Iteration Disambiguation for Parallelism Identification in Time-Sliced Applications

Shane Ryoo, Christopher I. Rodrigues, and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{`sryoo, cirodrig, hwu`} `@crhc.uiuc.edu`

Abstract. Media and scientific simulation applications have a large amount of parallelism that can be exploited in contemporary multi-core microprocessors. However, traditional pointer and array analysis techniques often fall short in automatically identifying this parallelism. This is due to the allocation and referencing patterns of time-slicing algorithms, where information flows from one time slice to the next. In these, an object is allocated within a loop and written to, with source data obtained from objects created in previous iterations of the loop. The objects are typically allocated at the same static call site through the same call chain in the call graph, making them indistinguishable by traditional heap-sensitive analysis techniques that use call chains to distinguish heap objects. As a result, the compiler cannot separate the source and destination objects within each time slice of the algorithm. In this work we discuss an analysis that quickly identifies these objects through a partially flow-sensitive technique called iteration disambiguation. This is done through a relatively simple aging mechanism. We show that this analysis can distinguish objects allocated in different time slices across a wide range of benchmark applications within tens of seconds even for complete media applications. We will also discuss the obstacles to automatically identifying the remaining parallelism in studied applications and propose methods to address them.

1 Introduction

The pressure of finding exploitable coarse-grained parallelism has increased with the ubiquity of multi-core processors in contemporary desktop systems. Two domains with high parallelism and continuing demands for performance are media and scientific simulation. These often operate on very regular arrays with relatively simple pointer usage, which imply that compilers may be able to identify the parallelism in these applications. Recent work has shown that contemporary analyses are capable of exposing a large degree of parallelism in media applications written in C [14].

However, there are still obstacles to be overcome in analyzing the pointer behavior of these applications. A significant percentage of these applications are

based on time-sliced algorithms, with information flowing from one time slice to the next. The code of these applications typically consists of a large loop, often with multiple levels of function calls in the loop body, where each iteration corresponds to a time slice. Results from a previous iteration(s) are used for computation in the current iteration. While there are typically dependences between iterations, there is usually an ample amount of coarse-grained parallelism within each iteration, or time slice, of the algorithm. For example, in video encoding applications, there is commonly a dependence between the processing of consecutive video frames, a fundamental time slice of video processing. However, there is an ample amount of parallelism within the video frame, where thousands of sub-pictures can be processed in parallel.

From our experience, time-sliced algorithms typically operate by reading from data objects that are written in the previous time slice, performing substantial computation, and writing to data objects that will be used by the next time slice. The primary step of parallelizing the computation of the time slice lies in the disambiguation between the input and output objects of the time slice. This proves to be a challenging task for memory disambiguation systems today. The difficulty lies in the fact that the code is cyclic in nature. The output objects of an iteration must become the input of the next iteration. That is, the input and output objects are coupled by either a copying action or a pointer swapping operation during the transition from one time slice to the next. Without specialized flow sensitivity, a memory disambiguation system will conclude that the input and output objects cannot be distinguished from each other.

Three different coding styles exist for transitioning output objects to input objects in time-sliced algorithms:

1. **Fixed purpose**: The data objects operated on are allocated in an acyclic portion of the program and designated specifically as input and output structures. At the end of an iteration, the data in the output structure are copied to the input structure for use in the next iteration. Previous parallelism-detection work [14] assumes this coding style.
2. **Swapped buffer**, or double-buffering: Two or more data objects are created in the acyclic portion of the program and pointed to by input(s) and output pointers. At the end of an iteration, the pointer values are rotated.
3. **Iterative allocation**: A new data object is allocated and written during each iteration of the primary program loop and assigned to the output pointer. At the end of the iteration, the output object of the current iteration is assigned to the input pointer in preparation for consumption by the next iteration. Objects that are no longer needed are deallocated.

Many compiler analyses, even some that are flow-sensitive, will see the pointers in the latter two categories as aliasing within the loop. This is true when considering the static code, since the stores in the currently-executing iteration are writing the objects that will be read in the next iteration. When the dynamic stream of instructions is considered, however, the pointers and any stores and loads from them are independent within a single iteration of the loop. In media and simulation applications, this is where much of the extractable loop-level

parallelism lies. The goal of our work is to create a targeted, fast, and scalable analysis that is *cycle-sensitive*, meaning that it can distinguish objects allocated or referenced in cyclic patterns. We will address the third category; the first is adequately disambiguated by traditional points-to analysis and the second can be handled by tracking the independence of the pointers via alias pairs [10] or connection analysis [4].

Figure 1 shows an example of this within a video encoder. At least two related images exist during MPEG-style P-frame encoding: the frame currently being encoded and reconstructed and the frame(s) that was/were previously reconstructed. During motion estimation and compensation, the encoder attempts to achieve a close match to the desired, current image by copying in pieces from the previous reconstructed frame. In terms of memory operations, it reads data from the previous frame and writes it to the current frame. In the reference MPEG-4 encoder [11], these objects can come from the same static call sites and calling contexts and must be disambiguated by determining that they were created in different iterations of a control flow cycle.
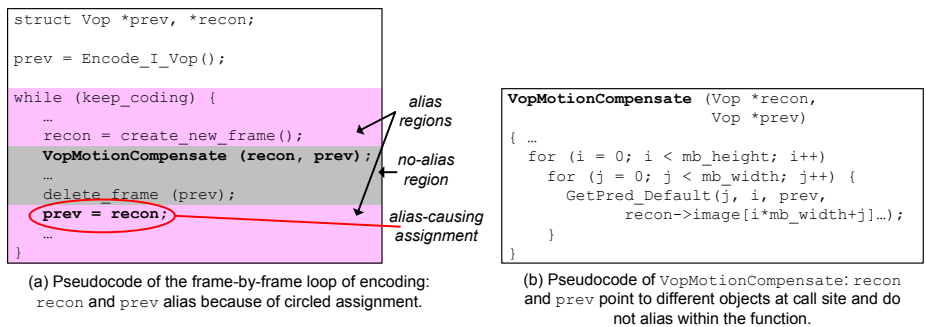

(a) Pseudocode of the frame-by-frame loop of encoding: `recon` and `prev` alias because of circled assignment.

(b) Pseudocode of `VopMotionCompensate`: `recon` and `prev` point to different objects at call site and do not alias within the function.

Fig. 1. An example of cycle-sensitivity enabling parallelism.

In the loop shown in Figure 1(a), the pointers `prev` and `recon` will point to the same object at the circled assignment. Within the middle shaded region, however, the two pointers will always point to different objects, since `recon` is allocated in the current loop iteration and `prev` is not. Thus, loops within this region, such as those within `VopMotionCompensate` as shown in Figure 1(b), can have their iterations executed in parallel. The goal of our analysis, *iteration disambiguation*, is to distinguish the most recently allocated object from older objects that are allocated at the same site. This and similar cases require an interprocedural analysis to be effective because the objects and loops involved nearly always span numerous functions and loop scopes.

We first cover related work in Section 2. We then describe the analysis in Section 3. Section 4 shows analysis results for several benchmark programs. We conclude with some final comments and future work.

2   Related Work

The intent of iteration disambiguation is to quickly distinguish objects that come from the same static call site and chain, but different iterations of a control flow loop. It is designed as an extension of a more general pointer analysis system. By doing this, the analysis is able to capture cases which general analyses are incapable of distinguishing or cannot accomplish in an inexpensive manner. For an overview of previous work in pointer analysis, we refer to [8].

The closest existing work to iteration disambiguation, in terms of disambiguation results, is connection analysis, as proposed by Ghiya and Hendren [4]. It attempts to overcome the limitation of basic points-to analysis and naming schemes by using a storeless model [3] to determine the sets of interacting, or connected, pointers that may alias. At each program point and for each calling context, the analysis maintains the sets of connected local and global pointer variables. Each pointer's connection set approximates the set of other pointers with which it may alias. Connections are removed, or "killed", for a pointer when it is assigned, and replaced with the connection information of the right-hand expression, if any. At a given program point, disjoint connection sets for two pointers indicates that they have not interacted in any way and do not conflict. Interprocedural analysis is handled by exhaustive inlining.

Connection analysis distinguishes new and old objects in a loop by making a newly-allocated object be initially unconnected to anything else. The basic example shown in Figure 1 can be disambiguated by connection analysis because the assignment to the variable `recon` kills `recon`'s connection to previous objects. However, control flow within the loop body can foil connection analysis. When a variable is assigned on only some paths through a loop, its connections are not killed on the other paths. This leads to pointer aliasing of the variable and its connected variables after the paths merge. This case has been observed in video encoders and obscures parallelism in those applications.

Shape analysis is a flow-sensitive analysis which attempts to analyze the pattern of pointer usage to express relationships between different instances of the same abstract heap object, examples of which are [5, 6, 15]. This can improve pointer disambiguation for recursive data structures. Generally, the possible types of structures must be known to the analysis, with the exception of [6]. The purpose of this work is not to identify the relationship between instances of recursive structures, but to disambiguate "top-level" pointers retained in local or global variables that refer to objects created in different iterations of a control flow cycle. Shape analysis generally does not focus on this aspect of pointer behavior.

Wu et al. [16] have developed a may-alias points-to analysis that can distinguish different elements of an array in loops where array elements are known to be accessed by a strictly monotonically increasing index variable in a loop. In their analysis, there can be no interaction between the array elements, whereas the objects targeted by iteration disambiguation are coupled through copying or pointer assignment at the end of loop iterations. Their work is complementary to ours and can be combined with ours to provide greater precision.

3 Analysis

This section describes iteration disambiguation, a dataflow algorithm that distinguishes objects allocated in different iterations of a cycle at compile-time. It does this by marking objects' references with different *ages*. Intuitively, if one considers a loop body as a code segment observed during dynamic execution, the objects created outside of the segment or in previous instances of the segment are distinct from any objects created in the examined segment.

3.1 Example

Figure 2(a) shows the control flow graph for a simple example of an iteration disambiguation opportunity, while Figure 2(b) shows an unrolled version of the loop to clarify the relationship between pointers a and b within each outer loop iteration. We define two memory objects A and B by their static allocation sites. Since object B lies within a loop, its instances are given subscripts to indicate the iteration of the loop in which they were allocated.
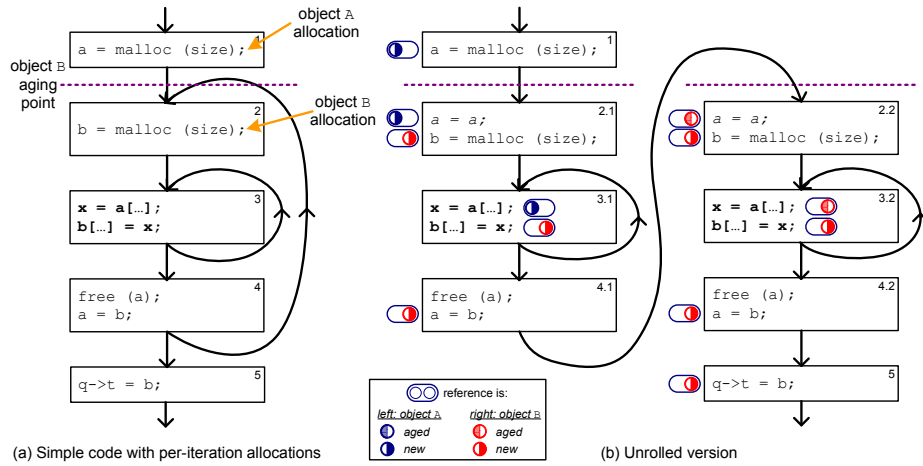

Fig. 2. Iteration disambiguation example.

In the first iteration of the loop in Figure 2(b), pointer a points to object A, while pointer b points to object B_1, created in block 2.1. These two pointers do not alias within block 3.1 but do in block 4.1. There is a similar situation in block 3.2, except that in this case a points to object B_1, created in block 2.1, while b points to object B_2, created in block 2.2. Additional iterations of the loop would be similar to the second iteration. Although b aliases with a within instances of block 4, the two do not alias within any instance of block 3: a points to A or B_{n-1}, while b points to B_n. The compiler can determine that loads are independent from stores within the smaller loop in block 3 and can combine this

with array index analysis to determine that parallel execution is safe for those loop iterations.

Another way of looking at this relationship is that object B_n is a separate points-to object from previously created objects. We call the B_n object *new*, while B_{n-1} and older objects are lumped together as *aged*. Objects become aged at *aging points*, which are ideally just before a new object is created. Between the *aging points* for an object, or between an aging point and the end of the program, two pointers that point to these two objects cannot alias for any n.

3.2 Algorithm

Our algorithm operates on non-address-taken local variables in the program's control flow graph. A pointer analysis, run prior to the algorithm, annotates variables with sets of objects they may point to. Points-to sets are represented as a set of abstract object IDs, where each ID stands for an unknown number of dynamic objects that exist at runtime. Intuitively, the analysis distinguishes `new` references to an abstract object created within the body of a loop from `aged` ones that must have been created prior to entering the loop or within a prior iteration of the loop. References are `ambiguous` when it becomes unclear whether they refer to `aged` or `new` objects. As long as the ages of two references are distinct and unambiguous, they refer to independent dynamic objects within the scope that contains both references and the allocation for that object. The algorithm is described for a single abstract object with a unique allocation site and calling context. When analyzing multiple objects, analyses of separate abstract objects do not interact. The compiler may choose the objects which are potentially profitable; at this time every heap object allocated within a cycle is analyzed.

The analysis uses *aging points* in the control flow graph to delineate the scope over which a reference is considered `new`. A reference becomes `aged` when it crosses an aging point. Aging points are placed at the entry point of each loop and function containing the allocation. Placing aging points at the beginning of loop iterations ensures that all `new` references within the loop are to objects created in the current loop iteration. New references outside the loop point to objects created in the last loop iteration. Aging at function entry points is necessary for recursive function calls.

Recursive functions require additional consideration. A `new` reference becomes `aged` at the entrance of a recursive function that may allocate it, and could be returned from the function. This is effectively traveling backwards across an aging point, creating a situation where the same reference is both `aged` and `new` in the same region. We avoid this error by conservatively marking an `aged` return value of a recursive function `ambiguous` when it is passed to callers which are also in the recursion. [1]

The example in Figure 2 obtained the `aged` reference via a local variable that was live across the back edge of the loop. However, many programs retain

---

[1] If a function could have multiple return values, the same situation can occur without recursion. However, only one return value is allowed in C/C++.

pointers to older objects in lists or other non-local data structures and load them for use. In order to detect these pointers as `aged`, the analysis must determine that the load occurs before any `new` references are stored to non-local memory. Once non-local memory contains a `new` reference, the abstract object is labeled `escaped` until control flow reaches the next aging point. Loads of an `escaped` object return references that are `ambiguous` instead of `aged`. Effectively, non-local memory is an implicit variable that is either ambiguous (`escaped`) or aged (not `escaped`). Escaped reference analysis runs concurrently with propagation of `new` and `aged` reference markings.

Setup. There are several items that need to be performed prior to executing the dataflow algorithm:

1. A heap-sensitive pointer analysis is run to identify dynamically-allocated objects, distinguished by call site and calling context [2], and find which variables may reference the object(s). Figure 3(a) shows a code example with initial flow-insensitive pointer analysis information.
2. SSA notation is constructed for each function in the program, with μ-functions[2] at loop entry points. Although constructing SSA notation prior to pointer analysis can improve the resolution of the input information via partial flow-sensitivity [7], this is not necessary for the algorithm to function correctly.
3. Aging points are marked, for each abstract object, at the entry of each loop or function containing an allocation of the object. This is a bottom-up propagation and visits strongly connected components in the program callgraph only once. In loops, μ-functions that assign references to objects created within the loops are aging points. The input parameters to a function that may create the object are also marked as aging points.
4. A dataflow predicate, representing an age, is initialized for each pointed-to object on each pointer assignment, including SSA's μ- and ϕ-functions, and function parameter. The latter are treated as implicit copy operations during interprocedural propagation. We initialize these predicates as follows:
   - Pointer variables which receive the return address of the allocation call are marked with `new` versions of the references.
   - The destination of μ-functions are marked `aged` for an object if the corresponding loop contains an allocation of the object.
   - Destinations of loads that retrieve a reference from memory are optimistically marked `aged`. Unlike the previous two cases, this initialized value may change during propagation of dataflow.
   - All other pointer assignments are marked `unknown`.

   Figure 3(b) shows the results of SSA construction, marking of aging points, and initialization of dataflow predicates for the example in Figure 3(a).

---

[2] μ-functions were proposed in the Gated Single Assignment notation [1]. Unlike that work, it is not necessary to know which references are propagated from the loop backedges; we use the form simply to mark entries of loops.
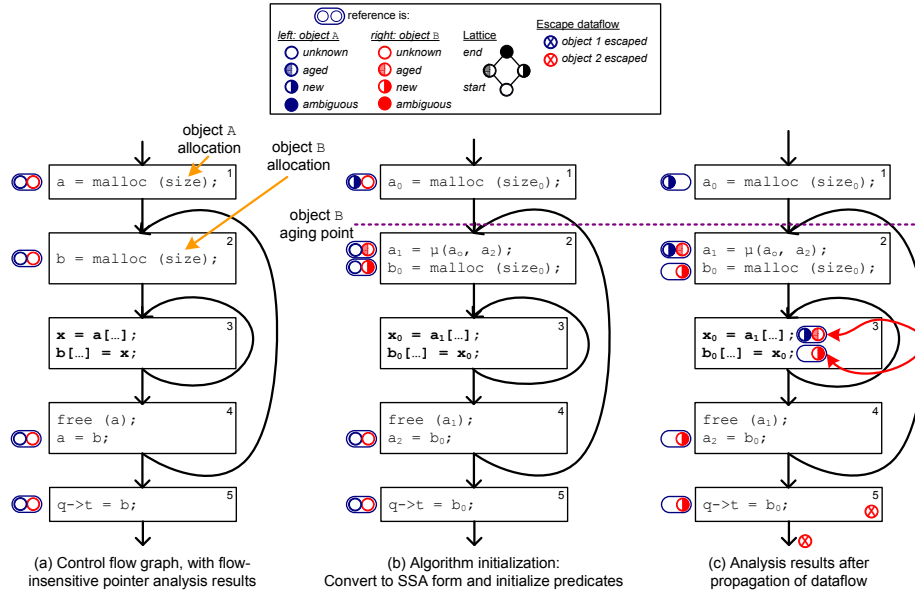
Fig. 3. Iteration disambiguation dataflow.

Propagation. Figure 3(c) shows the results of dataflow propagation for the given example. Conceptually, the algorithm marks a reference returned by an allocation routine as `new`. This dataflow predicate propagates forward through the def-use chain until it reaches an aging point, after which it becomes `aged`. This aged reference is also propagated forward. If `aged` and `new` references for the same object meet at control flow merges other than aging points, the result becomes `ambiguous`. Separate abstract objects do not interact in any way; for example, propagation from a_0 to a_1 remains `new` for object A because there is only an aging point for object B for that transition.

As mentioned previously, we desire that the analysis capture cases where older object references are loaded from non-local memory. For this, the analysis identifies regions of the program where `new` or `ambiguous` references have `escaped` to non-local memory. References loaded from memory outside this region are guaranteed to be `aged`, but those loaded within the region are `ambiguous` because a potentially `new` reference has been placed in memory.

Propagation of age markings for object references and detection of escaped references proceed concurrently. All propagation is done in the forward direction. Age markings propagate via SSA def-use chains, while escape markings propagate along intra- and inter-procedural control flow paths. Age propagation uses a three-stage lattice of values, depicted in the legend in Figure 3. The least value of the lattice is `unknown`. The other lattice values are `aged`, `new`, and `ambiguous`. The join or union of an `aged` and a `new` reference is `ambiguous`, which means that the compiler cannot tell the iteration relationship of the reference relative

to other object references in the loop. The contents of memory are `ambiguous` where an object has `escaped`; elsewhere, memory only contains `aged` references. Age and escape markings increase monotonically over the course of the analysis. The analysis provides a measure of flow-sensitivity if the base pointer analysis did not support it: references which remain `unknown` at analysis termination are not realizable.

Age markings propagate unchanged through assignment and address arithmetic. The union of the ages is taken for references passed in to ϕ-functions or to input parameters of functions that do not contain an allocation call. At aging points first the union of the ages is taken, then `new` or `ambiguous` ages are converted to `aged`. Function return values are handled differently depending on whether the caller and callee lie in a recursive callgraph cycle. For the nonrecursive case, return values are simply propagated to the caller. For the recursive case, `aged` returns are converted to `ambiguous`. This is necessary to preserve correctness; since the call itself is not an aging point but the callee may contain aging points, a `new` reference passed into the recursive call may be returned as `aged` while other references in the caller to the same dynamic object would remain `new`.

Propagation proceeds analogously for escaped markings. The union of escaped markings is taken at control flow merge points. Escaped markings are not propagated past aging points since all references in memory become `aged` at those points. At the return point of a call where the caller and callee lie in a recursive cycle, memory is conservatively marked `escaped`.

Age and escaped reference markings influence one another through load and store instructions. Stores may cause a `new` or `ambiguous` reference to escape past the bounds that can be tracked via SSA. For our implementation, this occurs when a pointer is stored to a heap object, global variable, or local variable which is address-taken. At that store instruction, the analysis sets an `escaped` marking which propagates forward through the program. The region where this escape dataflow predicate is set is called the *escaped region*.

The analysis optimistically assumes during setup that loaded references are `aged`. If a loaded reference is found to be in the escaped region, the analysis must correct the reference's marking to `ambiguous` and propagate that information forward through def-use chains. Conceptually, the compiler cannot determine whether the reference was the most recent allocation or an earlier one, since a `new` reference has been placed in memory. An example of an escaped reference is shown in the bottom block of Figure 3(c). `Aged` references do not escape because the default state of references loaded from memory is `aged`.

Iteration disambiguation preserves context-sensitivity provided by the base pointer analysis. The calling context is encoded for each object. When the analysis propagates object references returned from functions, the contexts are examined and objects with mismatched contexts are filtered out. The analysis also performs filtering to prevent the `escaped` dataflow from propagating to some unrealizable call paths: references can escape from a function only if they were created in that function or passed in as an input parameter. Our implementa-

tion currently is overly conservative for `escaped` dataflow when a reference is an input parameter which doesn't escape on some call paths. Handling this case requires an analysis to determine which input parameters may escape, and has not been prominent in studied applications.

3.3 Properties and Limitations

The iteration disambiguation algorithm explained here is only able to distinguish the object from the current/youngest iteration of a cycle from objects allocated during previous iterations. In other terms, the analysis is k-limited [9] to two ages. The benefit of this is that the analysis is relatively simple and can be formulated as an interprocedural, monotonic dataflow problem. In general only the most recently allocated object is written, while older ones are read-only, so a single age delineation is sufficient to identify parallelism.

The profitability of iteration disambiguation depends on how long a `new` object stays in a local variable and is operated on before escaping. In studied media and simulation applications, `new` references are often created at the top of loop bodies and escape towards the bottom of the loop after significant computation is performed. This exposes the available parallelism within the primary computation loops. However, is not uncommon for a reference to escape immediately on allocation and not be retained in a local variable, which prevents benefit from iteration disambiguation. The common case for this is sub-objects which are linked into an aggregate object, such as separate color planes of an image. Methods for resolving these objects are discussed in the next section.

The presented algorithm's effectiveness is also inversely tied to the distance between the aging points and the allocation of the object, since all objects between the aging locations and the allocation are `aged`. These cases might be disambiguable if the aging point was relocated, but this causes more complexity in utilizing the analysis results.

4 Experiments

This section presents empirical results that show that iteration disambiguation generally takes a small amount of time and can identify the distinction between cyclic objects. We covered two categories of benchmarks. For the first, we chose programs from SPEC CPU 2000 and 2006, excluding those from 2000 that have newer versions or equivalents in 2006, and those that the current version of our compiler cannot complete, notably gcc and Perl. For the second category, we used several applications from MediaBench I as well as a few independent ones. Our intent with the broad selection is to show that the analysis can disambiguate references in application domains other than media and scientific simulation. We use Fulcra [12] as our base pointer analysis.

4.1 Analysis Statistics

Figure 4 shows analysis statistics for the benchmark programs analyzed. The bars represent iteration disambiguation's time to analyze, annotate, and count statistics in seconds, on a 1.8 GHz Athlon 64. The dots connected by lines represent the number of distinct cyclic objects, distinguishable by call site and calling contexts. The number of heap objects is dependent on the degree of cloning (replication per call site and path) [13] performed by the base pointer analysis. For example, the MPEG-4 decoder and encoder applications have a large number of nested allocation functions which create the large number of objects seen in the figure. The "lowest-level" objects tend to be replicated the most, which affects some of our metrics.
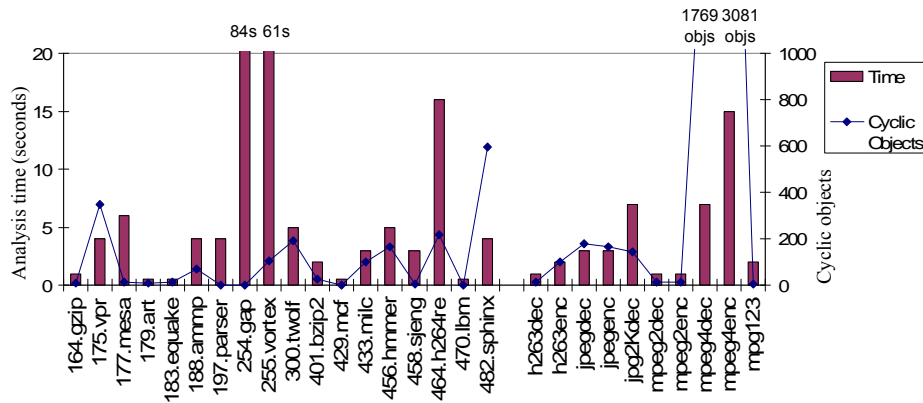

Fig. 4. Iteration disambiguation analysis time and object count.

In general, the analysis is fast and highly scalable. For most programs, which have few cyclic objects, the analysis takes only a few seconds. Even for programs with many cyclic objects, such as 464.h264ref and mpeg4enc, the analysis runs within 16 seconds. The majority of analysis time is spent in the setup phase and the time for propagation of age markings and escaped dataflow is usually insignificant. The primary outliers are two SPEC CPU2000 benchmarks, 254.gap and 255.vortex. These benchmarks are over twice as large as the majority of the benchmarks in the program, with a correspondingly higher setup time. The larger size also increases the amount of code that escaped reference dataflow must propagate through. Finally, they have an unusually high number of references relative to the size of the codes. Unlike the other benchmarks, the time for age propagation and escaped reference dataflow is on the same order as setup time.

4.2 Object Classifications

There are two special object classifications which are exposed by the analysis. First, some objects allocated within cycles are used as temporary storage and are

deallocated within the same iteration. These cases are interesting because they represent privatization opportunies. In iteration disambiguation, these objects are recognizable since only `new` references are used to load data from or store data to them. The percentage of only-new objects is shown in Figure 5. Benchmarks that have no cyclic objects are omitted.
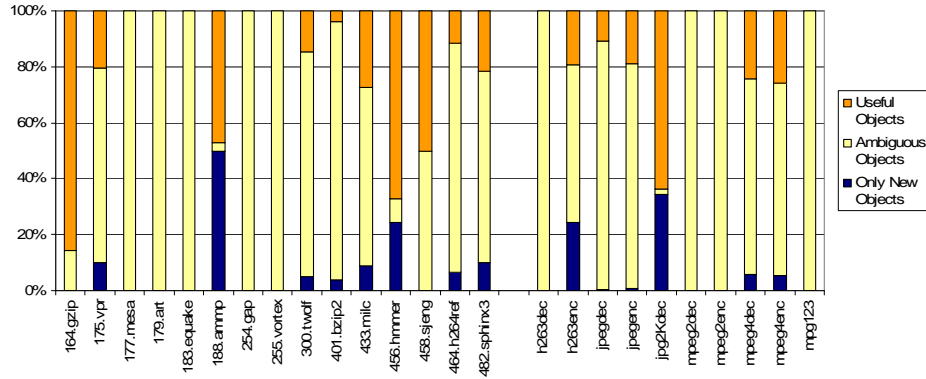

Fig. 5. Iteration disambiguation results: proportion of only new and ambiguous objects.

Second, for some cyclic objects, there are either no `new` markings or no `aged` markings. For these objects, iteration disambiguation has no useful effect. We term these *ambiguous objects*. In programs with inherent parallelism, these objects are commonly multidimensional arrays and sub-structures, which require complementary analyses when detecting parallelism. As mentioned previously, these lower-level objects are a significant portion of the total object count due to heap cloning, and thus increase the apparent number of ambiguous objects beyond a static count of call sites when heap cloning has an effect. Even so, direct inspection of several of the applications has shown that the majority of the heap objects are ambiguous.

4.3 Analysis results

Prior to discussing the analysis results, we break the categories of `aged` and `ambiguous` references into subcategories to gain a better understanding of the results and program properties. They are:

- **Loop Aged**: The reference was passed via a local variable across the backedge of a loop, or entered a recursive function that may allocate the object.
- **Loaded Aged**: The reference's source is loaded from memory in a region where a `new` or `ambiguous` reference has not escaped.
- **Merge Ambiguous**: The age of the reference is ambiguous due to a control flow or procedure call merge in which new and aged references of an object merge via dataflow, such as for conditional allocations.

– **Escape Ambiguous**: The age of the reference is ambiguous because it was obtained via a load in a code region where a `new` or `ambiguous` reference has escaped. We also include aged references returned from a recursive function to callers within the recursion in this category.
– **Combination Ambiguous**: This represents a merge of an escape-ambiguous reference with other types of references.

Figure 6 shows statistics of the results of iteration disambiguation. Results are shown as a percentage of the total static, heap-referencing memory operations, in an assembly-like representation, for each program. When a memory operation may access multiple cyclic objects, an equal fraction is assigned to each object. References to objects that are only `new` have been omitted because they inflate the apparent utility of the analysis. A significant percentage of both `new` and `aged` references indicates likely independence of operations within a loop body. Applications that have no useful cyclic objects have been omitted.

Although a more appropriate test of this analysis would be to show the amount of parallelism exposed by the analysis, we do not attempt this for this work. The objects of interest in many time-sliced applications are children objects of the top-level objects that iteration disambiguation can operate on, and are identified as ambiguous objects. We currently do not have an analysis to prove that children are unique to a parent object, so the amount of extractable parallelism is relatively small. In the future we hope to show application performance difference when the additional analyses are integrated into our framework.
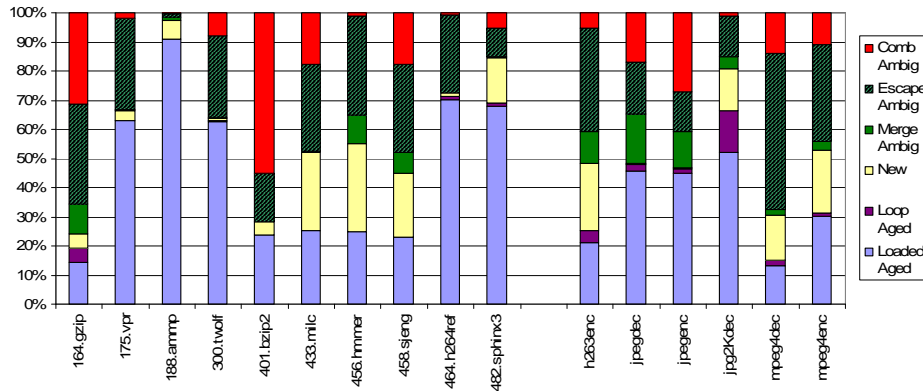

Fig. 6. Iteration disambiguation results: percentages of reference types.

The high percentage of escape and combination ambiguous references indicate that many static operations on heap objects use references that have been stored to and then loaded back from memory prior to an aging point. This is expected for programs that build up large aggregate structures and then operate on them, such as the larger SPEC CPU benchmarks. Despite the fact that

ambiguous objects tend to make up the majority of objects, they do not always dominate the references because of fractional counting per memory operation. We observed a tendency for operations to be one type of reference, because the objects they reference usually have similar usage patterns.

Media programs have a high percentage of ambiguous references because the majority of operations work on data substructures linked to top-level structures. As previously mentioned, escaped references are often multidimensional arrays and can be addressed with the appropriate analysis. Another case that is missed by iteration disambiguation is sub-structures of a cyclic object linked into an aggregate structure. We are currently developing a complementary analysis to address this shortcoming.

One interesting case is 464.h264ref, which is a video encoder application and thus expected to do well with iteration disambiguation. However, it has a smaller percentage of `new` references than most applications. The reason is the common use of an exit-upon-error function, which calls and is called by many of the allocation and free functions used in the application. This creates a large recursion in the call graph, which has the effect of aging `new` references rapidly. In addition, we discovered that half of the loaded `aged` references become escape `ambiguous` if the analysis does not prevent dataflow propagation through unrealizable paths, such as calls to `exit()`.

5   Conclusions and Future Work

This paper discusses iteration disambiguation, an analysis that distinguishes high-level, dynamically-allocated, cyclic objects in programs. This cyclic relationship is common in media and simulation applications, and the appropriate analysis is necessary for automatic detection and extraction of parallelism. We show that we can disambiguate a significant percentage of references in a subset of the presented applications. We also explain some of the reasons why the analysis was not able to disambiguate more references in cases where we would expect a compiler to be able to identify parallelism.

For future work, we will be developing complementary analyses which will enable a compiler to automatically identify parallelism within programs that are amenable to parallel execution. These include array analyses, analyses that identify structure relationships such as trees, and value flow and constraint analyses.

Acknowledgment

References

1. R. Ballance, A. Maccabe, and K. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271, 1990.

2. J. D. Choi, M. G. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

3. A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, April 1992.

4. Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, pages 515–533, August 1995.

5. Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 1–15, 1996.

6. B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

7. R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, June 1998.

8. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

9. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 244 – 256, 1981.

10. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

11. MPEG Industry Forum. http://www.mpegif.org/.

12. E. M. Nystrom. *FULCRA Pointer Analysis Framework*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

13. E. M. Nystrom, H.-S. Kim, and W. W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, June 2004.

14. S. Ryoo, S.-Z. Ueng, C. I. Rodrigues, R. E. Kidd, M. I. Frank, and W. W. Hwu. Automatic discovery of coarse-grained parallelism in media applications. *Transactions on High-Performance Embedded Architectures and Compilers*, 1(1):194–213, 2007.

15. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the ACM Symposium on Programming Languages*, pages 16–31, January 1996.

16. P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proceedings of the 16th International Conference on Supercomputing*, pages 262–273, 2002.