Generalized Profile-Guided Iterator Recognition

Generalized Profile-Guided Iterator Recognition

Stanislav Manilov
S.Z.Manilov@sms.ed.ac.uk
University of Edinburgh
United Kingdom

Christos Vasiladiotis
C.Vasiladiotis@sms.ed.ac.uk
University of Edinburgh
United Kingdom

Björn Franke
bfranke@inf.ed.ac.uk
University of Edinburgh
United Kingdom

Abstract

Iterators prescribe the traversal of data structures and determine loop termination, and many loop analyses and transformations require their exact identification. While recognition of iterators is a straight-forward task for affine loops, the situation is different for loops iterating over dynamic data structures or involving control flow dependent computations to determine the next data element. In this paper we propose a compiler analysis for recognizing loop iterators code for a wide class of loops. We initially develop a static analysis, which is then enhanced with profiling information to support speculative code optimizations. We have prototyped our analysis in the LLVM framework and demonstrate its capabilities using the SPEC CPU2006 benchmarks. Our approach is applicable to all loops and we show that we can recognize iterators in, on average, 88.1% of over 75,000 loops using static analysis alone, and up to 94.9% using additional profiling information. Existing techniques perform substantially worse, especially for C and C++ applications, and cover only 35–44% of the loops. Our analysis enables advanced loop optimizations such as decoupled software pipelining, commutativity analysis and source code rejuvenation for real-world applications, which escape analysis and transformation if loop iterators are not recognized accurately.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords Loop iterators, loop analysis

ACM Reference Format:
Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. 2018. Generalized Profile-Guided Iterator Recognition. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179511

1 Introduction

Advanced compiler optimization [43] is largely concerned with the optimization of loops as the largest proportion of time executing a program is spend in loops, where small per-iteration improvements multiply to greater overall effect. In fact, mathematical frameworks – such as the polyhedral model [6] – underpinning loop optimization and parallelization have been specifically designed to capture loop behavior for analysis and transformation. Central to loop analysis is knowledge of loop iterators, which determine the iteration ranges of loops, are updated in every iteration and prescribe how data structures are traversed. Whereas recognition of loop iterators for Fortran-style do-loops is trivial and can be accomplished using syntactic pattern matching, most compilers employ some variation of induction variable recognition at the intermediate representation level to capture a wider range of iterators resulting from a multitude of idioms and programming styles.

While advanced loop optimizations have been developed in the academic literature their deployment in commercial or open-source compilers such as LLVM has been hampered by the fact that real-world source code often defeats existing compiler analyses. Our generalized iterator recognition analysis developed in this paper represents a critical step towards enabling development and application of novel loop transformations, applicable to wider range of loops.

It is possibly surprising that different interpretations of what constitutes a loop iterator have been developed in the compiler and programming language communities. For example, in the polyhedral model underpinning many loop transformations each loop iteration within loop nest is modelled as a lattice point inside a polytope describing the loop iteration space. A loop iterator is then an enumerator for point vector with integer valued components with bounds representing the faces of the loop limiting polytope. In comparison, in the object-oriented programming community iterators are objects, which enable a programmer to traverse a container data structure, e.g. a list. Iterators are objects instantiated from iterator classes, which implement a common interface and provide possibly complex functionality through method invocations on iterator objects and abstract the implementation details of how exactly a data structure is traversed from the programmer. In this paper we aim to develop a generalized notion of loop iterators, which is not restricted to affine loops or data structure traversals using object-oriented iterator objects, but applicable to the widest possible range of loops.

Based on our generalized definition of loop iterators we develop a static analysis, which can be readily integrated in a compiler framework such as LLVM without relying on external tools. We then enhance this static analysis to incorporate profiling information to overcome inherent limitations, e.g. data dependence analysis is undecidable in general [40]. We show how profiling information can complement static analysis to enable more aggressive iterator recognition, i.e. fewer loop instructions are marked as "iterator code" and more as "payload". This can be useful for e.g. speculative optimizations, which benefit from a potentially larger loop payload and can better cope with "unsafe" information.

We evaluate the usefulness of our generalized iterator definition and analysis by its ability to separate loops into iterator code and loop "payload". For this we have developed an LLVM analysis pass and applied to the entire SPEC CPU2006 benchmark suite, including integer and floating-point applications written in C, C++ and Fortran. In total, we cover over 75,000 loops representing real-world applications. We show that iterator recognition can be applied to all loops and is able to successfully recognize iterators in the vast majority of cases. In particular, our iterator recognition can separate iterators from non-affine loops, which escape traditional analysis.

1.1 Motivating Examples and Use Cases

To motivate the technique developed in this paper we consider three loop analyses and transformations, which have in common that each of them can benefit from improved separation of code constituting the loop iterator from the rest of the code, forming the per-iteration "payload" of the loop.

Consider the affine loop in Figure 1, where the iterator i can be trivially recognized using syntactic pattern matching[1]. Optimizing and parallelizing compilers typically rely on the recognition of basic loop iterators to enable further analyses and transformations [29]. Induction variables induced by the surrounding loop (not present in this example) can be identified and substituted by closed form expressions using techniques developed in e.g. [35, 36].

Now consider the example in Figure 2. This loop traverses a recursive data structure – a singly-linked list – and applies a local update to each element of the list. It is clear that this loop does not have a natural iterator in the sense of an integer value loop index incremented by a fixed amount in every iteration [36], but instead a pointer is updated and checked. Such pointer chasing iterators can be recognized using ad-hoc pattern matching [39] or using an algorithm developed as part of DSWP+ [22, 45]. Following successful recognition of the loop iterator, highlighted in the example, the loop can be parallelized using a decoupled software pipelining (DSWP) approach.

[1]Please note that examples in this paper are shown using C/C++ source code for ease of illustration, whereas the techniques and their implementations operate on internal representations such as LLVM IR and are language agnostic (with exception of source code rejuvenation in Figure 3).

```
int array[100];
...
for(i = 0; i < 100; i++) {
  array[i]++;
}
```

Figure 1. An affine loop, where the highlighted loop iterator is trivially recognized using syntactic pattern matching. Further induction variables depending on this iterator i may be recognized using techniques developed in e.g. [29, 35, 36].

```
my_list ptr;
...
while(ptr) {
  ptr->val++;
  ptr = ptr->next;
}
```

Figure 2. A traversal of a recursive data structure loop with highlighted iterator code. Recursive Data Structure (RDS) loop iterators can be recognized using an ad-hoc technique [39] or a partitioning algorithm in DSWP+ [22, 45].

An alternative use of iterator recognition is in source code rejuvenation [34], where the loop from Figure 2 may be converted to the form shown in Figure 3, where the user-defined singly-linked list and its traversal have been replaced with an STL list container and its hidden iterator methods, another common programming language abstraction of iterators offered by the C++ programming language. Generalized iterator recognition (together with shape analysis [47] and abstract data structure recognition [12]) enables this kind of loop transformation aimed at raising code abstraction.

Our third example in Figure 4 is a code excerpt showing breadth-first graph search implemented in the C++ programming language. Conceptually, the loop spanning lines 15–30 of this example is similar to the loop in the pointer-chasing loop in Figure 2, but now the code makes use of STL's list container to store lists of adjacent nodes (adj) and a queue of nodes (queue) needed for BFS traversal. Within the loop, methods for checking and updating the iterator are invoked. Additionally, the loop contains an inner loop in lines 24–29 with its own iterator and further conditional inner control flow, adding to the complexity of the code contributing to the traversal of the graph data structure. Conceptually, we can say that a BFS graph iterator involves a queue and operations on it as well as an additional loop to enqueue newly discovered nodes, which matches our expectations for this algorithm.

In this example, we can separate out the output statements in line 18, which do not contribute to the loop traversal and its termination, and we therefore consider to form the

```
std::list<int> list;
...
for(int &val : list) {
  val++;
}
```

Figure 3. The loop from Figure 2 after source code rejuvenation, where the traversal of the user-defined linked list has been replaced with an equivalent STL `list` container and abstracted iterator methods. Iterator recognition enables further analyses driving this kind of loop transformation.

loop "payload". RDS loop recognition [22, 39, 45] is defeated by this loop comprising method invocations and complex control flow, whereas the technique presented in this paper successfully identifies the code forming the (highlighted) iterator of the outer `while`-loop. Accurate recognition of the iterator in this example can be used to enable further analysis, e.g. to drive DSWP-style parallelisation or loop commutativity analysis [41].

```
1  void Graph::BFS(int s)
2  {
3    // Mark all the vertices as not visited
4    bool *visited = new bool[V];
5    for(int i = 0; i < V; i++)
6      visited[i] = false;
7
8    // Create a queue for BFS
9    list<int> queue;
10
11   // Mark the current node as visited and enqueue it
12   visited[s] = true;
13   queue.push_back(s);
14
15   while(!queue.empty()) {
16     // Dequeue a vertex from queue and print it
17     s = queue.front();
18     cout << s << "␣";
19     queue.pop_front();
20
21     // Get all adjacent vertices of the dequeued
22     // vertex s. If a adjacent has not been visited,
23     // then mark it visited and enqueue it
24     for(auto i : adj[s]) {
25       if(!visited[i]) {
26         visited[i] = true;
27         queue.push_back(i);
28       }
29     }
30   }
31 }
```

Figure 4. Breadth-first search of a graph implemented in C++ and using STL's `list` container to store lists of adjacent nodes (`adj`) and a queue of nodes (`queue`) needed for BFS traversal. The iterator of the `while`-loop spanning lines 15-30 comprises complex control flow and method invocations, and can be separated from the loop "payload" in line 18, which prints a vertex id. Example adopted from [1].

The previous examples highlight different notions of iterators and how iterator recognition can drive novel optimizations. They also demonstrate that current techniques for iterator recognition are limited in their ability to process non-affine iterators. As a result, success of advanced loop transformations which require separation of loop iterator code from the payload is hampered. What is needed is a technique capable of processing real-world codes employing a broad range of different styles of loop iterators, which may make use of user-defined data structures, STL or Boost containers and iterators, and complex control flow alike.

1.2 Contributions

In this paper we propose a new definition of generalized loop iterators, which subsumes various existing notions of iterators. We then develop a novel iterator recognition algorithm, which can be used to separate iterator code from the loop "payload". We develop two versions of our iterator recognition algorithm: (a) based on static analysis only, which is fast, but conservative, and (b) using additional profiling information to enable more aggressive speculative optimizations. We evaluate our LLVM prototype implementation against the SPEC CPU2006 benchmark suite and demonstrate its ability successfully recognize more loop iterators than any other technique.

2 Background

Affine Iterators. In compiler theory iterators are typically associated with structured `for`-loops, where a normalized loop iterates over a sequence of consecutive integer numbers between affine lower and upper bounds. The iteration space of such a loop or loop nest is an ordered set of loop iterations, in which each iteration is represented by a point $(i_1, \ldots, i_n)$ for a loop nest of depth n. Loop iterators are then the space represented by a column vector $\mathbf{I} = [i_1, \ldots, i_n]^T$, and loop ranges form a system of inequalities $LB_n(i_1, \ldots, i_{n-1})^T \leq i_n \leq UB_n(i_1, \ldots, i_{n-1})^T$, where LB_k and UB_k are lower and upper loops bounds, respectively, at nesting level k.

RDS/DSWP Loop Partitioning. Parallelization of *RDS* loops is the main concern of DSWP [39]. Traversals of a RDS are matched against a syntactic pattern. Specifically, DSWP searches for load instructions that are data dependent on previous instances of the same instruction. These *induction pointer loads (IPL)* form the kernel of the traversal slice [31]. IPLs can be identified using augmented techniques for identifying induction variables [16, 31]. This relatively simple approach works well for one-dimensional pointer-chasing loops, but fails beyond that.

A more sophisticated technique for detecting cycling dependences in loops has been developed in [22, 32, 45] as part of DSWP+. It comprises an algorithm for systematic separation of dependence cycles, some of which may be involved in loop iterators. However, DSWP+ does not make any attempt to recognize iterators. Instead, all dependence cycles are treated equal. In Section 3 we extend this algorithm with the ability to extract the specific instructions and variables, which form the loop iterator while marking up the remaining code as "payload".

Object-oriented Iterators. Iterators have a different meaning in the OOP community [7, 27] where constructs such as C++ STL iterators are considered. Specifically, in C++ an iterator is any object that, pointing to some element in a range of elements (e.g. a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (*) operators). Using existing compiler techniques these iterators escape analysis and transformation.

Iteratorless and Unseparable Loops. Not all loops have iterators, which naturally advance the position in an iteration space or data structure. For example, consider a spin-lock loop, which spins until a flag is set by e.g. an interrupt handler or in another thread. In some cases iterators cannot be separated from a distinct loop "payload". An example of such an unseparable loop is a linear search in a linked list.

3 Methodology

In this section we present our methodology for iterator recognition. Initially, we will provide an overview and create an intuitive understanding of iterators before we develop a formal definition of our concept of iterators. This is followed by a static analysis for iterator recognition, which – as a useful addition for speculative loop transformation – is subsequently complemented with profiling information to significantly enhance its capability to separate iterator code. Finally, we share insights from our LLVM prototype implementation.

3.1 Definitions

Intuitively, a loop iterator is a variable (or a set of variables), which is updated in every iteration of a loop and is involved in controlling loop exits, e.g. as part of a conditional expression. This intuitive understanding is captured in the following definition:

Definition 3.1. *Generalized Loop Iterator.* A generalized loop iterator is a minimal set of variables and operations manipulating these variables, which form a Strongly Connected Component (SCC) in the Program Dependence Graph (PDG) and exhibit a loop-carried dependence of distance 1. Furthermore, this SCC has no incoming edges from other SCCs in the PDG.

For this definition we exploit that conditional expressions controlling loop exits will introduce control dependences to every operation contained in the loop body. Since we are also interested in variables, which are updated in every loop iteration, we are looking for data dependences in the other direction, i.e. from update operations in the loop body towards read operations in loop termination expressions. Together these dependences will form a loop-carried dependence cycle or, more generally, a SCC in the PDG. Other operations may depend on this SCC, but it is the dominant SCC of operations, which does not depend on any other operations and variables that determines loop termination and thus constitutes the loop iterator.

In counted or affine loops the conventional iterator is intuitively covered by our definition as it is this variable (often `i`) and its updates and checks, that form the dominant SCC controlling execution of the remaining loop operations, which in turn form the loop "payload". Similarly, iterators of pointer-chasing loops are covered by the same definition as pointer updates and checks introduce a cyclic dependence relation on which the remaining loop body depends. However, our definition also covers STL-like iterators, which are updated and checked in every single loop iteration, possibly involving calls to methods in other classes, which necessitate the use of inter-procedural analysis for their identification.

Abstraction and generalization of the properties of a loop iterator in our definition allows us to develop an iterator recognition analysis operating on the compiler IR, thus enabling a source-language agnostic approach.

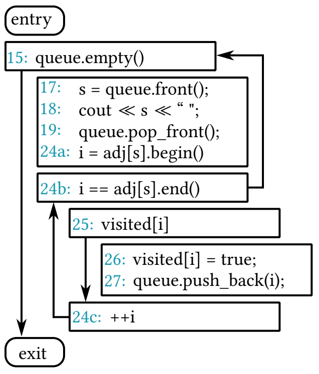
3.2 Static Analysis

Our analysis for determining loop iterators involves three stages closely following Definition 3.1:

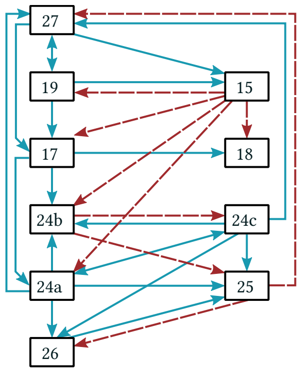
1. **PDG construction**. We assume the IR provides us with a Control Flow Graph (CFG) in Static Single Assignment (SSA) form (Figure 5a). We apply the algorithm from [11] to construct the Control Dependence Graph (CDG) of the loop. Additionally, we combine the implicit def-use chain present in the intermediate representation with a static dependence analysis of memory accesses based on [19] to build the Data Dependence Graph (DDG) of the loop. We produce the PDG by combining the CDG and DDG (Figure 5b).
2. **Determine SCCs.** Once we have the program dependence graph of a loop, we determine its strongly connected components. We build a directed acyclic graph (DAG) connecting the SCCs using Kosaraju's algorithm [2].
3. **Dominant SCC and iterator recognition.** Finally, we take the dominant SCC, i.e. the one that has no incoming edges in the SCC DAG. This dominant SCC represents the loop iterator and we label instructions represented by the SCC as "iterator instructions" and variables involved as "iterator variables" (Figure 5c). Inspection of the properties of these iterator instructions and variables reveals that together they satisfy Definition 3.1 by construction, showing that we have indeed recognized the loop iterator.

3.3 Incorporating Profiling Information

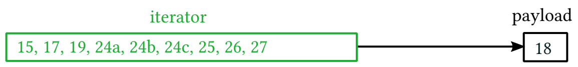
Conservatism of the static analysis used as part of the construction of the DDG in section 3.2 is a limiting factor. *May* dependences introduce spurious dependence relations, which may not materialize for any program execution on valid input data. We now investigate how incorporating profiling



(a) CFG.



(b) Corresponding PDG.



(c) SCCs of the PDG. The green SCC has no incoming edges and constitutes the iterator.

Figure 5. CFG, PDG & SCC for the `while`-loop in Fig. 4.

information obtained from instrumentation and execution of the target program can be used to improve iterator recognition. We use a profiling technique similar to [49] to capture data dependences.

Clearly, any approach relying on profiling information for the computation of data dependences is prone to errors such as missing a dependence, which did not materialize in a specific execution trace, but could well occur in another [46]. However, there is still benefit in incorporating such unsafe information for two reasons: (a) we can determine an upper bound, which enables us to quantify the scope for improvements of static analyses, and (b) some transformations, e.g. speculative parallelization [33], are inherently resilient against data dependence violations and can benefit from more aggressive loop transformations.

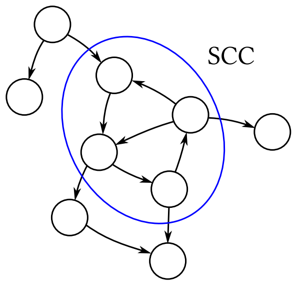
In principle, we complement our PDG construction algorithm (stage 1 above) with profiling information similar to [24], i.e. for each of the statically detected *may* data dependences we refer to profiling information to resolve this *may* dependence as either *must* dependence or *no* dependence, based on whether or not a dependence was observed in the execution profile. The remaining stages of the algorithm remain unchanged.

We expect users to profile their applications using representative, but reduced data sets, e.g. [14], and focussing their efforts on those parts of the program relevant to their targeted transformation in order to avoid excessive profiling costs.

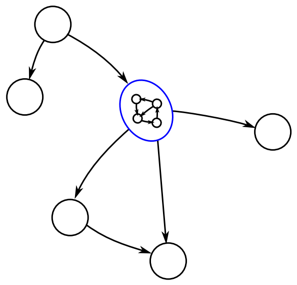
3.4 Implementation

We have implemented a prototype of our technique as an analysis pass in the LLVM compiler infrastructure. This allows us to analyze all of the SPEC CPU2006 benchmarks, since there are LLVM front-ends available for C/C++ (clang) and Fortran (flang). The LLVM IR is a CFG in SSA form as needed by the technique. Standard LLVM analyses allow for detecting loops in this graph and promoting memory accesses to virtual registers and thus simplifying the IR.

Instrumentation & Profiling. In order to collect memory access information about a subject program, we apply a context aware memory profiler. This begins with building a complete call graph of the program and computing context offsets for function calls. When these context offsets are accumulated for a given stack trace (i.e. a sequence of function calls) the result is always a context identifier that is unique for the given stack trace. We handle groups of recursive functions and indirect function calls. The former we do by reducing the call graph to a SCC DAG and assigning a unique context ID to every SCC: the calls within the recursive groups have offsets of zero (see Figure 6). The latter – by adding calls to the runtime that indicate indirect function calls and function returns and keeping a stack of indirect calls at runtime.



(a) A recursive call graph.



(b) Contracted SCC DAG.

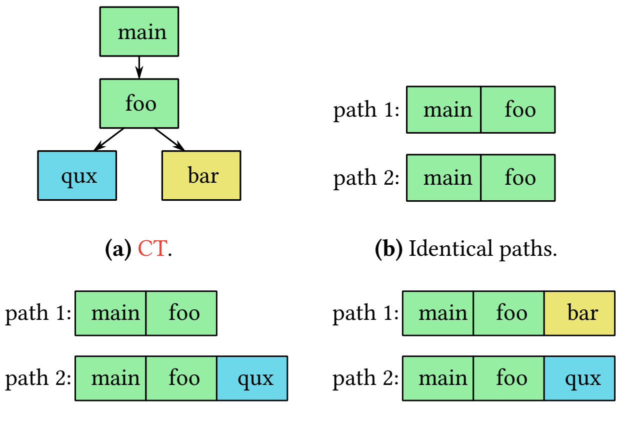
Figure 6. Handling recursion robustly. The strongly connected component in Figure 6a is represented by a single node in the contracted graph in Figure 6b. The resulting graph does not contain any further cycles, i.e. it is a DAG.

Once this map from function calls to context offsets is computed we instrument each call with calls to the runtime to advance and then restore the context ID. Once this is done we instrument all memory accesses with a call to a runtime function that records the instruction ID and the address that is accessed.

Lastly, we also instrument loops, since we want to know which iterations triggered a dependence and across which loop nesting level did the dependence occur. For this, we instrument loop entering and loop exiting edges in the CFG as well as loop backedges.

Once instrumentation is complete we execute the program and collect a profile that consists of a list of the dependences and the code coverage. Each dependence is described by its type (RAW, WAW, WAR); source and target instructions; source and target contexts; and list of loop iteration counters. Iteration counters are obtained by counting the number of times a backedge was encountered, rather than relating to our complex notion of iterators that we are trying to extract.

Incorporating Profiling Information. Once the profiling information is collected, we incorporate it back in the analysis to augment the static analysis results. Because we need to map unique contexts back to the functions in which they occur, we need to explicitly build the tree of all possible context IDs. This is the Call Tree (CT) of the program (see Figure 7a), with unique context IDs associated with each node.



(a) CT. **(b)** Identical paths.

(c) Prefix of another path. **(d)** Paths with common prefix.

Figure 7. Comparing the paths from the root of the CT to the instructions in an observed dependence is necessary in order to decide which instructions to associate with the dependence.

The size of the CT is asymptotically exponential in the number of distinct functions in the program, so it is impossible to construct in an efficient manner. For this reason we implement a lazy tree access procedure, which builds only parts of the tree which are required to compute the nodes for context IDs which have actually been observed during the profiling run. The results are cached, so that the same part of the tree does not need to be computed twice. This puts a limit on the complexity of the analysis and ensures that the number of nodes in the tree that will be explicitly computed will be at most the number of contexts seen during the profiling run, which is less than exponential.

With the CT built, for each dependence in the gathered profile we compute the paths from the root of the CT to the context in which the source and target instructions of the dependence were encountered. By taking the longest common prefix of these two paths we find the appropriate instructions to build the dependence between:

1. Between two memory instructions, if the paths are the same (instructions are part of the same function: Fig. 7b);
2. Between a memory instruction and a call instruction, if one path is a prefix of the other (one instruction happens before a function call in the function in which the other instruction belongs to: Figure 7c);
3. Between two call instructions, if no path is a prefix of the other (the memory instructions happen before different function calls within the same function: Figure 7d).

PDG Construction. Each of the dependences encountered during profiling is added to the PDG built by static analysis, where *may* dependences are treated as absent dependences for the purpose of enabling aggressive, speculative transformation, but are re-inserted as encountered. In our evaluation we discuss the impact of resolving statically determined *may* dependence, which have not been covered by profiling, as either *must* dependence (=conservative lower bound) or *no* dependence (=aggressive upper bound).

4 Evaluation

While iterator recognition is central to driving non-affine loop transformations we evaluate its usefulness by determining its capability of separating loops into iterator code and payload. We prefer this direct measurement of its analytic power over e.g. measuring the impact on a particular loop optimization as this avoids bias introduced by transformation itself or details of the target platform.

Experimental Set-up. We evaluate our methodology for iterator recognition and separation against SPEC CPU2006 application benchmarks. We include all integer and floating-point benchmarks covering codes written in C, C++ and Fortran. In order to limit the time required for profiling we have chosen to use the `test` input data set, as we found out there is no justified improvement from using the `ref` data set instead.

We use an LLVM implementation (version 3.9) of our technique as described in the previous section. For our iterator recognition pass we conduct two experiments: (a) we rely on static analysis only for PDG construction, and (b) we use both static analysis and additional profiling information obtained from running an instrumented version of the benchmarks, where we feed back profiling information to the dependence analysis pass of the LLVM compiler. The host system uses four AMD Opteron 6376 CPUs and has 1TB of RAM available.

For each benchmark we report the total number of loops, the number of loops with affine iterators identified by Polly [20], the number of statically separable loops by our iterator recognition technique, and the number of separable loops using additional profiling information.

Results. We present our main results in Table 1. We show results for > 75.000 loops across all SPEC CPU2006 applications, both integer and floating-point, grouped by programming language (C, C++ and Fortran). For each benchmark we report the total number of loops, the number of affine loops and their percentage of the total number of loops, the number and percentage of statically separable loops using our technique, and the number and percentage of dynamically separable loops. Since profiling with standard data sets does not guarantee that a particular loop is executed, we report a *range*, i.e. a lower and an upper bound, for each benchmark

Table 1. Results for the SPEC CPU2006 benchmarks, grouped by programming language. For each benchmark we provide the total number of loops and how many of them have affine iterators. We compare this to our generalized iterator recognition pass when driven (a) by static dependence analysis and (b) profile-guided dependence analysis indicating lower and upper bounds, respectively. We observe that our novel iterator recognition pass can identify and separate substantially more loop iterators – for either programming language – than what is possible with affine iterator recognition alone. Profiling information always increases the number of separable loops, in particular for the C++ benchmarks.

Benchmark	Loops	Affine Iterators		Statically Separable		Separable After Profiling			
C Benchmarks	#	#	%	#	%	lower bound (#/%)		upper bound (#/%)	
400.perlbench	1,333	364	27.3	1,285	96.4	1,292	96.9	1,297	97.3
401.bzip2	238	106	44.5	191	80.3	235	98.7	238	100
403.gcc	4,617	957	20.7	4,581	99.2	4,581	99.2	4,588	99.4
429.mcf	50	16	32.0	50	100	50	100	50	100
433.milc	426	257	60.3	424	99.5	424	99.5	425	99.8
445.gobmk	1,288	554	43.0	1,272	98.8	1,274	98.9	1,281	99.5
456.hmmer	876	337	38.5	867	99.0	867	99.0	868	99.1
458.sjeng	267	50	18.7	265	99.3	265	99.3	267	100
462.libquantum	98	36	36.7	98	100	98	100	98	100
464.h264ref	1,870	1,268	67.8	1,859	99.4	1,859	99.4	1,869	99.9
470.lbm	23	22	95.7	23	100	23	100	23	100
482.sphinx3	591	151	25.5	582	98.5	587	99.3	587	99.3
Total	Σ(11, 677)	Σ(4, 118)	⌀(35.3)	Σ(11, 495)	⌀(98.4)	Σ(11, 555)	⌀(99.0)	Σ(11, 591)	⌀(99.3)
C++ Benchmarks	#	#	%	#	%	#	%	#	%
444.namd	623	450	72.2	548	88.0	557	89.4	557	89.4
447.dealII	7,323	4,115	56.2	3,999	54.6	4,239	57.9	6,073	82.9
450.soplex	759	360	47.4	453	59.7	532	70.1	649	85.5
453.povray	1,357	438	32.3	975	71.8	1,035	76.3	1,220	89.9
471.omnetpp	481	133	27.7	162	33.7	178	37.0	233	48.4
473.astar	119	42	35.3	82	68.9	107	89.9	110	92.4
483.xalancbmk	3,617	676	18.7	1,907	52.7	2,169	60.0	2,712	75.0
Total	Σ(14, 279)	Σ(6, 214)	⌀(43.5)	Σ(8, 126)	⌀(56.9)	Σ(8, 817)	⌀(61.8)	Σ(11, 554)	⌀(80.9)
Fortran Benchmarks	#	#	%	#	%	#	%	#	%
410.bwaves	85	84	98.8	85	100	85	100	85	100
416.gamess	21,386	19,970	93.4	20,353	95.4	20,386	95.3	20,924	97.8
434.zeusmp	547	533	97.4	534	97.6	537	98.2	537	98.2
435.gromacs (Fortran/C)	2,364	1,717	72.6	2,088	88.3	2,120	89.7	2,273	96.2
436.cactusADM (Fortran/C)	1,903	1,150	60.4	1,523	80.0	1,567	82.3	1,724	90.6
437.leslie3d	405	403	99.5	403	99.5	404	99.8	404	99.8
454.calculix (Fortran/C)	4,610	3,309	71.8	3,877	84.1	3,980	86.3	4,409	95.6
459.GemsFDTD	1,181	1,161	98.3	1,165	98.6	1,174	99.4	1,178	99.7
465.tonto	10,791	10,464	97.0	10,610	98.3	10,636	98.6	10,710	99.2
481.wrf (Fortran/C)	7,739	7,390	95.5	7,568	97.8	7,576	97.9	7,618	98.4
Total	Σ(51, 011)	Σ(46, 181)	⌀(90.5)	Σ(48, 206)	⌀(94.5)	Σ(48, 465)	⌀(95.0)	Σ(49, 862)	⌀(97.7)

for the dynamically separable loops. The lower bound corresponds to cases where *may* dependences in non-profiled loops are conservatively approximated, whereas the upper bounds corresponds to a scheme where such unobserved *may* dependences are resolved aggressively.

Comparison to Affine Loop Iterators. Inspection of the data in Table 1 reveals that affine loop iterators are common in Fortran based programs and account for 90.5% of all loops in these applications, but are less frequently encountered in programs written in C and C++ (35.3% and 43.5%, respectively). However, even for C and C++ applications there exists great variance with individual programs, e.g. `458.sjeng`, `403.gcc` or `483.xalancbmk`, exhibiting only few affine iterators, whereas others including `470.lbm` make frequent use of such iterators.

For most C and Fortran programs almost all loop iterators can be separated from loop "payloads", resulting in 94.5% and 98.4% of statically separable loops. Note that failure to separate iterators is *not* a failure of our technique, but an inherent loop property ("unseparable loop").

For C++ codes we observe fewer separable iterators, although this figure (56.9%) is still well above that for affine iterators (43.5%) and can be improved substantially using profiling information.

Evaluation By Programming Language. Again, consider Table 1. Static iterator recognition works well for Fortran and C programs with 94.5% and 98.4%, respectively, of all loops separable using static analysis alone. This is a slightly surprising result given that the C based SPEC applications tend to be more irregular and pointer based than their Fortran counterparts. However, we find that some C benchmarks such as `401.bzip2` and some mixed Fortran/C codes such as `436.cactusADM` and `454.calculix` have a lower than average number of statically separable loops. For the mixed language this benchmarks these loops are contained in the C parts of the applications, though.

The situation is different for the applications written in C++, where the average percentage of statically separable loops is substantially lower at 56.9% than for C or Fortran based codes. For some applications, e.g. `471.omnetpp` this percentage can be as low as 33.7%, whereas for `444.namd` 88.0% of its loops are statically separable. Lower separability figures for C++ applications can be attributed to following two reasons: (a) the C++ applications comprise fewer separable loops, and (b) C++ applications are harder to analyse statically using the analyses provided in the LLVM compiler.

Some of the non-separable loops in C++ applications are caused by the programming language's specified requirement for zero initialization of arrays, where each element is zero-initialized. It is also known that some of the C++ applications in the SPEC CPU2006 suite contain a large number of non-natural and multi-exit loops [42], which defeat LLVM's static analysis. We have also observed that LLVM's ability to disambiguate accesses to fields and members in structures (and classes) is limited, e.g. in `473.astar`.

Impact of Profiling Information. For cases where static analysis already enables iterator separation for e.g. $> 98\%$ of all loops there is obviously limited scope for improvement (other than minimizing iterator instructions). This is the case for most Fortran and C benchmarks, although there are a number of notable exceptions. While only 191 out of 238 loops are statically separable for `401.bzip2` profiling enables separation of, at least, 235 loops, thus increasing recognition from 80.3% to 98.7%.

For C++ applications with their lower number of separable loops and iterators profiling improves separability by, on average, 5% (lower bound) and up to 24%. For `450.soplex`, for example, static analysis enables separation of 59.7% of all loops, whereas profiling contributes to an increase of greater than 10% and up to almost 26% depending on whether a conservative or aggressive scheme is used.

Coping with Limited Profiling Coverage. Profiling incurs a substantial overhead and it is in the interest of the user to reduce the time for profiling. This can be achieved by using a smaller input data set, e.g. the `test` instead of the `ref` data set for SPEC CPU2006. However, a typical data set often does not fully exercise every code path, i.e. there is limited loop coverage. We found that the `test` input data set for SPEC CPU2006 covered, on average, only 19% of all loops (see Figure 8).

We see that profiling information has a substantial impact on the ability to separate iterators, especially for the C++ applications. These findings suggest that profiling with standard data sets is not ideal, but a more targeted approach supported by techniques from the field of software testing, e.g. automated test case generation [3], should be considered, but this is beyond the scope of this paper.

Evaluation of Iterator Size and Complexity. Consider the three diagrams in Figure 9, where we plot the distribution of relative iterator sizes and their frequency across the SPEC CPU2006 benchmarks, broken down by programming language (C, C++, Fortran). We make two interesting observations: (a) Iterator sizes are not uniformly distributed, and (b) the distribution of iterator sizes varies significantly for the three programming languages used in the benchmark suite.

For the C benchmarks we observe a bimodal distribution of iterator sizes, where iterators are either very small (around 5-10% of loop instructions) or large (around 85% of loop instructions). The situation is different for Fortran codes, where the vast majority of iterators is small. The C++ applications exhibit the same bimodal trait as the C codes, but the peaks at the lower and higher end of the scale are less distinct.

Further inspection of the iterators reveals that small iterators are typically affine or near-affine iterators, which only require a few instructions to update and compare. Larger iterators often comprise control flow.

Analysis and Profiling Overhead. Static analysis needs to consider pairwise all instructions contained in a loop for dependence testing (using LLVM's DependenceAnalysis pass), resulting in $O(n^2)$ complexity. This is an LLVM restriction and a different implementation of LLVM's dependence analysis could improve this algorithmic complexity substantially. However, for most of the SPEC CPU2006 applications analysis is reasonably fast and only adds a few seconds to the overall compilation time. A notable exception is the `416.gamess` application, which with its 2M+ IR instructions and 17.5M+ static dependences spread over 21k+ loops, takes almost 11 minutes to analyse statically. None of the C or C++ benchmarks takes longer than 30 seconds to analyse, though, and most of them can be processed in under 10 seconds.

Profiling naturally incurs a much greater overhead than static analysis. For example, profiling of the SPEC CPU2006 applications using the `test` data set requires several minutes and up to several hours. In our (somewhat naïve) setup, the overhead resulting from instrumented execution of a program yields, on average, a 760× slowdown (compared to 289× of the technique in [26], which exhibits far more conceptual and practical restrictions). This is clearly prohibitive in a continuous edit-compile-test cycle or on very large data sets. However, these are not the envisaged use cases of our profiling technique. Instead, the preferred application as enabler of one-off transformations is inherently more tolerant to the observed profiling overhead in exchange for greater accuracy.

Use in Commutativity Based Parallelization. As an example of its usefulness to enable novel loop transformations we have applied iterator recognition to drive a dynamic commutativity based parallel loop detection method inspired by [41], which also uses machine learning to select profitable loops for OpenMP parallelization [44]. This loop parallelization approach critically relies on iterator recognition for handling of both affine and non-affine loops, possibly comprising traversals of dynamic data structures.

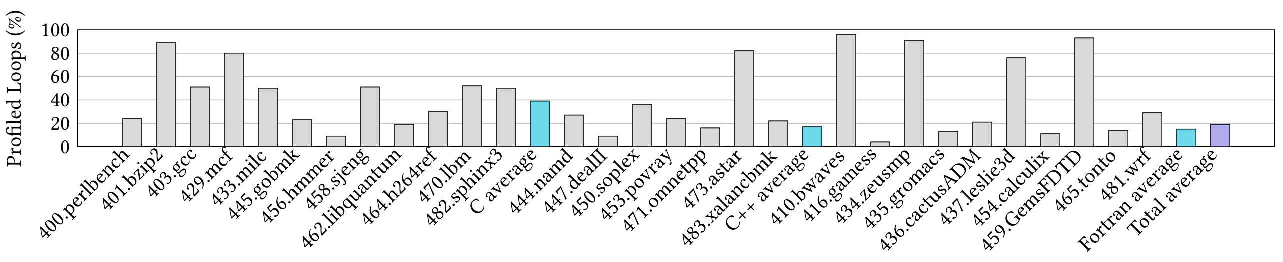
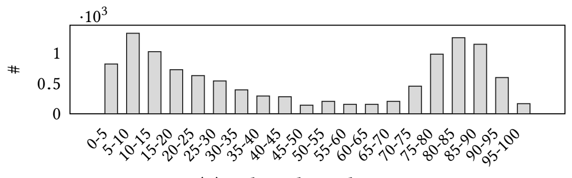
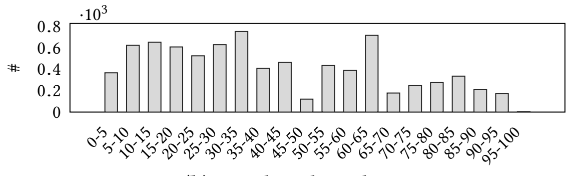


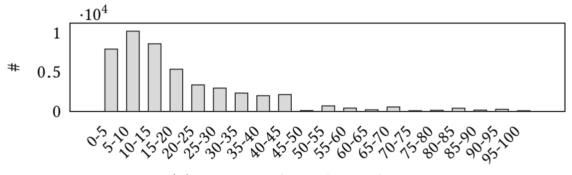
Figure 8. Percentages of loops covered by the SPEC CPU2006 `test` data set.



(a) C benchmarks.



(b) C++ benchmarks.



(c) Fortran benchmarks.

Figure 9. Distribution of iterator sizes as percentages of loop IR instructions (0% = iteratorless, 100% = unseparable).

In Figures 2 and 10 we compare this loop parallelization approach driven by our novel iterator recognition developed in this paper to three competing parallelizing compilers: Intel's ICC compiler (version 17.0) [10], LLVM/Polly [20], and constraint-based parallel idiom recognition [18]. Table 2 shows the number of loops each of the parallelization tools is capable to identify in the sequential C sources of the NAS benchmarks (version 3.3.1) [4]. We also directly compare the union of parallelizable loops, which Intel ICC, LLVM/Polly and parallel idiom recognition *together* can detect with the loops identified by our approach. As shown iterator recognition directly enables commutativity based parallelization, which as a single method combines as much parallelism as Intel ICC, LLVM/Polly and parallel idiom recognition together. Combined with a suitable machine learning based profitability analysis [44] this enables parallel performance levels exceeding those of any of parallelization methodology in isolation (see Figure 10).

5 Related Work

We have discussed existing notions of iterators in Section 2 and provided examples of code transformations enabled

Table 2. Parallel loops discovered by Intel ICC, LLVM/Polly, constraint-based idiom recognition and commutativity analysis enabled by our iterator recognition pass.

	Total	Polly	ICC	Idioms	Union ICC/Polly/Idioms	Comm.-based Parallelization
BT	50	24	40	5	42	42
CG	26	1	16	8	19	19
DC	75	1	23	12	33	33
EP	7	0	3	1	3	3
FT	16	0	3	1	4	4
IS	16	1	4	8	11	11
LU	63	17	40	0	42	42
MG	38	0	18	5	21	21
SP	53	27	42	2	44	44
UA	157	28	115	10	127	127
Total	501	99	304	52	346	346

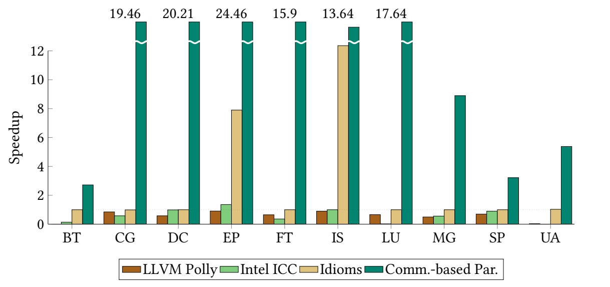


Figure 10. Parallel speedup for NAS 3.3.1 sequential benchmarks resulting from parallelization using (a) Intel ICC, (b) LLVM/Polly, (c) parallel idiom recognition and (d) commutativity analysis powered by iterator recognition.

by iterator recognition in Section 1.1. Loop concepts and iterators are as old as the oldest high-level programming languages [17] and have received particular attention in polyhedral loop analysis and programming language design. Loop identification [38] is the general problem of finding loops in programs. It is often based on Tarjan's interval-finding algorithm and is an essential step in performing various optimizations and transformations, but it is not concerned with identifying loop iterators. Decidability of termination of several variants of simple integer loops, without branching in the loop body and with affine constraints as the loop guard (and possibly a precondition) has been considered in [5]. Whilst this work is related to iterator recognition, it follows decision theoretical approach. Reducible and inrreducible loops are subject of [21]. Efficient symbolic analysis of chains of recurrences supporting induction recognition is presented in [13]. Pointer-based array traversals are analyzed and transformed to closed form array expressions in [15]. Static analysis is employed in [9] to determine loop iteration counts using polytope-based loop evaluation and program slicing. A constraint based approach to recognition of reductions is presented in [18], where a wide class of reductions including their loop iterators is recognized in the LLVM framework. However, generalized iterators as presented in this paper are beyond the scope of their work. In [28] a technique for computing SSA-PDGs and their SCCs is introduced, but no attempt is made to extract their dominating SCC as a loop iterator. Instead, all SCCs are treated equal and merged in the graph to coarsen the granularity of potential parallel regions by applying typed fusion.

HELIX [8, 30] is a speculatively parallelizing compiler, which would benefit from iterator recognition. While HELIX applies parallelizing loop transformations, it relies on normalizable loops (equivalent to `while` loops), but it does not attempt to separate loop iterator code. Instead, HELIX monitors *all* loop carried data dependences without further distinction. In [37] automatic parallelization of loops that iterate over user-defined containers that have interfaces similar to the lists, vectors and sets in the C++ STL is demonstrated. However, this approach relies on the user inserting OpenMP directives into a serial program and, effectively, marking up loop iterators. Partitioning of heap-allocated data structures and transformation of pointer-manipulating programs is a concern for high-level synthesis supporting FPGA design flows. Separation logic is used in [48] for static analysis and transformation enabling parallelization of programs with dynamic data structures and pointer-based memory accesses.

An early framework supporting profile-guided data dependence analysis and subsequent loop parallelization was developed in [44], but it has only been applied to a selection of benchmarks. A more scalable and efficient data dependence profiling methodology was presented in [26], which experiences slowdowns in the same order of magnitude as our technique, but suffers from more restrictions. In [25] another data dependence profiling technique is developed, which reduces slowdowns to around 150×, but does not overlay dependence to a dynamic call graph, but only merges dependences in a static call graph. An online dynamic dependence analysis is shown in [23].

6 Summary and Conclusions

In this paper we have developed a generalized notion of loop iterators, which enables compiler-based iterator recognition. This in itself is an enabling analysis, which supports novel code transformations including loop optimization, parallelization and general loop rewriting. In particular, iterator recognition enables the development of new loop transformations targeting non-affine loops, which in some domains make up the majority of loops. We show that static analysis works well for C and Fortran, but complex C++ code benefits from additional profiling information. We have demonstrated that our approach to iterator recognition works in practice and our LLVM prototype implementation is capable of separating a substantially larger number of loops and iterators than previous techniques based on our evaluation against the full SPEC CPU2006 suite. Future work will focus on integrating our technique with advanced loop parallelization.

References

[1] Breadth first traversal or BFS for a graph. http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/.

[2] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1983.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, Aug. 2013.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks&mdash;summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[5] A. M. Ben-Amram, S. Genaim, and A. N. Masud. On the termination of integer loops. *ACM Trans. Program. Lang. Syst.*, 34(4):16:1–16:24, Dec. 2012.

[6] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] K. Bierhoff. Iterator specification with typestates. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, SAVCBS '06, pages 79–82, New York, NY, USA, 2006. ACM.

[8] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[9] D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *2009 International Symposium on Code Generation and Optimization*, pages 136–146, March 2009.

[10] K. Craft. *Intel C++ Compiler 17.0 Release Notes*. Intel.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[12] R. Dekker and F. Ververs. Abstract data structure recognition. In *Proceedings of the 9th International Conference on Knowledge-Based Software Engineering*, KBSE'94, pages 133–140, Piscataway, NJ, USA, 1994. IEEE Press.

[13] R. v. Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 118–132, London, UK, 2001. Springer-Verlag.

[14] V. Escuder and R. Rico. Reduced input data sets selection for SPEC CPUint2006. Technical Report TR-HPC-02-2009, Department of Computer Engineering, Universidad de Alcalá, Spain, April 2009.

[15] B. Franke and M. O'Boyle. Array recovery and high-level transformations for DSP applications. *ACM Trans. Embed. Comput. Syst.*, 2(2):132–162, May 2003.

[16] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, Jan. 1995.

[17] W. K. Giloi. Konrad Zuse's Plankalkül: The first high-level, "non Von Neumann" programming language. *IEEE Ann. Hist. Comput.*, 19(2):17–24, Apr. 1997.

[18] P. Ginsbach and M. F. P. O'Boyle. Discovery and exploitation of general reductions: A constraint based approach. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 269–280, Piscataway, NJ, USA, 2017. IEEE Press.

[19] G. Goff, K. Kennedy, and C.-W. Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 15–29, New York, NY, USA, 1991. ACM.

[20] T. Grosser, A. Groesslinger, and C. Lengauer. Polly – Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, Dec. 2012.

[21] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, July 1997.

[22] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 121–130, New York, NY, USA, 2010. ACM.

[23] A. Jimborean, P. Clauss, J. M. Martinez, and A. Sukumaran-Rajam. Online dynamic dependence analysis for speculative polyhedral parallelization. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 191–202, Berlin, Heidelberg, 2013. Springer-Verlag.

[24] N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 148–159, Piscataway, NJ, USA, 2017. IEEE Press.

[25] A. Ketterlin and P. Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 437–448, Washington, DC, USA, 2012. IEEE Computer Society.

[26] M. Kim, N. B. Lakshminarayana, H. Kim, and C.-K. Luk. Sd3: An efficient dynamic data-dependence profiling mechanism. *IEEE Trans. Comput.*, 62(12):2516–2530, Dec. 2013.

[27] M. H. Kim. A new iteration mechanism for the C++ programming language. *SIGPLAN Not.*, 30(1):20–26, Jan. 1995.

[28] F. Li, A. Pop, and A. Cohen. Automatic extraction of coarse-grained data-flow threads from imperative programs. *IEEE Micro*, 32(4):19–31, July 2012.

[29] S. P. Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012.

[30] N. Murphy, T. Jones, R. Mullins, and S. Campanoni. Performance implications of transient loop-carried data dependences in automatically parallelized loops. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 23–33, New York, NY, USA, 2016. ACM.

[31] E. M. Nystrom, R. D.-C. Juy, and W.-M. W. Hwuz. Characterization of repeating data access patterns in integer benchmarks. In *Proceedings of the 28th International Symposium on Computer Architecture*, September 2001.

[32] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[33] V. Packirisamy, A. Zhai, W.-C. Hsu, P. C. Yew, and T. F. Ngai. Exploring speculative parallelism in SPEC2006. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 77–88, April 2009.

[34] P. Pirkelbauer, D. Dechev, and B. Stroustrup. Source code rejuvenation is not refactoring. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '10, pages 639–650, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 444–448, New York, NY, USA, 1995. ACM.

[36] W. M. Pottenger. Induction variable substitution and reduction recognition in The Polaris parallelizing compiler. Master's thesis, University of Illinois at Urbana-Champaign, 1995.

[37] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers*, pages 524–538, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[38] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2):175–188, Mar. 1999.

[39] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[40] T. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, Jan. 2000.

[41] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

[42] R. E. Rodrigues, P. Alves, F. Pereira, and L. Gonnord. Real-World Loops are Easy to Predict: A Case Study. In *Workshop on Software Termination (WST'14)*, Vienna, Austria, July 2014.

[43] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[44] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI*, pages 177–187. ACM, 2009.

[45] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[46] T. J. K. E. von Koch and B. Franke. Variability of data dependences and control flow. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 180–189. IEEE Computer Society, 2014.

[47] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 1–17, London, UK, UK, 2000. Springer-Verlag.

[48] F. J. Winterstein, S. R. Bayliss, and G. A. Constantinides. Separation logic for high-level synthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 9(2):10:1–10:23, Dec. 2015.

[49] H. Yu and Z. Li. Fast loop-level data dependence profiling. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 37–46, New York, NY, USA, 2012. ACM.