Unification of Static and Dynamic Analyses to Enable Vectorization

Ashay Rane[1(✉)], Rakesh Krishnaiyer[2], Chris J. Newburn[2], James Browne[1], Leonardo Fialho[1], and Zakhar Matveev[2]

[1] The University of Texas at Austin, Austin, USA
{ashay.rane,fialho}@utexas.edu, browne@cs.utexas.edu
[2] Intel Corporation, Santa Clara, USA
{rakesh.krishnaiyer,chris.newburn,zakhar.a.matveev}@intel.com

Abstract. Modern compilers execute sophisticated static analyses to enable optimization across a wide spectrum of code patterns. However, there are many cases where even the most sophisticated static analysis is insufficient or where the computation complexity makes complete static analysis impractical. It is often possible in these cases to discover further opportunities for optimization from dynamic profiling and provide this information to the compiler, either by adding directives or pragmas to the source, or by modifying the source algorithm or implementation. For current and emerging generations of chips, vectorization is one of the most important of these optimizations. This paper defines, implements, and applies a systematic process for combining the information acquired by static analysis by modern compilers with information acquired by a targeted, high-resolution, low-overhead dynamic profiling tool to enable additional and more effective vectorization. Opportunities for more effective vectorization are frequent and the performance gains obtained are substantial: we show a geometric mean across several benchmarks of over 1.5x in speedup on the Intel Xeon Phi coprocessor.

Keywords: Performance optimization · Dynamic profiling · Vectorization

1 Introduction

Modern compilers leverage sophisticated static analyses to enable aggressive optimizations for pursuing performance and power efficiency. There are, however, many circumstances where the information required to enable optimizations cannot be determined by static analysis alone. Additionally, compilers are bound by safety requirements to be conservative to avoid producing a different result. It is often possible in these cases to discover further opportunities for optimization from dynamic profiling and provide this information to the compiler, either by adding directives or pragmas to the source, or by modifying the algorithm or implementation. While many modern compilers have some capabilities

J. Brodman and P. Tu (Eds.): LCPC 2014, LNCS 8967, pp. 367–381, 2015.
DOI: 10.1007/978-3-319-17473-0_24

for dynamic profiling and can feed back dynamic profiling information into optimization algorithms, they are seldom used because of their high cost (runtime overhead) and low benefit (optimization potential). There are also separate tools for dynamic profiling and for performance optimization based on these dynamic profiles.

This paper defines, implements and applies a systematic process for combining the information acquired by static analysis by modern compilers with information acquired by a targeted, high-resolution, low-overhead dynamic profiling tool to enable additional and more effective vectorization. Vectorization was chosen as the target for this study because it is becoming one of the most important optimizations for modern multicore and manycore architectures.

Our end-to-end process of discovering and applying optimizations is based on: (a) an analysis of the minimal information that is needed for effective vectorization but is missing from static analysis which is critical to minimizing instrumentation overhead, (b) how this information can be used to define targeted, high-resolution, low-overhead dynamic profiling, (c) how dynamic profiling can collect the missing information and (d) insights from dynamic profiling. MACVEC suggests the exact location and text of the changes, but the user remains in control of whether to apply them. These steps enable MACVEC to exceed the performance that could ever be possible with static analysis alone.

The implementation of our dynamic profiling system for maximizing vectorization instruments only those loops which are not fully vectorized and which consume a significant program execution time. To minimize the overhead, instrumentation is specifically targeted to the additional information required for most effective vectorization. The compiler carries out all of the optimizations.

The key contributions of the paper include the following:

1. Development of the knowledge base required for unification of static analyses and dynamic profiling for enhancing vectorization (Sect. 2).
2. Implementation of the workflow using the Intel compiler and extending an existing dynamic profiling and performance optimization tool (Sects. 3 and 4).
3. Demonstration of the effectiveness of the unified process using small to moderate-sized applications and using a mix of benchmarks and a few real-word applications (Sect. 5).

2 Enhancing Vectorization Through Dynamic Profiling

Careful analysis of the vectorization reports generated by the Intel® Composer XE 2013 SP1 Update 3 (v14.0.3) compiler reveals information that the compiler needs for effective vectorization but which could not be derived using static analysis alone. Based on our studies, the commonly occuring reasons for not efficiently vectorizing a loop include the compiler not being able to determine: (a) the trip count, (b) the access strides for the arrays in the loop, (c) the alignment of arrays in the loop, (d) whether there were backward loop carried dependences among the arrays in the loop body, (e) failure to recognize non-temporal or streaming stores, and (f) the outcomes of branches.

The mapping from vectorization messages to the information gathered using dynamic profiling was done by comparing compiler messages and code patterns. The measurements which must be made by dynamic profiling to obtain this information are specified in the following paragraphs. MACVEC uses attribute, pragmas and directives that are in standard use by GCC wherever they are available, and those supported exclusively by the Intel® Composer XE 2013 SP1 Update 3 (v14.0.3) compiler where necessary. The association between the measurements and the specification of the information back to the compiler (in the form of pragmas or code modifications) required detailed knowledge of the large set of pragmas available with these compilers. The following paragraphs explain the metrics measured for each of these cases, the pragmas and directives required to optimize the code based on the observed measurements, and the applications or benchmarks that were used to validate the measurements.

All information in the recommendations made at the end pertain only to application code properties (ensuring portability across compilers where applicable). They are also mostly applicable for multiple target platforms with only a few exceptions. In most application codes for example, vectorization remains important on Intel Xeon processor and Intel Xeon Phi coprocessor.

Loop Trip Count: Loops tend to have poor vectorization efficiency unless the alignment- and remainder-handling costs are non-existent or are amortized away with a high trip count. Trip counts for each loop may also impact the choice of loops to vectorize or parallelize. Vectorization is normally thought of as being applicable to inner loops while outer loops are commonly parallelized. However, the compiler may choose (based on heuristics like the trip count and access patterns) to vectorize the outer loop rather than the inner loop. Loop trip counts (measured using simple counters in the loop body) that are smaller than a pre-decided threshold (1024) may be indicated using `#pragma loop_count`. The LCD benchmark was used to validate the loop trip count analysis.

Array Access Strides: Efficient vectorization of each statement in the loop body requires that all of the array accesses for each array in the loop have a unit stride with respect to the vectorized (inner or outer) loop. If the code does not exhibit unit strides, less efficient vectorization (requiring use of gather/scatter operations) may be possible. For each array referenced in the loop, the difference in the addresses for the references across consecutive iterations is recorded as the stride for that array. Based on the stride metrics, the compiler may be instructed to prefetch data (for example targeting indirect accesses that follow no regular strides) using `#pragma prefetch <indirect-array>`. The array access stride information may also be used to tell the compiler to generate an alternate gather sequence via the `-opt-gather-scatter-unroll` option. A synthetic array-of-struct microbenchmark was written to collect the measurements and to validate them.

Alignment of Arrays: Besides unit strides, an additional consideration for efficient vectorization is that the first loop iteration access to each array occur at

cache-line aligned addresses or that all array references have the same relative alignment. Unless all vectorized references are cache line aligned, masking is required in a peeling loop, which reduces efficiency. Furthermore, unless all references are mutually aligned with respect to cache line boundaries, some shifting is likely to be required, also reducing efficiency. The address of the first referenced element for each array referenced in the loop is recorded. The measurements recorded are the addresses of the first referenceto each array in each statement in the loop body. Arrays can be aligned using `__mm_malloc()` for heap memory and using the `__attribute__((aligned(64)))` clause for global, static and stack memory. Vectorizable loops containing all aligned array references may be tagged using `#pragma vector aligned`[1]. A NBody application was used to validate array alignment measurements.

Overlapping Arrays: A loop is vectorizable only if arrays referenced in the loop do not have lexically backward dependences. Such dependencies may be introduced in the code if arrays overlap in memory. The span of addresses (derived from the first and last referenced addresses for each array) is used to identify if two differently-named arrays overlap in memory. Such a dynamic check provides a fast and accurate way of determining whether arrays overlap. The fact that pointers do not overlap can be conveyed to the compiler using the `restrict` keyword. The STREAM benchmark was used to validate measurements about overlapping arrays.

Non-Temporal or Streaming Stores: The non-temporal property for store instructions for arrays implies the array will not be referenced in the near future and this property can be derived on the basis of reuse distance. Using streaming stores for arrays requires that the sequence of stores write the entire cache line, the arrays is accessed with unit strides, without using a mask register and is aligned to cache-line boundary. Alignment, branch outcomes, strides and reuse distances are measured and a combination of these metrics with rudimentary static analysis (for identifying write-only arrays) is used to recommend the use of non-temporal stores or streaming stores. To indicate to the compiler to use streaming store instructions in vectorizable loops, `#pragma vector nontemporal` or the `-opt-streaming-stores=always` option may be used. The STREAM benchmark was used to validate the relevant measurements for this metric.

Branch Path Outcomes: Although branches are less suitable for vectorization, the compiler may still benefit if it knows that the direction taken by the branch is the same for most iterations of the loop. The data recorded is the count of true and false outcomes for each branch. Branches that always evaluate in one direction may be indicated so to the compiler using the `__builtin_expect()` attribute. Knowing that the control-flow will be coherent for all vector-lanes allows the compiler to often generate a more efficient code-path for the vector-loop. The LCD benchmark was used to validate branch-outcome metrics.

[1] http://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.

3 Workflow

This section describes the high-level workflow for integration of compiler static analysis with dynamic profiling and gives the tools used in our implementation of the workflow. The details of our implementation are given in Sect. 4.

The high-level workflow is generic and can be implemented with different compilers and instrumentation tools. This study uses the Intel® Composer XE 2013 SP1 Update 3 (v14.0.3) compiler and uses performance analysis results from the PerfExpert [7] open source profiling based optimization tool to implement a new tool called MACVEC. MACVEC is an instrumentation and analysis tool built by adding the measurements described in Sect. 2 to the instrumentation framework provided by MACPO [19]. PerfExpert, MACVEC and MACPO are open source tools readily available for download[2]. Steps (b) through (g) are auomated in MACVEC.

(a) **Selecting Execution Environment:** We used production inputs to enable the application to spend significant time executing each phase of its workflow. Different representative input-sets may be needed to exercise different parts of the algorithm.
(b) **Determining Important Loops:** The PerfExpert performance optimization system was used to determine the loops which use more than a chosen fraction of the execution time of the application.
(c) **Identifying the Loops that are Not Fully Vectorized:** We used the vectorization reports generated by the Intel compiler (`-vec-report=6`) to identify loops that can benefit from better vectorization.
(d) **Building the Set of Loops to be Instrumented:** This is formed by the intersection of the loops identified in steps (b) and (c).
(e) **Instrumenting Selected Loops:** This is one of the key steps implemented in MACVEC. Section 4.1 explains the details of the instrumentation calls.
(f) **Generating Measurements:** By executing the user program, the instrumentation from the previous step records dynamic profiling measurements into in-memory data structures.
(g) **Identifying Recommendations for Additional Vectorization:** At the end of the user program execution, MACVEC runs various analyses over the collected measurements to generate recommendations for optimizing the program. Details of the analyses (and the steps used to convert the measurements into recommendations) are illustrated in Sect. 4.2.
(h) **Analyzing Validity of Optimizations:** In our setup, this process is currently performed manually.
(i) **Implementing Recommendations:** The program source code was manually changed to include the recommendations from MACVEC. Automation of this step using PerfExpert is in progress.
(j) **Evaluating Performance Gain:** Performance improvements resulting from the optimizations are described in Sect. 5.

[2] https://www.tacc.utexas.edu/perfexpert.

4 Implementation of Dynamic Profiling

MACVEC instruments the specified functions and loops using the Rose [18] (v0.9.5a) compiler's Abstract Syntax Tree (AST) of the user source code andRose compiler APIs. AST-level instrumentation (as opposed to IR-level instrumentation or binary instrumentation) helps MACVEC in producing metrics derived directly from the user code structure. Further, MACVEC's instrumentation-based analysis yields recommendations that are agnostic to the generated code.

The fact that MACVEC provides *source-level information* isolates MACVEC from the maturity and aggressiveness of the production compiler used to compile the user's code. For instance, a particularly aggressive vectorizing compiler may split a loop into a prolog *peel loop* to get references aligned to specific boundaries, a *main loop* to handle the steady state, and an epilog *remainder loop* to handle unaligned leftover iterations. The Rose compiler AST API operates at the source level, not at the level of loops in the generated code.

4.1 Compiler-Based Instrumentation

During the instrumentation phase, MACVEC adds function calls to the program source code to record the metrics identified in Sect. 2. MACVEC currently does not use alias-analysis information from the compiler. MACVEC inserts the function calls either before or after the loop body. As the function calls are placed outside of the loop body, the iteration count has only a small effect on the overhead. However, the number of nested loops and the trip count of outer loops influences the overhead substantially.

We compared the execution times with and without instrumentation using the Rose compiler. We instrumented all functions and loops in the codes from the Rodinia [4] benchmark suite that consumed at least 40 % of the total execution time. The mean of slowdowns (ratio of times from instrumented execution to non-instrumented execution) was 1.13x for array alignment checks, 1.08x for loop trip count measurements, 1.07x for branch-path analysis, 1.06x for array overlap checks and 1.05x for stride checks.

4.2 Generation of Recommendations

MACVEC analyses internally maintain a histogram of collected values, which are iterated over just before the program terminates. The nature of the generated recommendations naturally allow a few optimizations in the data collection process, which in turn helps to reduce the instrumentation overhead. For instance, analysis of loop trip count needs to check only whether the trip count is lower than a threshold. If the observed trip count is greater than the threshold, then the histogram update is skipped. Other similar optimizations (lazy initialization, fast-path and slow-path separation, among others) are useful in reducing the overhead of instrumentation.

MACVEC resolves measurements by thread. The data collection phase records each thread's information separately. MACVEC generates conservative recommendations from this per-thread information. For instance, MACVEC reports accesses to the arrays as aligned only if array references from all threads are aligned.

The steps to generate recommendations for each instrumentation case are explained below:

1. **Loop Trip Count:** The loop trip count is compared against a threshold (1024) to estimate whether vectorizing the loop will likely be inefficient. If so, MACVEC recommends inserting `#pragma loop_count` so that the compiler is aware of the low trip count before attempting to vectorize the loop.
2. **Stride Analysis:** If the measurements for array references indicate that the code exhibits strides that are not of length 1, MACVEC recommends converting array-of-structs references to struct-of-arrays references. If the stride values are more than 4 cache lines apart and if the code is being compiled for the Xeon Phi coprocessor, MACVEC recommends adding `#pragma prefetch <indirect-array>` and using the `-opt-gather-scatter-unroll`.
3. **Array Alignment Check:** The first-referenced address of the referenced array is used to understand the alignment of the data structure. If *all* arrays are aligned and if the loop is vectorizable, MACVEC recommends using the `#pragma vector aligned` directive. If arrays are not aligned or if they are mutually aligned to the same alignment value, MACVEC recommends aligning the arrays. Arrays can be requested to be aligned using `__mm_malloc()` for heap memory and using the `__attribute__((aligned(64)))` clause for global, static and stack memory.
4. **Non-temporal and Streaming Stores:** Arrays inside vectorizable loops that are written but never read within the loop, which are accessed using unit strides without a mask register and which exhibit high reuse distance (derived using reuse distance analysis) are good candidates for using streaming store instructions. In such cases, MACVEC recommends using the `-opt-streaming-stores=always` option. If an array simply exhibits low reuse and if the loop is vectorizable, MACVEC recommends adding the `#pragma vector nontemporal` directive for that array instead of recommending streaming stores.
5. **Array-Overlap Check:** MACVEC uses the difference between the lower and upper addresses of the referenced array as the span of the array. Using the calculated spans, MACVEC checks whether array references overlap. If the pointers do not overlap, MACVEC recommends adding the `restrict` keyword to the relevant pointer declarations.
6. **Branch Path Analysis:** Based on the branch outcomes, MACVEC determines whether the branch evaluates to mostly ($\geq$85 %) true, mostly ($\geq$85 %) false, always true or always false. If the branch was observed to evaluate to always true or always false, MACVEC recommends indicating the branch outcomes to the compiler using the `__builtin_expect()` attribute.

5 Case Studies

Four types of case studies are reported. The goal of the first case study was to get an upper bound on the performance improvement which could potentially be obtained by fully effective vectorization. The goal for the second set of case studies was to validate that MACVEC would arrive at the same recommendations for supplying additional information to the compiler as human experts. The third set of case studies applies the full workflow to small benchmark applications that had previously been hand-tuned by experts. This set of case studies includes codes from benchmark suites (including two from the Rodinia benchmarks) and one moderate-sized application. The goal for the fourth set of case studies was to check whether MACVEC could generate recommendations that improve the running time of the presumably well-tuned standard applications.

5.1 Execution Environment

All performance measurements and tests were carried out on the Stampede supercomputer at the Texas Advanced Computing Center. Each node on Stampede is comprised of two eight-core Intel® Xeon E5-2680 processors. Each of the eight cores on a chip has a 32 KB L1 instruction cache, a 32 KB L1 data cache and a 256 KB L2 cache. The L3 cache (20 MB) is shared by all eight cores on the chip. Each node also contains an Intel® Xeon Phi™ (Knights Corner)[3] coprocessor. Profiling runs both before and after optimization used the code generated by the Intel® Composer XE 2013 SP1 Update 3 (v14.0.3) compiler using the `-O3` and `-fopenmp` flags. Applications compiled to run on the Xeon Phi used the `-mmic` flag as well. Thus, all applications were run with parallelization enabled using OpenMP. Although we have used OpenMP for parallelization, we believe the framework we describe in this paper will be applicable to codes using other parallelization techniques such as MPI, Intel® Cilk™ Plus[4], etc. All applications run on the Xeon processor were run with 16 threads while those on the Xeon Phi coprocessor were run with 244 threads. Applications were run on the Xeon Phi coprocessor using the native mode. Performance results are based on timing the computational kernel, as printed in the application output. Code instrumented by MACVEC was run on the Xeon processor only[5].

5.2 Upper Bound Case Studies

An upper bound on the potential performance enhancement obtainable through enhancing vectorization can be estimated by applying the first three steps of the workflow to an application and then determining the percentage of the execution

[3] http://software.intel.com/en-us/mic-developer.
[4] https://software.intel.com/en-us/intel-cilk-plus.
[5] The Rose compiler framework is not yet available on the Intel Xeon Phi coprocessors hence the code could be instrumented to run only on the Intel Xeon processor and not the Intel Xeon Phi coprocessor.

Table 1. Time spent in loops not fully-vectorized in the Rodinia suite.

Application	Execution time of non-fully-vectorized loops	Main reason(s) for not fully Vectorizable (as per compiler's static analysis)
backprop	32.52 %	Vector dependence.
euler	12.42 %	Non-standard loop, vector dependence.
euler_double	78.99 %	Non-standard loop.
pre_euler	75.94 %	Non-standard loop, vector dependence.
pre_euler_double	71.60 %	Non-standard loop, vector dependence.
heartwall	7.03 %	Vector dependence, statement cannot be vectorized.
lavaMD	37.42 %	Vector dependence.
kmeans	19.54 %	Vector dependence, non-standard loop.
leukocyte	35.01 %	Unsupported loop, vector dependence.
srad_v1	48.45 %	Vector dependence.
streamcluster	85.58 %	Unsupported loop, vector dependence

time consumed by the set of loops that are not fully vectorized. Table 1 shows the percentage of such exectution time for codes in the Rodinia [4] benchmark suite. Codes that finished executing in less than 10 seconds were not used in this study. The percentage of execution time in the loops which can, in the best-case scenario, potentially benefit from additional vectorization ranges from about 85 % for StreamCluster to about 7 % for Heartwall. Indeed, some of these loops may not be vectorizable because of backward dependencies or irregular strides. However the data does suggest that significant performance enhancements can be obtained if loops are fully vectorized.

Table 1 also shows the dominant reasons why loops were not vectorized. In many cases, the compiler assumed dependence among iterations of the (inner or outer) loop or the loop was not a counted loop with a single entry and a single exit. The dependence information can be verified using dynamic profiling.

5.3 Validation Case Studies

Satish et al. [21] and Krishnaiyer et al. [12] present a variety of codes where experts in compilers and performance analysis manually identified specific loops. These loops were characterized by required modifications in compiler directives or pragmas and relatively modest code changes that could enhance the degree and effectiveness of vectorization. Among these, the NBody and STREAM [16] (and a subset of the LCD benchmark codes) were chosen for determining the effectiveness of MACVEC's dynamic profiling. The NBody code implements a standard $O(N^2)$ algorithm to calculate force among a given set of bodies. The STREAM benchmark measures the memory bandwidth based on simple loops

that copy values between arrays, possibly performing some arithmetic operations in the process. We use a variant of the STREAM benchmark that allocates memory dynamically. The LCD benchmarks are a collection of 100 Fortran loops that test the vectorization effectiveness. We use two loops from a C version of this benchmark for validation. The source-level pragmas, tuning modifications and compiler options were removed from the codes and the full workflow applied to each of these codes. The results from the runtime-measurement-based studies matched the output of the human experts.

5.4 Benchmark Case Studies

The LavaMD and SRAD codes from the Rodinia [4] benchmark suite and the Conjugate Gradient (CG) code from the NAS benchmarks [1] suite were used as test applications. The Rodinia codes were chosen because Rodinia focuses on benchmarking for accelerators. The LavaMD code calculates particle potential and relocation due to mutual forces between particles within a large 3D space. The SRAD code is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). The NAS-CG code, with its unpredictable memory access patterns and considerable synchronization, provides a different execution pattern than the more regularly structured computations of applications in the Rodinia suite.

Table 2. Performance results on benchmark applications.

Category	Application	Sources of improvement	Improvement on Xeon	Improvement on Xeon Phi
Validation case studies	NBody	Array alignment.	0.93x	1.45x
	STREAM	Array alignment non-temporal stores, restrict keyword.	copy: 1.06x scale: 1.41x add: 1.30x triad: 1.29x	copy: 1.00x scale: 1.32x add: 1.29x triad: 1.30x
Benchmark case studies	NAS CG	Loop count, gather/scatter unroll, prefetch pragma.	1.06x	2.18x
	LavaMD	Restrict keyword.	2.19x	8.99x
	SRAD	Array alignment.	0.99x	1.09x
Application case studies	MILC	AOS to SOA transformation.	1.10x	1.60x
	LBM	Non-temporal stores, restrict keyword.	1.06x	1.20x
	LULESH	Restrict keyword.	1.03x	1.00x
-	Overall (geo. mean)	-	**1.18x**	**1.55x**

5.5 Application Case Studies

We chose MILC [22], LBM [20] and LULESH [10] for the application case studies because these codes are known to have complex loop structures and complex data structures and thus offer opportunities for potential benefit from dynamic profiling and analysis. The version of MILC used in these studies is used as a benchmark for system acceptance by the National Science Foundation (NSF). The LBM code[6] was provided to us by Carlos Rosales-Fernandez of the Texas Advanced Computing Center. We used the optimized version of LULESH [11] available from the Lawrence Livermore National Laboratory website[7].

5.6 Analysis of Case Studies

The validation case studies demonstrated that the automatic process for dynamic profiling and recommending code modifications matches the recommendations of human experts. These case studies indicated that the automated process for integration of dynamic profiling information into the compilation process can frequently yield substantial performance improvement with a small effort investment. Only minor changes in the application were required except for conversion of Arrays-of-Structures to Structures-of-Arrays.

The benchmark applications were tuned by changing at most ten lines of source code. Table 2 shows the applications, the sources of improvement and the resulting speedup on the Intel® Xeon processor and the Intel® Xeon Phi coprocessors. Three of the eight codes (LBM, LavaMD and Lulesh) were optimized using compiler flags alone. The speedup values are medians of five consecutive runs. The extent of the difference between the original and optimized running times varies from a 7 % regression to 8.9x speedup.

Data layout can have a significant impact on performance, as illustrated in the case of MILC. Using a structure-of-arrays layout to make consecutive references have a unit stride enables the use of vector-loads and vector-stores instead of gathers and scatters. Vector-loads and vector-stores have a shorter execution latency in the front-end of the processor and have a much better utilization of the memory system, as many requests now map to a single cache line.

5.7 Safety of Recommended Optimizations

The optimization process defined and applied in this paper has captured and structured the knowledge necessary to generate applicable recommendations for optimization. This step is particularly important for vectorization-related optimizations which require in-depth knowledge of compiler pragmas and runtime libraries (knowledge not generally known to application developers and users). The measurements from dynamic profiling, and thus the recommendations are, however, specific to the input chosen for dynamic profiling so that the recommendations may not be valid or at least as effective, across all possible input sets.

[6] http://code.google.com/p/mplabs.
[7] https://codesign.llnl.gov/lulesh.php.

The recommendations to inform the compiler of loop counts, array access strides, existence of streaming or non-temporal stores and branch path outcomes are "safe" in the sense that the correctness of the application will not be impacted if the information is not valid for different inputs although performance may be impacted. The recommendations informing the compiler of array alignments and array overlap (vector dependence) are not safe if alignments or array overlap depend on inputs or control flow path. It is, however, possible to generate source code checks to verify that the recommended optimization is valid for the current input and execution environment. These checks will cause invocation of the appropriate one of the vectorized- or the non-vectorized-version of the loop. Application developers will usually be able to readily verify safety of a recommended optimization due to their familiarity with the application code. These source code checks for safety can be automatically generated and presented to the user for use if needed.

To test the robustness of the vectorization optimizations, we ran MILC with 19 different inputs (formed by doubling each of 19 values accepted by the code as input, one by one). We chose to run this test on MILC (instead of any other benchmark) because the MILC application is relatively complex and also because the MILC input affects not just the operating problem size but also the control paths taken by the code – statements that are not applicable to the other codes. The MILC application includes some basic tests (that are roughly equivalent to assertion failures) which ensure that portions of the computation are valid. Apart from ensuring that our optimizations to MILC did not cause assertion failures, we also verified the 'residue' value printed by the application at the end of its output was the same for the naive and optimized versions of the application. The optimized code on the different inputs ran 4 % to 25 % faster on the Intel Xeon processor and about 36 % faster on the Intel Xeon Phi coprocessor. We omit the details of the experiment due to space constraints.

6 Related Research

There have been previous efforts to combine measurements from static and dynamic analyses. In recent years, most compilers have added profile guided optimization capabilities where a user-selected option causes the compiler to generate instrumentation to gather runtime information on specific execution behaviors of the program and have the compiler use this information in future compilations. There have been at least two such performance optimization systems [3,5]. Oancea and Rauchwerger [17] have combined static analysis and dynamic profiling to enhance parallelism in loop, resolving the independence of the loop's memory references using runtime analysis. Vector Seeker [6] optimistically measures the vector parallelism using dynamic profiling. Hornung and Keasler [9] argue that efficient vectorization is possible with current compiler technology and offer suggestions on how it can be accomplished. Multiple other works [14,23,24] analyze data dependencies in order to discover potential parallelism.

More recent research in this area done by Holewinski et al. [8] describes an approach to infer a program's SIMD parallelization potential by analyzing

the dynamic data-dependence graph derived from a sequential execution trace. Maleki et al. [15] give a evaluation of how well modern compilers (as of 2011) vectorize. They conclude that there is a gap between modern compiler auto-vectorized loops and loops which could be successfully vectorized manually with additional information on the execution behavior. Their paper was a motivation for the research and tools of our work. Other works [12,13,21] show how human experts can combine knowledge of compilers and applications to enhance vectorization across multiple types of applications.

Similar to MACPO, Intel Advisor XE[8] [2] "Survey" and Advisor "Suitability" tools profile hotspots, and make recommendations on how source code and parallel runtime could be tweaked or modified to achieve greater (threading SMP) parallelism, and hence better performance. An analogous form of Advisor could be considered, that helps improve vector parallelism for targets with SIMD hardware. The results of this paper bolster the motivation for such an extension.

7 Summary and Future Work

A systematic and comprehensive process for integration of information from dynamic profiling into the compilation process was formulated, implemented, and applied. The steps requiring detailed knowledge of the compiler, analysis of the vectorization reports, and determining the relevant code modifications (i.e. steps (b) through (g) of the Workflow) have been automated. Our results on various workloads show a geomean of 1.5x improvement from supplementing static analysis with dynamically-profiled analysis of six conditions, and using commercially-de facto-standard pragmas and code modifications to communicate the results back to the compiler, under selective user control. This pilot study was found compelling enough to motivate possible inclusion of a MACVEC-like mechanism in the Intel Advisor tool, which uses various analyses to guide users to improved threading and vectorization.

One of the byproducts of this work will be a user guide to vectorization which will illustrate how applications should be structured to take maximum advantage of vector instructions present on today's computer architectures. Application of this process may be most beneficial for the emerging generations of many-core accelerator chips where performance is heavily dependent on both vectorization and parallelization.

There are three primary tasks to be completed in the future. The workflow will be integrated into the PerfExpert [7] framework enabling automated implementation of the recommended optimizations and safety checks. Second, instrumentation and the analysis will be extended from the currently-supported C and C++ languages to Fortran as well. Finally, this automated process will be extended to optimizations other than vectorization by replicating the automated process for the optimization reports generated by the compiler.

[8] http://software.intel.com/en-us/intel-advisor-xe.

Acknowledgments. This work is funded in part by Intel corporation and by the National Science Foundation under OCI award #0622780.

References

1. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks - summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing 1991, pp. 158–165. ACM, New York (1991)
2. Brett, B., Kumar, P., Kim, M., Kim, H.: CHiP: a profiler to measure the effect of cache contention on scalability. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops, IPDPSW 2013, pp. 1565–1574. IEEE Computer Society, Washington, DC (2013)
3. Callahan, D., Dongarra, J., Levine, D.: Vectorizing compilers: a test suite and results. In: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Supercomputing 1988, pp. 98–105. IEEE Computer Society Press, Los Alamitos (1988)
4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 44–54, October 2009
5. Chung, I.H., Cong, G., Klepacki, D., Sbaraglia, S., Seelam, S., Wen, H.F.: A framework for automated performance bottleneck detection. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–7, April 2008
6. Evans, G.C., Abraham, S., Kuhn, B., Padua, D.A.: Vector seeker: a tool for finding vector potential. In: Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing, WPMVP 2014, pp. 41–48. ACM, New York (2014)
7. Fialho, L., Browne, J.: Framework and modular infrastructure for automation of architectural adaptation and performance optimization for HPC systems. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 261–77. Springer, Heidelberg (2014)
8. Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.N., Rountev, A., Sadayappan, P.: Dynamic trace-based analysis of vectorization potential of applications. SIGPLAN Not. **47**(6), 371–82 (2012)
9. Hornung, R., Keasler, J.: A case for improved C++ compiler support to enable performance portability in large physics simulation codes. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2013)
10. Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B.L., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., Still, C.H.: Exploring traditional and emerging parallel programming models using a proxy application. In: Parallel and Distributed Processing Symposium, International, pp. 919–932 (2013)
11. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Technical report LLNL-TR-641973, Lawrence Livermore National Laboratory (2013)
12. Krishnaiyer, R., Kultursay, E., Chawla, P., Preis, S., Zvezdin, A., Saito, H.: Compiler-based data prefetching and streaming non-temporal store generation for the intel(r) xeon phi(tm) coprocessor. In: 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops Ph.D. Forum (IPDPSW), pp. 1575–1586, May 2013

13. Kristof, P., Yu, H., Li, Z., Tian, X.: Performance study of simd programming models on intel multicore processors. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops Ph.D. Forum (IPDPSW), pp. 2423–2432, May 2012
14. Larus, J.: Loop-level parallelism in numeric and symbolic programs. IEEE Trans. Parallel Distrib. Syst. **4**(7), 812–26 (1993)
15. Maleki, S., Gao, Y., Garzaran, M., Wong, T., Padua, D.: An evaluation of vectorizing compilers. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 372–382, October 2011
16. McCalpin, J.D.: A survey of memory bandwidth and machine balance in current high performance computers. IEEE TCCA Newsl. 19–25 (1995)
17. Oancea, C.E., Rauchwerger, L.: Logical inference techniques for loop parallelization. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 509–520. ACM, New York (2012)
18. Quinlan, D.J.: ROSE: compiler support for object-oriented frameworks. Parallel Process. Lett. **10**(2/3), 215–26 (2000)
19. Rane, A., Browne, J.: Enhancing performance optimization of multicore/multichip nodes with data structure metrics. ACM Trans. Parallel Comput. **1**(1), 3:1–3:20 (2014)
20. Rosales, C., Whyte, D.S.: Dual grid lattice boltzmann method for multiphase flows. Int. J. Numer. Meth. Eng. **84**(9), 1068–84 (2010)
21. Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., Dubey, P.: Can traditional programming bridge the Ninja performance gap for parallel computing applications? In: Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA 2012, pp. 440–451. IEEE Computer Society, Washington, DC (2012)
22. Shi, G., Kindratenko, V., Gottlieb, S.: The bottom-up implementation of one MILC lattice QCD application on the cell blade. Int. J. Parallel Program. **37**(5), 488–507 (2009)
23. Zhong, H., Mehrara, M., Lieberman, S., Mahlke, S.: Uncovering hidden loop level parallelism in sequential applications. In: IEEE 14th International Symposium on High Performance Computer Architecture, HPCA 2008, pp. 290–301, February 2008
24. Zhuang, X., Eichenberger, A., Luo, Y., O'Brien, K., O'Brien, K.: Exploiting parallelism with dependence-aware scheduling. In: 18th International Conference on Parallel Architectures and Compilation Techniques, PACT 2009, pp. 193–202, September 2009