Exploring and Evaluating Array Layout Restructuration for SIMDization

Christopher Haine, Olivier Aumage, Enguerrand Petit, and Denis Barthou

Univ. Bordeaux, LaBRI / INRIA, France
firsname.lastname@labri.fr

Abstract. SIMD processor units have become ubiquitous. Using SIMD instructions is the key for performance for many applications. Modern compilers have made immense progress in generating efficient SIMD code. However, they still may fail or SIMDize poorly, due to conservativeness, source complexity or missing capabilities. When SIMDization fails, programmers are left with little clues about the root causes and actions to be taken.
Our proposed guided SIMDization framework builds on the assembly-code quality assessment toolkit MAQAO to analyzes binaries for possible SIMDization hindrances. It proposes improvement strategies and readily quantifies their impact, using *in vivo* evaluations of suggested transformation. Thanks to our framework, the programmer gets clear directions and quantified expectations on how to improve his/her code SIMDizability. We show results of our technique on TSVC benchmark.

Keywords: SIMDization, performance tuning, performance model

1 Introduction

Nowadays microprocessors feature SIMD vector units, potentially providing substantial performance improvement by concurrently applying the same instruction to all the elements of a vector. A rich and complex API of SIMD instructions has been developed on multiple architectures (such as AVX for Intel or NEON for ARM). Thus, the performance of a code is highly dependent on the use of the SIMD instructions, and compilers are virtually unavoidable for performing SIMDization in an efficient and portable manner. However, the performance of an SIMD code is itself highly dependent on data structure layouts. Unfortunately, although commercial compilers (e.g. IBM xlc, Intel icc, PGI pgcc) have made significant advances in auto-SIMDization, a lot of source codes remain too complicated for a compiler to SIMDize, especially when complex data structures or memory access patterns are involved.

Optimizing for SIMDization may require transformations on code and data structure. A lot of research work has been devoted to improve the capability of compilers to perform appropriate transformations on the code structure [1–5]. On the data structure side instead, compilers usually do not override the data layout chosen by the programmer. Several works have studied data layout restructuring

for specific applications (e.g. stencils [6]). For general purpose compilers, the impact of such transformations on the whole application is difficult to assess at compile-time, even with inter-procedural optimization enabled. It is also difficult for a compiler to determine whether the layout of a data structure should be changed for the whole application or for a limited scope (inducing extra copies), such as a performance critical kernel. The consequence is that the choice of the right data structure still largely depends on the programmer. And, when SIMDization fails due to sub-optimal data layouts, the compilers leave the user with little clue about what may be the cause of performance inefficiency, let alone about how the source code and data structures could be transformed in order to improve SIMDization. This is unfortunate, as in certain cases only a moderate amount of modifications would be required from the programmer to enable SIMDization by the compiler.

Tools such as Intel VTune [7] may suggest that the user hand-SIMDizes his code using intrinsics on the x86 architecture. However, resorting to intrinsics may hinder the portability of the code. More elaborate works focus on specific code optimization [8][9], other works [6] suggest data restructuring, but are limited to very specific cases such as stencils here. In a recent work [10] we proposed a framework built on the MAQAO toolkit, to analyze binary codes and to formulate user-targeted hints about SIMDization potentials and hindrances. These hints provide the user with possible strategies to remove SIMDization hurdles, such as code transformations or data restructuring. However, this preliminary work conducted a qualitative analysis only, thus lacking worthiness quantification in applying advised transformations.

We propose a new integrated qualitative and quantitative approach to guided SIMDization. Our approach reports possible code improvement strategies involving data layout restructuring. Moreover, it offers a fast assessment of the performance improvement (or lack thereof) to be expected from applying each such strategy respectively using the concept of *in vivo* transformation evaluation.

This paper is organized as follows. Section 2 gives the context and motivating example for this work. Section 3 presents the big picture of our proposal. Section 4 exposes the binary analysis stage of our proposal identifying SIMDization issues related to data layout and memory access patterns. Section 5 exposes the assessment stage of our proposal. Section 6 presents evaluation results on kernels from the TSVC benchmark suite. Section 7 discusses positioning of our contribution with respect to related works. Section 8 concludes this paper.

2 Motivating Example

The listing on Figure 1 shows a kernel extracted from the TSVC benchmark suite [11, 12], a suite of codes for the evaluation of SIMDization capabilities of compilers. This kernel is part of the function `s1115`.

Array `c` is stored row-major, but accessed column-major, which hinders SIMDization. A possible strategy is to transpose the `c` array. Another classi-

```
1 for (i = 0; i < N; i++) {
2   for (j = 0; j < N; j++) {
3     a[i][j] = a[i][j]*c[j][i]
4               + b[i][j];
5   }
6 }
```

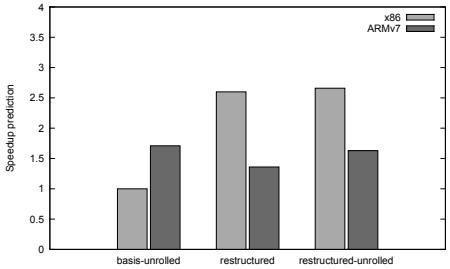

Fig. 1: Kernel `"s1115"` from TSVC suite, showing the need for quantified optimization strategy assessment.

cal strategy is to perform partial loop unrolling. A third strategy is to combine both data transpose and loop unrolling.

The barplot on Figure 1 shows the relative speed-ups of these strategies compared to the original version. Comparisons are made both for an x86 architecture, the Intel Sandy Bridge E5-2650 @2GHz using icc 13.0.1, and an ARM architecture, the ST-Ericsson Snowball (ARMv7 Cortex-A9 @800MHz) using gcc 4.6.3. Both architectures clearly show very dissimilar behaviors on this example, to the point that the two strategies showing good results on x86 perform poorly on the ARM architecture. To address such situations, we propose a framework enabling the fast assessment of SIMDization strategies.

3 Principle of Fast Data Layout Exploration

Our framework reports possible code improvement strategies involving data layout restructuring using a binary-level static and dynamic analysis. It then enables the programmer to explore the value of these strategies before actually engaging into expensive source code modifications. It offers a fast assessment of the performance improvement (or lack thereof) to be expected from applying these strategies. For that, it uses the concept of *in vivo* transformation evaluation.

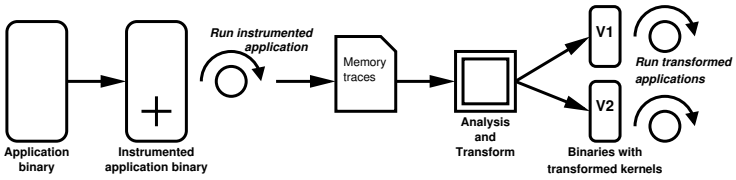

Fig. 2: Overview

The benefit of strategies is evaluated as shown on Picture 2 through the following steps: 1) The original application binary is instrumented with MAQAO [13]

in order to trace the targeted hotspot region (a key application function for instance). 2) The instrumented binary is then run to generate a trace of memory references. This trace consists of the sequence of addresses accessed for each read-/write operation and for each thread. 3) The application binary together with the generated memory traces are analyzed in order to propose possible transformation strategies to improve SIMDizability. The analysis determines which arrays are accessed and how they are accessed, then the transformation changes this data layout in order to enhance spatial locality. Transformations are array contraction, transposition and transformation from Arrays of Structures (AoS) into Structures of Arrays (SoA). A SIMDization step is performed if possible, translating instructions into their SIMD counterparts. Finally, a transformed code is generated for each such strategy. Since these codes are generated with transformations relying on trace information, they may not be semantically equivalent to the initial code. They are called *mockups*. 4) Each transformed mockup is then run within the context of its host application. Its performance is measured to assess the relevance of the corresponding strategy.

In order to speed-up the process, the application is checkpointed during step 2 just before executing the hotspot region. Then, each transformed kernel test at step 4 restarts from the checkpointed state of the application.

4 Static/Dynamic Binary Code Analysis for SIMDization

A static and dynamic analysis of the binary code is performed. We assume a limited code fragment (a function for instance), identified with usual profiling tools, is the target of this analysis. This analysis finds loops, blocks and functions in the binary, instruments the code in order to produce traces of memory addresses and computes the dependence graph, essential for vectorization. For the instrumentation and generation of the binary code we use the MAQAO framework. We recall its main features thereafter. The following section explains how the array structures accessed by the code are detected through trace analysis.

4.1 Framework for Static/Dynamic Analysis of Binary Code

We use the MAQAO framework for performing both the static and dynamic analysis of binary code. MAQAO is a performance tuning tool [13] that analyzes the binary code of applications. It builds the control flow graph and the call graph of the code, and detects loop nests. It also proposes an API and a domain-specific language called MIL to instrument a binary code [14]. This instrumentation is able to capture any value in the code, and in particular can be used to trace memory accesses, count loop iterations, capture function parameters. Compared to PIN [15], a tool with similar functionalities, MAQAO performs static rewriting-based instrumentation from binary to binary and further analyzes the collected information in a post-mortem fashion. PIN, on the contrary, dynamically rewrites binary codes while they execute, and performs analysis on the fly. As most of MAQAO work is done offline, the overall cost for

analyzing a binary with MAQAO is much smaller than with PIN. With MAQAO, it is possible to captures memory streams by instrumenting all instructions that access memory, on a specified code fragment (such as a function or loop). For each instruction instrumented, the flow of addresses captured is compressed on-the-fly using a lossless algorithm, NLR, designed by Ketterlin and Clauss [16]. This compressed trace represents the accessed regions with a union of polyhedra, captures access strides and multidimensional array indexing (when possible).

Beyond instrumentation probes, the MIL language makes it possible to add any kind of assembly instruction to the binary, thus to implement code transformation. Our framework does not yet attempt to manage register allocation. Consequently the code transformation must use identical registers, completely unused registers, or spill/fill. However, this current limitation may be lifted in the future. Moreover, MAQAO can perform an induction variable detection mechanism removing the need for dynamic pointer dependence checking through tracing, as part of the code transformation process.

4.2 Analysis of Data Structure Accesses

This analysis consists in determining the layout of data accessed by each assembly instruction. The data layout detection focuses on finding out: How many arrays are accessed, with how many dimensions, with which element structure (that is, how many fields per array element). This assumes that each load/store assembly instruction of the studied function accesses exactly one single array (no indirection, no pointer aliasing in particular).

Memory addresses accessed for each load/store of the code fragment are compacted as shown in Figure 3. Traces are here represented by loops iterating over the successive address values taken during the execution. The first step of the analysis consists in identifying the load/store that access the same array. To this end, each region accessed by a load/store is converted to a simpler representation, using a strided interval (lower address, upper address, access stride). Out of clarity, in the trace the name of the array in Figure 3 is shown in comments. Actually, the analyzer only knows the assembly instruction accessing this memory region. The goal is to find which traces share the same arrays.

Formally, let I denote the set of assembly instructions accessing memory. We define a relation $\equiv_{array}$ between instructions $i_1, i_2 \in I$ as: $i_1 \equiv_{array} i_2$ iff i_1 and i_2 access to the same array. $\equiv_{array}$ is an equivalence relation, the classes representing the different arrays. The idea is that two instructions with overlapping accessed memory region are equivalent. Algorithm 1 finds the different arrays by merging overlapping regions. Its complexity of is $O(N \log N)$, due to the sorts.

Lemma 1. *Algorithm 1 finds the sets of instructions that access to the same arrays. More formally, it computes* $I/\equiv_{array}$.

Proof. All instructions within a set CLASS are equivalent according to $\equiv_{array}$. This boils down to merging overlapping intervals. Consider an instruction added to CLASS, with index k in L. According to the algorithm, the lower address

```
1 for (nl = 0; nl < ntimes; nl++) {
2   for (i = 0; i < N; i+=2) {
3     a[i] = a[i-1] + b[i];
4   }
5 }
```

```
1  # Trace for access a[i-1]          Strided interval for a[i-1]
2  for i0 = 0 to 999
3    for i1 = 0 to 1535
4      val 0x1525d40 + 8*i1      => [ 0x1525d40 ; 0x1525d40 + 8*999; 8 ]
5  endfor
6  endfor
7  # Trace for access b[i]            Strided interval for b[i]
8  for i0 = 0 to 999
9    for i1 = 0 to 1535
10     val 0x1528d84 + 8*i1      => [ 0x1528d84 ; 0x1528d84 + 8*999; 8 ]
11 endfor
12 endfor
13 # Trace for access a[i]            Strided interval for a[i]
14 for i0 = 0 to 999
15   for i1 = 0 to 1535
16     val 0x1525d44 + 8*i1      => [ 0x1525d44 ; 0x1525d44 + 8*999; 8 ]
17 endfor
18 endfor
```

Fig. 3: Trace example on function `s111` from TSVC. Each trace is compacted with NLR algorithm into loops in this simple example, iterating over successive addresses. A simplified representation with strided intervals is used.

Algorithm 1: Identifying distinct arrays from access traces.

Data: I = list of load/store triplets $[lower_i, upper_i, stride_i]$, $i = 1..N$
Result: OUT = I/R_{array}, the set of instructions grouped by array they access.

```
L = {lower_i, i = 1..N} ;
U = {upper_i, i = 1..N} ;
sort L by increasing address;
sort U by increasing address;
CLASS = {I_1} ;
for k = 2..N do
    if L_k > U_{k-1} then
        OUT = OUT ∪ {CLASS};
        CLASS = ∅ ;
    CLASS = CLASS ∪ {I_k} ;
OUT = OUT ∪ {CLASS};
```

bound L_k of the region accessed by this instruction is such that $L_k \leq U_{k-1}$, with U_{k-1} the upper bound of an other instruction in CLASS. There are k regions starting before L_k and only $k-1$ closed at U_{k-1}. Hence at least one region starting before L_k is ending after U_{k-1}, this shows that instruction k accesses the same interval as another instruction of the set CLASS. Reciprocally, we can show that if two instructions are not in the same set CLASS, they do not access the same array. Let p and q be their index in L, assuming $p < q$. Since they do not belong to the same set ARRAY, there exists k such that $L_k > U_{k-1}$ and $p < k \leq q$. It implies that there are $k-1$ regions that are ending before L_k,

that is, all $k-1$ regions starting before L_k are also ending before L_k. The arrays of these regions are different from the array accessed by L_k. In particular, since $p < k$, the array accessed by instruction p is different from the array accessed by instruction q.

For example, the analysis of the three regions from Figure 3 builds the sets $L = \{0x1525d40, 0x1525d44, 0x1528d84\}$ and $U = \{0x1525d40+8*999, 0x1525d44+8*999, 0x1528d84+8*999\}$. The two first regions are found as being accesses to the same array, while the third one is not since $0x1526d84 > 0x1525d44+8*999$.

Now, the second step of the analysis consists in finding load/store instructions accessing the same element field within an array of structures (e.g. `t[0].x, t[1].x, .., t[n].x`). Among the instructions accessing to the same array of structures, we define a relation $\equiv_{field}$ between each pair of instructions accessing the same field in the same array. Thus formally, for each two instructions i_1, i_2 accessing the same array ($i_1 \equiv_{array} i_2$), the relation $i_1 \equiv_{field} i_2$ is verified iff $i_1 \equiv_{array} i_2$ and:

$$lower_1 \equiv lower_2 \pmod{\gcd_{i \in [i_1]_{\equiv_{array}}} (stride_i)}.$$

The gcd of the strides of all accesses on the array corresponds to the size in bytes of the structure. The values of $lower_1$ and $lower_2$ modulo this size correspond to the offset of the field within a structure. Then, fields for each partition I/R_{array} can be sorted according to their *lower* value modulo the gcd of the strides. Determining the field layout of an array of structures can be done with a $O(N \log N)$ complexity.

In the previous example, the two strided intervals $[0x1525d40; 0x1525d40 + 8*999; 8]$ and $[0x1525d44; 0x1525d44+8*999; 8]$ are not found equivalent, since $0x1525d40 \not\equiv 0x1525d44 \pmod 8$. Therefore the two instructions access different fields in an array of structures. Note that in the initial C code, there is no structure. However, the stride 2 on the loop counter entails that all loads on `a` are on even indices while the stores are on odd indices, a behavior similar to an access to a 2 field structure. The following section explains how these structures are transformed.

Last, for all instructions accessing the same fields of the same array, strides of the NLR trace are considered in order to determine whether a transposition is required or not. In the NLR trace with its `for` loops, this boils down to determine whether the stride of the innermost loop is the smallest one. If a transposition is required for all instructions accessing the same field, then it is performed by the proposed transformation described in the following.

5 Fast Exploration and Assessment of Data Restructuration for SIMDization

Data layout together with access pattern knowledge provides precious clues about SIMDization issues and ways to address them. However the corrective

steps to enable or improve SIMDization may have other, unwanted side-effects leading to an overall performance degradation instead of the expected gain. Therefore the second part of the transformation process we propose is to directly and quickly assess the potential of such a transformation on the binary code. For that, we adopt an *in vivo* approach, by running a "mock-up" of a transformed kernel within its host application. Thanks to the use of a checkpoint/restart mechanism, only the relevant part of the application is tested with the kernel mock-up. In this section we discuss technical aspects of our *in vivo* approach.

5.1 Checkpointing for Fast Exploration

One solution for assessing the potential of code transformations is to run the different versions generated and measure their execution times. This auto-tuning technique has proved its efficiency in particular for tuning performance of library functions. For application codes, this would mean to run the whole application multiple times, only for assessing an optimization on a limited code fragment, entailing large execution times.

We propose to resort instead to checkpoint/restart technique in order to be able to execute multiple versions of the same code, within the same context. The principle of this technique was first described by Lee and Hall [17]. The idea is to first run the application, instrumented so that there is a checkpoint at the entry of the function to optimize. The binary code of the function is then modified in order to change data structures and/or SIMDize one of its loops. This modified code is then restarted using the previous checkpoint. Note that the binary codes used for the checkpoint and for the restart are not the same. This technique works if binaries have the same size and if the only modified function is the function where the restart occurs. Binary sizes are kept the same by using code padding, and the instrumentation with MAQAO allows pinpointed code transformation (to functions or loops).

The advantages of this technique are numerous:

- By restarting different versions of the code, these codes can be evaluated within the same applicative context, at no cost.
- The method works with parallel, multithreaded codes.
- Array addresses, pointers keep the same value after a restart as at the time of checkpoint. It implies that the optimization can be dependent on the values collected by traces, in particular address traces. We use this approach to restructure data layouts. A first run after restart collects all memory accesses, and the analysis technique proposed in the previous section is applied. Then a new version of the code is generated (see next section) with restructured data layout and it is restarted in order to measure its performance.

The following section presents how the code is modified once the trace has been collected and arrays and structures are discovered. In our implementation, we use the Berkeley lab checkpoint/restart (BLCR) tool [18].

5.2 Array Contraction and AoS to SoA Transformation

From the addresses collected by the trace, the different arrays and structures are identified by the algorithm presented in Section 4.2. To transform Arrays of Structures into Structures of Arrays, new arrays are allocated. The restructuring corresponds to a mapping function, mapping indices of the initial array to indices in the new array. The mapping we propose has two objectives:

- Reduce the stride between elements accessed successively whenever possible
- Transform AoS into SoA. For this transformation, the size of the array is deduced from the trace.
- Transpose arrays if successive accesses are not performed along successive addresses, for multi-dimensional accesses detected through the NLR traces.

When the first instruction to access these data is a read, a copy performing this mapping is required. When the first instruction is a write, no copy is needed.

New arrays are allocated inside the function analyzed, before the loop to vectorize. Besides, if a copy from the initial array to the new array is required, it is also inserted right after the allocation of the new array. Allocation and copy are then placed after the last write to the initial array (if any). Several locations may be possible. We choose to place the copy at the earliest possible location in the code, so that the impact of the copy on performance may be reduced. The sizes of the new arrays allocated on the heap are determined by the trace and the type of transformation involved (whether applying contraction or not).

The array contraction consists in removing unnecessary strides separating elements of an array. Consider an array $[a; b; s]$, assuming each data is 4 bytes long, it is contracted into a new array $[a'; a'+(b-a)*4/s; 4]$ with starting address a'. When multiple regions access the same array, as analyzed by the previous algorithm, the gcd of the strides for all regions is considered and the extreme addresses accessed define the boundaries of the initial array. The code allocating the contracted array is inserted for the mockup code.

Now consider an array of structures $[a; b; 4]$. The size of the structure, n, and the number of elements in the array, N are deduced from the trace analysis (see Section 4.2). A field of this structure is characterized by an offset k corresponding to the displacement in bytes from the beginning of a structure element. The i^{th} element of field k in the array is positioned at $k + n * i$ bytes from the beginning of the array, for $0 \leq i < N$. To transform this AoS into SoA, the i^{th} element of field k is mapped to the i^{th} element of subarray k, at offset $k * N + 4 * i$. Therefore if the AoS $[a; b; 4]$ is remapped into the SoA $[a''; b''; 4]$ with $b'' - a'' = b - a$ (same size), each access region $[a_k; b_k; s_k]$ of the field k is remapped into $[a''+k*N+4*(a_k-k-a)/n; a''+k*N+4*(b_k-k-a)/n; 4*s_k/n]$.

For the example of Figure 3, considering the two regions accessing the same array but different fields, the size of the structure found is 8 byte long. The size of the new structure of arrays replacing the array of structure is $1000 * 8$ bytes. The two regions are mapped to a new array starting at index 0 for the first one and $1000 * 4$ for the second one. Their stride is now 4 instead of 8. The creation

of this array and copy of the elements (only those that are read) is inserted in this case right before the loop.

In terms of code transformation, this implies that for any load and store instruction, its address is changed into addresses inside the new array. For a transformation of AoS into SoA, using the previous notation, an address $addr$ is changed into $a'' + k * N + 4 * (addr - k - a)/n$ with $a'' + k * N$ a constant corresponding to the address within the new array where the array for fields k starts. The assembly code for a load instruction `load .., [address]` for instance is changed into the following code (all constants are prefixed by #):

```
XOR  RDX, RDX
LEA  RAX, address
SUBQ RAX, #a+k
MULQ RAX, #4
DIVQ #n
MOVQ RAX, a'' // newly allocated array
ADDQ RAX, #k*N
load .. , [RAX]
```

This code requires that registers `%RDX` and `%RAX` are available since the integer division makes an implicit use of them. This may requires to perform register reallocation on the modified code, or a spill/fill for these two registers. While this transformation is correct for any access, its impact on performance can be important due to the memory access of the spill/fill and the integer division. The later is replaced by a shift whenever n is a power of 2 (removing also the constraint on the use of $RDX : RAX$). A simpler transformation is possible whenever an induction variable detection computation on the initial code finds that the address accessed is of the form $a + n * i$ with i an induction variable (a register here). The transformation then consists in adding a new induction variable with stride 4 and modify the base register of the load. There is no integer division involved then. We use in our implementation this transformation whenever possible. The code modification is similar for the array contraction transformation. The code modification in case of a transposition is similar to the previous case.

5.3 SIMDization

Once the data layout has been transformed, the code is SIMDized. The transformation we propose here is simple and only vectorizes arrays of floats or doubles.

In order to trigger SIMDization, all arrays accessed have to fulfill two conditions: (i) Elements are either 4 or 8 byte long, and instructions are floating point operations; (ii) The strides used by all modified arrays is either 4 or 8 (depending on the data type). If one of the condition is not fulfilled, SIMDization is not performed and a warning is emitted.

Besides, the memory traces are used to compute a dependence graph, taking into account register and memory dependences, as presented in [19]. The conditions for a possible vectorization are deduced from such dependence graph and instruction schedule compatible with dependences is generated.

Arithmetic operations are vectorized using a simple correspondence between scalar/SIMD instructions. Load/store instructions are rewritten as aligned or unaligned accesses, depending on the address alignments. SSE registers are allocated. Finally, the iteration count is changed by adding a new loop counter. These transformations are eased by the fact that the trace provides the iteration count and array alignment information.

6 Experimental Results

We assess the accuracy of our mockup-driven predictions on a suite of benchmarks, TSVC. TSVC consists of 151 functions intended to explore the typical difficulties a compiler can meet in the context of vectorization. Out of these 151 benchmarks, 31 matches data layout access issues, our primary focus in this study. The others correspond to control issues, mostly already well handled by compilers. In Figure 4, we report speedups obtained by *mockup* kernels and by manually restructured (*correct*) kernels over compiled basis kernel compiled with icc 13.0.1, on Intel Sandy Bridge E5-2650 @2GHz. These kernels are a subset of the data layout issue category. More complex kernels of this category will be studied using future evolutions of our prototype infrastructure.

Non-contiguous stride accesses are an obstacle to vectorization, compilers may see the opportunity of vectorization but consider it not efficient enough to vectorize. This is the case for benchmarks *s111* and *s128*, performing accesses with strides 2. Their respective mockups show significant gain to expect from data restructuring.

When 2-dimensional arrays are accessed column-wise (in C) or row-wise (in Fortran), accesses with large strides are performed, and one may resort to data transposition prior to massive computations, in order to allow vectorization. Code mockup here predicts significant gains to expect from restructured kernel, which is effectively perfectly reached.

One big challenge for compiler autovectorization is brought by rescheduling issues, that is, codes where compilers see vector dependences it can not resolve, although such dependences could be fixed by permuting instructions or loop peeling. All *s241*, *s243*, *s211*, *s212*, *s1213*, *s244* and *s1244* benchmarks have rescheduling issues and are not vectorized by the compiler. Here, the dynamic dependence graph enables to find a correct schedule for SIMD code.

In some cases, a non-contiguous data pattern may not cause performance issues, as they are already well handled by the compiler and/or the architecture. Here, benchmarks showing no speedup over the basis kernel (*s1111*, *s131*, *s121*, *s151*) correspond to alignment issues, which can be solved by unaligned accesses or vector permutations. On this very architecture, unaligned accesses do not produce a significant performance overhead, therefore data restructuring will not bring better performance.

For all these measures, the time to restructure data (using a copy) is not included. Indeed, the benchmarks are small functions and a copy is, with a few exception, not amortized. Performance of the mockup is in most cases close to the

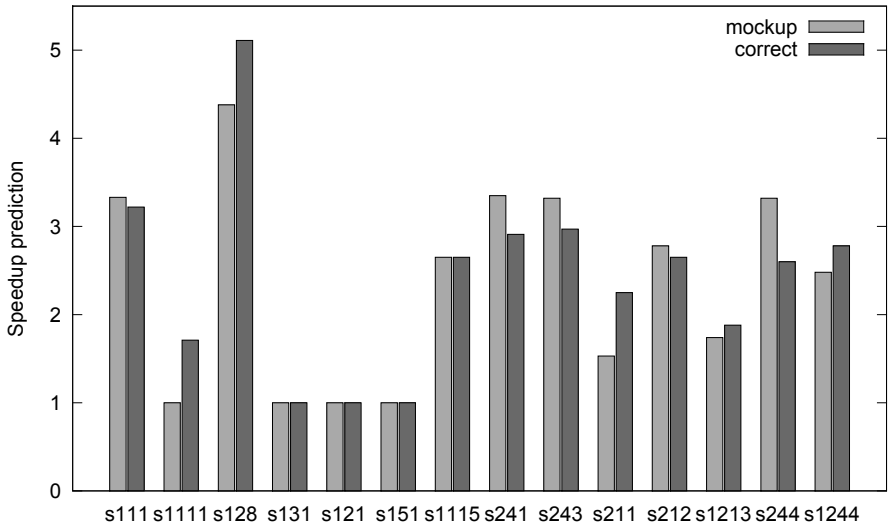

Fig. 4: TSVC Mockup Prediction on x86

real transformed code. When there are differences, this is explained by the fact that the mockup results from a binary transformation, while the correct hand-tuned code results from a source-to-source transformation. Hence, the binary code resulting from hand-tuning the source code may not be exactly the same as the mockup code due to compiler optimizations.

7 Vectorization Issues and Related Works

A lot of work has already been devoted to help programmers harness the power of SIMD instruction sets, once called "multimedia" extensions. Multiple approaches have been followed in this attempt to group isomorphic elementary computations together. Loop-level approaches have been explored for several decades. Back in 1992, Hanxleden and Kennedy [1] made proposals for balancing loop nest iterations over multiple lanes of a SIMD machine. MMX instruction sets and alike started to gather interest with works such as Krall's [2] proposing to apply vectorization techniques to generate code for SIMD extensions. Larsen introduced the concept of Superword-Level Parallelism (SLP) [3] which groups isomorphic statements together, when potentially packable and executable in parallel. More recently Nuzman et al. [4] explored SIMDization at the outer-loop level. Much work has also been devoted into employing polyhedral methods [5]. Several works have been conducted to allow compilers to accept more complex code and data structures, such as alignment mismatch [20], flow-control [21], non-contiguous data accesses [22] or minimizing in-register permutations [23], for examples. SIMDizing in the context of irregular data structures is also being

studied [24]. All these works have in common that they accept unmodified source code as input and they attempt to generate the best SIMDized binary code for the target hardware. They do not involve the programmer in their attempt to produce good SIMD code and usually give little information back to the programmer when their attempt fails. Our proposal differs and complement these works in that it offers to diagnose and alter produced binary codes in order to help programmers improve they source code with respect to SIMDizability.

Some works have followed the path to act on the source code side. Frameworks such as the Scout [25] source-to-source compiler enable the programmer to annotate the source code with `pragma` directives that are subsequently translated to SIMDized code. A recent work of Evans *et al.* [26] presents a method to analyze vector potential, based on source annotated code and an on-the-fly dependence graph analysis using the PIN framework. Other works aim at specialized fields and will produce efficient code for a selected class of applications, e.g. stencil computations [27, 6] for instance, or allow to auto-tune specific kernels [8]. Our solution differs in that it aims at helping the programmer to improve his/her *original* code in a generic manner instead of specializing it or augmenting it with annotations. It also again complements these works because it can analyse their output for quality assessment as well as to devise and experiment further optimizations.

Profiling tools such as Intel VTune or the suite Valgrind, for instance, can diagnose code efficiency and pinpoint issues such as memory access patterns with bad locality. Intel VTune may even suggest that the programmer resorts to SIMD intrinsics. However, such an action is not always desirable for code readability and maintainability.

Our approach is able to make higher-level transformation suggestions based on instruction flow dependence analysis of the binary code, and to quantify their expected performance gain over the code using *in vivo* code evaluation. Several previous works have proposed techniques to extract pieces of code for evaluation [17, 28], and shown that such an operating mode is viable for performance measurements [29].

8 Conclusion

This paper presented a new technique for transforming and evaluating data layout transformations for SIMDization, directly from the binary code. This method changes Arrays of Structures for instance into Structures of Arrays, performs SIMDization if possible on the code and provides a quick assessment "*in-vivo*" of any performance gain/loss resulting from this transformation. Being trace-based, the transformation proposed is not to be integrated in a static compiler but provides the user with a good estimation of the real source-code transformation, before deciding to perform expensive source code modification. The transformation is achieved at the binary level, ensuring no interference with compiler optimizations. Moreover, we have proposed an original use of checkpoint/restart techniques in order to reduce the cost of our method. The prelimi-

nary results on TSVC benchmarks show that performance estimations are close to those actually obtained after a source code transformation.

For future work, we plan to extend the use of our technique to larger applications and to a wider range of data layout transformations. We are also working on addressing multithreaded applications. The trace collection stage already supports application with multiple threads, and most of the work will now concentrate on the strategy building stage.

Acknowledgement This work has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreements no 610402 Mont-Blanc 2.

References

1. v. Hanxleden, R., Kennedy, K.: Relaxing simd control flow constraints using loop transformations. In: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (1992)
2. Krall, A., Lelait, S.: Compilation techniques for multimedia processors. Intl. J. of Parallel Programming (2000)
3. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. In: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (2000)
4. Nuzman, D., Zaks, A.: Outer-loop vectorization: revisited for short simd architectures. In: ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT). (2008)
5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (2008)
6. Henretty, T., Stock, K., Pouchet, L.N., Franchetti, F., Ramanujam, J., Sadayappan, P.: Data layout transformation for stencil computations on short-vector simd architectures. In: Intl. conference on compiler construction: part of the joint European conferences on theory and practice of software. (2011)
7. Intel: Vtune (2014) http://software.intel.com/en-us/intel-vtune-amplifier-xe.
8. Videau, B., Marangozova-Martin, V., Genovese, L., Deutsch, T.: Optimizing 3d convolutions for wavelet transforms on cpus with sse units and gpus. In: Intl. europar conference on parallel processing. (2013)
9. Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.N., Sadayappan, P.: When polyhedral transformations meet SIMD code generation. In: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (2013)
10. Aumage, O., Barthou, D., Haine, C., Meunier, T.: Detecting simdization opportunities through static/dynamic dependence analysis. In: Workshop on Productivity and Performance (PROPER). (2013)
11. Callahan, D., Dongarra, J., Levine, D.: Vectorizing compilers: a test suite and results. In: Conf. on Supercomputing. (1988)
12. Maleki, S., Gao, Y., Garzarn, M.J., Wong, T., Padua, D.A.: An evaluation of vectorization compilers. In: Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT). (2011)

13. Barthou, D., Rubial, A.C., Jalby, W., Koliai, S., Valensi, C.: Performance tuning of x86 OpenMP codes with MAQAO. In: Tools for High Performance Computing. Springer Berlin Heidelberg (2010)
14. Charif-Rubial, A.S., Barthou, D., Valensi, C., Shende, S., Malony, A., Jalby, W.: Mil: A language to build program analysis tools through static binary instrumentation. In: IEEE Intl. High Performance Computing Conf. (HiPC), Hyberabad, India (December 2013) 206–215
15. 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: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (2005)
16. Ketterlin, A., Clauss, P.: Prediction and trace compression of data access addresses through nested loop recognition. In: ACM/IEEE Intl. Conf. on Code Generation and Optimization, New York, NY, USA, ACM (2008) 94–103
17. Lee, Y.J., Hall, M.: A code isolator: Isolating code fragments from large programs. In: Langages and Compilers for High Performance Computing. (2004)
18. Hargrove, P.H., Duell, J.C.: Berkeley lab checkpoint/restart (blcr) for linux clusters. Journal of Physics: Conf. Series **46**(1) (2006) 494
19. Aumage, O., Barthou, D., Haine, C., Meunier, T.: Detecting SIMDization Opportunities through Static/Dynamic Dependence Analysis. In: Workshop on Productivity and Performance (PROPER), Aachen, Germany (September 2013)
20. Eichenberger, A.E., Wu, P., O'Brien, K.: Vectorization for SIMD architectures with alignment constraints. In: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (2004)
21. Shin, J., Hall, M., Chame, J.: Superword-level parallelism in the presence of control flow. In: ACM/IEEE Intl. Conf. on Code Generation and Optimization. (2005)
22. Nuzman, D., Rosen, I., Zaks, A.: Auto-vectorization of interleaved data for SIMD. In: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (2006)
23. Ren, G., Wu, P., Padua, D.: Optimizing data permutations for SIMD devices. In: ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation. (2006)
24. Ren, B., Agrawal, G., Larus, J.R., Mytkowicz, T., Poutanen, T., Schulte, W.: SIMD parallelization of applications that traverse irregular data structures. In: ACM/IEEE Intl. Conf. on Code Generation and Optimization. (2013)
25. Krzikalla, O., Feldhoff, K., Muller-Pfefferkorn, R., Nagel, W.E.: Scout: a source-to-source transformator for SIMD-optimizations. In: Workshop on Productivity and Performance (PROPER). (2011)
26. Evans, G.C., Abraham, S., Kuhn, B., Padua, D.A.: Vector seeker: A tool for finding vector potential. In: Workshop on Prog. Models for SIMD/Vector Processing, New York, NY, USA, ACM (2014) 41–48
27. Jaeger, J., Barthou, D.: Automatic efficient data layout for multithreaded stencil codes on cpus and gpus. In: IEEE Intl. High Performance Computing Conf., Pune, India, IEEE Computer Society (December 2012) 1–10
28. Petit, E., Bodin, F., Papaure, G., Dru, F.: ASTEX: a hot path based thread extractor for distributed memory system on a chip. In: HiPEAC Industrial Workshop. (2006)
29. Akel, C., Kashnikov, Y., de Oliveira Castro, P., Jalby, W.: Is source-code isolation viable for performance characterization? In: Intl. Workshop on Parallel Software Tools and Tool Infrastructures. (2013)