of variables. However, unlike in the AST, a single inter-node in the instruction tree can represent several instructions enclosed inside an instruction block. Steps 1 and 2 in Figure 4 visualise the difference between the AST and the corresponding instruction tree. The initial translation from the AST to the instruction tree is straightforward, as it only requires a one-to-one mapping of each instruction to ● *instruction block* node; each array reference to □ *array reference* leaf; and each array slice to 🟥 *array slice* leaf. For simplicity, and because it does not generate any execution regions, we remove the leaf of the reference `A3` and instead, we store it in the corresponding instruction block (grey set in step 3 in Figure 4).

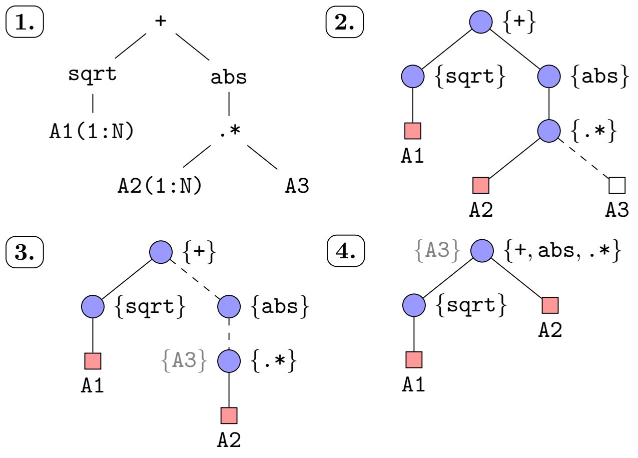

Fig. 4: Steps required for obtaining the minimal instruction tree: (1) conversion to the AST; (2) one-to-one translation of the AST to the instruction tree with the removal of array references; (3) application of Algorithm 1; (4) the result minimal form of the instruction tree.

Minimal Instruction Tree. At this point, our instruction tree is in the maximal form, because each instruction has its block. However, the JIT compilation in MATLAB can combine such blocks. Algorithm 1 expresses this merging as a repetitive process which combines pairs of instruction blocks as long as there exists any pair of combinable blocks left. Therefore, the result of Algorithm 1 is a *minimal instruction tree* consisting of a minimum possible number of instruction blocks, such as the tree from step 4 in Figure 4. The resulting tree indicates the order of execution regions occurring during the execution of MATLAB expressions.

Algorithm 1 Building of a minimal instruction tree by the repetitive merging of instruction blocks inside an instruction tree.

```
Input: the root node of the initial instruction tree
Output: minimal instruction tree

 1: function canMerge($node$)
 2:     $correctInst \leftarrow node.inst \subseteq combinableFunctions$
 3:     return isInstructionBlock($node$)$\wedge correctInst$

 4: function buildMinimalTree($node$)
 5:     $revChildren \leftarrow$ reverseList($node.children$)          ▷ Visit from right-to-left
 6:     for $child$ in $revChildren$ do
 7:         buildMinimalTree(child)                              ▷ Recursive visit of the tree
 8:     if canMerge($node$)$\wedge$canMerge($node.parent$) then
 9:         if $\neg$hasRightSibling($node$) then
10:             $node.parent.inst \leftarrow node.inst \uplus node.parent.inst$
11:             attach($node.children$, $node.parent$)
12:             removeFromParent($node$)
```

Obtaining the Minimal Instruction Tree. The main function buildMinimalTree from Algorithm 1 is a recursive method which traverses the input instruction tree and finds candidates of instruction blocks for merging. The routine traverses the tree in a post-order, but with children visited right-to-left (lines 5–7). The reason for the reversed visit of children is the evaluation order of arguments in MATLAB, which is left-to-right. Hence, MATLAB evaluates expression `e1 + e2` starting with arguments `e1`, `e2` and finishing with the `+` operator which creates an evaluation sequence: `e1`, `e2`, `+`. With the standard post-order traversal and left-to-right visiting of children, we would never merge `e1` with `+` (even if possible), because the second expression `e2` stands on the way — `e2` evaluates in between `e1` and `+`. However, if we visit children right-to-left, then we could merge `e2` with `+` and give to `e1` an opportunity to combine with a newly created block of instructions {`e2`, `+`}. Therefore, the merge of a node with its parent is possible only when the node has none of the right siblings (line 9).

The presented perspective on merging nodes (instruction blocks) is related to the structure of the instruction tree, which encodes the evaluation order of MATLAB operations. However, the lack of right siblings node is not the only condition required for merging instruction blocks. The other condition, even more important, is if both instruction blocks contain only *combinable functions* which can be merged. The function canMerge from Algorithm 1 encapsulates the condition, and it is used in line 8. Moreover, the canMerge can work only on instruction blocks; hence, the use of the isInstructionBlock predicate. Finally, we merge two nodes by adding together the instructions from both merged blocks (line 10). The operation $\uplus$ is a *multiset sum*, which not only performs a union between two multisets, but it also adds repeated elements multiple times inside the result multiset. The subsequent operations rebuild the structure of the instruction tree by connecting child nodes (line 11) and removing one of the merged nodes (line 12).

Instruction Chain. Finally, we flatten the minimal instruction tree (step 4 from Figure 4) by traversing it in the post-order manner (left branch, right branch, root). The result is an *instruction chain* which represents the order of execution regions during the program run presented in Figure 5.

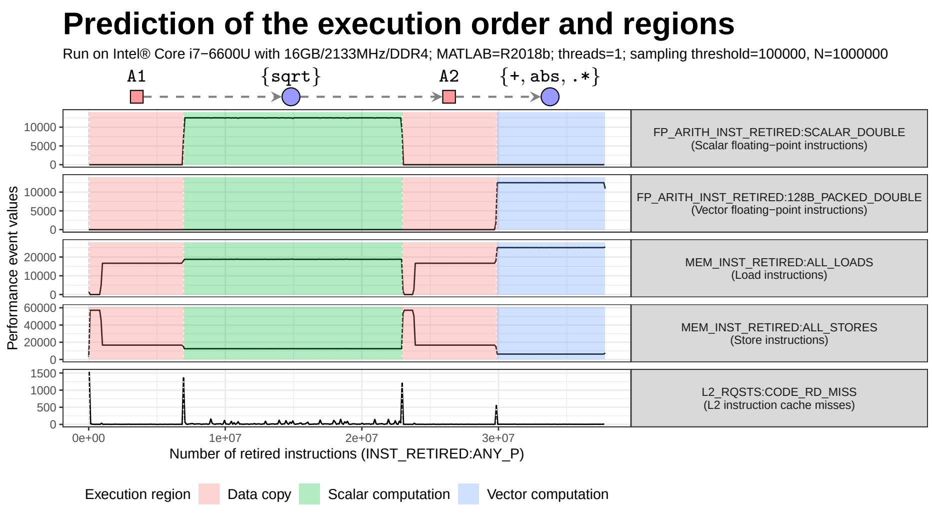

Fig. 5: Prediction of execution order and regions from the instruction chain.

4 Repacking of Array Slices

Array slicing affects the performance of MATLAB programs in two ways: (1) it creates a copy of the requested subset of an array; and (2) it sometimes prevents the JIT compiler from merging two or more operations into one instruction block. Usually, array slicing is necessary for programs, and we need to perform the copy at some point. However, it is possible to move the copying before the computation. Therefore, the computation uses only references to the already copied array slices, which allows the JIT compiler to merge and execute operations together. In this section, we show how to extract array slices and better schedule computations by repacking array slices into new variables.

Listing 1.2: Repacking of array slices on `crni3` loop from the LCPC16 suite[3].

```
1 % Original code
2 X(1:(N-1)) = (B(1:(N-1)) - C(1:(N-1)) .* X(2:N)) ./ D(1:(N-1));
3 % After repacking of array slices
4 tmp_b = B(1:(N-1));
5 tmp_c = C(1:(N-1));
6 tmp_x = X(2:N);
7 tmp_d = D(1:(N-1));
8 X(1:(N-1)) = (tmp_b - tmp_c .* tmp_x) ./ tmp_d;
```

Transformation. Listing 1.2 presents the simple idea behind the repacking of array slices. The transformation replaces every indexed read reference (array slice) in the right-hand side of the assignment (line 2) by a reference to a temporary variable `tmp_*` which holds the array slice (line 8). The code on lines from 4 to 7 depicts how array slices are repacked into new temporary variables `tmp_*`.

Application. The transformation is especially useful in cases where array slicing prevents the JIT compiler from executing operations together. For finding good candidates of the repacking, we propose to use our execution model, which predicts the order of computations in the expression before and after the repacking. Furthermore, the model works directly from the source code of the program and requires no prior execution (static model).
The knowledge about the order of computations after the repacking shows whether the repacking creates new instruction blocks, regions where many operations execute together. In general, the repacking yields two results: (1) no change to the order of computations; or (2) creation of new instruction blocks by merging other blocks. In the first case, the repacking does not improve program performance. However, the sole existence of new instruction blocks (2) is not sufficient to guarantee the performance improvement.

Results. Figure 6 presents the relative increase of the performance after the repacking of array slices measured on three kernels: `crni3` loop from LCPC16 [3], `state_fragment` (kernel 7) from Livermore [6], and `s211` from TSVC [12] benchmark suites.
One occurring pattern in the data from Figure 6 is a better performance of the repacking obtained on the newer version of MATLAB R2018b. A possible explanation of this is the improved JIT compiler. In that case, the repacking reveals a massive opportunity for the JIT compilation, which the results show.
The repacking for `TSVC:s211` kernel decreases the performance for the majority of tests. However, the execution model can predict this outcome. The result is

[6] https://www.netlib.org/benchmark/livermore

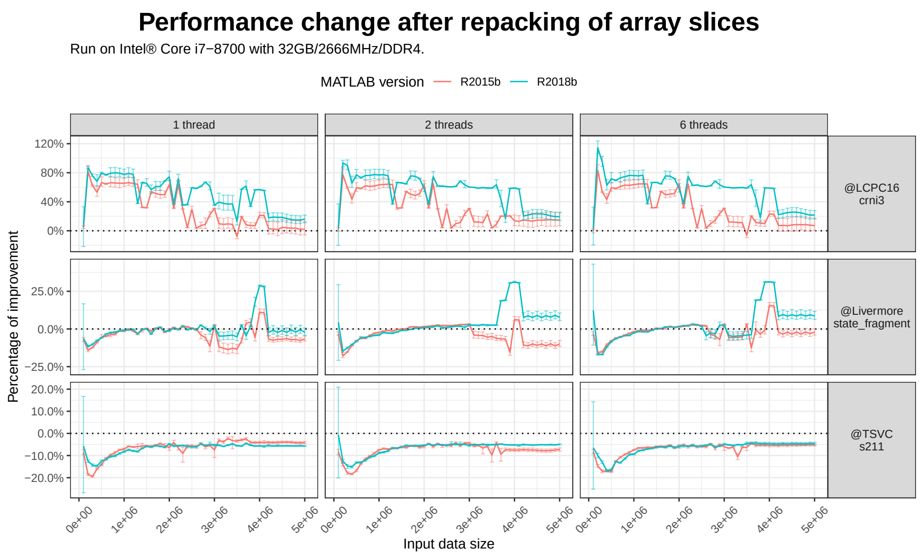

Fig. 6: Result of applying repacking of array slices. Not every computation benefits from the transformation (`TSVC/s211`).

the same instruction chain of the code before and after the repacking. In other words, the repacking is not profitable because it does not decrease the number of instruction blocks required for the execution.
Other two loops `LCLP16:crni3` and `Livermore:state_kernel` are perfect examples of how the performance gain from the repacking depends on the size of input data, the number of threads, the MATLAB version, and the code itself. In the current form, the execution model is insufficient to answer if the repacking increases the performance. Thus, the repacking should be considered with the code profiling to find out if the transformation is beneficial in the given context.

5 Related Works

In our work, we have utilised and combined various concepts, such as: analysing large-scale and time-varying behaviours, characterising programs with performance counters or vertical profiling. Nevertheless, we have not found works similar to ours. We suspect that the main reason for the lack of similar works is because the bulk of research work analysing programs behaviour deals only with open-source, freely available solutions. Therefore, the problems which we deal with in this paper are unnoticed, as the source code is provided.

Time-Varying Behaviour. One of the first studies of program properties changing through the program execution was a study of large-scale patterns in the SPEC95 benchmark suite by Sherwood and Calder [14]. The work looked for patterns in performance profiles of, e.g. instructions per cycles (IPC), cache miss rate or branch prediction miss rate, in terms of committed instructions. A new result of the study was finding *cyclic behaviours* in benchmarks which specify how long we should run the benchmarks to obtain representative results. Subsequent studies by Sherwood et al. improved the idea of time-varying, large-scale patterns by either creating an automatic machine-independent technique for finding large-scale pattern [15] or proposing a hardware (and software) tracking and prediction method for reoccurring phase behaviours [13]. Although our work deals with small scale time-varying changes and lacks phase behaviour, the work of Sherwood et al. is an early example of the detailed analysis of performance profiles. In another interesting study, Duesterwald et al. [5] argued that time-varying behaviours are essential and should be incorporated into adaptive systems to improve program performance and energy consumption. The results show that programs present time-varying behaviour at even small scale which could be used, e.g. to predict the value of one metrics based on another.

Characterising Programs with Performance Counters. The study by Eeckhout et al. [6] used hardware performance counters to analyse interactions between various Java Virtual Machines (JVM), processors, and programs. The results show that differences between JVMs implementations are more significant than differences from running various benchmarks on the same implementation. Similarly, to our study, Eeckhout et al. looked directly into the performance events to see how interpreted programs perform on processors. The use of traces with performance events (similar to traces from `mPAPI` tool) was visible in the study by Sweeney et al. [17] which described a methodology to analyse the performance of Jikes RVM, research virtual machine. The traces allow understanding better the interactions between various components of Java program execution: the application, virtual machine, operating system, and the μ-architecture. However, the authors noted that traces of performance counters are not enough to explain certain performance phenomena. Therefore, the work was just a prequel to *vertical profiling.*

Vertical Profiling. Across three papers, Hauswirth et al. [9,7,8] explored the idea of vertical profiling, a methodology for understanding and correlating performance data over time from multiple levels of abstractions: server, hardware, virtual machine (VM), operating system (OS), application. At the core, in correlating performance from multiple levels of abstraction lies the same idea as in our use of performance event profiles, however, in our case, we correlate multiple performance events coming from the same abstraction — a processor. In the second paper, authors evaluated several techniques for automated trace alignment coming from different measurements [7]. So far, in our work, we have used only a small amount of performance events which are measured at once. Nevertheless, in the future, we plan to consider trace alignment as well.

6 Conclusion

In the paper, we have described two concepts of *performance event profile* and *execution region* which originated from the need for describing program execution of the closed-source MATLAB environment (Section 2). The combination of *performance event profiles* with *execution regions* captures the time-varying behaviour of programs by tracking their execution directly on CPU. Moreover, we have presented our open-source tool `mPAPI` for accessing hardware performance counters in MATLAB/Octave programs and building performance profiles.
Furthermore, we have used *performance event profiles* and *execution regions* to build a tree-based execution model for expressions in the vectorised MATLAB programs (Section 3). The model predicts execution regions and their order for a given MATLAB expression. Moreover, the model highlights functions scheduled for the combined execution, by the JIT compiler (as *instruction blocks*).
Finally, with the knowledge about program execution in MATLAB, we have introduced a simple, yet powerful code transformation *repacking of array slices* (up to 80 % of performance increase on benchmark `LCPC16:crni3`; but when misused, the repacking decreases performance up to −20 %). However, for now, the transformation should be considered in the context of Profile-Guided Optimisations (PGO), until we further develop the execution model to predict the performance gain of the repacking.
The work presented in this paper is the first step to obtain an entirely usable execution model for MATLAB programs. The future work includes: (1) consideration of vector instructions; (2) prediction of parallel execution; (3) extending the model with control-flow; (4) quantifying the precision of the model and accuracy of measurements of performance events. Moreover, we plan to test the approach on other interpreters with JIT compilers: PyPy for Python[7] and Julia [8]. However, we do not plan to go beyond the domain of scientific computing.

References

1. Aycock, J.: A brief history of just-in-time. ACM Computing Surveys **35**(2), 97–113 (2003). https://doi.org/10.1145/857076.857077
2. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: International Symposium on Code Generation and Optimization, CGO 2003. pp. 265–275 (2003). https://doi.org/10.1109/CGO.2003.1191551
3. Chen, H., Krolik, A., Lavoie, E., Hendren, L.: Automatic Vectorization for MATLAB. Lecture Notes in Computer Science, vol. 10136 LNCS, pp. 171–187 (2017). https://doi.org/10.1007/978-3-319-52709-3_14
4. Doweck, J., Kao, W.F., Lu, A.K.y., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A., Yoaz, A.: Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. IEEE Micro **37**(2), 52–62 (2017). https://doi.org/10.1109/MM.2017.38

[7] https://pypy.org/
[8] https://julialang.org/

5. Duesterwald, E., Cascaval, C., Sandhya Dwarkadas: Characterizing and predicting program behavior and its variability. In: Parallel Architectures and Compilation Techniques, PACT 2003. pp. 220–231 (2003). https://doi.org/10.1109/PACT.2003.1238018
6. Eeckhout, L., Georges, A., De Bosschere, K.: How java programs interact with virtual machines at the microarchitectural level. ACM SIGPLAN Notices **38**(11), 169 (2003). https://doi.org/10.1145/949343.949321
7. Hauswirth, M., Diwan, A., Sweeney, P.F., Mozer, M.C.: Automating vertical profiling. ACM SIGPLAN Notices **40**(10), 281 (2006). https://doi.org/10.1145/1103845.1094834
8. Hauswirth, M., Sweeney, P.F., Diwan, A.: Temporal vertical profiling. Software - Practice and Experience **40**(8), 627–654 (2010). https://doi.org/10.1002/spe.972
9. Hauswirth, M., Sweeney, P.F., Diwan, A., Hind, M.: Vertical profiling: Understanding the Behavior of Object-Oriented Applications. In: Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004. vol. 39, p. 251 (2004). https://doi.org/10.1145/1028976.1028998
10. Hennessy, J.L., Patterson, D.: Computer Architecture: A Quantitive Approach. Morgan Kaufmann (2017)
11. Luk, C.k., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Programming Language Design and Implementation, PLDI 2005. p. 190. ACM Press (2005). https://doi.org/10.1145/1065010.1065034
12. Maleki, S., Gao, Y., Garzarán, M.J., Wong, T., Padua, D.A.: An evaluation of vectorizing compilers. Parallel Architectures and Compilation Techniques, PACT 2011 **7**, 372–382 (2011). https://doi.org/10.1109/PACT.2011.68
13. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. IEEE Micro **23**(6), 84–93 (2003). https://doi.org/10.1109/MM.2003.1261391
14. Sherwood, T., Calder, B.: Time Varying Behavior of Programs. Tech. rep. (1999), https://cseweb.ucsd.edu/~calder/papers/UCSD-CS99-630.pdf
15. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Architectural Support for Programming Languages and Operating Systems, ASPLOS 2002. vol. 36, p. 45 (2002). https://doi.org/10.1145/605397.605403
16. Sprunt, B.: The basics of performance-monitoring hardware. IEEE Micro **22**(4), 64–71 (2002). https://doi.org/10.1109/MM.2002.1028477
17. Sweeney, P.F., Hauswirth, M., Cahoon, B., Cheng, P., Diwan, A., Grove, D., Hind, M.: Using hardware performance monitors to understand the behavior of java applications. Virtual Machine Research And Technology Symposium, VM 2004 p. 5 (2004)
18. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting Performance Data with PAPI-C. In: Tools for High Performance Computing 2009. pp. 157–173 (2010). https://doi.org/10.1007/978-3-642-11261-4_11
19. Weaver, V.M., McKee, S.A.: Can hardware performance counters be trusted? In: International Symposium on Workload Characterization, IISWC 2008. pp. 141–150 (2008). https://doi.org/10.1109/IISWC.2008.4636099
20. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM **52**(4), 65 (2009). https://doi.org/10.1145/1498765.1498785

21. Yasin, A.: A Top-Down method for performance analysis and counters architecture. In: International Symposium on Performance Analysis of Systems and Software, ISPASS 2014. pp. 35–44 (2014). https://doi.org/10.1109/ISPASS.2014.6844459
22. Zaparanuks, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In: International Symposium on Performance Analysis of Systems and Software, ISPASS 2009. pp. 23–32 (2009). https://doi.org/10.1109/ISPASS.2009.4919635