Loop-Oriented Array- and Field-Sensitive Pointer Analysis for Automatic SIMD Vectorization

Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue
School of Computer Science and Engineering, UNSW, Sydney, Australia 2052
{ysui, fanx, haozhou, jingling}@cse.unsw.edu.au

Abstract

Compiler-based auto-vectorization is a promising solution to automatically generate code that makes efficient use of SIMD processors in high performance platforms and embedded systems. Two main auto-vectorization techniques, superword-level parallelism vectorization (SLP) and loop-level vectorization (LLV), require precise dependence analysis on arrays and structs in order to vectorize isomorphic scalar instructions and/or reduce dynamic dependence checks incurred at runtime.

The alias analyses used in modern vectorizing compilers are either intra-procedural (without tracking inter-procedural data-flows) or inter-procedural (by using field-insensitive models, which are too imprecise in handling arrays and structs). This paper proposes an inter-procedural **L**oop-oriented **P**ointer **A**nalysis, called LPA, for analyzing arrays and structs to support aggressive SLP and LLV optimizations. Unlike field-insensitive solutions that preallocate objects for each memory allocation site, our approach uses a fine-grained memory model to generate *location sets* based on how structs and arrays are accessed. LPA can precisely analyze arrays and nested aggregate structures to enable SIMD optimizations for large programs. By separating the location set generation as an independent concern from the rest of the pointer analysis, LPA is designed to reuse easily existing points-to resolution algorithms.

We evaluate LPA using SLP and LLV, the two classic vectorization techniques on a set of 20 CPU2000/2006 benchmarks. For SLP, LPA enables it to vectorize a total of 133 more basic blocks, with an average of 12.09 per benchmark, resulting in the best speedup of 2.95% for `173.applu`. For LLV, LPA has reduced a total of 319 static bound checks, with an average of 22.79 per benchmark, resulting in the best speedup of 7.18% for `177.mesa`.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, Languages, Performance

Keywords Pointer Analysis, SIMD, Loop, Array, Field

LCTES'16, June 13–14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4316-9/16/06...$15.00
http://dx.doi.org/10.1145/2907950.2907957

1. Introduction

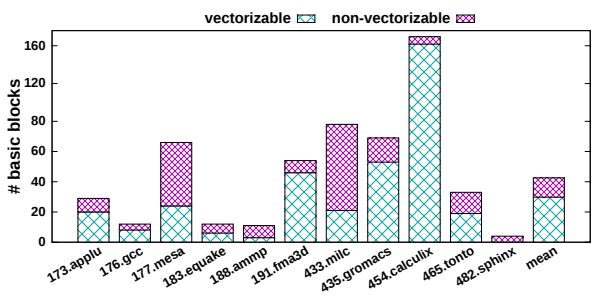
SIMD (Single-Instruction Multiple-Data) technology is ubiquitous in both desktop computers and embedded systems (e.g., Intel's AVX, ARM's NEON and MIPS's MDMX/MXU) and DSPs (e.g., Analog Devices's SHARC and CEVA's CEVA-X) in order to improve performance and energy-efficiency. Existing vectorizing compilers (e.g., LLVM) enable two main vectorization techniques to extract data-level parallelism from a loop: (1) basic block or superword-level parallelism (SLP) vectorization [3, 7, 19, 22, 33], which packs isomorphic scalar instructions in the same basic block into vector instructions, and (2) loop-level vectorization (LLV) [15, 16, 21, 29], which combines multiple consecutive iterations of a loop into a single iteration of vector instructions.

Generating efficient vector code using these two optimizations relies on precise dependence analysis. For example, in order to successfully vectorize isomorphic instructions in Figure 1(a), SLP checks conflicting memory accesses using the alias information before packing the four isomorphic instructions into a vector instruction (line 2 in Figure 1(b)). Given a write to an element of an array (e.g., A[0] = ...), any subsequent store or load (e.g., = B[1]) should not access the same memory address as &A[0]. Figure 1(c) shows another example that can be vectorized by LLV. In order to disambiguate memory addresses inside a loop where aliases cannot be determined statically, LLV performs loop versioning by inserting code that performs runtime alias checks to decide whether the vectorized version or scalar version of a loop is executed. As illustrated in Figure 1(d), LLV creates two versions of the loop and places code that checks, at run time, whether the pointers A and B point to disjoint memory regions. In the case of any overlap being detected, the scalar version (line 3) is executed. Otherwise, the vectorized version (line 5) is used, instead.

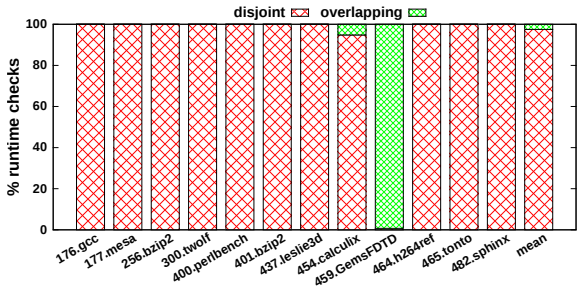
1.1 Motivation

A conservative alias analysis may cause either some vectorization opportunities to be missed or some redundant but costly runtime checks to be introduced. Figure 2 shows the impact of LLVM's BasicAA alias analysis on the effectiveness of SLP and LLV on all relevant SPEC CPU2000/2006 benchmarks compiled by LLVM.

Figure 2(a) gives the number of vectorizable and non-vectorizable basic blocks by SLP in all 11 relevant SPEC CPU2000/2006 benchmarks. A SPEC benchmark is included if and only if SLP-related must-not-alias queries are issued to some basic blocks but not answered positively. These are the SPEC benchmarks for which SLP may benefit from more precise alias information. A basic block that receives some SLP-related alias queries is said to be *vectorizable* if SLP can generate at least one vectorized instruction for the basic block. `433.milc` has the largest number of basic blocks (57) that cannot be vectorized, representing 73.07% of the



(a) SLP: number of vectorizable and non-vectorizable basic blocks



(b) LLV: percentage of runtime checks for determining disjoint and overlapping memory (i.e., array) regions (under the reference inputs)

Figure 1: Impact of LLVM's BasicAA alias analysis on the effectiveness of SLP and LLV in LLVM.

```
1  foo (float *A, float *B) {
2     A[0] = B[0]; A[1] = B[1]; A[2] = B[2]; A[3] = B[3];
3  }
```

(a) Code before SLP

```
1  foo (float *A, float *B) {
2     A[0:3] = B[0:3];
3  }
```

(b) Code after SLP

```
1  foo (float *A, float *B) {
2     for (int i = 0; i < N; i++) A[i] = B[i] + K;
3  }
```

(c) Code before LLV (N%4==0)

```
1  foo (float *A, float *B) {
2     if ((&A[N-1] ≥ &B[0]) && (&B[N-1] ≥ &A[0]))
3        for (int i = 0; i < N; i++) A[i] = B[i] + K;
4     else
5        for (int i = 0; i < N; i+=4) A[i:i+3] = B[i:i+3] + K;
6  }
```

(d) Code after LLV (N%4==0)

Figure 2: Examples for SLP and LLV vectorizations.

total number of basic blocks (78) with alias queries. Across the 11 benchmarks, 30.04% of basic blocks are vectorizable on average.

Figure 2(b) gives the percentage of runtime alias checks that return "disjoint" or "overlapping" for the two memory regions (e.g., arrays) pointed by two pointers for all the 12 CPU2000/CPU2006 benchmarks that contain dynamic alias checks inserted by LLV. LLV relies on these checks to disambiguate the aliases that cannot be resolved at compile time. Compared to SLP, the impact of alias analysis on the effectiveness of LLV can be more pronounced for some benchmarks. Across the 12 benchmarks evaluated, an average of 96.35% of dynamic alias checks return "disjoint regions". In fact, the vectorized rather than scalar version of a loop is always executed in all the benchmarks except `454.calculix` and `459.GemsFDTD`. Thus, runtime checks are redundant and can be eliminated if a more precise alias analysis is applied, reducing both instrumentation and code-size overheads incurred.

1.2 Challenges and Insights

The main source of imprecision in alias analysis for SIMD vectorization is lack of a precise inter-procedural analysis for aggregate data structures including arrays and structs. The alias analysis used in LLVM is intra-procedural, which is overly conservative without tracking the inter-procedural data flows. Existing field-sensitive pointer analyses for C [4, 17] use a field-index-based approach to distinguish the fields by their unique indices (with nested structs expanded). However, this approach ignores the size information for each field, by treating all the fields as having the same size. *As C is not strongly-typed, the types of a pointer and its pointed-to objects may be incompatible due to casting. Such a field-sensitive solution may not obtain sound results to support SIMD optimizations.*

To the best of our knowledge, location sets [31] represent still the most sophisticated field-sensitive memory model to be used to enable pointer analysis for C programs. A location set $\langle off, s\rangle \in \mathbb{Z} \times \mathbb{N}$ represents a set of memory locations $\{off + i \times s \mid i \in \mathbb{Z}\}$ accessed from the beginning of a memory block B, where *off* is an offset within B and s is a stride, both measured in bytes. The stride s is 0 if the location set contains a single element. Otherwise, it represents an unbounded set of locations.

Although location sets are byte-precise when used in analyzing the fields in a struct, there are several limitations preventing them from being used to enable developing precise alias analyses for auto-vectorization. First, arrays are modeled monolithically, with all the elements in the same array collapsed. Second, the lengths of array are not recorded. Thus, an array inside a memory block is assumed to extend until the end of the block, making it difficult to handle nested arrays and structs accurately. Finally, the loop information, which is critical for loop-oriented optimization, such as SIMD vectorization, is ignored. Therefore, how to perform loop-oriented memory modeling for arrays and structs to enable precise alias analysis required for SIMD vectorization remains open.

1.3 Our Solution

To address the above challenges for analyzing arrays and nested data structures, including arrays of structs and structs of arrays, we introduce a fine-grained access-based memory modeling method that enables a **L**oop-oriented array- and field-sensitive **P**ointer **A**nalysis (LPA) to be developed, with one significant application to automatic SIMD vectorization. The novelty lies in disambiguating aliases by generating *access-based location sets* lazily so that location sets are dynamically created during the on-the-fly points-to resolution based on how arrays and structs are accessed.

Access-based location sets are a generalization of location sets [31] so that both arrays and structs are handled in a uniform manner. Unlike the location-set model [31], which ignores the loop information and does not distinguish the elements of an array, LPA leverages the loop trip count and stride information to precisely model array accesses including nested aggregate structures. The symbolic ranges of an array access expression are fully evaluated if they are statically determined (e.g., constant values) or partially evaluated using our value range analysis, developed based on LLVM's SCEV (SCalar EVolution) pass.

To make LPA scalable in whole-program SIMD optimizations for large programs, LPA provides tunable parameters to find a right balance between efficiency and precision by merging location sets. In addition, LPA separates memory modeling as an independent concern from the rest of the pointer analysis, thereby facilitating the development of different pointer analyses (e.g., flow-insensitive and flow-sensitive variants) with desired efficiency and precision tradeoffs by reusing existing pointer resolution frameworks.

This paper makes the following contributions:

- We introduce LPA, a new loop-oriented array- and field-sensitive inter-procedural pointer analysis based on access-based location sets built in terms of a lazy memory model.
- We apply LPA to improve the effectiveness of SLP and LLV, by enabling SLP to vectorize more basic blocks and LLV to insert fewer dynamic runtime checks.
- We evaluate LPA with 20 SPEC CPU2000/2006 benchmarks, for which SLP or LLV can benefit from more precise alias information. For SLP, LPA enables a total of 133 more basic blocks to be vectorized, with an average of 12.09 more per benchmark, resulting in the best speedup of 2.95% observed in `173.applu`. For LLV, LPA has successfully reduced a total of 319 static bound checks, with an average of 22.79 per benchmark, resulting in the best speedup of 7.18% in `177.mesa`. We also provide a detailed discussion about where our fine-grained alias analysis is applicable and its limitations.

2. Background

We introduce the partial SSA form used in LLVM for representing a program and the standard inclusion-based pointer analysis based on a simple field-insensitive memory model.

2.1 Program Representation

A program is represented in LLVM's partial SSA form [4, 8, 32]. The set of all program variables, $\mathcal{V}$, is separated into two subsets: $\mathcal{A}$ containing all possible targets, i.e., *address-taken variables* of a pointer and $\mathcal{T}$ containing all *top-level variables*, where $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$.

The following statements and expressions are relevant:

$$\begin{array}{lll} \text{Statement} & s ::= & p = \&a \mid p = q \mid q = e_p \mid e_p = q \\ \text{MemExpr} & e_p ::= & *p \mid p \rightarrow f \mid p[i] \end{array}$$

There are four types of statements: $p = \&a$ (ADDROF), $p = q$ (COPY), $q = e_p$ (LOAD), and $e_p = q$ (STORE), where $p, q \in \mathcal{T}$, $a \in \mathcal{A}$ and e_p denotes a memory access expression involving a pointer p, including a pointer dereference, a field access or an array access. Our memory expressions are considered to be ANSI-compliant. For example, given a pointer to an array, using pointer arithmetic to access anything other than the array itself has undefined behavior [5], thus not allowed in our model.

Top-level variables are put directly in SSA form, while address-taken variables are only accessed indirectly via memory access expressions. For an ADDROF statement $p = \&a$, known as an *allocation site*, a is a stack or global variable with its address taken or a dynamically created heap object (at, e.g., a `malloc()` site). Interprocedural parameter assignments and procedure returns are modeled using COPY statements (e.g., $p = q$).

Figure 3 gives a code fragment and its partial SSA form, where $p, q, y, z, t \in \mathcal{T}$ and $a, b, x \in \mathcal{A}$. Here, a is accessed indirectly at a store $*p = t$ by introducing a top-level pointer t in partial SSA form. Any field access $x.f$ via an address-taken variable is transformed into a field dereference via a top-level pointer, e.g., $q \rightarrow f$. Similarly, an array access, e.g., $x[i]$ is transformed to $q[i]$.

Passing arguments into and returning results from functions are modeled by copies. The complex statements like $*p = *q$ are decomposed into the basic ones $t = *q$ and $*p = t$ by introducing a top-level pointer t. Accessing a multi-dimensional array as in $q = p[i][j]$ is transformed into $q = p[k]$, where $k = i * n + j$ and n represents the size of the second dimension of the array.

p = &a;	***p = &a;***
a = &b;	***t = &b;***
	****p = t;***
q = &x;	***q = &x;***
x.f = y	***q->f = y***
x[i] = z	***q[i] = z***
(a) C code	(b) Partial SSA

Figure 3: A C code fragment and its partial SSA form.

2.2 Inclusion-Based Field-Insensitive Pointer Analysis

Figure 4 gives the rules used in a field- and flow-insensitive inclusion-based analysis [1] with statements transformed into constraints for points-to resolution until a fixed point is reached.

A field-insensitive solution [4, 8] treats every address-taken variable at its allocation site as a single abstract object. Field and array memory access expressions, $p \rightarrow f$ and $p[i]$, in terms of a pointer p are handled in the same way as a pointer dereference $*p$. The objects are pre-allocated so that the total number of objects remains unchanged during points-to resolution. The two pointer dereferences $*p$ and $*q$ are not aliases if the intersection of their corresponding points-to sets $pt(p)$ and $pt(q)$ is empty.

$$\texttt{[I-ALLOC]}\ \frac{p = \&a}{\{a\} \subseteq pt(p)} \qquad \texttt{[I-LOAD]}\ \frac{p = *q \quad a \in pt(q)}{pt(a) \subseteq pt(p)}$$

$$\texttt{[I-COPY]}\ \frac{p = q}{pt(q) \subseteq pt(p)} \qquad \texttt{[I-STORE]}\ \frac{*p = q \quad a \in pt(p)}{pt(q) \subseteq pt(a)}$$

Figure 4: Field-insensitive inclusion-based pointer analysis.

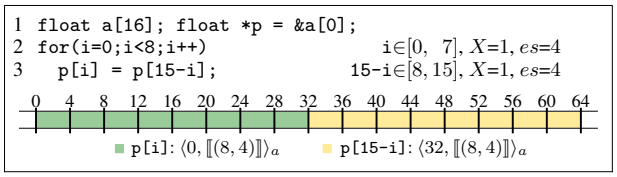
3. The LPA Analysis

We first describe our memory model (AMM) on access-based location sets (Section 3.1). We then discuss how to perform our loop-oriented array- and field-sensitive pointer analysis based on AMM (Section 3.2), including required value-range analysis and location set disambiguation. Finally, we focus on field unification and handling positive weight cycles (Section 3.3).

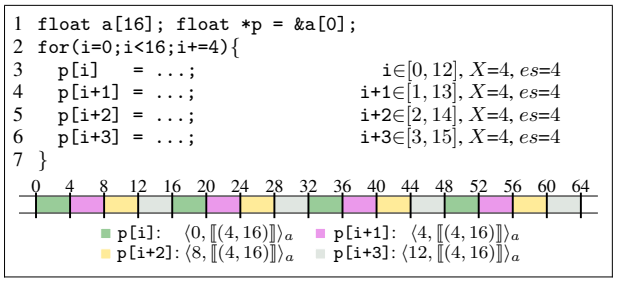
3.1 AMM: Access-based Memory Modeling

In the field-insensitive approach, an object at an allocation site is considered monolithically. In contrast, our access-based memory modeling achieves field-sensitivity by representing an abstract object in terms of one or more location sets based on how the object is accessed. A location set σ represents memory locations in terms of numeric offsets from the beginning of an object block. Unlike [31], which ignores the loop and array access information, AMM models field-sensitivity in accessing an array, e.g., $a[i]$ by maintaining a range interval $[lb, ub]$, where $lb, ub \in \mathbb{N}$ and an access step $X \in \mathbb{N}^+$ (with $X = 1$ if a is accessed consecutively inside a loop) by leveraging the loop information using our value-range analysis.

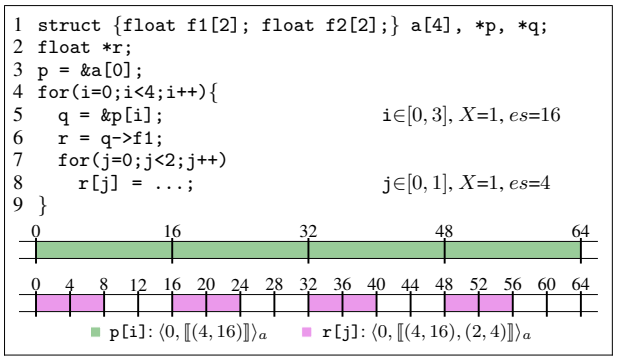
To precisely model the locations based on the array access information, we introduce a new concept called *access trip*, which



(a) An array with **consecutive** accesses



(b) An array with **non-consecutive** accesses



(c) **Nested arrays and structs** with consecutive accesses

Figure 5: Examples for access-based location sets.

is a pair (t, s) consisting of a trip count $t = (ub - lb)/X + 1$ and a stride $s = es * X$, where es is the size of an array element.

An access-based location set σ derived from an object a is:

$$\sigma = \langle off, [\![(t_1, s_1), ..., (t_m, s_m)]\!] \rangle_a \quad (1)$$

where $off \in \mathbb{N}$ is an offset from the beginning of object a, and $T = [\![(t_1, s_1), ..., (t_m, s_m)]\!]$ is an access-trip stack containing a sequence of (trip count, stride) pairs for handling a nested struct of arrays. Here, m is the depth of an array in a nested aggregate structure. In Figure 5(c), $a[4]$ is an array of structs containing two array fields $f1[2]$ and $f2[2]$ whose depths are $m = 2$.

Finally, $LS(\sigma)$ denotes a set of positions from the beginning of an object a:

$$LS(\sigma) = \{off + \sum_{k=1}^{m} (n_k \times s_k) \mid 0 \le n_k < t_k\} \quad (2)$$

Let us go through three examples with consecutive and non-consecutive accesses to single and nested arrays in Figure 5.

EXAMPLE 1 (Consecutive Array Access). *Figure 5(a) shows symmetric assignments from the last eight to the first eight elements of an array, $a[16]$. Two expressions $p[i]$ and $p[15-i]$, where $i \in [0, 7]$ and $(15 - i) \in [8, 15]$, always access disjoint memory locations, as highlighted in green and yellow, respectively. Therefore, loop-level vectorization can be performed without adding dynamic alias checks due to the absence of dependences between $p[i]$ and $p[15-i]$.*

The location set for representing the consecutive accesses of $p[i]$ is $\sigma = \langle 0, [\![((7-0)/1+1, 4*1)]\!] \rangle_a = \langle 0, [\![(8, 4)]\!] \rangle_a$, where the size of an array element $es = 4$ and step $X = 1$. According to Equation 2, $LS(\sigma) = \{0 + n * 4 \mid 0 \le n < 8\}$ = {0, 4, 8, 12, 16, 20, 24, 28}. Similarly, the location set for $p[15-i]$ is $\sigma' = \langle 32, [\![((15-8)/1+1, 4*1)]\!] \rangle_a = \langle 32, [\![(8, 4)]\!] \rangle_a$, representing a set of locations with offsets: $LS(\sigma') = \{32, 36, 40, 44, 48, 52, 56, 60\}$. Therefore, when accessing an array element, σ and σ' always refer to disjoint memory locations.

EXAMPLE 2 (Non-Consecutive Array Access). *Figure 5(b) gives a program obtained after loop unrolling with a step $X = 4$. Four expressions p[i], p[i+1], p[i+2], and p[i+3] also access disjoint memory locations, which can be disambiguated statically without inserting runtime checks by* LLV. *The location set for representing the non-consecutive accesses of $p[i]$ is $\langle 0, [\![((12-0)/4+1, 4*4)]\!] \rangle_a = \langle 0, [\![(4, 16)]\!] \rangle_a$, where $i \in [0, 12]$, $es = 4$ and $X = 4$, representing a set of positions from the beginning of object a: {0, 16, 32, 48}, which is disjoint with all other location sets shown.*

EXAMPLE 3 (Nested Array Access). *Figure 5(c) gives a more complex program that requires several (trip count, stride) pairs to model its array access information precisely. Here $a[4]$ is an array of structs containing two array fields $f1[2]$ and $f2[2]$. The outer loop iterates over the array $a[4]$ via $p[i]$ while the inner loop iterates over the array elements of field $f1[2]$ via $r[j]$.*

The location set for representing the consecutive accesses of $p[i]$ is $\langle 0, [\![((3-0)/1+1, 16*1)]\!] \rangle_a = \langle 0, [\![(4, 16)]\!] \rangle_a$, where $i \in [0, 3]$, $es = 16$ (the size of the structure including four floats) and $X = 1$.

The location set of $r[j]$ in accessing the inner field $f1[2]$ is $\langle 0, [\![(4, 16), ((1 - 0)/1 + 1, 4 * 1)]\!] \rangle_a = \langle 0, [\![(4, 16), (2, 4)]\!] \rangle_a$, representing a set of locations with offsets: {0, 4, 16, 20, 32, 36, 48, 52}, where $j \in [0, 1]$, $es = 4$ and $X = 1$.

3.2 Pointer Analysis Based on AMM

AMM is designed by separating the location set generation as an independent concern from the rest of the pointer analysis. It facilitates the development of a more precise field- and array-sensitive analysis by reusing existing points-to resolution algorithms.

Figure 6 gives the rules for an inclusion-based pointer analysis based on AMM. Unlike the field-insensitive counterpart given in Figure 4, the points-to set of a field-sensitive solution contains location sets instead of objects. For each allocation site, e.g., $p = \&a$ (`[S-ALLOC]`), the location set $\sigma = \langle 0, [\![]\!] \rangle_a$ is created, representing the locations starting from the beginning of object a. `[S-LOAD]` and `[S-STORE]` handle not only pointer dereferencing but also field and array accesses by generating new location sets via $GetLS$. `[S-COPY]` is the same as in the field-insensitive version.

For a field access $p \to f$, $GetLS(\langle off, T \rangle_a, p \to f)$ generates a new location set by adding *off* with the offset (measured in bytes after alignment has been performed) of field f in object a while keeping the access trip information T unchanged.

For an array access, there are two cases. In one case, the array index i is a constant value C so that $p[C]$ accesses a particular array element. AMM generates a new location set with a new offset $off + C * es$. In the other case, i is a variable $i \in [lb, ub]$ with an access step X, where lb, ub and X are obtained by our value-range analysis. As a range interval obtained statically by our analysis is always over-approximated, the resulting range is the intersection between $[lb, ub]$ and array bounds, i.e., $[lb', ub'] = [0, m - 1] \sqcap [lb, ub] = [\max(0, lb), \min(m - 1, ub)]$, where m is the length of the array.

$$\texttt{[S-ALLOC]}\ \frac{p = \&a \quad \sigma = \langle 0, [\![]\!]\rangle_a}{\{\sigma\} \subseteq pt(p)} \qquad \texttt{[S-LOAD]}\ \frac{q = e_p \quad \sigma \in pt(q) \quad \sigma' = GetLS(\sigma, e_q)}{pt(\sigma') \subseteq pt(q)}$$

$$\texttt{[S-COPY]}\ \frac{p = q}{pt(q) \subseteq pt(p)} \qquad \texttt{[S-STORE]}\ \frac{e_p = q \quad \sigma \in pt(p) \quad \sigma' = GetLS(\sigma, e_p)}{pt(q) \subseteq pt(\sigma')}$$

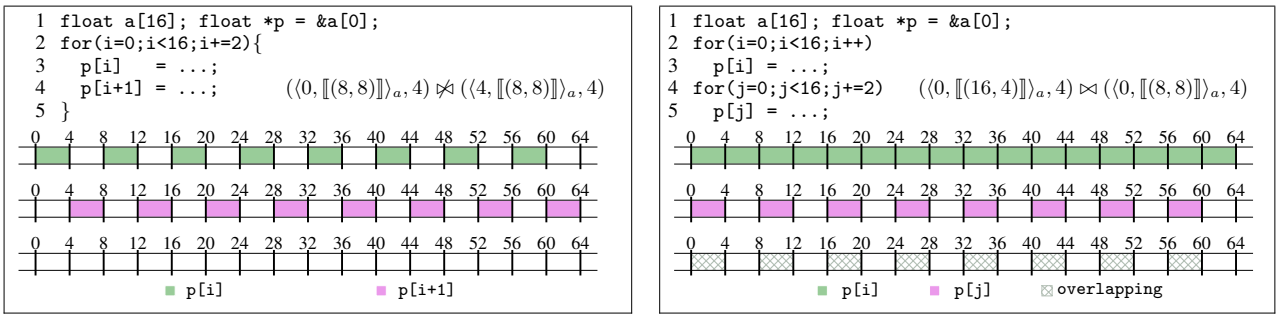
$$GetLS(\langle off, T\rangle_a, e_p) = \begin{cases} \langle off, T\rangle_a & \text{if } e_p \text{ is } *p \\ \langle off + off_f, T\rangle_a & \text{else if } e_p \text{ is } p \to f, \text{ where } off_f \text{ is the offset of field } f \text{ in array object } a \\ \langle off + C * es, T\rangle_a & \text{else if } e_p \text{ is } p[i], \text{ where } i \text{ is constant C} \\ \langle off + lb * es, T.\texttt{push}(\frac{ub'-lb'}{X} + 1, X * es)\rangle_a & \text{else if } e_p \text{ is } p[i], \text{ where } i \in [lb, ub] \text{ with step } X, \\ & [lb', ub'] = [lb, ub] \sqcap [0, m-1] \text{ and } m \text{ is size of array object } a \end{cases}$$

Figure 6: Rules for field- and array-sensitive inclusion-based pointer analysis equipped with an access-based memory model.

$$alias(e_p, e_q) = \begin{cases} true & \text{if } \exists\, \sigma'_p \in pt(p) \wedge \sigma'_q \in pt(q) : (\sigma_p, sz_p) \bowtie (\sigma_q, sz_q), \text{where } \sigma_p = GetLS(\sigma'_p) \wedge \sigma_q = GetLS(\sigma'_q) \\ false & \text{otherwise} \end{cases} \tag{3}$$

$$(\sigma_p, sz_p) \bowtie (\sigma_q, sz_q) = \begin{cases} true & \text{if } obj(\sigma_p) = obj(\sigma_q) \text{ and } \exists\, l_p \in LS(\sigma_p) \wedge l_q \in LS(\sigma_q) : (l_p < l_q + sz_q) \wedge (l_q < l_p + sz_p) \\ false & \text{otherwise} \end{cases} \tag{4}$$

Figure 7: Disambiguation of location sets (where $obj(\sigma)$ denotes the object on which σ is generated).



(a) Disjoint location sets

(b) Overlapping location sets

Figure 8: Examples for disjoint and overlapping location sets.

Finally, the new offset is $off + lb' * es$ and the new trip stack is generated by pushing the trip count and stride pair into the stack T, i.e., $T.\texttt{push}(\frac{ub'-lb'}{X} + 1, X * es)$, so that the hierarchical trip information is recorded when accessing arrays nested inside structs.

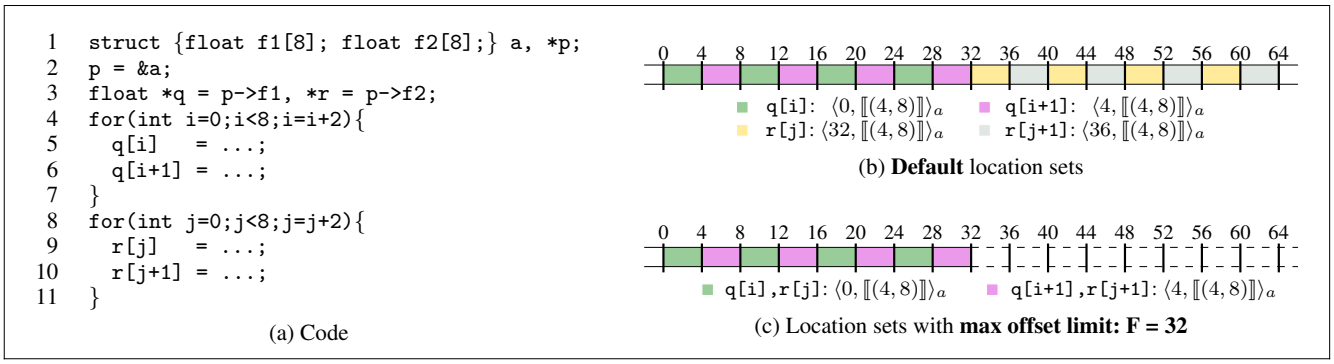
Value Range Analysis Our value range analysis estimates conservatively the range of values touched at a memory access based on LLVM's SCEV pass, which returns closed-form expressions for all top-level scalar integer variables (including top-level pointers) in the way described in [30]. This pass, inspired by the concept of *chains of recurrences* [2], is capable of handling any value taken by an induction variable at any iteration of its enclosing loops.

As we are interested in analyzing the range of an array index inside a loop in order to perform SIMD optimizations, we only extract the value range from an integer variable if it can be represented by an *add recurrence SCEV* expression. For other SCEV expressions that are non-computable in the SCEV pass or outside a loop, our analysis approximates their ranges as $[-\infty, +\infty]$ with their steps being $X = 1$. This happens, for example, when an array index is a non-affine expression or indirectly obtained from a function call.

An add recurrence SCEV has the form of $<se_1, +, se_2>_{lp}$, where se_1 and se_2 represent, respectively, the initial value (i.e. the value for the first iteration) and the step per iteration for the containing loop lp. For example, in Figure 5(b), the SCEV for the array index i inside the `for` loop at line 2 is $<0, +, 4>_2$, where its lower bound is $lb = 0$ and its step is $X = 4$.

The SCEV pass computes the trip count of its containing loop, which is also represented as a SCEV. A trip count can be non-computable. For a loop with multiple exits, the worst-case trip count is picked. Similarly, a loop upper bound is also represented by a SCEV, deduced from the trip count and step information.

Disambiguation of Location Sets In a field-insensitive analysis, two pointer dereferences are aliases if they may refer to the same object. In AMM, every object may generate multiple location sets. Two location sets can refer to disjoint memory locations even if they are generated originally from the same object.

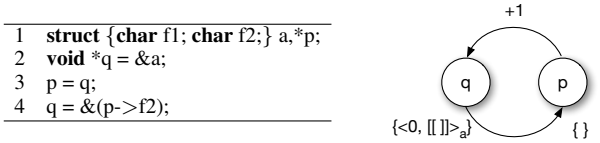


(a) Code

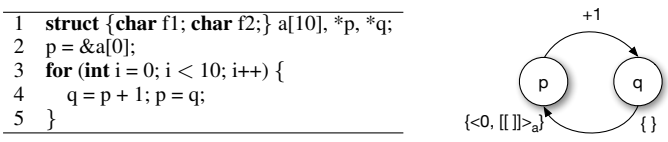
(b) **Default** location sets

(c) Location sets with **max offset limit: F = 32**

Figure 9: Field unification for location sets.



(a) C code and its PWC formed with field accesses



(b) C code and its PWC formed with array accesses

Figure 10: Handling of positive weight cycles.

Our analysis checks whether two memory expressions e_p and e_q are aliases or not by using both their points-to information and their memory access sizes sz_p and sz_q obtained from the types of the points-to targets of the two pointers p and q, as shown in Figure 7. We say that e_p and e_q are aliases if Equation 3 holds, where $(\sigma_p, sz_p) \bowtie (\sigma_q, sz_q)$ denotes that two locations may overlap, i.e., a particular memory location may be accessed by both e_p and e_q.

According to Equation 4, $(\sigma, sz_p) \bowtie (\sigma', sz_q)$ holds if and only if σ and σ' are generated from the same memory object (i.e., $obj(\sigma)$ = $obj(\sigma')$) and there exists an overlapping zone accessed by the two expressions based on the size information sz_p and sz_q (measured in bytes). In all other cases, two location sets, e.g., two generated from two different memory objects, are disjoint.

EXAMPLE 4 (Disjoint and Overlapping Location Sets). *Figure 8(a) illustrates disjoint memory accesses. Two expressions e_p and e_q are not aliases since their location sets are disjoint. According to Equation 1, the location sets of $p[i]$ and $p[i+1]$ are $\langle 0, [\![(8, 8)]\!]\rangle_a$ and $\langle 4, [\![(8, 8)]\!]\rangle_a$, respectively. The sizes of both accesses to the elements of an array with the float type are 4. According to Equation 4 $(\sigma_p, sz_p) \not\bowtie (\sigma_q, sz_q)$, $p[i]$ and $p[i+1]$ always access disjoint regions. In contrast, Figure 8(b) shows a pair of overlapping location sets, with their overlapping areas shown in gray.*

3.3 Field Unification and PWCs

Field Unification Optimization For some programs, a field-sensitive analysis may generate a large number of location sets due to deeply nested aggregates, which may affect the efficiency of points-to propagation during the analysis. To make a tradeoff between efficiency and precision, we introduce a simple yet effective unification technique call *field-unification*, which aims to reduce analysis overhead by merging existing location sets. It provides an offset limit parameter *F* for the starting offset of a location set. The parameter allows users to find a right balance between efficiency and precision by tuning the number of location sets.

A location set using field unification is represented by $\sigma = \langle off \,\%\, F, T\rangle_a$ and the trip stack of array access information remains unchanged in order to exploit vectorization opportunities.

EXAMPLE 5 (Field Unification). *Figure 9(a) gives a field unification example with a struct containing two array fields $f1[8]$ and $f2[8]$. Field $f1[8]$ is accessed via $q[i]$ and $q[i+1]$ inside the `for` loop (lines 4-7). Field $f2[8]$ is accessed via $r[j]$ and $r[j+1]$ inside the other loop (lines 8-11). The default location sets generated for the four memory accesses are shown in Figure 9(b). If we limit the maximum starting field offset to be 32, then the location set of $r[j]$ is merged into $q[i]$ and $r[j+1]$ is merged into $q[i+1]$, so that only two location sets are generated. However, for each loop, our memory modeling can still distinguish the two array accesses (e.g., q[i] and q[i+1]) even after unification, as illustrated in Figure 9(c).*

Handling Positive Weight Cycles With field-sensitivity, one difficulty lies in handling positive weight cycles (PWCs) [17] during points-to resolution. Without field-sensitivity [4, 18], a cycle on a constraint graph formed by copy edges is detected and collapsed to accelerate convergence during its iterative constraint resolution.

In a field-sensitive constraint graph, a cycle may contain a copy edge with a specific field offset, resulting in a PWC. Figure 10(a) shows a PWC with an edge from p to q, indicating a field offset causing infinite derivations unless field limits are bounded. Eventually, p and q always have the same solution. Thus, all derived fields are redundant and unnecessary for precision improvement. Simply collapsing p and q may be unsound, as they can point to other fields of the struct a during the on-the-fly derivation. To handle PWCs efficiently while maintaining precision, we follow [17, 20] by marking the objects in the points-to set of the pointers inside a PWC to be field-insensitive, causing all their fields to be merged.

AMM models both field and array accesses of an object. This poses another challenge for PWC handling as a cycle may involve pointer arithmetic when array elements are accessed. Figure 10(b) shows a PWC example simplified from `181.mcf`. The pointer `p` iterates over all the elements in an array of structs, $a[10]$, inside the

Program	KLOC	#Stmt	#Ptrs	#Objs	#CallSite
173.applu	3.9	3361	20951	159	346
176.gcc	226.5	215312	545962	16860	22595
177.mesa	61.3	99154	242317	9831	3641
183.equake	1.5	2082	6688	236	235
188.ammp	13.4	14665	56992	2216	1225
191.fma3d	60.1	119914	276301	6497	18713
197.parser	11.3	13668	36864	1177	1776
256.bzip2	4.6	1556	10650	436	380
300.twolf	20.4	23354	75507	1845	2059
400.perlbench	168.1	130640	296288	3398	15399
401.bzip2	8.2	7493	28965	669	439
433.milc	15	11219	30373	1871	1661
435.gromacs	108.5	84966	224967	12302	8690
436.cactusADM	103.8	62106	188284	2980	8006
437.leslie3d	3.8	12228	38850	513	2003
454.calculix	166.7	135182	532836	18814	23520
459.GemsFDTD	11.5	25681	107656	3136	6566
464.h264ref	51.5	55548	184660	3747	3553
465.tonto	143.1	418494	932795	28704	58756
482.sphinx3	25	20918	60347	1917	2775
Total	1208.2	1457541	3898253	117308	182338

Table 1: Program characteristics.

`for` loop. Simply marking object a as being field-insensitive may lead to a loss of precision. Although p and q can access any element in a, the two fields of an array element are still distinguishable, i.e., $p \rightarrow f1$ and $p \rightarrow f2$ refer to two different memory locations. Our analysis performs a partial collapse for array-related PWCs so that only the location sets generated by array accesses are merged, while the location sets generated by field accesses remain unchanged.

4. Evaluation

Our objective is to demonstrate that LPA (our loop-oriented array- and field-sensitive pointer analysis) can improve the effectiveness of SLP and LLV, two classic auto-vectorization techniques, on performing whole-program SIMD vectorization. For comparison purposes, we have selected publicly available sound alias analyses, LLVM's BasicAA and SCEVAA, as the baselines. *The field-index-based Andersen's analysis [4, 17], which ignores the size information for each field, is not considered due to its unsoundness in supporting SIMD optimization, as also discussed in Section 1.*

We have included all the SPEC CPU2000/CPU2006 benchmarks for which SLP or LLV can benefit from more precise alias analysis (as discussed in Figure 2). There are a total of 20 benchmarks (totaling 1208.2 KLOC) qualified, including the 18 benchmarks shown in Figure 2 and two more benchmarks, `197.parser` and `436.cactusADM`, for which some dynamic alias checks are eliminated by LPA but not executed under the reference inputs. For SLP, LPA enables a total of 133 basic blocks to be vectorized, with an average of 12.09 per benchmark, resulting in the best speedup of 2.95% in `173.applu`. For LLV, LPA eliminates a total of 319 static bound checks, with an average of 22.79 per benchmark, resulting in the best speedup of 7.18% in `177.mesa`.

It should be remarked that most auto-vectorization techniques reported in the literature have been evaluated with small kernels. Compared to a few, e.g., [11, 19], targeting whole-program performance evaluation, LPA has delivered comparable performance improvements across a large number of SPEC 2000/2006 benchmarks. More importantly, LPA (with an open-source implementation) represents a general-purpose and complementary approach that can be directly used to improve the effectiveness of existing auto-vectorization techniques, including SLP and LLV, as demonstrated in this paper.

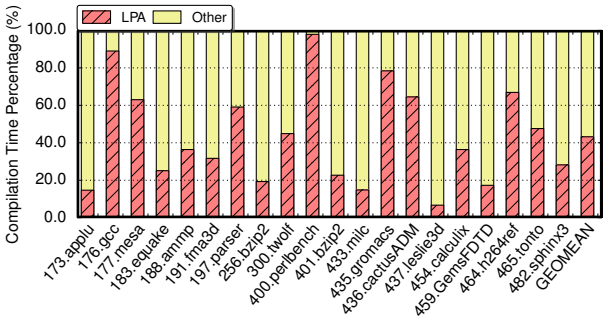


Figure 11: Percentage of analysis time over total compilation time.

Below we describe first our implementation of LPA (Section 4.1), then our experimental setup (Section 4.2), and finally, our experimental results and case studies (Section 4.3).

4.1 Implementation

We have implemented LPA on top of the open-source software, SVF [24] based on LLVM. Every allocation site is modeled as a distinct abstract object. The size of each object is recorded. For a global or stack object, its size is statically known according to the type information at its allocation site, while the size of a heap object created by an allocator function, e.g., $malloc(sz)$ is obtained according to its parameter sz. The size of a heap object is set to ∞ if sz can only be determined at runtime (e.g., under a program input). The location sets are generated according to the rules in Section 3.2. The default field limit (Section 3.3) is set to 1024. LLVM's ScalarEvolution pass is executed before LPA. Then the SCEVAddRecExpr class is used to extract loop information including trip count, step, and bounds for array accesses.

For pointer analysis, we have used the wave propagation technique [18, 32] for constraint resolution. The positive weight cycles (*PWC*s) [17] are detected by using Nuutila's SCC detection algorithm [14]. A program's call graph is built on the fly and points-to sets are represented using sparse bit vectors.

4.2 Experiment Setup

Our experiments are conducted on an Intel Core i7-4770 CPU (3.40GHz) with an AVX2 SIMD extension, which supports 256-bit floating point and integer SIMD operations. The machine runs a 64-bit Ubuntu (14.0.4) with 32 GB memory.

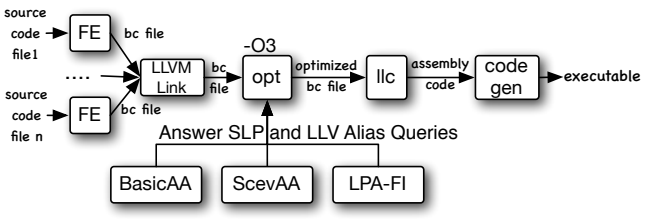


Figure 12: The compilation workflow.

Figure 12 describes the compilation workflow used in our experiments. The source code is compiled into bit-code files using clang (for C code) and gfortran and dragonegg (for Fortran code),

and then linked together using llvm-link. Next, the generated bitcode file is fed into LLVM's opt module to perform vectorization. The effects of an alias analysis on LLV and SLP are evaluated separately. When testing SLP, the compiler flags used are "*-O3 -march=core-avx2 -disable-loop-vectorization*" (with LLV disabled). When testing LLV, the compiler flags used are "*-O3 -march=core-avx2 -disable-slp-vectorization*" (with SLP disabled). llc is used as the back-end to emit assembly code. Finally, executables are generated using clang and gfortran code generators.

We have applied LLV and SLP by using three different alias analyses: (1) LLVM's BASICAA, (2) LLVM's SCEVAA, (3) LPA. To compare fairly the effects of these three alias analyses on the performance benefits achieved by SLP and LLV, we have modified LLVM's alias interface to allow these different alias results to be used only in the SLP and LLV passes. All the other optimizations use BASICAA.

We will focus on a set of 20 SPEC CPU2000/CPU2006 benchmarks, for which LPA is more precise than either BASICAA or SCEVAA: (1) LPA enables more basic blocks to be vectorized by SLP or (2) LPA eliminates some static bounds checks that would otherwise be inserted by LLV. The remaining benchmarks are excluded as LPA has the same capability in answering alias queries as the other two. Table 1 lists some statistics for the 20 benchmarks selected. In our experiments, the execution time of each program is the average of five runs under its reference input.

4.3 Results and Analysis

We first describe the compilation overhead incurred by LPA for the 20 benchmarks examined. To demonstrate how LPA helps harness vectorization opportunities and improve program performance, we provide (1) the number of new basic blocks that are vectorized by SLP under LPA (but not under LLVM's alias analyses), (2) the number of dynamic alias checks that are eliminated under LPA (but introduced under LLVM's alias analyses), and more importantly, (3) the performance speedups obtained given (1) and (2).

4.3.1 Compile-Time Statistics

Analysis Times Figure 11 gives the percentage of LPA analysis time over the total compilation time. LPA is fast in analyzing programs under 100KLOC, with under one minute per benchmark. For larger programs (with $\geqslant$100KLOC), such as `176.gcc`, `435.gromacs` and `465.tonto`, LPA takes longer to finish, with the analysis times ranging from 94.4 to 240.8 secs.

Benchmark	BASICAA	SCEVAA	LPA
173.applu	20	4	26
176.gcc	4	3	6
177.mesa	24	23	64
183.equake	2	1	4
188.ammp	1	2	4
191.fma3d	46	23	53
433.milc	21	13	69
435.gromacs	53	35	57
454.calculix	161	92	166
465.tonto	19	21	32
482.sphinx	0	0	1
Total	351	217	482

Table 2: Number of basic blocks vectorized by SLP under the three alias analyses (larger is better).

***Static Results of* SLP** Table 2 lists the number of basic blocks vectorized by SLP with its alias queries answered by the four analyses compared across the 11 benchmarks. Among the 20 benchmarks listed in Table 1, these 11 benchmarks are the only ones for which LPA is more effective than either BASICAA or SCEVAA or both.

Benchmark	BASICAA	SCEVAA	LPA
176.gcc	4	8	2
177.mesa	121	137	88
197.parser	1	1	0
256.bzip2	1	6	0
300.twolf	11	13	10
400.perlbench	23	21	13
401.bzip2	6	9	5
436.cactusADM	71	112	2
437.leslie3d	21	21	4
454.calculix	83	90	57
459.GemsFDTD	65	79	16
464.h264ref	30	32	2
465.tonto	110	118	38
482.sphinx3	4	5	1
Total	551	652	238

Table 3: Number of static alias checks inserted by LLV under the three alias analyses (smaller is better).

There are totally 351 and 217 basic blocks vectorized by SLP under BASICAA and SCEVAA, respectively. LPA has improved these results to 482, outperforming BASICAA and SCEVAA by 1.38× and 2.23×, respectively. The most significant improvements happen at `177.mesa` and `433.milc`. In each case, LPA enables SLP to discover 40+ vectorizable basic blocks, yielding an improvement of around 3× over BASICAA and SCEVAA. LPA provides more precise aliases since it is more precise in analyzing arrays and nested data structures and in disambiguating must-not-aliases for the arguments of a function.

***Static Results of* LLV** Table 3 gives the number of static alias checks inserted by LLV under the four analyses compared across the 14 benchmarks. Again, among the 20 benchmarks listed in Table 1, these 14 benchmarks are the only ones for which LPA can avoid some checks introduced either by BASICAA or SCEVAA or both.

LLV introduces totally 551 and 652 checks under BASICAA and SCEVAA, respectively, but only 238 under LPA-FI, representing a reduction by 2.32× and 2.74×, respectively. Although the number of static checks is not large, the number of dynamic checks can be huge. For example, a static check inserted for a loop in `Utilities_DV.c` of `454.calculix` is executed up to 28 million times at run time under its reference input (Table 5). Even if a check is inserted at the preheader of a loop, it may still be executed frequently if the loop is nested inside another loop or in recursion.

When LPA is applied, all runtime checks have been eliminated in `197.parser` and `256.bzip2`. For benchmarks such as `176.gcc`, `436.cactusADM`, `437.leslie3d`, `459.GemsFDTD`, `464.h264ref`, `465.tonto`, `482.sphinx3`, over 50% of static checks introduced by LLVM's alias analyses have been eliminated.

4.3.2 Runtime Performance

Let us examine the performance gains obtained under LPA given the above compile-time improvements achieved by SLP and LLV. For a program, the baseline is the smaller of the two execution times achieved by a vectorization technique (either SLP or LLV) considered under BASICAA and SCEVAA.

***Performance Improvements of* SLP** Figure 13 gives the whole-program speedups achieved by SLP under LPA normalized with respect to LLVM's alias analyses. Table 4 lists the code locations and execution frequencies for the new basic blocks that are vector-

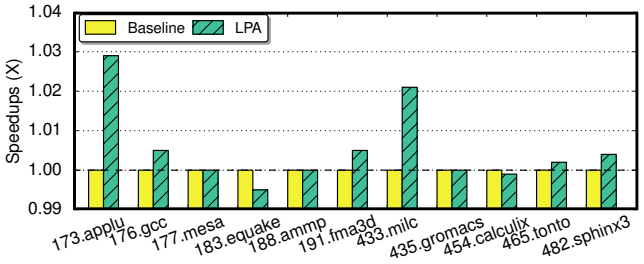


Figure 13: SLP: whole-program speedups (with the baseline being the better of BASICAA and SCEVAA).

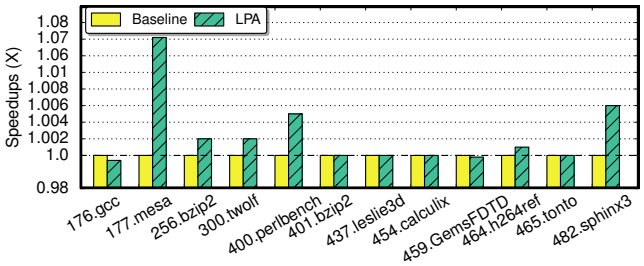


Figure 14: LLV: whole-program speedups (with the baseline being the better of BASICAA and SCEVAA).

ized by SLP under LPA and executed under the reference inputs. Note that for `177.mesa`, its new basic blocks vectorized under LPA are never executed using its reference input.

For all the benchmarks evaluated, `173.applu` and `433.milc` achieve the best speedups of 1.0295× and 1.0218×, respectively, because many new basic blocks vectorized under LPA are frequently executed according to Table 4. For `176.gcc`, `191.fma3d` and `482.sphinx3`, many new basic blocks are also vectorized under LPA, but their performance improvements are small, ranging from 0.1% to 0.5%, because some of these blocks are executed either infrequently or zero times (under their reference inputs).

Interestingly, `183.equake` experienced a performance slowdown under LPA, because LLVM's code motion optimization is not as effective in handling the vectorized version of a loop as it does in handling the scalar version of the same loop. This benchmark has a loop (executed 151173 times) that contains one basic block (executed 6 times) inside a conditional branch. In its scalar version, all `getelementptr` instructions in the loop are moved into its preheader. In the vectorized version, however, some of these `getelementptr` instructions stay inside the loop. Vectorizing this loop is thus not beneficial.

***Performance Improvements of* LLV** Figure 14 is an analogue of Figure 13 to demonstrate the performance speedups achieved by LLV under LPA with the same baseline. Correspondingly, Table 5 is an analogue of Table 4 except that we are here concerned with the loops whose runtime alias checks are completely removed by LPA but introduced by BASICAA or SCEVAA.

For these benchmarks, the performance improvements achieved vary, depending on how costly their removed runtime checks are. We have omitted `197.parser` and `436.cactusADM` as their eliminated runtime checks are not executed under the reference inputs.

For `177.mesa`, we observe a speedup of 1.0718×, as its removed runtime checks involve complex range checks for 10 different pointer pairs, with each pair executed 96512 times. For many other benchmarks, such as `256.bzip2`, `300.twolf`, `400.perlbench`, `464.h264ref` and `482.sphinx3`, the performance improvements are under 1.01×, as their removed runtime checks are not costly relative to their total execution times.

Benchmark	Basic Blocks Vectorized and Executed		Execution Frequency
	Source Files	Line Numbers	
173.applu	applu.f	2688-2715,2838-2865 2988-3015	68484312
		2738-2747,2888-2897 3038-3047	63761256
176.gcc	regclass.c	904-906	1781066
		1597-1600	537972
177.mesa	not executed		
183.equake	quake.c	913-914	6
188.ammp	rectmm.c	323-351	843216
		1028-1043	559731147
191.fma3d	platq.f90	1986-1990,1998-2002	163182300
433.milc	addvec.c	11-13	33600000
	s_m_mat.c	29-48	6400000
	s_m_vec.c	15-18	800000
	s_m_a_vec.c	18-21	446480000
	make_ahmat.c	40-45	657920000
	s_m_a_mat.c	17-20	302080000
	su3mat_copy.c	13-16	270080000
	rephase.c	44-47	14720000
435.gromacs	coupling.c	77-79	7001
	vec.h	487-495	21006
454.calculix	results.f	803-808	3417876
	incplas.f	669-672	278437
465.tonto	rys.fppized.f90	1179-1195	6769676
		1198-1218	4768547
		1221-1241	3759643
482.sphinx3	utt.c	384-387	2808

Table 4: The code locations and execution frequencies for the basic blocks that are vectorized by SLP under LPA but not LLVM's alias analyses and executed under the reference inputs.

For `176.gcc`, a performance slowdown is observed despite removal of some of its runtime checks. We examined its vectorized code and found that the slowdown is caused by function inlining. There is a loop in function `gen_rtvec_v` of `emit-rtl.c`. By removing the runtime checks for the loop, its containing function becomes smaller. As a result, LLVM has decided to inline this function in its callers, causing the performance slowdown. If we add "__attribute__ ((noinline))" for this function, the slowdown will disappear.

4.3.3 Case Studies

To further understand the performance improvements observed in Figures 13 and 14, we have selected four representative kernels from the 18 benchmarks evaluated to show how LPA facilitates SIMD vectorization in some real code scenarios, where both BASICAA and SCEVAA are ineffective. We consider two kernels for improving SLP and two kernels for improving LLV, as listed in Figures 15(a) and (b), and their code snippets in Figures 15(c) – (f). Figures 15(g) and (h) give the speedups achieved by SLP and LLV, respectively, under LPA over BASICAA and SCEVAA.

SLP *Kernels* In `SLP_K1`, the loop given at lines 2 – 3 in Figure 15(c) is fully unrolled by LLVM's code optimizer, due to its small loop trip count (N_REG_CLASSES=7), before it is passed to LLVM's vectorizer. In order to vectorize the eight isomorphic statements after loop unrolling, SLP needs to check if any dependence exists when accessing the array field `cost[j]` via the two pointers `p` and `q` that are the parameters of the containing function. BASICAA and SCEVAA are ineffective as disambiguating `p` and `q` requires an inter-procedural analysis. Guided by LPA, SLP has successfully vectorized this kernel, resulting in a speedup of 2.03×.

Kernel	Function	Benchmark
SLP_K1	`regclass`	176.gcc
SLP_K2	`rephase`	433.milc

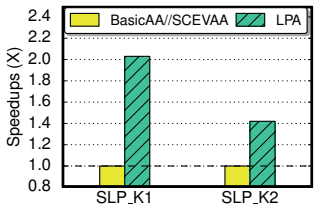
(a) Kernels with new basic blocks vectorizable by SLP under LPA

```
1   p->mem_cost += q->mem_cost * loop_cost;
2   for (j = 0; j < N_REG_CLASSES; j++)
3     p->cost[j] += q->cost[j] * loop_cost;
```

(c) Source code of SLP_K1

```
1  for(dir=XUP;dir<=TUP;dir++){
2    for(j=0;j<3;j++)for(k=0;k<3;k++){
3      s->link[dir].e[j][k].real *= s->phase[dir];
4      s->link[dir].e[j][k].imag *= s->phase[dir];
5    }
6  }
```

(d) Source code of SLP_K2



(g) Kernel speedups for SLP

Kernel	Function	Benchmark
LLV_K1	`hbCreateDecodeTables`	256.bzip2
LLV_K2	`UPDATE`	437.leslie3d

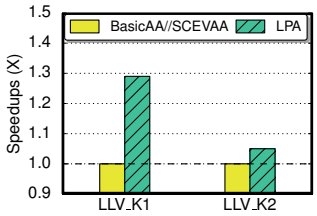
(b) Kernels with runtime alias checks avoided by LLV under LPA

```
1   for (i = minLen + 1; i <= maxLen; i++)
2     base[i] = ((limit[i-1] + 1) << 1) - base[i];
```

(e) Source code of LLV_K1

```
1   Q(1:I2,J,K,1,M) = (RNM1 * Q(1:I2,J,K,1,M)
2     + Q(1:I2,J,K,1,N) + DU(1:I2,J,K,1)) * RNI
```

(f) Source code of LLV_K2



(h) Kernel speedups for LLV

Figure 15: Case studies for four selected kernels with their improved performance under LPA (over BASICAA and SCEVAA).

`SLP_K2` represents an example demonstrating the power of LPA on analyzing deeply nested arrays of structs. The inner loop at lines 2 – 5 in Figure 15(d) is fully unrolled by LLVM's code optimizer. Thus, a basic block with 18 isomorphic statements is formed. All the data in the basic block are accessed via one pointer, `s`. However, enabling SLP requires the struct fields `link` and `phase` to be modeled separately in the nested aggregate, which is supported by LPA but not by BASICAA or SCEVAA. With LPA, SLP has vectorized this kernel, resulting in a speedup of 1.42×.

LLV *Kernels* `LLV_K1` is a loop containing array accesses via pointers `base` and `limit`, which are parameters of its containing function. To vectorize this loop and avoid runtime checks, LLV needs to recognize that the two memory accesses `base[i]` and `limit[i-1]` are actually disjoint, which cannot be done by the intra-procedural alias analyses, BASICAA and SCEVAA. By disambiguating the two array accesses, LPA enables their redundant runtime checks to be avoided. Thus, a speedup of 1.29x is achieved.

`LLV_K2` is a Fortran loop containing accesses to a multi-dimensional array `Q`. To vectorize the loop without introducing runtime checks, the dependence for the two array accesses `Q(1:I2, J, K, 1, M)` and `Q(1:I2, J, K, 1, N)` needs to be analyzed. LPA generates two location sets that access disjoint locations of `Q` under different indices for its highest dimension (i.e., `M` and `N`) based on our value-range analysis. Therefore, runtime checks are eliminated and the performance is improved (by 1.05×).

5. Related Work

Pointer Analysis As a fundamental enabling technique, pointer or alias analysis [1, 4, 6, 23, 28] paves the way for software bug detection [9, 10, 25] and compiler optimizations [13, 26, 27]. In automatic SIMD vectorization, statements grouped together for vectorization must be dependence-free. A recent evaluation on vectorizing compilers [12] reveals some limitations of existing dependence analyses, calling for more precise alias analyses in order to uncover more vectorization opportunities.

The alias analyses used in modern compilers (e.g., LLVM) are intra-procedural, yielding conservative answers to many alias queries. In the literature on inter-procedural alias analysis for C programs, many field-sensitive pointer analysis algorithms [4, 17] rely on a field-index-based model to distinguish fields in a struct by treating all fields to have the same size, which are not sound to support SIMD optimizations. Wilson and Lam [31] introduced a byte-precise model based on location sets, without handling loops and arrays precisely enough to support SIMD optimizations.

This paper introduces an inter-procedural pointer analysis that precisely analyzes aggregate data structures including deeply nested arrays, arrays of structs and structs of arrays to enable advanced SIMD optimizations in vectorizing compilers.

Auto-Vectorization Loops are the main target of the two important vectorization techniques, superword-level parallelism vectorization (SLP) [3, 7, 19, 22, 33] and loop-level vectorization (LLV) [16, 21, 29]. The first SLP approach is proposed in [7], which obtains isomorphic statement groups by tracing data flows starting from consecutive memory accesses. Dynamic programming [3] is later adopted to consider different possibilities of combining isomorphic statements and search for an effective vectorization solution. How to generalize SLP on predicated basic blocks in the presence of control flows is discussed in [22]. More recently, some researchers [19] focused on transforming non-isomorphic statement sequences into isomorphic ones in order to broaden the scope of SLP [7]. Loop-level vectorization is developed based on the technology originally designed for vector machines. Many improvement have been made, by handling interleaved data accesses [16], control flow divergence [21], and loop transformations [29]. Recently, Zhou and Xue [34] introduce an approach to exploiting both SLP and loop-level SIMD parallelism simultaneously by reducing the data reorganization overhead incurred.

6. Conclusion

This paper proposes a new loop-oriented pointer analysis for precisely analyzing arrays and structs to uncover vectorization opportunities that would otherwise be missed by existing alias analyses. Our approach employs lazy memory modeling to generate access-based location sets according to how structs and arrays are accessed. Our experimental results show that LPA improves the performance of LLVM's SLP and LLV across a number of SPEC benchmarks.

Benchmark	Loops With Their Runtime Checks Removed		Execution
	Source Files	Line Numbers	Frequency
176.gcc	tree.c	1155-1156	62263
	emit-rtl.c	469-470	53
177.mesa	osmesa.c	746-748	96512
256.bzip2	bzip2.c	1081-1082	2674
300.twolf	qsortg.c	60-64	1091
		141-145	312
400.perlbench	pp_sort.c	116	526879
	regcomp.c	580-581	10
		583-584	191784
		623-624	3
		637-638	315240
401.bzip2	huffman.c	239	22554
437.leslie3d	tml.f	3572,3573,3576,3577, 3578,3581,3582,3583, 3586,3587,3588,3648	8863680
454.calculix	A2_util.c	1320-1324	15990856
	Chv_swap.c	505-515	1261407
	IV_util.c	486-488	222
	InpMtx_init.c	182-192	111
	Utilities_DV.c	59-61	28864631
		118-120	2490749
		1147-1152	6
		1184-1188	34
	Utilities_IV.c	121-123	4922420
	Utilities_newsort.c	1130-1134,1136-1140	36075
		1424-1428,1430-1434	1354311
459.GemsFDTD	huygens.fppized.f90	706-707,714-715,725-726, 735-736,820-821,826-827, 839-840,847-848	192000
		709-710,717-718,730-731, 738-739,818-819,824-825, 833-834,844-845	191000
		465-468,478-481,493-496, 506-509,521-524,534-537, 555-558,568-571,583-586, 596-599,611-614,624-627	192
		470-473,483-486,498-501, 511-514,526-529,539-542, 550-553,563-566,578-581, 591-594,606-609,619-622	191
	NFT.fppized.f90	805,818,831,844,859,872, 890,903,916,929,942,955	724
		811,824,837,850,865,878, 896,909,922,935,948,961	728
464.h264ref	macroblock.c	2059-2060	250704
	mv-search.c	236	16708450
	rdopt.c	1858-1860	8110368
465.tonto	realmat.fppized.f90	3087,3095	348490
482.sphinx3	new_fe_sp.c	207-209	64584

Table 5: The code locations and execution frequencies for the loops without (with) runtime alias checks under LPA (LLVM's alias analyses), executed under the reference inputs.

Acknowledgments

We thank all the reviewers for their constructive comments on an earlier version of this paper. This research is supported by ARC grant, DP150102109.

References

[1] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, 1994.

[2] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *ISSAC '94*, pages 242–249, 1994.

[3] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *MICRO '10*, pages 201–212, 2010.

[4] B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*, pages 289–298, 2011.

[5] ISO90. ISO/IEC. international standard ISO/IEC 9899, programming languages - C. 1990.

[6] M. Jung and S. A. Huss. Fast points-to analysis for languages with structured types. In *Software and Compilers for Embedded Systems*, pages 107–121. Springer, 2004.

[7] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00*, pages 145–156, 2000.

[8] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16, 2011.

[9] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for java. In *ECOOP '14*, pages 27–53. Springer, 2014.

[10] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program tailoring: Slicing by sequential criteria. In *ECOOP '16*, 2016.

[11] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. In *PLDI '12*, pages 347–358, 2012.

[12] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *PACT '11*, pages 372–382, 2011.

[13] P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC '05*, pages 9–18, 2015.

[14] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[15] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short SIMD architectures. In *PACT '08*, pages 2–11. ACM, 2008.

[16] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI '06*, pages 132–143, 2006.

[17] D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems*, 30(1):4, 2007.

[18] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135, 2009.

[19] V. Porpodas, A. Magni, and T. M. Jones. PSLP: Padded SLP automatic vectorization. In *CGO '15*, pages 190–201, 2015.

[20] R. R. Rick Hank, Loreena Lee. Implementing next generation points-to in open64. In *Open64 Developers Forum*, 2010. URL `http://www.affinic.com/documents/open64workshop/2010/`.

[21] J. Shin. Introducing control flow into vectorized code. In *PACT '07*, pages 280–291, 2007.

[22] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO '05*, pages 165–175, 2005.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96*, pages 32–41. ACM, 1996.

[24] Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, 2016. `https://github.com/unsw-corg/SVF`.

[25] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264, 2012.

[26] Y. Sui, Y. Li, and X. Jingling. Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*, CGO '13, pages 1–11, 2013.

[27] Y. Sui, S. Ye, J. Xue, and J. Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Software: Practice and Experience*, 44(12): 1485–1510, 2014.

[28] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *CGO '16*, pages 160–170, 2016.

[29] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT '09*, pages 327–337, 2009.

[30] R. van Engelen. Efficient symbolic analysis for optimizing compilers. In *CC '01*, pages 118–132, 2001.

[31] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI '95*, pages 1–12, 1995.

[32] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336. Springer, 2014.

[33] H. Zhou and J. Xue. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization*, 13(1):11:1–11:26, 2016.

[34] H. Zhou and J. Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *CGO '16*, pages 59–69, 2016.