Loop-Oriented Pointer Analysis for Automatic SIMD Vectorization

YULEI SUI, University of Technology Sydney (UTS), Australia
XIAOKANG FAN, HAO ZHOU, and JINGLING XUE, University of New South Wales (UNSW), Australia

Compiler-based vectorization represents a promising solution to automatically generate code that makes efficient use of modern CPUs with SIMD extensions. Two main auto-vectorization techniques, superword-level parallelism vectorization (SLP) and loop-level vectorization (LLV), require precise dependence analysis on arrays and structs to vectorize isomorphic scalar instructions (in the case of SLP) and reduce dynamic dependence checks at runtime (in the case of LLV).

The alias analyses used in modern vectorizing compilers are either intra-procedural (without tracking inter-procedural data-flows) or inter-procedural (by using field-sensitive models, which are too imprecise in handling arrays and structs). This article proposes an inter-procedural **L**oop-oriented **P**ointer **A**nalysis for C, called LPA, for analyzing arrays and structs to support aggressive SLP and LLV optimizations effectively. Unlike field-insensitive solutions that pre-allocate objects for each memory allocation site, our approach uses a lazy memory model to generate *access-based 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 so that existing points-to resolution algorithms (e.g., flow-insensitive and flow-sensitive pointer analysis) can be reused easily.

We have implemented LPA fully in the LLVM compiler infrastructure (version 3.8.0). We evaluate LPA by considering SLP and LLV, the two classic vectorization techniques, on a set of 20 C and Fortran CPU2000/2006 benchmarks. For SLP, LPA outperforms LLVM's BASICAA and SCEVAA by discovering 139 and 273 more vectorizable basic blocks, respectively, resulting in the best speedup of 2.95% for `173.applu`. For LLV, LLVM introduces totally 551 and 652 static bound checks under BASICAA and SCEVAA, respectively. In contrast, LPA has reduced these static checks to 220, with an average of 15.7 checks per benchmark, resulting in the best speedup of 7.23% for `177.mesa`.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Pointer analysis, SIMD, loop-oriented, compiler optimisation

ACM Reference format:
Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue. 2018. Loop-Oriented Pointer Analysis for Automatic SIMD Vectorization. *ACM Trans. Embed. Comput. Syst.* 17, 2, Article 56 (January 2018), 31 pages.
https://doi.org/10.1145/3168364

This work is supported by ARC grants, DE170101081, DP150102109, DP170103956.
Authors' addresses: Y. Sui is with Centre for Artificial Intelligence (CAI) at Faculty of Engineering and Information Technology, University of Technology Sydney (UTS), Australia; email: yulei.sui@uts.edu.au; J. Xue, X. Fan, and H. Zhou are with School of Computer Science and Engineering, University of New South Wales, Australia; emails: {jingling,fanx,haozhou}@cse.unsw.edu.au.

https://doi.org/10.1145/3168364

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) 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 (Barik et al. 2010; Larsen and Amarasinghe 2000; Shin et al. 2005; Porpodas et al. 2015; Zhou and Xue 2016a), which packs isomorphic scalar instructions in the same basic block into vector instructions, and (2) loop-level vectorization (LLV) (Nuzman and Zaks 2008; Nuzman et al. 2006; Trifunovic et al. 2009; Shin 2007), which combines multiple consecutive iterations of a loop into a single iteration of vector instructions.

To generate efficient vector code, these two optimizations reliy on precise dependence analysis. For example, to successfully vectorize isomorphic instructions (on four-element vectors) in Figure 1(a), SLP checks conflicting memory accesses by 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] = $\cdots$), any subsequent store or load (e.g., $\cdots$ = B[1]) should not access the same memory address as &A[0]. Figure 1(c) shows another example that can be vectorized by LLV (where N is assumed to be divisible by 4 for illustration purposes). 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 according to SLP in all 12 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 analysis. 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. We note that `433.milc` has the largest number of basic blocks (57) that cannot be vectorized based on the alias results obtained from LLVM BasicAA, representing 73.07% of the total number of basic blocks (78) with alias queries. Across the 11 benchmarks, 153 out of total 504 basic blocks (30.04%) that issue alias queries in SLP are non-vectorizable using LLVM's BasicAA, but potentially vectorizable if a more precise alias analysis is used.

Figure 2(b) gives the number of runtime alias checks for determining whether two memory regions (e.g., arrays) pointed by two pointers are disjoint or overlapping for all the 11 SPEC CPU2000/CPU2006 benchmarks that contain dynamic alias checks inserted by LLV. LLV relies on these checks to disambiguate 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

```
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

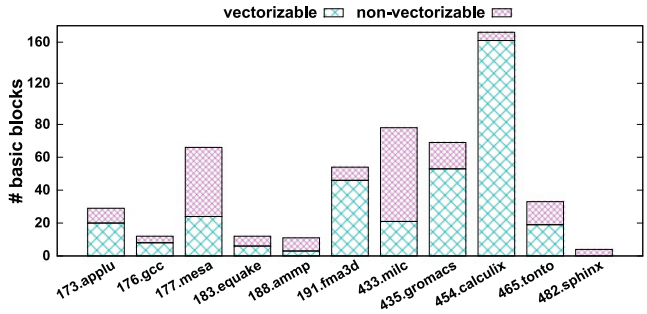
```
1   foo (float *A, float *B) {
2     for (int i = 0; i < N; i++) A[i] = B[i] + K;
3   }
```

(c) Code before LLV

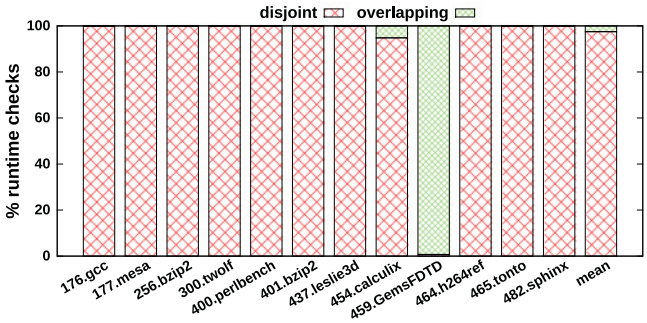
```
1   foo (float *A, float *B) {
2     if ((&A[N-1] ≥ &B[0]) & (&B[N-1] ≥ &A[0]))
3       for (int i = 0; i < N; i++) A[i] = B[i] + K;
4     else
5       for (int i = 0; i < N; i+=4) A[i:i+3] = B[i:i+3] + K;
6   }
```

(d) Code after LLV

Fig. 1. Examples for SLP and LLV vectorizations (assuming a four-element vector).

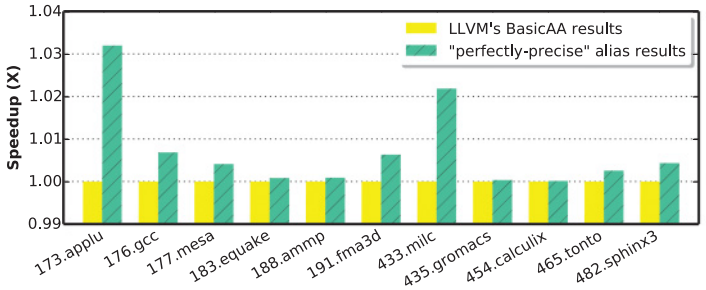


(a) SLP: number of vectorizable and non-vectorizable basic blocks

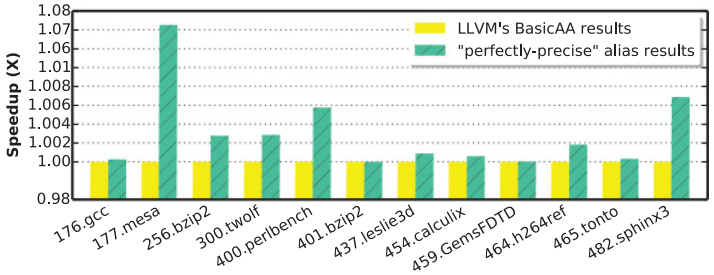


(b) LLV: percentage of runtime checks for determining disjoint and overlapping memory (i.e., array) regions (under the reference inputs)

Fig. 2. Impact of LLVM's BasicAA analysis on the effectiveness of SLP and LLV.



(a) SLP: whole-program speedups.



(b) LLV: whole-program speedups.

Fig. 3. Potential performance benefits for SLP and LLV achieved with a "perfectly-precise" (i.e., oracle) alias analysis relative to LLVM's BasicAA.

is disjoint. 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, most runtime checks are redundant and can therefore be eliminated if a more precise alias analysis is applied, resulting in a reduction in both instrumentation and code-size overheads.

Figure 3 shows the potential performance benefits for the two vectorization techniques with a "perfectly-precise" (i.e., oracle) static alias analysis with respect to LLVM's BasicAA for whole-program vectorization. However, such static analysis does not exist due to the undecidability of the aliasing problem in the presence of potentially an unbounded number of program paths (Ramalingam 1994). To obtain the ideal performance for SLP and LLV in Figures 3(a) and 3(b), we have manually added the `__restrict__` attribute to disambiguate the aliases for the potentially vectorizable basic blocks in Figure 2(a). We have also removed the redundant static instrumentation code identified in Figure 2(b) based on the reference inputs in the SPEC benchmarks.

We can see that many benchmarks show room for improvement, with the speedups ranging from 1.01× to 1.07×, which are significant when compared with existing techniques for whole-program vectorization (Porpodas et al. 2015; Nuzman et al. 2006; Karrenberg 2015). More importantly, developing a more precise alias analysis for discovering more vectorization opportunities is complementary, resulting in extra performance benefits on top of the previous work. Note that there may not be a direct translation from static improvement (measured in terms of vectorizable basic blocks and redundant instrumentation code removed) to the actual runtime performance speedups, since some vectorized code may not be executed frequently under some particular inputs.

The aim of this article is to develop a more precise pointer analysis than the existing LLVM's alias analyses to achieve more effective SIMD vectorization.

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. The existing field-sensitive pointer analyses for C (Pearce et al. 2007; Hardekopf and Lin 2011) 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 the objects that it points to may be incompatible due to type casting. A field-index-based analysis may generate unsound results for partial aliases (as illustrated in Section 2.3), which may cause the compiler to generate incorrect vectorized code. Therefore, such approach is not appropriate for supporting SIMD optimizations.

To the best of our knowledge, location sets (Wilson and Lam 1995) represent still the most sophisticated byte-precise field-sensitive memory modeling in pointer analysis for C programs. A location set $\langle off, s\rangle_a \in \mathbb{Z} \times \mathbb{N}$ represents a set of memory locations $\{off + i \times s \mid i \in \mathbb{Z}\}$ accessed from the beginning of a memory object a, where off is an offset within a 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 in developing precise alias analyses for auto-vectorization. First, arrays are modeled monolithically, with all the elements in the same array collapsed. Second, their sizes 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 optimizations, 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 (Wilson and Lam 1995) so that both arrays and structs are handled in a uniform manner. Unlike the location-set model (Wilson and Lam 1995), 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 accesses to arrays including nested aggregate structures. The 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 versions) with desired efficiency and precision tradeoffs by reusing

Table 1. Statements and Memory Expressions

Statement s	$::= p = alloc_a(sz) \mid p = malloc_a(sz) \mid p = q \mid p = \&e_p \mid q = e_p \mid e_p = q$
MemExpr e_p	$::= *p \mid p \rightarrow f \mid p[i]$

existing pointer resolution frameworks. A pointer analysis is *flow-sensitive* if the flow of control in the program is distinguished and *flow-insensitive* otherwise.

In summary, this article makes the following contributions:

- We introduce Lpa, a new loop-oriented array- and field-sensitive inter-procedural pointer analysis for C programs based on access-based location sets built in terms of a lazy memory model.
- We apply flow-insensitive and flow-sensitive versions of Lpa to improve the effectiveness of SLP and LLV, by enabling more basic blocks to be vectorized by SLP and some redundant dynamic checks that are inserted by LLV to be eliminated.
- We have implemented Lpa in LLVM (3.8.0). We evaluate Lpa with a total of 20 C and Fortran CPU2000/2006 benchmarks, for which SLP or LLV can benefit from a more precise alias analysis. These include all 18 benchmarks shown in Figure 2 as explained before, and two more benchmarks, `197.parser` and `436.cactusADM`, for which some alias checks are eliminated by Lpa but not executed under the reference inputs. For SLP, Lpa outperforms LLVM's BasicAA and ScevAA by discovering 139 and 273 more vectorizable basic blocks, respectively, resulting in the best speedup of 2.95% for `173.applu`. For LLV, LLVM introduces totally 551 and 652 static bound checks under BasicAA and ScevAA, respectively. In contrast, Lpa has reduced these static bound checks to 220, with an average of 15.7 checks per benchmark, resulting in the best speedup of 7.23% for `177.mesa`. We also provide a detailed discussion about the scenarios where our approach is applicable and its limitations.

The rest of this article is organized as follows. Section 2 provides the background information. Section 3 presents our loop-oriented array and field-sensitive but flow-insensitive pointer analysis (Lpa). Section 4 introduces our flow-sensitive version of Lpa. Section 5 discusses and analyzes our experimental results. Section 6 describes the related work. Finally, Section 9 concludes the article.

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 field-insensitive memory model.

2.1 Program Representation

A program is represented in LLVM's partial SSA form Hardekopf and Lin (2011), Ye et al. (2014b), and Lhoták and Chung (2011). 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 $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$. Top-level variables are put directly in SSA form, while address-taken variables are only accessed indirectly via memory expressions.

Table 1 gives the statements and expressions relevant to our analysis, where $p, q \in \mathcal{T}$, $a \in \mathcal{A}$, and e_p denote a memory access expression involving a pointer p, including a pointer dereference, a field access, or an array access. The 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 (ISO90 1990). There are six types of statements:

$p = \&a;$ $a = \&b;$	$p = \&a;$ $t = \&b;$ $*p = t;$
$a.f = r;$	$p \rightarrow f = r;$
$a[i] = q;$	$p[i] = q;$
$x = \&a.f;$	$x = \&p \rightarrow f;$
(a) C code	(b) Partial SSA

Fig. 4. A C code fragment and its partial SSA form.

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

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

Fig. 5. Field-insensitive inclusion-based pointer analysis.

—StcAlloc: $p = alloc_a(sz)$ represents static allocation, where a is either a stack or global object. The size sz of object a is determined statically (bytewise).
—DynAlloc: $p = malloc_a(sz)$ represents heap allocation, where the size sz of object a is either known statically or determined at runtime, e.g., user inputs.
—Copy: $p = q$ is either (1) a LLVM `bitcast` instruction, where p and q are different pointer types or (2) decomposed from a LLVM `phi` instruction, i.e., p = `phi`(q, r) is translated into $p = q$ and $p = r$.
—AddrOf: $p = \&e_p$, represents the fact that the address of a memory expression (e.g., struct field $\&p \rightarrow f$) can be taken in weakly-typed languages (e.g., C), whereas it is not permitted in strongly-typed languages (e.g., Java).
—Load: $q = e_p$ reads a value from memory.
—Store: $e_p = q$ writes a value to memory.

Figure 4 gives a code fragment and its partial SSA form, where $p, q, r, t \in \mathcal{T}$ and $a, b \in \mathcal{A}$. Here, a is accessed indirectly at a store $*p = t$ by introducing a top-level pointer t in partial SSA form. Any field access $a.f$ via an address-taken variable is transformed into a field dereference via a top-level pointer, e.g., $p \rightarrow f$. Similarly, an array access, e.g., $a[i]$ is transformed to $p[i]$. The address of a struct field, e.g., $x = \&a.f$ is transformed to AddrOf $x = \&p \rightarrow f$.

For our interprocedural analysis, passing arguments into and returning results from functions are modeled by Copy statements. 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.

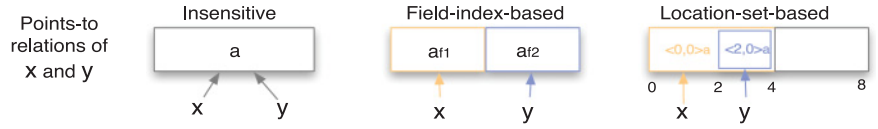
2.2 Field-Insensitive Pointer Analysis

Figure 5 gives the rules used in a field- and flow-insensitive inclusion-based analysis (Andersen 1994) with program statements transformed into constraints for points-to resolution until a fix point is reached.

```
typedef struct { int f1; int f2;} A;
typedef struct { short f1; short f2; int f3} B;
```

1. $A * p = malloc_a(8)$;
2. $B * q = (B)p$;
3. $int * x = \&p \rightarrow f1$;
4. $short * y = \&q \rightarrow f2$;

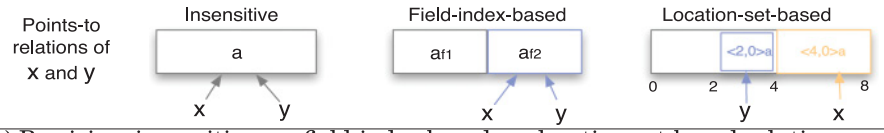
	Insensitive	Field-index	Location-set
points-to	$pt(x) = \{ a \}$ $pt(y) = \{ a \}$	$pt(x) = \{ a_{f1} \}$ $pt(y) = \{ a_{f2} \}$	$pt(x) = \{ \langle 0,0 \rangle_a \}$ $pt(y) = \{ \langle 2,0 \rangle_a \}$
$*x$ and $*y$	aliasing	not aliasing	aliasing
soundness	sound	unsound	sound



(a) Soundness: insensitive v.s. field-index based v.s. location-set-based solution.

1. $A * p = malloc_a(8)$;
2. $B * q = (B)p$;
3. $int * x = \&p \rightarrow f2$;
4. $short * y = \&q \rightarrow f2$;

	Insensitive	Field-index	Location-set
points-to	$pt(x) = \{ a \}$ $pt(y) = \{ a \}$	$pt(x) = \{ a_{f2} \}$ $pt(y) = \{ a_{f2} \}$	$pt(x) = \{ \langle 4,0 \rangle_a \}$ $pt(y) = \{ \langle 2,0 \rangle_a \}$
$*x$ and $*y$	aliasing	aliasing	not aliasing
soundness	sound	sound	sound



(a) Precision: insensitive v.s. field-index based v.s. location-set-based solution.

Fig. 6. Comparing two field-sensitive solutions.

A field-insensitive solution (Hardekopf and Lin 2007; Lhoták and Chung 2011) 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 field-insensitive points-to resolution. The pointers dereferences $*p$ and $*q$ are must-not aliases if the intersection of their points-to sets $pt(p)$ and $pt(q)$ is empty and may aliases otherwise.

2.3 Field-Sensitive Modeling: Index-based vs. Location-set-based Solution

In a field-insensitive analysis, an object at an allocation site is considered monolithically. In contrast, a field-sensitive analysis distinguishes different sub-components of an aggregate object. We compares two previously used solutions to field-sensitivity.

— *Field-index-based solution*, which distinguishes the fields of an aggregate object by their unique indices with all fields treated as having the same size. For each object a, a_f is used to represent the sub-object that corresponds to the field f of a.
— *Location-set-based solution* (Wilson and Lam 1995), which models the fields of an object byte-precisely based on their declared types. In Figure 6(a), p at line 1, which is declared with type $A*$, is made to point to object a, where A is a struct consisting of two 4-byte-integer fields. At line 3, $x = \&p \rightarrow f1$ obtains the address of a's first field represented by location set $\langle off, s \rangle_a$, where $off = 0$, denoting the offset from the beginning of object a, and stride $s = 0$, because the location set has a single element. Based on the type $int*$ of x, we know that $*x$ accesses the memory between the 0st and the 4th byte of a, as illustrated in Figure 6(a).

Location-set-based modeling can give three kinds of alias results for a pair of memory expressions, e.g., $*p$ and $*q$: (1) *may aliases* if $*p$ and $*q$ may refer to the same location set derived from

an object; (2) *must-not aliases* if $*p$ and $*q$ refer to completely different abstract objects; and (3) *partial aliases* (LLVM-Alias-Analysis 2017) if $*p$ and $*q$ may refer to different location sets derived from the same object with overlapping memory locations. In this article, two memory expressions are said to be *aliases* if they are either may aliases or partial aliases.

A location-set-based analysis is more sound and precise than a field-index-based analysis in identifying partial aliases for C. There are two reasons. First, the address of an subobject (e.g., a struct field or an array element) can be taken. Second, an object or its subobjects can be accessed via pointers of different types due to pointer casting.

We use the example in Figure 6(a) (adopted from the example in Pearce et al. (2007)) to demonstrate that field-index-based approach is unsound in terms of answering the partial alias query $*x$ and $*y$. We then use the example in Figure 6(b) with the code at line 3 slightly changed to demonstrate that field-index-based solution is also less precise than the location-set-based approach in the presence of casting in C.

— As shown in Figure 6(a), both field-sensitive solutions can distinguish the two fields compared to field-insensitive analysis, however, the field-index-based approach ignores the sizes of fields, reporting unsoundly that x and y point to different targets a_{f1} and a_{f2}. In fact, $*x$ and $*y$ are partial aliases as they access overlapping memory, i.e., locations between the 2nd to 4th byte of object a as illustrated.
— As illustrated in Figure 6(b), x and y point to the same object a_{f2} according to the field-index-based approach. However, the bytewise location-set-based approach is able to disambiguate $*x$ (between the 4th and 8th byte of a) and $*y$ (between the 2nd and 4th byte of a), based on the two disjointed location sets $\langle 4, 0\rangle_a$ and $\langle 2, 0\rangle_a$.

3 THE LPA ANALYSIS

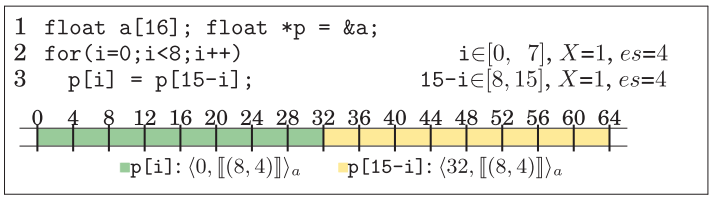
Both field-index and location-set-based approaches are loop-unaware and array-insensitive. Our memory modeling approach is built on top of a location-set-based solution to enable sound handling of partial aliases. By incorporating loop information, LPA produces precise points-to results for an important compiler client, i.e., SIMD vectorization, which requires byte-precise modeling to be effective.

This section first describes 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 value-range analysis and location set disambiguation. Finally, we focus on field unification, together with how to handle positive weight cycles and flow-sensitivity (Section 3.3).

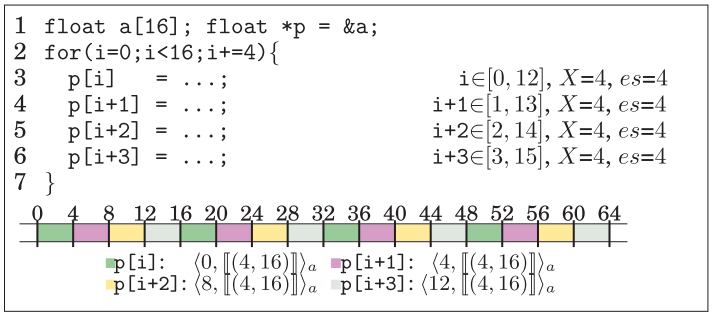
3.1 Amm: Access-based Memory Modeling

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 (Wilson and Lam 1995), 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.

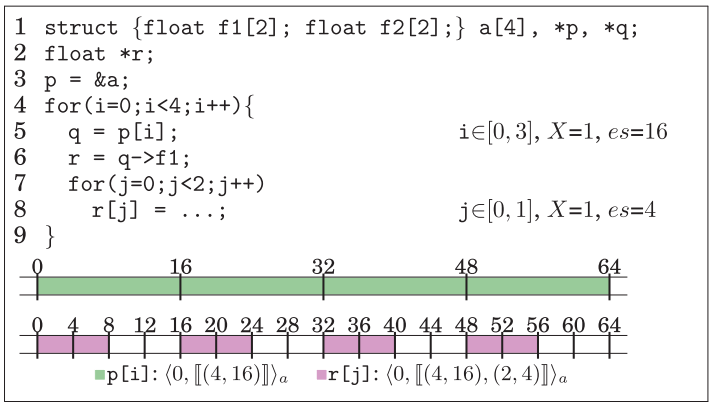
To precisely model the locations based on the array access information, we introduce a new concept called *access trip*, which is a pair (t, s) consisting of a trip count $t = (ub - lb)/X + 1$ and a stride $s = es * X$, where es is the size of an array element.



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



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



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

Fig. 7. Examples for access-based location sets.

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

$$\sigma = \langle off, [\![(t_1, s_1), \ldots, (t_m, s_m)]\!] \rangle_a, \tag{1}$$

where $off \in \mathbb{N}$ is an offset from the beginning of object a, and $T = [\![(t_1, s_1), \ldots, (t_m, s_m)]\!]$ 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 7(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 of σ from the beginning of an object a:

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

Let us go through three examples with consecutive and non-consecutive accesses to single and nested arrays as given in Figure 7.

Example 3.1 (Consecutive Array Access). Figure 7(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 dependencies between $p[i]$ and $p[15-i]$.

Our 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 locations.

Example 3.2 (Non-Consecutive Array Access). Figure 7(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 3.3 (Nested Array Access). Figure 7(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 8 gives the rules for an inclusion-based pointer analysis based on Amm. Unlike the field-insensitive counterpart given in Figure 5, 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*. Rule [S-COPY] is the same as in the flow-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 alway 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 length of the array.

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

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

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

Fig. 8. Rules for field- and array-sensitive inclusion-based pointer analysis equipped with an access-based memory model.

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.

3.2.1 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 van Engelen (2001). This pass, inspired by the concept of *chains of recurrences* (Bachmann et al. 1994), 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 to perform SIMD optimizations, we extract only 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 7(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 $X = 4$.

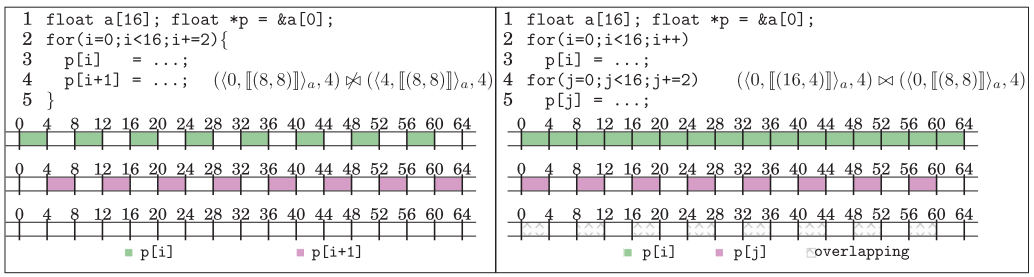
The SCEV pass computes the trip count of its containing loop, which is also represented as a SCEV. A trip count can be non-computable. For a loop with multiple exits, the worst-case trip count is picked. Similarly, a loop upper bound is also represented by a SCEV, deduced from the trip count and step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

3.2.2 Disambiguation of Location Sets. In a field-insensitive analysis, two pointer dereferences are aliased if they may refer to the same object. In Amm, every object may generate multiple

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

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

Fig. 9. Disambiguation of location sets (where $obj(\sigma)$ denotes the object on which σ is generated).



(a) Disjoint location sets (b) Overlapping location sets

Fig. 10. Examples for disjoint and overlapping location sets.

location sets. Two location sets can refer to disjoint memory locations even if they are generated originally from the same object.

Our analysis checks whether two memory expressions e_p and e_q are aliased 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 9. We say that e_p and e_q are aliased if Equation (3) holds, where $(\sigma_p, sz_p) \rhd\lhd (\sigma_q, sz_q)$ 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), $(\sigma, sz_p) \rhd\lhd (\sigma', sz_q)$ 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 3.4 (Disjoint and Overlapping Location Sets). Figure 10(a) illustrates disjoint memory accesses. Two expressions e_p and e_q are not aliased, since their location sets are disjoint. According to Equation (1), the location sets of $p[i]$ and $p[i+1]$ are $\langle 0, [\![(8,8)]\!]\rangle$ and $\langle 4, [\![(8,8)]\!]\rangle$, respectively. The sizes of both accesses to the elements of an array with the float type are 4. According to Equation (4), $(\sigma_p, sz_p) \rhd\!\lhd (\sigma_q, sz_q)$, $p[i]$ and $p[i+1]$ always access disjoint regions. In contrast, Figure 10(b) shows a pair of overlapping location sets, with their overlapping areas shown in gray.

3.3 Field Unification, PWC and Flow-Sensitivity

3.3.1 Field Unification Optimization. For some programs, a field-sensitive analysis may generate a large number of location sets due to deeply nested aggregate structures, which may affect the efficiency of points-to propagation during the analysis. To make a tradeoff between efficiency and

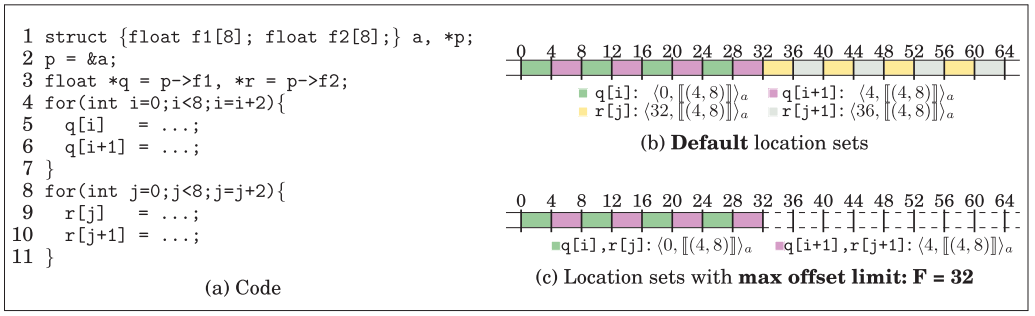


Fig. 11. Field unification for location sets.

precision, we introduce a simple yet effective unification technique call *field-unification*, which aims to reduce analysis overhead by merging existing locations. 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 generated.

With field unification, a location set is represented as $\sigma = \langle off \,\%\, F, T\rangle$. Note that the trip stack of array access information remains unchanged in order to exploit vectorization opportunities.

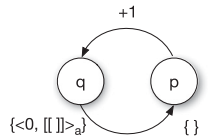
Example 3.5 (Field Unification). Figure 11(a) gives a field unification example with a struct containing two array fields $f1[8]$ and $f2[8]$. Field $f1[8]$ is accessed via $q[i]$ and $q[i+1]$ inside the `for` loop (lines 4–7). Field $f2[8]$ is accessed via $r[j]$ and $r[i+1]$ inside the other loop (lines 8–11). The default location sets generated for the four memory accesses are shown in Figure 11(b). If we limit the maximum starting field offset to be 32, then the location set of $r[i]$ is with into $p[i]$ and $r[i+1]$ is merged with $p[i+1]$, 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 11(c).

3.3.2 Handling Positive Weight Cycles. With field-sensitivity, one difficulty lies in handling positive weight cycles (PWCs) (Pearce et al. 2007) during points-to resolution. Without field-sensitivity (Hardekopf and Lin 2007; Pereira and Berlin 2009), 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 12(a) shows a PWC with an edge from p to q, indicating a field offset that causes 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 PWC efficiently while maintaining precision, we follow (Pearce et al. 2007; Rick Hank 2010) by marking the objects in the points-to set of the pointers inside a PWC to be field-insensitive, causing all its fields to be merged.

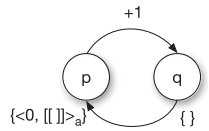
Amm models both field and array accesses of an object. This poses another challenge for PWC handling as a cycle may involve pointer arithmetic when array elements are accessed. Figure 12(b) shows a PWC example simplified from `181.mcf`. The pointer `p` iterates over all the elements in an array of structs, $a[10]$, inside the `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

```
1  struct S {char f1; char f2;} a,*p;
2  void *q = &a;
3  p = q;
4  q = &(p->f2);
```



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

```
1  struct S {char f1; char f2;} a[10], *p, *q;
2  p = &a;
3  for (int i = 0; i < 10; i++) {
4     q = p + 1; p = q;
5  }
```



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

Fig. 12. Handling of positive weight cycles.

sets generated by array accesses are merged, while the location sets generated by field accesses remain unchanged.

4 FLOW-SENSITIVE LPA

In previous flow-sensitive analyses, such as Hardekopf and Lin (2011) and Ye et al. (2014b), field-sensitivity is realized based on a field-index-based model, which has the same limitations as flow-insensitive analysis for analyzing loop-oriented clients like SIMD vectorization, as discussed in Section 2.3. Our access-based memory model is designed to be independent from any pointer analysis algorithm. For example, the Sparse Flow-Sensitive (SFS) points-to analysis algorithm (Hardekopf and Lin 2011) can be easily ported by adopting our byte-precise memory model, which incorporates loop information to obtain precise aliases by disambiguating disjoint location sets.

Figure 13 gives the rules of a field-insensitive SFS for resolving points-to information flow-sensitively. For a variable v, $pt(\ell, v)$ denotes its points-to set computed immediately after statement ℓ. Here, $\ell' \overset{v}{\hookrightarrow} \ell$ represents the pre-computed def-use (value-flow) relation of variable $v \in \mathcal{V}$ from statement ℓ' to statement ℓ (Sui and Xue 2016b; Hardekopf and Lin 2011; Sui and Xue 2016a).

The first four rules deal with the four types of statements introduced in Section 2.1 with every memory access expression e_p treated as a pointer dereference $*p$ in a field-insensitive manner. `[F-COPY]`, `[F-LOAD]`, `[F-STORE]` resolve and propagate the points-to information sparsely by following the pre-computed def-use chains $\hookrightarrow$. The last rule `[F-SU/WU]` enables a strong or weak update at a store, whichever is appropriate, where singletons (Lhoták and Chung 2011) form the set of objects in $\mathcal{A}$ representing unique locations by excluding heap, array, and local variables in recursion.

Figure 14 gives the rules of our field- and array-sensitive sparse flow-sensitive analysis based on Amm. In this case, a points-to set contains location sets instead of objects, with the location sets generated based on memory expressions $p \rightarrow f$ and $p[i]$ via $GetLS$ given in Figure 8.

We use the same notation as in Figure 13 to represent the points-to set $pt(\ell, p)$ of a top-level pointer p. For an address-taken variable a, we compute the points-to set $pt(\ell, \sigma)$ for every location set σ derived from the base object a, denoted as $a = obj(\sigma)$.

Two location sets σ and σ' may have overlapping memory locations when derived from the same base object as discussed in Section 3.2. Our analysis performs strong updates for every location set individually; i.e., strong updates on σ do not affect the values of σ'. As in Figure 13, only the location sets generated from singleton objects (Lhoták and Chung 2011) are allowed to be

$$\texttt{[P-ADDR]} \quad \frac{\ell : p = \&a}{\{a\} \subseteq \mathit{pt}(\ell, p)}$$

$$\texttt{[P-COPY]} \quad \frac{\ell : p = q \quad \ell' \overset{q}{\hookrightarrow} \ell}{\mathit{pt}(\ell', q) \subseteq \mathit{pt}(\ell, p)}$$

$$\texttt{[P-LOAD]} \quad \frac{\ell : p = *q \quad \ell'' \overset{q}{\hookrightarrow} \ell \quad \ell' \overset{a}{\hookrightarrow} \ell \quad a \in \mathit{pt}(\ell'', q)}{\mathit{pt}(\ell', a) \subseteq \mathit{pt}(\ell, p)}$$

$$\texttt{[P-STORE]} \quad \frac{\ell : *p = q \quad \ell'' \overset{p}{\hookrightarrow} \ell \quad \ell' \overset{q}{\hookrightarrow} \ell \; a \in \mathit{pt}(\ell'', p)}{\mathit{pt}(\ell', q) \subseteq \mathit{pt}(\ell, a)}$$

$$\texttt{[P-SU/WU]} \quad \frac{\ell : *p = \_ \quad \ell' \overset{a}{\hookrightarrow} \ell \quad a \in \mathcal{A} \backslash \mathit{kill}(\ell, p)}{\mathit{pt}(\ell', a) \subseteq \mathit{pt}(\ell, a)}$$

$$\mathit{kill}(\ell, p) = \begin{cases} \{a'\} & \text{if } \mathit{pt}(\ell, p) = \{a'\} \wedge (a' \in \mathit{singletons}) \\ \mathcal{A} & \text{else if } \mathit{pt}(\ell, p) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Fig. 13. Field-insensitive sparse flow-sensitive pointer analysis. (ℓ, v) represents a definition of a variable $v \in \mathcal{V}$ at a program statement ℓ, and $\ell' \overset{v}{\hookrightarrow} \ell$ represents the pre-computed def-use of variable v from ℓ' to ℓ.

$$\texttt{[F-ADDR]} \quad \frac{\ell : p = \&a \quad \sigma = \langle 0, [\![]\!] \rangle_a}{\{\sigma\} \subseteq \mathit{pt}(\ell, p)}$$

$$\texttt{[F-COPY]} \quad \frac{\ell : p = q \quad \ell' \overset{q}{\hookrightarrow} \ell}{\mathit{pt}(\ell', q) \subseteq \mathit{pt}(\ell, p)}$$

$$\texttt{[F-LOAD]} \quad \frac{\ell : q = e_p \quad \ell'' \overset{p}{\hookrightarrow} \ell \quad \ell' \overset{\sigma}{\hookrightarrow} \ell \quad \sigma \in \mathit{pt}(p) \quad \sigma' = \mathit{GetLS}(\sigma, e_p)}{\mathit{pt}(\ell', \sigma') \subseteq \mathit{pt}(\ell, q)}$$

$$\texttt{[F-STORE]} \quad \frac{\ell : e_p = q \quad \ell'' \overset{p}{\hookrightarrow} \ell \quad \ell' \overset{q}{\hookrightarrow} \ell \quad \sigma \in \mathit{pt}(p) \quad \sigma' = \mathit{GetLS}(\sigma, e_p)}{\mathit{pt}(\ell', q) \subseteq \mathit{pt}(\ell, \sigma')}$$

$$\texttt{[F-SU/WU]} \quad \frac{\ell : *p = \_ \quad \ell' \overset{\sigma}{\hookrightarrow} \ell \quad \sigma \in \mathcal{A} \backslash \mathit{kill}(\ell, p)}{\mathit{pt}(\ell', \sigma) \subseteq \mathit{pt}(\ell, \sigma)}$$

$$\mathit{kill}(\ell, p) = \begin{cases} \{\sigma'\} & \text{if } \mathit{pt}(\ell, p) = \{\sigma'\} \wedge (\mathit{obj}(\sigma') \in \mathit{singletons}) \\ \mathcal{A} & \text{else if } \mathit{pt}(\ell, p) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Fig. 14. Field- and array-sensitive sparse flow-sensitive pointer analysis based on Amm.

strongly updated. To answer alias queries from a client, like SIMD vectorization, the same alias disambiguation rules in Figure 9 are reused.

5 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, the default alias analyses in LLVM including BasicAA and SCEVAA are used as the baselines.

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 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 outperforms BasicAA and ScevAA by discovering 139 and 273 more vectorizable basic blocks, respectively, resulting in the best speedup of 2.95% for `173.applu`. For LLV, LLVM introduces totally 551 and 652 static bound checks under BasicAA and ScevAA, respectively. In contrast, Lpa has reduced these checks to 220, with an average of 15.7 checks per benchmark, resulting in the best speedup of 7.23% for `177.mesa`.

Below, we describe our implementation of Lpa (Section 5.1), our experimental setup (Section 5.2), our experimental results and case studies (Section 5.3), and, finally, some limitations of Lpa and possible future improvements (Section 5.4).

5.1 Implementation

We have implemented Lpa on top of our open-source software tool, SVF (Sui and Xue 2016b), based on LLVM (version 3.8.0). The LLVM compiler (Lattner and Adve 2004), which is designed as a set of reusable libraries with a well-defined intermediate representation, has been recognized as a common infrastructure to support program analysis and transformation.

In LPA, 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 assumed to be infinite if *sz* can only be determined at runtime (e.g., under a program input), following the location-set-based approach (Wilson and Lam 1995). The location sets are generated according to the rules in Section 3.1. The default field limit (Section 3.3) is set to 1,024. 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 LPA's points-to resolution, we use the wave propagation technique (Pereira and Berlin 2009; Rick Hank 2010) for constraint solving. The positive weight cycles (*PWC*s) (Pearce et al. 2007) are detected using Nuutila's SCC detection algorithm (Nuutila and Soisalon-Soininen 1994). A program's call graph is built on the fly and points-to sets are represented using sparse bit vectors.

We have chosen LLVM for an implementation of LPA for three reasons. First, LLVM, as one of the mainstream compilers, has relatively informative documents and an easy-to-understand interface compared to GCC for implementing and integrating new alias algorithms in its framework. Second, the SIMD vectorizers in LLVM exhibit comparable performance as those in GCC according to https://llvm.org/docs/Vectorizers.html#performance. Finally, LPA is complementary to the existing vectorization techniques, as it exposes more vectorization opportunities due to more precise aliases discovered.

5.2 Experiment Setup

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

Figure 15 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),

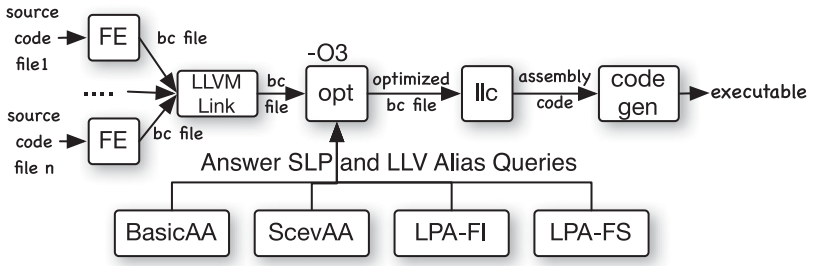


Fig. 15. The compilation work flow.

and then linked together using llvm-link. Next, the generated bit-code 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 four different alias analyses: (1) LLVM's BasicAA, (2) LLVM's SCEVAA, (3) the flow-insensitive version of Lpa (Lpa-FI), and (4) the flow-sensitive version of Lpa (Lpa-FS). To compare fairly the effects of these four 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 2 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.

5.3 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 basic blocks vectorized by SLP under Lpa (but not under LLVM's alias analyses), (2) the number of dynamic alias checks eliminated under Lpa (but introduced under LLVM's alias analyses), and more importantly, (3) the performance speedups obtained given (1) and (2).

5.3.1 Compile-Time Statistics. Some compile-time statistics are analyzed below.

Analysis Times. Table 3 and Figure 16 give the analysis times of Lpa-FI and Lpa-FS, and their percentage contributions to total compilation times. Both are fast in analyzing programs under 100KLOC, by spending under one minute per benchmark. For larger programs (with $\geqslant$100KLOC), such as `176.gcc`, `435.gromacs`, and `465.tonto`, Lpa-FI and Lpa-FS take longer to finish, with the analysis times of Lpa-FI ranging from 94.4 to 1740.5s and of Lpa-FS ranging from 134.8 to 7112.2s. For `400.perlbench`, which has a large number of pointers, statements, and indirect callsites, Lpa-FS finishes its analysis in around 2h.

Static Results of SLP. Table 4 lists the number of basic blocks vectorized by SLP with its alias queries answered by the four analyses compared across the 11 benchmarks. For the 20 benchmarks

Table 2. Program Characteristics

Program	KLOC	#Stmt	#Ptrs	#Objs	#CallSite
173.applu	3.9	3,361	20,951	159	346
176.gcc	226.5	215,312	545,962	16,860	22,595
177.mesa	61.3	99,154	242,317	9,831	3,641
183.equake	1.5	2,082	6,688	236	235
188.ammp	13.4	14,665	56,992	2,216	1,225
191.fma3d	60.1	119,914	276,301	6,497	18,713
197.parser	11.3	13,668	36,864	1,177	1,776
256.bzip2	4.6	1,556	10,650	436	380
300.twolf	20.4	23,354	75,507	1,845	2,059
400.perlbench	168.1	130,640	296,288	3,398	15,399
401.bzip2	8.2	7,493	28,965	669	439
433.milc	15	11,219	30,373	1,871	1,661
435.gromacs	108.5	84,966	224,967	12,302	8,690
436.cactusADM	103.8	62,106	188,284	2,980	8,006
437.leslie3d	3.8	12,228	38,850	513	2,003
454.calculix	166.7	135,182	532,836	18,814	23,520
459.GemsFDTD	11.5	25,681	107,656	3,136	6,566
464.h264ref	51.5	55,548	184,660	3,747	3,553
465.tonto	143.1	418,494	932,795	28,704	58,756
482.sphinx3	25	20,918	60,347	1,917	2,775
Total	1,208.2	1,457,541	3,898,253	117,308	182,338

Table 3. Total Analysis Time Including Lpa-FI and Lpa-FS

Benchmark	Analysis Times (s)	
	Lpa-FI	Lpa-FS
173.applu	0.3	0.4
176.gcc	390.1	1,215.2
177.mesa	28.7	45.6
183.equake	0.2	0.3
188.ammp	2.1	3.2
191.fma3d	15.7	30.4
197.parser	3.0	5.8
256.bzip2	0.2	0.3
300.twolf	5.1	7.9
400.perlbench	1,740.5	7,112.2

Benchmark	Analysis Times (s)	
	Lpa-FI	Lpa-FS
401.bzip2	2.3	6.2
433.milc	1.5	3.6
435.gromacs	94.4	134.8
436.cactusADM	53.2	1,901.2
437.leslie3d	0.7	1.3
454.calculix	30.9	46.8
459.GemsFDTD	4.3	6.5
464.h264ref	20.7	57.7
465.tonto	159.1	752.9
482.sphinx3	3.9	8.2

listed in Table 2, these 11 benchmarks are the only ones for which Lpa is more effective than either BasicAA or SCEVAA or both.

There are totally 351 and 217 basic blocks vectorized by SLP under BasicAA and SCEVAA, respectively. Lpa has improved these results to 482 (under Lpa-FI) and 490 (under Lpa-FS). The final results of Lpa-FS outperformsBasicAA and SCEVAA by 1.39× and 2.26×, respectively. The most significant improvements happen at `177.mesa` and `433.milc`. In each case, Lpa enables SLP to discover 40 more vectorizable basic blocks, yielding an improvement of about 3x 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.

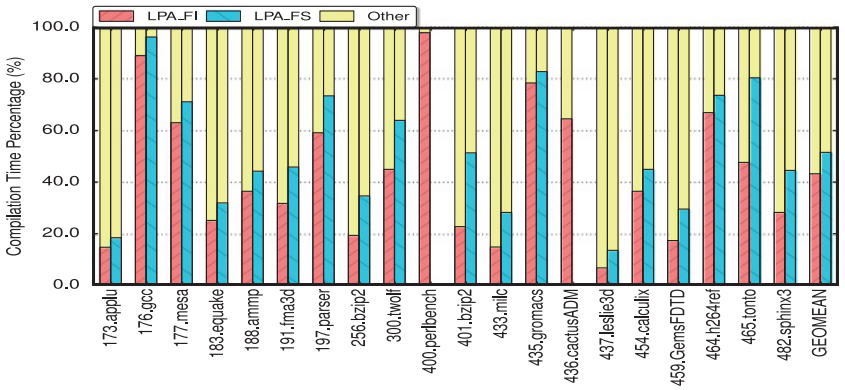


Fig. 16. Percentage of analysis time over total compilation time.

Table 4. Number of Basic Blocks Vectorized by SLP Under the Four Alias Analyses

Benchmark	Number of Basic Blocks Vectorized by SLP			
	BasicAA	SCEVAA	Lpa-FI	Lpa-FS
173.applu	20	4	26	26
176.gcc	4	3	6	6
177.mesa	24	23	64	66
183.equake	2	1	4	4
188.ammp	1	2	4	4
191.fma3d	46	23	53	56
433.milc	21	13	69	69
435.gromacs	53	35	57	58
454.calculix	161	92	166	168
465.tonto	19	21	32	32
482.sphinx	0	0	1	1
Total	351	217	482	490

Static Results of LLV. Table 5 gives the number of static alias checks inserted by LLV under the four analyses compared across the 14 benchmarks. Again, for the 20 benchmarks listed in Table 2, 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. The Lpa-FS further reduces the redundant checks to 220. 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 runtime under its reference input (Table 7). 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-FS is applied, over 50% of static checks introduced by LLVM's alias analyses have been eliminated in: `176.gcc`, `436.cactusADM`, `437.leslie3d`, `459.GemsFDTD`, `464.h264ref`, `465.tonto`, `482.sphinx3`. For `197.parser` and `256.bzip2`, all of their checks have been eliminated.

Table 5. Number of Static Alias Checks Inserted by LLV Under the Four Alias Analyses

Benchmark	Number of Static Alias Checks Inserted by LLV			
	BasicAA	SCEVAA	Lpa-FI	Lpa-FS
176.gcc	4	8	2	2
177.mesa	121	137	88	80
197.parser	1	1	0	0
256.bzip2	1	6	0	0
300.twolf	11	13	10	9
400.perlbench	23	21	13	7
401.bzip2	6	9	5	5
436.cactusADM	71	112	2	1
437.leslie3d	21	21	4	4
454.calculix	83	90	57	56
459.GemsFDTD	65	79	16	16
464.h264ref	30	32	2	2
465.tonto	110	118	38	37
482.sphinx3	4	5	1	1
Total	551	652	238	220

```
606   void cl_or (..., struct regnode_charclass_class *cl,
606                 struct regnode_charclass_class *or_with) {
637     for (i = 0; i < ANYOF_BITMAP_SIZE; i++)
638       cl->bitmap[i] |= or_with->bitmap[i];
661   }
683   STATIC I32 S_study_chunk(...) {
1095    oclass = data->start_class;
1096    data->start_class = &this_class;
1115    data->start_class = oclass;
1118    cl_or (..., data->start_class, &this_class);
1684  }
```

Fig. 17. Imprecise flow-insensitive pointer analysis example in `400.perlbench`. Weak updates at line 1115 result in aliases (cl->bitmap[i] and or_with->bitmap[i]).

Figure 17 shows a code snippet from `400.perlbench`, in which `cl` and `or_with` at line 638 point to the same object `&this_class` according to Lpa-FI, which leads to a weak update at line 1115 so that the old value of `data→start_class` is preserved. Thus, `cl` and `or_with` at line 606 are may-aliases, resulting in redundant checks inserted by LLV. However, a flow-sensitive analysis can strongly update `data→start_class` by killing its old contents, resulting in a must-not-alias for `cl` and `or_with` at line 606. Thus, the redundant checks are eliminated, resulting in more efficient vectorized code.

Partial Aliases. A total of 34 aliases queries issued by SLP and LLV are answered as partial aliases that are missed by the field-index-based approach in `176.gcc`, `177.mesa`, `435.gromacs` and `482.sphinx3`. All these partial aliases are generated, since the fields of a `union` object are accessed via pointers of different types. For example, a union type, `union` { `float32` *f*; `int32` *l*;}, in `482.sphinx3` is used to convert between floats and integers. In `435.gromacs`, a query is issued for an access to an object of `struct` {`int` *type*; `union` {`int*` *i*; `real*` *r*, `char**`*c*;} *old_contents*; }, where the struct contains a union used for implementing polymorphism in C with *type* as its tag

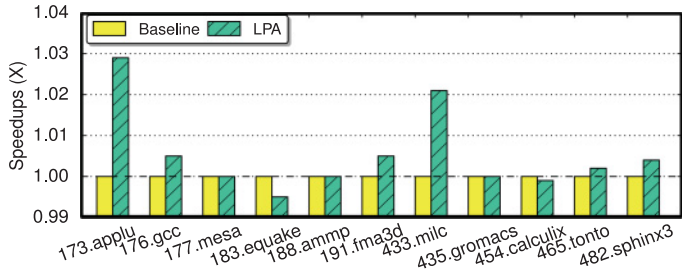


Fig. 18. SLP: whole-program speedups (with the baseline being the better of BasicAA and SCEVAA).

to indicate the type of the union object. Finally, the partial aliases in `176.gcc` are due to low-level bit operations over a whole union object, which can be accessed via multiple struct types in the form of `union { struct char` *common*`[]; struct { unsigned int` *code* `: 8; unsigned` *side_flag* `: 1; unsigned` *constant_flag* `: 1; } }`

5.3.2 Runtime Performance. Let us examine the performance gains obtained under Lpa given the above compile-time improvements achieved by SLP and LLV. In what follows, the results of Lpa represents the performance achieved by Lpa-FS. For a program, the baseline is the smaller of the two execution times achieved by SLP under BasicAA and SCEVAA.

Performance Improvements of SLP. Figure 18 gives the whole-program speedups achieved by SLP under Lpa normalized with respect to LLVM's alias analyses. Table 6 lists the code locations and execution frequencies for the new basic blocks that are vectorized by SLP under Lpa and executed under the reference inputs.

For all the benchmarks evaluated, `173.applu` and `433.milc` achieve the best speedups of 1.03× and 1.02×, respectively, because many new basic blocks vectorized under Lpa are frequently executed according to Table 6. 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, for `183.equake`, performance becomes worse when a certain basic block enclosed by a loop in the function `main` is vectorized (only) by SLP with LPA. As illustrated in Figure 19, compared to the basic block's scalar code, LLVM's code motion optimizer acts differently and does not hoist the computation on some loop-invariant variables (several `getelementptr` instructions) inside the loop to the loop preheader for the vectorized code. Due to the repeated execution of these getelementptr instructions (151,173×) and the relatively less frequent execution of the vectorized basic block (6× only due to the conditional branch), the extra overhead introduced outweighs the benefit of vectorization achieved.

Performance Improvements of LLV. Figure 20 is an analogue of Figure 18 to demonstrate the performance speedups achieved by LLV under Lpa with the same baseline. Correspondingly, Table 7 is an analogue of Table 6, except that we are here concerned with the loops whose runtime alias checks are completely removed by Lpa but would be 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 1.07×, as its removed runtime checks involve complex range checks for 10 different pointer pairs, with each pair executed 96,512×. For many other

Table 6. Code Locations and Execution Frequencies for the Basic Blocks that Are Vectorized by SLP Under LPA-FS but not LLVM's Alias Analyses, Under the Reference Inputs

Benchmark	Basic Blocks Vectorized and Executed		Execution Frequency
	Source Files	Line Numbers	
173.applu	applu.f	2688–2715,2838–2865 2988–3015	68,484,312
		2738–2747,2888–2897 3038–3047	63,761,256
176.gcc	regclass.c	904–906	1,781,066
		1597–1600	537,972
177.mesa	not executed		
183.equake	quake.c	913–914	6
188.ammp	rectmm.c	323–351	843,216
		1028–1043	559,731,147
191.fma3d	platq.f90	1986–1990,1998–2002	163,182,300
433.milc	addvec.c	11–13	33,600,000
	s_m_mat.c	29–48	6,400,000
	s_m_vec.c	15–18	800,000
	s_m_a_vec.c	18–21	446,480,000
	make_ahmat.c	40–45	657,920,000
	s_m_a_mat.c	17–20	302,080,000
	su3mat_copy.c	13–16	270,080,000
	rephase.c	44–47	14,720,000
435.gromacs	coupling.c	77–79	7,001
	vec.h	487–495	21,006
454.calculix	results.f	803–808	3,417,876
	incplas.f	669–672	278,437
465.tonto	rys.fppized.f90	1179–1195	6,769,676
		1198–1218	4,768,547
		1221–1241	3,759,643
482.sphinx3	utt.c	384–387	2,808

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.

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, then the slowdown will disappear.

5.3.3 Case Studies. To further understand the performance improvements of SLP and LLV observed in Figures 18 and 20, we have selected four representative kernels from the 20 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

Table 7. Code Locations and Execution Frequencies for the Loops Without (With) Runtime Alias Checks Under Lpa-FS (LLVM's Alias Analyses), Under the Reference Inputs

Benchmark	Loops With Their Runtime Checks Removed		Execution Frequency
	Source Files	Line Numbers	
176.gcc	tree.c	1155–1156	62,263
	emit-rtl.c	469–470	53
177.mesa	osmesa.c	746–748	96,512
256.bzip2	bzip2.c	1081–1082	2,674
300.twolf	qsortg.c	60–64	1,091
		141–145	312
400.perlbench	pp_sort.c	116	526,879
	regcomp.c	580–581	10
		583–584	191,784
		623–624	3
		637–638	315,240
401.bzip2	huffman.c	239	22,554
437.leslie3d	tml.f	3572,3573,3576,3577, 3578,3581,3582,3583, 3586,3587,3588,3648	8,863,680
454.calculix	A2_util.c	1320–1324	15,990,856
	Chv_swap.c	505–515	1,261,407
	IV_util.c	486–488	222
	InpMtx_init.c	182–192	111
	Utilities_DV.c	59–61	28,864,631
		118–120	2,490,749
		1147–1152	6
		1184–1188	34
	Utilities_IV.c	121–123	4,922,420
	Utilities_newsort.c	1130–1134,1136–1140	36,075
		1424–1428,1430–1434	1,354,311
459.GemsFDTD	huygens.fppized.f90	706–707,714–715,725–726, 735–736,820–821,826–827, 839–840,847–848	192,000
		709–710,717–718,730–731, 738–739,818–819,824–825, 833–834,844–845	191,000
		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	250,704
	mv-search.c	236	16,708,450
	rdopt.c	1858–1860	8,110,368
465.tonto	realmat.fppized.f90	3087,3095	348,490
482.sphinx3	new_fe_sp.c	207–209	64,584

```
double a[4], b[4];
for (...) {
  if (...) {
    a[0] = b[0] * c[0]; a[1] = b[1] * c[1];
    a[2] = b[2] * c[2]; a[3] = b[3] * c[3];
    ...
  }
}
```
(a) original code

```
for.preheader:
  %a0 = getelementptr [4 x double]* %a, 0, 0;
  %b0 = getelementptr [4 x double]* %b, 0, 0;
  %c0 = getelementptr [4 x double]* %c, 0, 0;
  ...
  %a3 = getelementptr [4 x double]* %a, 0, 3;
  %b3 = getelementptr [4 x double]* %b, 0, 3;
  %c3 = getelementptr [4 x double]* %c, 0, 3;
for.body:
  ...
if.then:
  ...
  %0 = load double* %b0;
  %1 = load double* %c0;
  %2 = fmul double %0, %1;
  store double %2, double* %a0;
  ...
```
(b) scalar code

```
for.preheader:
  %av = bitcast [4 x double]* %a to <4 x double>*;
  %bv = bitcast [4 x double]* %b to <4 x double>*;
  %cv = bitcast [4 x double]* %c to <4 x double>*;
for.body:
  %a1 = getelementptr [4 x double]* %a, 0, 1;
  %b1 = getelementptr [4 x double]* %b, 0, 1;
  %c1 = getelementptr [4 x double]* %c, 0, 1;
  ...
  %a3 = getelementptr [4 x double]* %a, 0, 3;
  %b3 = getelementptr [4 x double]* %b, 0, 3;
  %c3 = getelementptr [4 x double]* %c, 0, 3;
  ...
if.then:
  ...
  %0 = load <4 x double>* %bv;
  %1 = load <4 x double>* %cv;
  %2 = fmul <4 x double> %0, %1;
  store <4 x double> %2, <4 x double>* %a0;
  ...
```
(c) vector code

Fig. 19. LLVM-IR before and after SLP transformation of code snippet in `183.equake`.

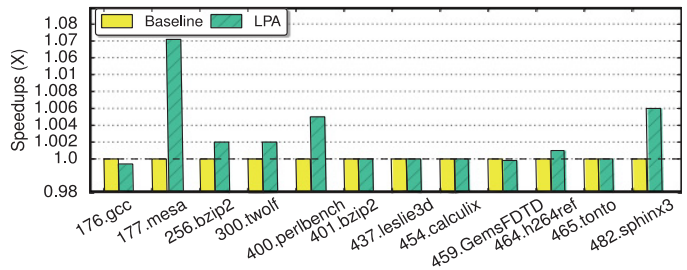


Fig. 20. LLV: whole-program speedups (with the baseline being the better of BasicAA and SCEVAA).

for improving LLV, as listed in Figures 21(a) and 21(b), and their code snippets in Figures 21(c)–21(f). Figures 21(g) and 21(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 21(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. 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 inter-procedural analysis. Guided by Lpa, SLP has successfully vectorized this kernel, resulting in a speedup of 2.03×.

`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 21(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×.

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

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

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   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  }
```

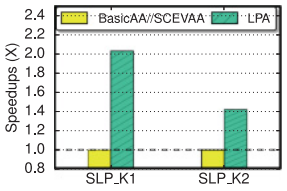
(d) Source code of SLP_K2

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

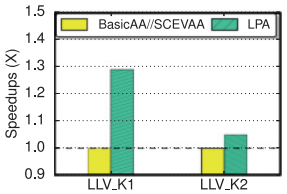
(e) Source code of LLV_K1

```
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



(g) Kernel speedups for SLP



(h) Kernel speedups for LLV

Fig. 21. Case studies for four selected kernels with their improved performance under LPA (over BASICAA and SCEVAA).

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 1.29× 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×).

5.4 Discussions

We describe two principal directions along which our approach can be further improved. One future direction is to develop a more sophisticated value range analysis. In our current implementation, the SCEV-based range analysis can be overly conservative, since the estimated ranges of values are sometimes crude over-approximations of their actual runtime ranges. This can happen in the case of indirect array accesses (e.g., a[*p] and a[b[i]]) or irregular loops (e.g., iterating arrays inside a loop with variant bounds).

Figure 22 shows an indirect array access in a loop from a frequently executed function `inl1130` in benchmark `435.gromacs` from CPU2006. LPA fails to enable LLV to remove redundant checks, since it cannot disambiguate the indirect array accesses made by different iterations of the loop at

```
3971    for (k = nj0; k ¡ nj1; k++)
3972        j = 3*ijnr[k];
4141        fac[j] = fac[j] - dx*fs
4176    }
```

Fig. 22. An imprecise value range analysis example, where indirect array accesses made by different iterations of the loop at line 3972 cannot be disambiguated.

```
110   #define fswap(zz1, zz2)        \
111   { Int32 zztmp = zz1; zz1 = zz2; zz2 = zztmp; }
113   #define fvswap(yyp1, yyp2, yyn)                    \
114   {                                                  \
118     while (yyn > 0) {                                \
119       fswap(fmap[yyp1], fmap[yyp2]);                 \
120       yyp1++; yyp2++; yyn–;                          \
121     }                                                \
122   }
125   #define fmin(a,b) ((a) < (b)) ? (a) : (b)
140   void fallbackQSort3 (UInt32* fmap, ...) {
213     n = fmin(ltLo-lo, unLo-ltLo);
213     fvswap(lo, unLo-n, n);
214   }
```

Fig. 23. An imprecise value range analysis, where `fmap[yyp1]` and `fmap[yyp2]` cannot be disambiguated due to variable loop bounds.

line 3972. Thus, validity runtime checks must be inserted outside the loop to perform the runtime pointer disambiguation.

Figure 23 shows an irregular loop code segment as a part of qsort in `401.bzip2` from CPU2006. The fswap at line 119 can be vectorized by LLV without requiring validity checks, because `fmap[yyp1]` and `fmap[yyp2]`, where $yyp1 \in [lo, lo + n - 1]$ and $yyp2 \in [unLo - n, unLo - 1]$, always access two different elements of the array fmap. However, LPA fails to remove those redundant checks due to variable loop bounds.

To improve the precision of value range analysis, one option is to use some advanced loop analysis and transformation frameworks, such as Polly (Grosser, Zheng, Aloor, Simbürger, Größlinger, and Pouchet Grosser et al.). Another is to handle pointer arithmetic more precisely by using symbolic range analysis (Paisante et al. 2016) possibly with a integer linear programming (ILP) solver (Rugina and Rinard 2000).

An inter-procedural flow-sensitive analysis based on our fine-grained memory model AMM can be time-consuming in analyzing large programs, such as 400.perlbench. LPA-FS takes around two hours to finish the analysis (Table 5).

Another future direction is to improve the scalability of our loop-oriented pointer analyses using demand-driven approach, like CFL-Reachability (Zheng and Rugina 2008; Sridharan and Bodík 2006; Sui and Xue 2016a), so that we can accelerate precise analysis, such as flow-sensitive analysis, at compile-time to generate vectorized code more efficiently.

6 RELATED WORK

Pointer Analysis. As a fundamental enabling technique, pointer or alias analysis (Andersen 1994; Jung and Huss 2004; Hardekopf and Lin 2011; Sui et al. 2016) paves the way for software bug detection (Sui et al. 2012; Ye et al. 2014a; Sui and Xue 2016a), enforcing control-flow integrity (Fan et al. 2017) and compiler optimizations (Nguyen and Xue 2015; Sui et al. 2013). In automatic SIMD vectorization, statements grouped together for vectorization must be dependence-free. A

recent evaluation on vectorizing compilers (Maleki et al. 2011) reveals some limitations of existing dependence analyses, calling for more precise alias analyses 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 (Hardekopf and Lin 2011; Pearce et al. 2007) 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 (1995) introduced a byte-precise model based on location sets, without handling loops and arrays precisely enough to support SIMD optimizations.

Recently, cclyzer (Balatsouras and Smaragdakis 2016) presented a fine-grained field-sensitive Andersen's analysis that infers lazily the types of heap objects by leveraging the type casting information to filter out redundant field derivations. By assuming that every (concrete) object has a single type throughout its lifetime, cclyzer improves the precision of field-sensitivity in the presence of factory methods and heap allocation wrappers, achieving the heap cloning results without explicit context-sensitivity. Unlike cclyzer, LPA does not infer the type of an abstract object. Instead, LPA calculates the size of an object by looking at the types of the pointers that point to the object, with a conservative assumption that one object can have multiple types. In cclyzer, type casting information can be leveraged to enable heap cloning analysis. However, the alias queries from the current vectorization algorithms are all issued from the same function. Therefore, the two heap wrappers must reside in the same function to make `cclyzer` effective in disambiguating heap-related aliases. Loop-oriented clients, such as SIMD vectorization, limit the capability of loop-unware points-to analyses, such as `cclyzer`. In addition, `cclyzer`, which assumes that one object has a single type throughout its lifetime, is unable to handle partial aliases queries issued in some programs (Section 5.3.1) when two pointers of different types access the same union object (as also discussed in Balatsouras et al. [2016]) or the same struct object being cast multiple times along a single program path (illustrated in Figure 6).

This article introduces an inter-procedural loop-oriented pointer analysis that precisely analyzes aggregate data structures, including deeply nested arrays, arrays of structs, and structs of arrays to enable effective SIMD vectorization.

Auto-Vectorization. Loops are the main target of the two important vectorization techniques, superword-level parallelism vectorization (SLP) (Barik et al. 2010; Larsen and Amarasinghe 2000; Shin et al. 2005; Porpodas et al. 2015; Zhou and Xue 2016a) and loop-level vectorization (LLV) (Nuzman et al. 2006; Trifunovic et al. 2009; Shin 2007). The first SLP approach is proposed in Larsen and Amarasinghe (2000), which obtains isomorphic statement groups by tracing data flows starting from consecutive memory accesses. Dynamic programming (Barik et al. 2010) 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 Shin et al. (2005). More recently, some researchers (Porpodas et al. 2015) focused on transforming non-isomorphic statement sequences into isomorphic ones to broaden the scope of SLP (Larsen and Amarasinghe 2000). Loop-level vectorization is developed based on the technology originally designed for vector machines. Many improvements have been made, by handling interleaved data accesses (Nuzman et al. 2006), control flow divergence (Shin 2007), and loop transformations (Trifunovic et al. 2009). Recently, Zhou and Xue (2016b) introduced an approach to exploiting both SLP and loop-level SIMD parallelism simultaneously by reducing the data reorganization overhead incurred.

7 CONCLUSION

This article 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 based on how structs and arrays are accessed. Our results show that LPA is more effective than LLVM's BasicAA and SCEVAA in improving the performance speedups achieved by SLP and LLV across a number of SPEC benchmarks.

8 ACRONYMS, ABBREVIATIONS AND NOTATIONS

8.1 Acronyms and Abbreviations

LLVM	Low Level Virtual Machine
GCC	GNU Compiler Collection
SPEC	Standard Performance Evaluation Corporation
SLP	Superword-Level Parallelism Vectorization
LLV	Loop-Level Vectorization
LPA	Loop-Oriented Pointer Analysis
SIMD	Single Instruction, Multiple Data
BASICAA	Basic Alias Analysis
SCEVAA	Analysis Analysis Based on SCalar Evolution expression
LPA-FI	Flow-Insensitive LPA
LPA-FI	Flow-Sensitive LPA
SSA	Static Single Assignment Form
KLOC	Thousands (kilo) of Lines of Code
AVX	Advanced Vector Extensions

8.2 Abbreviations and Notations

$\mathcal{V}$	Program variables (Section 2.1)
$\mathcal{T}$	Top-level variables (Section 2.1)
$\mathcal{A}$	Address-taken variables (Section 2.1)
$alloc_a$	Static allocation where a is either a stack and global object (Section 2.1)
$malloc_a$	Dynamic allocation where a is a heap object (Section 2.1)
$p[i]$	Array memory expression (Section 2.1)
$*p$	Pointer dereference memory expression (Section 2.1)
$p \rightarrow f$	Field dereference memory expression (Section 2.1)
$pt(v)$	Flow-insensitive points-to set of v (Section 2.2)
σ	Access-based location set (Section 3.1)
off	Offset from the beignning of an object (Section 3.1)
X	Access step (Section 3.1)
lb	Lower bound of an interval range (Section 3.1)
ub	Upper bound of an interval range (Section 3.1)
$[lb, ub]$	Interval range (Section 3.1)
$\sqcap$	Range intersection (Section 3.1)
T	Access-trip stack (Section 3.1)
t	Trip count (Section 3.1)
s	Stride (Section 3.1)

es	Size of an array element (Section 3.1)
$LS(\sigma)$	Locations of an access-based location set (Section 3.1)
$\bowtie$	Two location sets have overlapping memory locations (Section 3.2.2)
$pt(\ell, v)$	Flow-sensitive points-to set of v at a program statement ℓ (Section 4)
(ℓ, v)	Definition of a variable $v \in \mathcal{V}$ at a program statement ℓ (Section 4)
$\ell' \xhookrightarrow{v} \ell$	The pre-computed def-use of variable v from ℓ' to ℓ (Section 4)

REFERENCES

Lo Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation.

Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. 1994. Chains of recurrences—A method to expedite the evaluation of closed-form functions. In *Proceedings of the ISAAC'94*. 242–249.

George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive points-to analysis for C and C++. In *Proceedings of the SAS'16*.

Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2010. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the Micro'10*. 201–212.

Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT'17*. 329–340. DOI: http://dx.doi.org/10.1145/3092703.3092729

Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in {LLVM}. In *Proceedings of the IMPACT'11*.

Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the PLDI'07*. ACM, 290–299.

B. Hardekopf and C. Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO'11*. 289–298.

ISO90. 1990. ISO/IEC. international standard ISO/IEC 9899, programming languages C.

Michael Jung and Sorin Alexander Huss. 2004. Fast points-to analysis for languages with structured types. In *Software and Compilers for Embedded Systems*. Springer, 107–121.

Ralf Karrenberg. 2015. Whole-function vectorization. In *Proceedings of the CGO'11*. Springer, 85–125.

Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the PLDI'00*. 145–156.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the CGO'04*. IEEE Computer Society, 75.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *Proceedings of the POPL'11*. 3–16.

LLVM-Alias-Analysis. 2017. Retrieved from http://llvm.org/docs/AliasAnalysis.html.

Saeed Maleki, Yaoqing Gao, Mara J. Garzaran, Tommy Wong, David Padua, et al. 2011. An evaluation of vectorizing compilers. In *Proceedings of the PACT'11*. IEEE, 372–382.

Phung Hua Nguyen and Jingling Xue. 2015. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *Proceedings of the ACSC'05*. 9–18.

Esko Nuutila and Eljas Soisalon-Soininen. 1994. On finding the strongly connected components in a directed graph. *Inform. Process. Lett.* 49, 1 (1994), 9–14.

Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the PLDI'06*. 132–143.

Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: Revisited for short SIMD architectures. In *Proceedings of the PACT'08*. ACM, 2–11.

Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2016. Symbolic range analysis of pointers. In *Proceedings of the CGO'16*. ACM, 171–181.

David J. Pearce, Paul H. J. Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *Proceedings of the TOPLAS'07* 30, 1 (2007), 4.

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the CGO'09*. 126–135.

Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP automatic vectorization. In *Proceedings of the CGO'15*. IEEE, 190–201.

Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM TOPLAS* 16, 5 (1994), 1467–1471.

Rajiv Ravindran Rick Hank, Loreena Lee. 2010. Implementing next generation points-to in open64. In *Open64 Developers Forum*. Retrieved from http://www.affinic.com/documents/open64workshop/2010/.

Radu Rugina and Martin Rinard. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the PLDI'00*, Vol. 35. ACM, 182–195.

Jaewook Shin. 2007. Introducing control flow into vectorized code. In *Proceedings of the PACT'07*. 280–291.

Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-level parallelism in the presence of control flow. In *Proceedings of the CGO'05*. 165–175.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *Proceedings of the PLDI'06*, 387–400.

Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the CGO'16*. 160–170.

Yulei Sui, Yue Li, and Jingling Xue. 2013. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the CGO'13*. 1–11.

Yulei Sui and Jingling Xue. 2016a. On-demand strong update analysis via value-flow refinement. In *Proceedings of the FSE'16*.

Yulei Sui and Jingling Xue. 2016b. SVF: Interprocedural static value-flow analysis in LLVM. https://github.com/unsw-corg/SVF. In *Proceedings of the CC'16*. 265–266.

Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the ISSTA'12*. ACM, 254–264.

Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the PACT'09*. 327–337.

Robert van Engelen. 2001. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the CC'01*. 118–132.

Robert P. Wilson and Monica S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the PLDI'95*. ACM, 1–12.

Ding Ye, Yulei Sui, and Jingling Xue. 2014a. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *Proceedings of the CGO'14*. ACM, 154.

Sen Ye, Yulei Sui, and Jingling Xue. 2014b. Region-based selective flow-sensitive pointer analysis. In *Proceedings of the SAS'14*. Springer, 319–336.

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the POPL'08*. 197–208.

Hao Zhou and Jingling Xue. 2016a. A compiler approach for exploiting partial SIMD parallelism. *ACM Trans. Arch. Code Optim.* 13, 1 (2016), 11:1–11:26.

Hao Zhou and Jingling Xue. 2016b. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the CGO'16*. 59–69.

Received December 2016; revised August 2017; accepted November 2017