On the Importance of Points-To Analysis and Other Memory Disambiguation Methods For C Programs

Rakesh Ghiya
Intel Corporation
2200 Mission College Blvd
Santa Clara CA, 95052
(408) 765-5807
rakesh.ghiya@intel.com

Daniel Lavery
Intel Corporation
2200 Mission College Blvd
Santa Clara CA, 95052
(408) 765-0884
daniel.m.lavery@intel.com

David Sehr
Intel Corporation
2200 Mission College Blvd
Santa Clara CA, 95052
(408) 765-5372
david.c.sehr@intel.com

ABSTRACT

In this paper, we evaluate the benefits achievable from pointer analysis and other memory disambiguation techniques for C/C++ programs, using the framework of the production compiler for the Intel® Itanium™ processor. Most of the prior work on memory disambiguation has primarily focused on pointer analysis, and either presents only static estimates of the accuracy of the analysis (such as average points-to set size), or provides performance data in the context of certain individual optimizations. In contrast, our study is based on a complete memory disambiguation framework that uses a whole set of techniques including pointer analysis. Further, it presents how various compiler analyses and optimizations interact with the memory disambiguator, evaluates how much they benefit from disambiguation, and measures the eventual impact on the performance of the program. The paper also analyzes the types of disambiguation queries that are typically received by the disambiguator, which disambiguation techniques prove most effective in resolving them, and what type of queries prove difficult to be resolved. The study is based on empirical data collected for the SPEC CINT2000 C/C++ programs, running on the Itanium processor.

1. INTRODUCTION

Pointer analysis has recently been an active topic of research. Its goal is to compute potential targets of pointers in the program, and enable more accurate disambiguation of pointer-based indirect memory references. Recent research has led to the development of efficient pointer analysis techniques that can effectively analyze very large programs in reasonable time [5,6,7]. Most researchers in this area have used the metric of average points-to set size to evaluate the effectiveness of the analysis. While this metric provides a good measure of the static results, it does not reflect the actual benefits achievable from the analysis information in terms of program performance. To evaluate pointer analysis in this context, one requires a framework where points-to information is used by all compiler analyses and optimizations that can benefit from it. Previous work has focused on individual optimizations like parallelization [13], common subexpression elimination [14], and redundant load removal [15] to evaluate the benefits of points-to information. Cheng et. al conducted a more thorough study [16], but for a simulated processor and a different set of programs. A detailed study of the overall benefits of pointer analysis has not been undertaken on real hardware. That is the focus of this paper.

PLDI 2001 6/01 Snowbird, Utah USA
© 2001 ACM ISBN 1-58113-414-2/01/06/…$5.00

Pointer analysis itself is a component of the memory disambiguation framework of an optimizing compiler. A memory disambiguator uses a variety of techniques like symbol table information, address-taken analysis, base-offset calculations, and use-def chains, in addition to pointer analysis. Thus, another interesting data point in evaluating the effectiveness of pointer analysis is how often the disambiguator needs to use points-to information, and how crucial are those disambiguation queries. This would reflect the added benefits of pointer analysis over simpler heuristics and enable a comparison of its contributions versus the cost of the analysis.

The memory disambiguation framework implemented in the Intel Itanium compiler [11,12], provides the required infrastructure for such a study. All compiler analyses and optimizations that need to disambiguate memory references query the disambiguator. The disambiguator, in turn, uses information from pointer analysis, address-taken analysis, array dependence analysis, language semantics and other sources to answer the query. It also provides a mechanism to translate queries for low-level memory references from optimizations like instruction scheduling and software-pipelining into corresponding high-level constructs.

The main contributions of this paper include:

1. An optimizing compiler framework that brings together all the clients and sources of memory disambiguation.
2. A detailed study of the overall benefits of pointer analysis and its eventual impact on program performance.
3. A comprehensive analysis of the effectiveness of various disambiguation techniques, providing insight into which techniques are most often used.
4. A detailed analysis of the cases that prove difficult to disambiguate.

5. Experimental evaluation based on data collected from the industry standard SPEC CINT2000 benchmarks running on the Itanium processor.

The rest of this paper is organized as follows. In section 2, we introduce the overall Itanium compiler framework, presenting the various compiler analyses and optimizations. In section 3, we describe in detail our memory disambiguation framework and how it interfaces with its clients. Section 4 provides details on the pointer analysis implemented in our compiler. We present the empirical data on our study in section 5, along with suitable observations. Section 6 discusses related work, and finally we draw our conclusions in section 7.

2. INTRODUCTION TO THE INTEL® ITANIUM™ COMPILER

The Intel Itanium compiler is designed to extract the full potential of the Itanium architecture [10]. It incorporates a number of leading edge technologies, including profile guidance, multi-file interprocedural analysis and procedure integration, global code scheduling, and a large number of optimizations that make use of speculation and predication.

One of the key goals of the compiler is to eliminate or hide memory latency. One part of this is eliminating as many memory references as possible and taking advantage of the Itanium processor's large register files. Another part is scheduling to hide latency. Register variable promotion and scheduling rely intimately on the best possible memory disambiguation technology. Researchers and compiler writers have developed numerous techniques to prove memory locations independent or non-overlapping. The Itanium compiler incorporates the best-known practical techniques for points-to analysis and data dependence analysis.

Points-to computation is used as an input to disambiguation, but also performs several other functions. It may convert indirect function calls to direct function calls, sharpening the analysis and exposing opportunities for procedure integration. Points-to information can also be used to build the basis sets for MOD/REF analysis, which computes the set of locations modified/referenced by each function in the program. The compiler currently performs limited forms of MOD/REF analysis, such as for standard library functions (e.g., strlen).

Disambiguation and optimizations interact in many ways, so an effective disambiguator needs to incorporate information from a variety of semantic levels of the intermediate language (IL). For instance, generating efficient code for the register indirect addressing requires lowering to base and offset early in the compilation. Doing so naively may make disambiguation more difficult by obscuring such simple facts as two scalar variables can never conflict. Therefore the disambiguator needs to retain "high-level" information about storage locations. Relying solely on high-level information, though, may result in missed information as well. For example, if the program contains pointer arithmetic such as the following fragment, we need lowered addressing and constant propagation to prove that we can registerize s.b across the store whenever i is zero.

```
struct { int a, b; } s;
int *p = &s.a;
s.b = 0;
*(p + i) = 1;
... = s.b;
```

The interprocedural optimizer performs inlining and partial inlining of function bodies into call sites. After either optimization, post-inlining cleanup performs forward substitution of variables and indirect to direct reference conversion. This is particularly important for Fortran and C++ by-reference parameters that can become direct references after inlining. This has implications for disambiguation that will become apparent in later sections.

The clients of the disambiguator are the optimization and code scheduling modules in the compiler, including partial redundancy elimination (PRE) [2], partial dead store elimination (PDSE), dead code elimination, structure copy optimization, the global code scheduler [3], the local scheduler, and the software pipeliner.

PRE uses the disambiguator to determine if a store kills an available load, while PDSE wants to know if a load kills a potentially dead store. Removal of unnecessary loads and stores also enables many other optimizations that operate on temporaries (virtual registers), for example copy propagation or recognition of induction variables. Dead code elimination removes stores to local variables that are never read again for the remainder of the function. The local and global schedulers and the software pipeliner query the disambiguator to determine if a load and store or two stores can be reordered. The software pipeliner requests information about both loop-independent and loop-carried dependences, while the other schedulers query only about loop-independent dependences.

3. THE DISAMBIGUATOR FRAMEWORK

As described earlier, the disambiguator needs to retain high-level information about memory references. Many of the optimizations that rely on memory disambiguation occur in the compiler backend. Typically, after the program representation is lowered and optimizations are performed, much of the source-level information is lost and the code is transformed in ways that make it more difficult for the compiler to perform memory disambiguation. To solve this problem, the Itanium compiler maintains a link from each load or store to a high-level symbolic representation of the memory reference and other information that is crucial for disambiguation. We call our disambiguator *DISAM*, which stands for *DIS*ambiguation using *A*bstract *M*emory locations.

3.1 Abstract Memory Locations

DISAM decomposes the storage space (memory and registers) into a set of abstract storage locations called LOCs. There are different types of LOCs representing global variables, local variables, formal and actual function parameters, functions, registers, and dynamically allocated objects. Each LOC represents a storage object that is independent of all other storage objects. LOCs are part of the symbol table and have links to the symbol table information about the variables that they represent. For dynamically allocated objects, the LOC has a link to the call site in the source where the object was allocated.

LOCs can be grouped together in sets called LOC sets. For example, the set of memory locations that could be accessed through a pointer is represented using a LOC set. References to the same memory objects can be detected by intersecting LOC sets.

3.2 Retention of Source Level Information

Each memory reference is linked to source-level information through a DISAM token, which provides access to all the information necessary to perform memory disambiguation. This information includes a high-level symbolic representation of the memory reference, type information, and a link to an array data dependence graph for disambiguation of different elements of the same array.

For direct memory references, the disam token contains a LOC representing the memory object that is accessed. For indirect references, the token contains a LOC representing the pointer, and a dereference level. The disambiguator must use the results of points-to analysis to determine the set of locations that could be accessed by dereferencing the pointer. While the IL representation of the memory reference and its associated addressing changes greatly as a result of IL lowering and optimizations, it remains linked to its DISAM token, and the LOC representation preserves the high-level source-like representation of the memory reference. DISAM tokens are created early in the compiler when the high-level information is still available.

DISAM tokens are part of the memory referencing IL, and as such are automatically carried along whenever a memory reference is moved or copied. There is a small amount of DISAM token maintenance that must be done. For example, if the compiler creates a memory reference that did not exist in the source program (e.g. stack locations for parameter passing) a token is created to represent the memory reference. As described earlier, forward substitution of address expressions can cause an indirect reference to become direct. The DISAM tokens are updated after forward substitution to reflect this. Overall, DISAM token maintenance is a relatively simple task.

3.3 Performing Memory Disambiguation

Figure 1 shows a block diagram of the various modules involved in memory disambiguation and the interfaces between them. Each arrow is labeled with the type of data structure that is used for the interface. The disambiguator module receives queries from a client, consults the symbol table, array data dependence graph and points-to information if necessary, performs memory disambiguation, and returns a disambiguation result.

The following is an outline of the disambiguation methods employed. A full description of the details of each method is beyond the scope of this paper, but this gives the reader the flavor of what is done and sets up the various categories of disambiguation methods for the experimental results. The disambiguator currently applies the methods in the sequence presented in the subsequent paragraphs. This ordering is predominantly driven by the compile time cost of the query portion of the method. For example, determining if the address of a global variable is taken anywhere in the program requires sophisticated interprocedural analysis. However the cost to lookup this information as an attribute on a variable for each disambiguation query is trivial. In contrast, the cost to intersect points-to sets or to determine the base and offset for an address expression is larger. As soon as definite dependence or independence is determined, disambiguation stops and the result is returned.

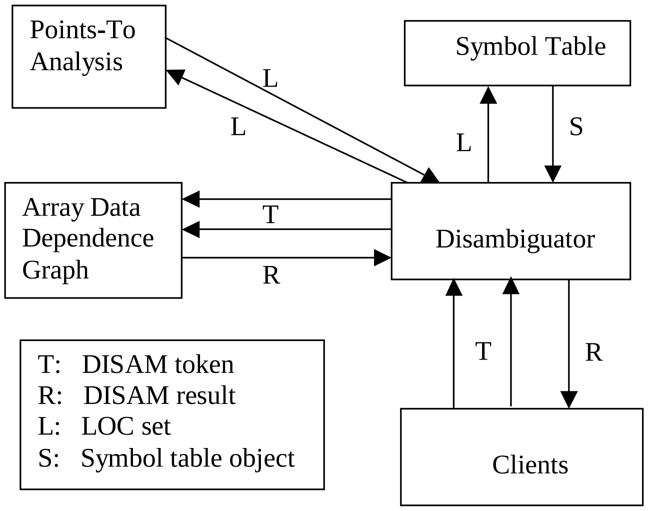

Figure 1 Disambiguation System

Compiler-generated references can often be easily disambiguated from all other memory references. For example, references to read-only storage areas can be disambiguated from all stores. Registers are spilled to a special area on the stack, so these loads and stores can easily be disambiguated from each other and from all memory references that are not spills. The disambiguator can trivially prove the above types of references independent; hence they are not considered in the statistics we present later.

If both memory references are direct (note that direct vs. indirect is easily determined using the dereference level in the DISAM token), their LOCs are compared to determine whether or not the same memory object is accessed. If the same object is accessed, the disambiguator then attempts to determine if overlapping portions of the object are accessed. For example, the array data dependence information is used to determine if the same array element is accessed and structure type information from the symbol table is used to determine if overlapping fields of a structure are accessed.

If at least one of the memory references is indirect, the disambiguator first attempts to prove independence without knowing where the indirect references point to. For example, an indirect reference off an unmodified parameter could not possibly access a stack allocated local variable from the function in which the two references appear. Also, an indirect dereference cannot possibly access a local variable that has not had its address taken. When the compiler is run with interprocedural optimization, it has the ability to automatically detect that it is seeing the whole program. That is it can detect whether or not there are calls to functions that it has not seen and does not know the behavior of. When the compiler can see the whole program, the disambiguator knows that an indirect reference cannot possibly access a global variable that has never had its address taken.

If simple rules such as those above do not allow the disambiguator to prove independence of the memory references, the results of points-to analysis are consulted. The compiler contains two points-to analysis phases: an intraprocedural (local) analysis and a flow-insensitive interprocedural analysis. The disambiguator checks the results of the local points-to analysis first. If that yields a maybe dependent result then the interprocedural points-to analysis is consulted.

With the exception of the compiler-generated references, the disambiguation methods discussed above, all make use of high-level symbol table information and analyses. In cases where the memory references cannot be disambiguated by the above methods, the disambiguator resorts to a method that utilizes the lowered addressing. It analyzes the address expression of each memory reference and tries to determine a base and offset. If successful it compares the base and offset for the two memory references. If they have the same base, the disambiguator can use the offsets and sizes of the memory references to determine whether or not they overlap. This simple base and offset analysis is useful for two memory references off the same pointer (with the pointer having the value at the two references), references whose addresses have been modified by adding offsets or performing pointer arithmetic in an unstructured way, and array accesses with constant subscripts.

Finally, for ANSI C compliant programs the disambiguator can perform type-based disambiguation based on the ANSI type aliasability rules. For example, under the ANSI rules, a reference to an object of type float cannot overlap with a reference to an object of type integer. This method is applied last because it is enabled by a user assertion that the program complies with the ANSI C standard. Applying the other methods first gives the compiler the opportunity to detect potential cases where the user may make the assertion for a program that is not truly compliant. In these cases it can ignore the assertion, perhaps avoiding a runtime failure.

4. POINTS-TO ANALYSIS FRAMEWORK

In our compiler framework, we have implemented the flow-insensitive interprocedural pointer analysis proposed by Andersen[4], which we call WPT (whole-program interprocedural points-to analysis). To counter its cubic time complexity, we have augmented the analysis with off-line variable substitution[5], which pre-computes pointers with identical points-to sets, and online cycle elimination[6], which identifies cycles in the points-to graph and collapses them as the analysis proceeds. All these approaches have been proposed in the literature.

We differ from published work in our handling of structure fields. Standard practice is to collapse the entire structure, and let the structure name represent all its fields. This leads to very imprecise analysis, as points-to sets of distinct fields of a structure can no longer be distinguished. In our approach we distinguish between distinct fields of a structure type, but do not distinguish between its individual instances. For example, consider the following code fragment:

```
        struct foo {int *p; int *q;} s1, s2;
        int x,y ;
        s1.p = &x;
        s2.q = &y;
```

With our approach, we distinguish between fields p and q of structure type foo, but not between its individual instances, s1 and s2. So, the points-to information we collect, indicates both s1.p and s2.p as pointing to x, and s1.q and s2.q to y. With the structure collapsing approach, we would have both s1.p and s1.q pointing to x, and similarly both s2.p and s2.q pointing to y.

In our experience with analyzing large programs, our approach provides substantially more accurate points-to information. This occurs because pointer analyses typically are not able to distinguish between different structure instances, because many structures of a given type tend to be allocated at a given malloc-site. With the structure collapsing approach, we also lose the distinction between fields and the points-to sets of the fields. For example, consider a structure type with one field as a function pointer, and another as integer pointer. If we collapse them, the points-to sets get merged, resulting in very imprecise information. Our approach solves this problem. For correctness in the presence of type-casting, when we encounter a composite structure assignment between different structure types, we analyze it as a sequence of assignments between their corresponding/overlapping fields. Note that in the case of a structure copy assignment of the form *a = *b, we only need to look-up the underlying type signatures of *a and *b, and not their points-to sets, thus reducing the analysis overhead.

Another added feature of our points-to implementation is identification of malloc-like functions. Typically, programmers use a wrapper function, like my_malloc, for dynamically allocating memory via the library call malloc, to check for potential errors and for modularity. This is even more evident in C++ programs, where overloaded new operators are common. Points-to analysis abstracts each static malloc-site in the program as a distinct memory location. With the use of wrapper functions, the analysis sees only one static malloc-site, and is not able to distinguish between memory locations allocated through different calls to this function. To solve this problem, we try to identify if the wrapper function behaves like malloc, in that in each invocation it returns a fresh memory location.

This is achieved by building the SSA (static single assignment) representation of the function, and walking back the use-def chain starting from the return value of the function. If the chain terminates at an unconditional malloc call, we know the wrapper function to be malloc-like. We also check that the wrapper function does not store the address of the allocated memory to any variable/structure field that can be live out of the function. We construct the SSA representation and perform the safety checks in the absence of points-to information, and hence need to be conservative in the presence of indirect references. Still, we are able to identify malloc-like functions in several benchmarks. That substantially improves the accuracy of points-to information and effectiveness of memory disambiguation.

Our final modification to the points-to analysis is that we analyze the assignment statements in the program in a particular order. We build a directed assignment graph of the program, where an assignment *l_loc* = *r_loc*, is represented by adding a directed edge from the node representing *r_loc* to the node representing *l_loc*. Next, we perform a topological sort on the nodes in the graph. The analysis then proceeds by visiting the nodes in the topological order. When a given node representing *x_loc* is being visited, all assignments where *x_loc* appears on the right hand side are analyzed. The analysis continues till all nodes are visited, and iterates until a fixed-point solution is reached. For example, consider the analysis of the following set of assignments: *{(p = q), (q = r), (r = &x)}*. For this set, the topological order of the nodes in the assignment graph is (*&x, r, q, p),* and the assignments will be analyzed in the following order: *{(r = &x), (q = r), (p = q)}*. The analysis of assignments in the original order will require four iterations, as compared to only two using the sorted order.

In the presence of indirect references, we cannot arrive at the best topological order of assignments, but can still improve upon the original sequence. For our benchmark suite, we noticed up to a 50% reduction in the number of iterations with the sorted assignment sequence. Finally, note that off-line variable substitution also requires the topologically sorted assignment graph, so we do not incur any additional overhead for this improvement.

With the above modifications, we were able to implement a very efficient and effective points-to analysis in our compiler framework. We have found that the time spent in points-to analysis is small compared to the overall compilation time.

4.1 Local Points-to Analysis

To achieve better disambiguation in the absence of interprocedural analysis, we have implemented an intraprocedural version of points-to analysis, we term local points-to analysis (LPT). For LPT, we use the same analysis engine as for WPT. Since LPT only sees assignments within the analyzed procedure, we need to make conservative assumptions about the points-to sets of global variables and formal parameters at procedure entry. Additionally we need to conservatively incorporate the effect of function calls on the points-to relationships.

We use the concept of a symbolic location, *nloc,* which stands for non-local location. This is used to represent all locations in the program, excluding the local variables of the analyzed procedure. All global variables and formal parameters of the procedure are initialized to be pointing to *nloc* at the onset of the analysis. This initialization embodies the assumption that at the entry of the analyzed procedure, these variables can point to any location in the program, except the set of locations created after the entry to the given procedure. This set includes all non-static local variables, and dynamic memory allocated inside the procedure, represented by static malloc-sites within the procedure. Additionally, *nloc* is initialized to point to itself, because locations represented by *nloc* can point to each other.

After this initialization, the analysis proceeds in the same iterative fashion on the set of assignments in the procedure as for WPT. On reaching a fixed-point, we mark all local locations accessible from a global variable or an actual parameter via pointer indirection, as address-escaped. The address-escaped locations can be both modified/referenced outside the procedure through function calls. Thus any location visible outside the analyzed procedure can point to a location whose address has escaped and vice versa. To take this into account, we include the symbolic location, *nloc,* in the points-to set of each address-escaped location. Furthermore, *nloc* is considered to additionally represent all address-escaped locations.

Since points-to sets have been updated, LPT is performed again. Identification of address-escaped locations and LPT is iterated until a fixed-point is reached. Typically, it takes fewer than three iterations. LPT is performed on the SSA representation of the procedure. This enables it to achieve limited flow-sensitivity: only for local pointer variables whose address is not taken and are replaced with SSA temporaries. In our framework, LPT is performed even when WPT has been conducted. This is because LPT can sometimes provide sharper information, as it is performed after inlining and SSA construction, providing the benefits of both context- and flow-sensitivity in a limited fashion.

5. EXPERIMENTS

Using the Itanium™ compiler and an Itanium-based system, we have collected data on the twelve C/C++ programs from the SPEC CINT2000 benchmark suite. The Itanium processor [9] contains two integer units, two memory/ALU units, and three branch units. Integer multiplies and divides can be executed on the processor's two floating-point units, adding additional integer throughput for some programs. The processor has a three-level on-package cache hierarchy. The highest optimization levels possible are used to generate the binaries for the experiments in this paper. The SPEC CINT2000 benchmarks are aggressively compiled using the switches that are used for a "peak" SPEC build, including all of the optimizations and analysis described in sections 2-4. The compiler's support for data speculation is turned off for these experiments because the focus of this paper is on traditional compile-time memory disambiguation without hardware data speculation support.

We have instrumented the disambiguator to collect data on the characteristics of the memory references and points-to sets, the number of queries, and the reason for each disambiguation result. Switches have been added to turn on and off the individual disambiguation methods, so that we could see their effect on performance.

5.1 Memory Reference Characteristics

Table 1 shows for each program, the percentage of static memory references that are direct references to local variables, global variables, and indirect references via pointers. The majority of memory references are either accesses to global variables (164.gzip, 186.crafty, 256.bzip2), or indirect pointer references (176.gcc, 181.mcf, 197.parser, 253.perl, and 254.gap). With the exception of 252.eon and 255.vortex, we do not see many references to local variables, as only accesses to local arrays and address-taken local scalars are considered memory references. Other local variables are promoted to registerizable temporaries. The address of a local variable is typically taken to effect call-by-reference semantics. The address-exposure is no longer required if the given call is inlined and forward substitution is applied (section 2). This further reduces the number of address-taken local variables. Forward substitution also eliminates indirect references off the formal parameters in the inlined copy of the callee function. This effect is most pronounced for 252.eon (the only C++ program in the benchmark suite), where indirect references off the C++ "this" pointer in the inlined callee, become direct references in the caller.

Table 1 Program Memory Reference Characteristics

Program	Local %	Global %	Ind %	Avg Set Size	Total Queries
164.gzip	7	84	9	2.4	26118
175.vpr	16	39	45	1.3	40093
176.gcc	8	31	61	22.1	1237456
181.mcf	8	11	80	1.3	10195
186.crafty	4	87	9	3.7	321026
197.parser	7	39	5	6.9	67642
252.eon	27	40	33	147.7	507662
253.perl	6	36	58	427.3	1192815
254.gap	4	22	74	196.3	286053
255.vortex	34	22	44	39.3	405790
256.bzip2	15	67	18	1.00	13544
300.twolf	2	46	52	3.4	443028

The column labeled "Avg Set Size" in Table 1 shows the average size of the points-to sets for the indirect references in each program as determined by our interprocedural points-to analysis (WPT). It is reasonably small for the majority of the benchmarks, with the exception of 252.eon, 253.perl and 254.gap. The loss of accuracy in the first two benchmarks occurs due to the presence of indirect calls, with numerous potential target functions (assigned to an array of function pointers). This forces WPT to consider all possible assignments between the formals of the numerous target functions and the actuals at call-sites, resulting in loss of accuracy. The analysis for 252.eon can be improved by more accurate handling of virtual function calls.

The benchmark 254.gap primarily uses dynamically allocated structures. However, the memory allocating routine for this benchmark, called NewBag, uses free lists and complex pointer arithmetic, and cannot be automatically identified as a malloc function. Thus WPT cannot distinguish between different structures, and provides very inaccurate points-to information. The use of free lists maintained with global pointers would inhibit even a flow- and context-sensitive analysis from identifying that each invocation of the function NewBag returns a disjoint piece of memory. Thus the loss in accuracy is due to poor basis information available to the points-to analysis, and not because of lack of flow- or context-sensitivity.

The last column in Table 1 shows the total number of distinct disambiguation queries for each benchmark. Throughout this paper, each pair of memory references is accounted for only once, irrespective of the number of times the disambiguator is queried with a given pair of references. The filtering out of repeated queries is done using a hash table mechanism. Note that we start accounting for unique memory references after inlining (this is when the DISAM tokens are created), so the memory references in different inlined copies of a given function are considered unique. Compilation of 176.gcc and 253.perl generates the largest number of unique queries, approximately 1.2 million each. Thus the compile time cost for each query is important.

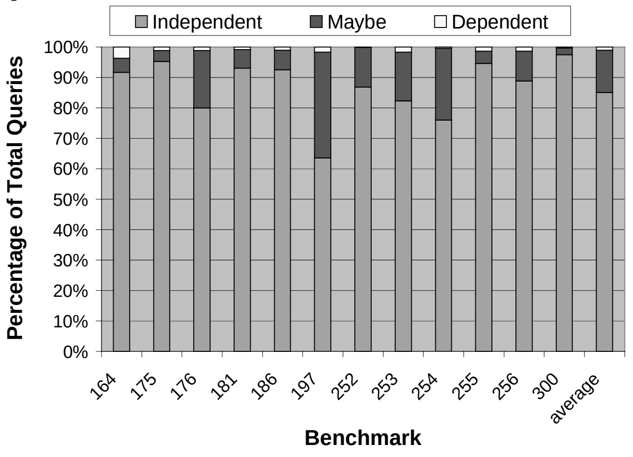

Figure 2 Disambiguation Result Summary

5.2 Analyzing the Disambiguation Queries

We now focus on the characteristics of the disambiguation queries received from various compiler phases, and the effectiveness of our disambiguation techniques in resolving them. Figure 2 shows, for each program, the percentage of unique disambiguation queries for which the disambiguator returns definitely independent, definitely dependent, and maybe dependent. The three categories add up to 100%. Over all the queries received for the same pair of memory references, the best result (definite is better than maybe) is recorded. For example, the simple base and offset analysis is used only after the addresses are lowered. Thus for a pair of references that requires this method to determine independence, the disambiguator would return a maybe result before the address-lowering phase, followed by an independent result thereafter. The definitely dependent results are either determined by the direct method or by simple base and offset. The "average" bar on the chart shows that on average, the disambiguator returns an independent result for more than 85% of the queries, indicating the effectiveness of our suite of disambiguation techniques.

5.2.1 Analyzing the Independent Queries

In this section, we study the relative contribution of different disambiguation methods described in section 3.3, in resolving disambiguation queries. In Figure 3, the portion of each bar representing the definitely independent queries in Figure 2, is expanded and scaled to 100%. For a given bar, each section represents the percentage contribution of a particular method in resolving queries. Note that the percentage basis is the total number of queries proven *independent*, and not the total number of queries received. The accounting is done by applying the methods in a specific order and crediting the first method that determines independence.

For these experiments, the methods are applied in an order that is slightly different from that described in section 3.3. We believe this revised order gives better insight into the real benefits of the disambiguation methods. The methods are ordered according to

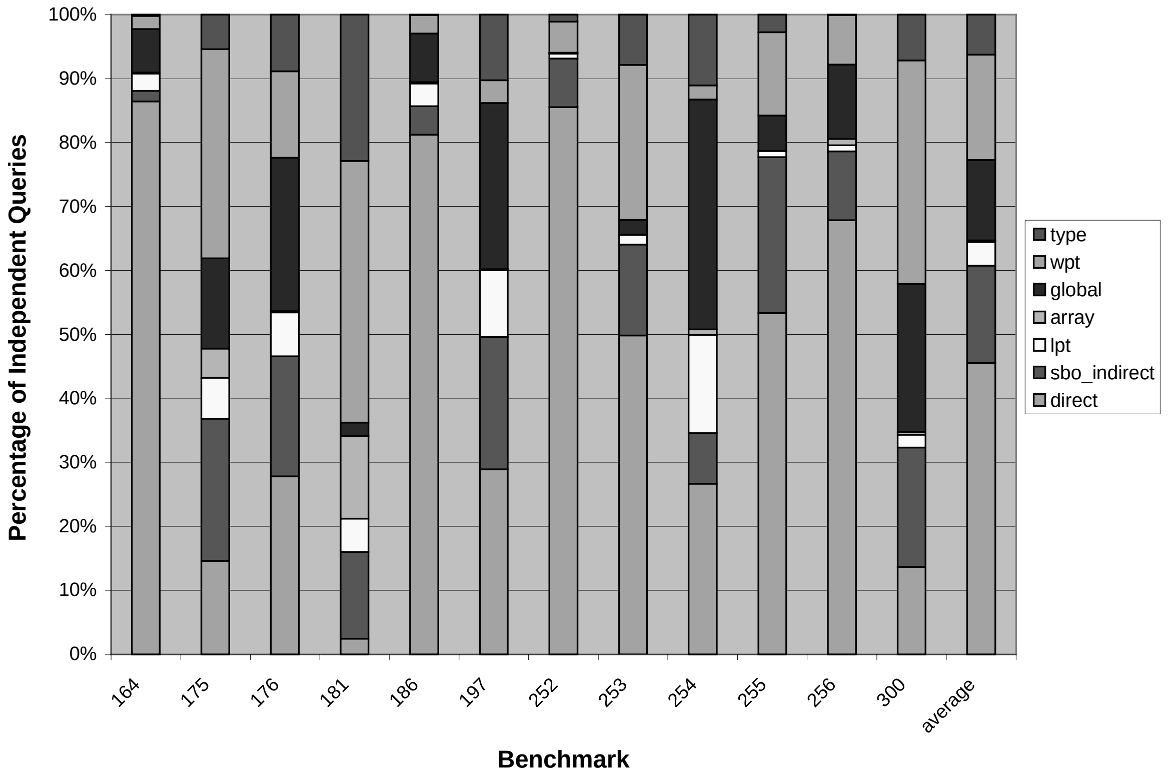

Figure 3 Breakdown of Independent Queries by Method

two criteria. The first is whether the method is used by default in the compiler without interprocedural analysis. This gives insight into the level of disambiguation that can be achieved without user assertions or interprocedural alias/address analysis. The second criterion is the complexity of the method. The rationale here is that the more complex methods should be credited only for queries where that method is really needed. Thus for example, the reader can see the additional benefit provided by interprocedural points-to analysis (WPT) over the simpler methods that would normally be implemented in a compiler before WPT. Type-based disambiguation is the exception here. Even though it is simpler than WPT, it requires a user assertion. We believe it is interesting to see how much benefit the user assertion provides over what the compiler analyses can determine on their own. The order of the methods is as follows:

1. direct: Disambiguation of direct references, either different memory objects or different parts of the same memory object (not including array element analysis).
2. sbo_indirect: Simple base and offset analysis, and simple rules to disambiguate indirect references from direct references.
3. array: Array data dependence analysis to disambiguate different elements of the same array.
4. LPT: Intraprocedural points-to analysis.
5. global: Disambiguation of an indirect reference from a direct reference to a global variable that has not had its address taken.
6. WPT: Interprocedural flow-insensitive points-to analysis.
7. type: type-based disambiguation.

Methods 1-4 are enabled at all optimization levels. Methods 5 and 6 are applied only at the highest levels of optimization when multifile interprocedural analysis is enabled.

We can make several important observations from the data presented in Figure 3. First, simple techniques like direct and sbo_indirect, which do not require sophisticated program analysis, resolve over 60% of the queries determined independent, on average. The direct technique is especially effective for the benchmarks 164.gzip, 186.crafty, 252.eon, and 256.bzip2. This is consistent with the fact that the majority of memory references for these benchmarks fall into the direct reference category (Table 1). For all benchmarks except 186.crafty, nearly 100% of the queries resolved by the direct method are for two references to different memory objects. The preservation of high-level information makes resolution of these queries very easy. The LOCs are simply compared to determine independence. For 186.crafty about 25% of the queries resolved by the direct method are to different fields of the same (statically allocated) structure. Structure type information accessible from the DISAM token is used to determine the position of the field accessed within the structure.

The sbo_indirect disambiguation technique makes consistent contributions across all benchmarks. It is useful in disambiguating accesses to different structure fields with the same base pointer. This case arises frequently in loops traversing a linked data structure (175.vpr, 176.gcc, and 181.mcf), where the loop body contains accesses to different fields with respect to the navigating pointer: for example, the memory references t->item and t->next in the following code fragment.

```
while (t != NULL) {
   t->item = t->item + 1;
   t = t->next;
}
```

Local points-to analysis (LPT) makes visible contributions for 175.vpr, 176.gcc, 197.parser and 254.gap. In the former two benchmarks, several procedures initialize locally declared pointers via explicit calls to malloc (as opposed to via wrapper functions). Indirect references off these pointers are then easily disambiguated against other memory references, using their local points-to sets consisting of the distinct malloc-sites. In 197.parser and 254.gap, local points-to succeeds in disambiguating accesses to local pointers used to navigate arrays, like pointers p and q in the code fragment below:

```
p = &a[i]; q = &b[j];
while (…) {
    *p = *q + 1;
     p++; q++;
}
```

Array dependence analysis (array) does not make a significant contribution to independent queries in the static count for the benchmarks with many arrays like 164.gzip and 256.bzip2. However, it does prove to be crucial in the runtime performance context because disambiguation of a few array-based memory references can enable optimization and scheduling of a critical loop. Array dependence analysis is important statically for 181.mcf. This benchmark contains loops that go through arrays of arcs and nodes. The software pipeliner queries the disambiguator about loop-carried dependences for some of these loops and the array data dependence analysis is able to determine that a new array element is accessed for each iteration.

Address-taken analysis for global variables (global) proves effective across the majority of the benchmarks. In the given benchmark set, global variables are mostly scalars of integer or pointer type, and their address is not typically taken as they can be directly accessed in any section of the program. The technique is less effective for 181.mcf with only 11% of memory references as global accesses, and for 253.perl, which has address-taken attribute set on several global variables.

Interprocedural points-to analysis (WPT) is most effective for the benchmarks 175.vpr, 181.mcf, and 300.twolf. These benchmarks use structures with pointer fields, and the different fields point to dynamically allocated arrays associated with distinct static malloc-sites. The majority of memory references in these benchmarks involve indirect accesses to these disjoint arrays, which can be accurately disambiguated. Both 175.vpr and 300.twolf use wrapper functions for memory allocation. Our analysis is able to identify them as malloc functions, leading to accurate points-to information. Otherwise all pointers would be reported as pointing to a single malloc site, providing almost no disambiguation. Also note that our strategy of distinguishing between distinct fields of a given structure type proves critical to obtaining accurate disambiguation for this benchmark set. The contribution of WPT for 176.gcc and 256.bzip2 is also attributable to identification of dynamically allocated arrays with distinct malloc-sites.

For 252.eon and 254.gap, we have very inaccurate points-to information with average points-to set size over 100, and subsequently little contribution from WPT. However, for 253.perl, even with an average set-size of 427, WPT is responsible for over 20% of independent queries. The majority of these queries involve disambiguation of address-taken global variables against indirect pointer references. On the contrary, for 197.parser, even though the set-size is small (6.92), the WPT-based disambiguation is ineffective because the majority of pointers in the program have the same points-to set. Thus average points-to set size is not always a reliable indicator of the usefulness of points-to information.

Finally, note that simpler techniques like LPT and address-taken analysis for globals (global), steal a significant number of independent queries that would otherwise be attributed to WPT. Thus, disambiguation frameworks that do not use the simpler techniques, may overstate the added benefits of using interprocedural points-to analysis[14, 16].

Type-based disambiguation makes significant contributions for 181.mcf and 254.gap. The two benchmarks require frequent disambiguation of pointers against objects of types long and short, respectively. The technique also pays off for 176.gcc, 197.parser, and 253.perl. Again the contribution comes from queries disambiguating pointers against scalar objects of types integer and float.

Overall, one can notice that each disambiguation technique pays off for one or another benchmark. The technique that proves most effective depends on the memory reference mix of the program, and the type of queries posed by the various optimizations in the compiler. This depends on the regions of the program considered more important by the optimizations based on program structure and profile information. Thus, our strategy of employing a suite of disambiguation techniques is a viable approach to the problem.

5.2.2 *Analyzing the Maybe Dependent Queries*

In this section, we analyze in detail the cases for which our disambiguator reports a maybe dependence. In Figure 4, we highlight the maybe dependent portion of the bars shown in Figure 2. As opposed to Figure 3, each bar in Figure 4 shows the percentage of maybe dependent queries with respect to the *total* number of queries *received* by the disambiguator. On an average, we report 12% of queries as maybe dependent. For 6 out of 12 benchmarks, it is in the range of 5%, indicating very accurate disambiguation. We have over 20% maybe queries for 197.parser and 254.gap, with 176.gcc and 253.perl falling in the 15-20% range. Before presenting a detailed analysis, we first explain the breakdown of data presented in Figure 4.

The top section of each bar in Figure 4 (labeled *direct*), represents the maybe dependent queries which involve two direct memory references (accesses to global or address-taken local variables). The bottom section represents maybe dependent queries involving

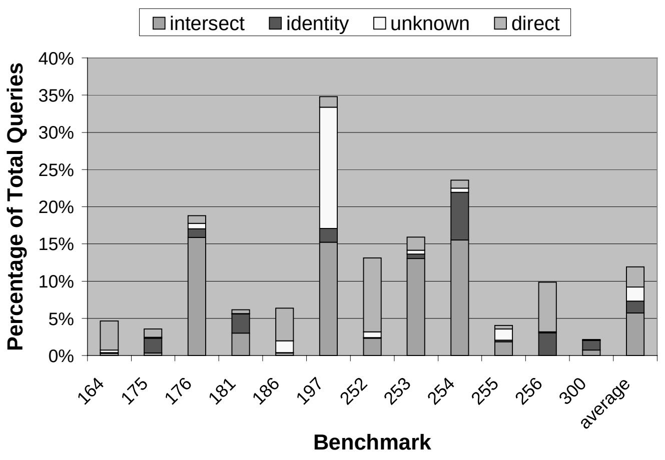

Figure 4 Breakdown of Maybe Queries by Method

at least one indirect memory reference. The maybe cases for indirect references are further subdivided into three categories, *intersect, identity,* and *unknown*, explained below:

- *intersect*: The sets of locations accessible from the two memory references intersect. Since at least one reference is an indirect reference, one of the location sets is obtained from points-to analysis.
- *identity*: Both memory references are indirect references off the same pointer, for example the two indirect references in the statement: {*p = *p + x;}
- *unknown*: The points-to set for one of the dereferenced pointers is not known, and the pointer is conservatively assumed to be able to point to any address-exposed location in the program. This category includes pointers initialized via library calls like file pointers, pointers to i/o buffers, the argv pointer used as a formal argument to function main, and pointers never initialized in the program (dereferences to such pointers are guarded by conditions that always evaluate to false at runtime).

The points-to sets considered above are the ones obtained from WPT analysis. Each category in the above classification pinpoints a specific area of potential improvement for points-to analysis. For example, the *unknown* queries can be better resolved by more accurate modeling of pointers, which are not explicitly initialized in the source program. The requirement is improving the basis information for points-to analysis, and not necessarily its propagation strategy. The *identity* queries can benefit from program point specific points-to information, which requires introduction of flow-sensitivity. Finally, the queries in the *intersect* category may benefit only from a more sophisticated points-to analysis. We now focus on the data for individual benchmarks.

We first consider 197.parser, that reports the highest percentage of maybe dependent queries (35%). The dominant categories are *unknown* and *intersect*. The *unknown* queries arise from dereferences of file pointers and pointers to i/o buffers. Currently our analysis does not accurately model such pointers, and assumes them to be possibly pointing to any address-exposed location, resulting in very conservative disambiguation. Our hand analysis indicates that with more accurate modeling of these pointers (by making them point to specific symbolic locations), the *unknown* queries can either be disambiguated or identified as *identity* queries.

Further, the benchmark 197.parser uses a graph data structure, with all nodes allocated via a wrapper function called xalloc. This function uses free lists and pointer arithmetic, and our analysis cannot recognize it as a malloc function. As a result, queries involving accesses to disjoint nodes of the graph cannot be disambiguated, and fall in the *intersect* category. By explicitly recognizing xalloc as a malloc function (for experimental purposes), we see a drop in *intersect* queries from 15% to 5%. Most of the added disambiguation is achieved in program regions where a new node is allocated via xalloc, and inserted in the graph data structure (all references to the newly allocated node can be disambiguated).

The benchmark, 254.gap, also uses a memory allocator which is based on free lists and pointer arithmetic. All data structures in the program are allocated via calls to this function, and majority of pointers in the program end up having identical points-to sets as explained in section 5.1. This results in over 15% queries falling in the *intersect* set. The queries in the *identity* set arise from our strategy of merging all instances of a given structure field. For example, the memory references *(p->ptr) and *(q->ptr), are both considered as the reference *(ptr) by our disambiguation framework. Such dereferences of structure field pointers frequently occur in 254.gap.

For the benchmark, 176.gcc, the majority of maybe queries arise inside loops traversing linked data structures, like a list of instructions, or a chain of tree nodes. To be able to distinguish between different nodes of such data structures, we need to determine the acyclic property of the navigating pointer fields, which requires sophisticated data structure analysis [18]. However, even advanced data structure analyses need to identify the statements where new heap nodes are allocated. The complex user-defined memory management in this benchmark practically obscures this information from the analysis.

For 253.perl, the points-to information is very inaccurate, and we get over 10% queries in the *intersect* category. In the benchmark 256.bzip2, we have several global pointers, which are initialized only once in the program through malloc, and are used as dynamic arrays. All *identity* queries arise from dereferences of a given global pointer at different program points. Note that since the pointer points to the same location across the entire program, flow-sensitive information will not be able to improve disambiguation.

For other benchmarks, we see noticeable number of maybe queries in the *direct* category. These queries mostly involve array references, subscripted with a pointer reference, a structure field, or another array reference: cases too complex for the array dependence analyzer. Other queries require disambiguation of an array reference outside a loop, with those inside the loop, as illustrated below. Such a query may be posed by the scheduler attempting to move the post-loop array load above the loop.

```
for (i = 0; i < x; i++) {
    a[i + k] = rhs;
}
y = a[m];
```

Another source of *direct* maybe queries we have identified, involves accesses to elements of an array of (statically allocated) structures. For two references of the form a[i].fieldA and a[k].fieldB, our disambiguator returns a maybe dependence. Since we are dealing with static structures, different fields can be safely assumed to access disjoint memory locations irrespective of the array indices. The additional checks required to get this disambiguation, can be easily implemented. For now, majority of the direct maybe queries in 186.crafty and 252.eon are attributable to this reason.

Overall, our maybe analysis indicates that majority of maybe dependent queries arise in programs using dynamic data structures, where the user-defined memory allocating routines cannot be automatically identified by our simple SSA-based recognizer. Development of more sophisticated techniques for malloc function recognition, holds the potential to substantially improve disambiguation for a large number of programs. This is amply demonstrated by the accurate disambiguation achieved for 175.vpr and 300.twolf, and the improvement we get for 197.parser on explicitly recognizing the malloc function. Alternatively, the user can be asked to provide this information via an assertion.

5.3 Performance Analysis

We now focus on the impact of our memory disambiguation techniques on program performance. Figure 5 shows the percentage speedup obtained by successively enabling the disambiguation methods. The 0% speedup line represents the base performance with no memory disambiguation. The speedup for each method is computed by dividing the execution time for the base case by the execution time with that method (and all the methods to the left) enabled.

Basic disambiguation of direct references is very important for 164.gzip and 252.eon, consistent with the static data presented in Figure 3. Disambiguation of direct references is not very important for 256.bzip2 performance, despite the importance of that method in the static data. The frequently executed loops in 256.bzip2 do not contain direct reads and writes of different memory objects. Likewise, direct reference disambiguation is not as important for 186.crafty performance as the static data would indicate.

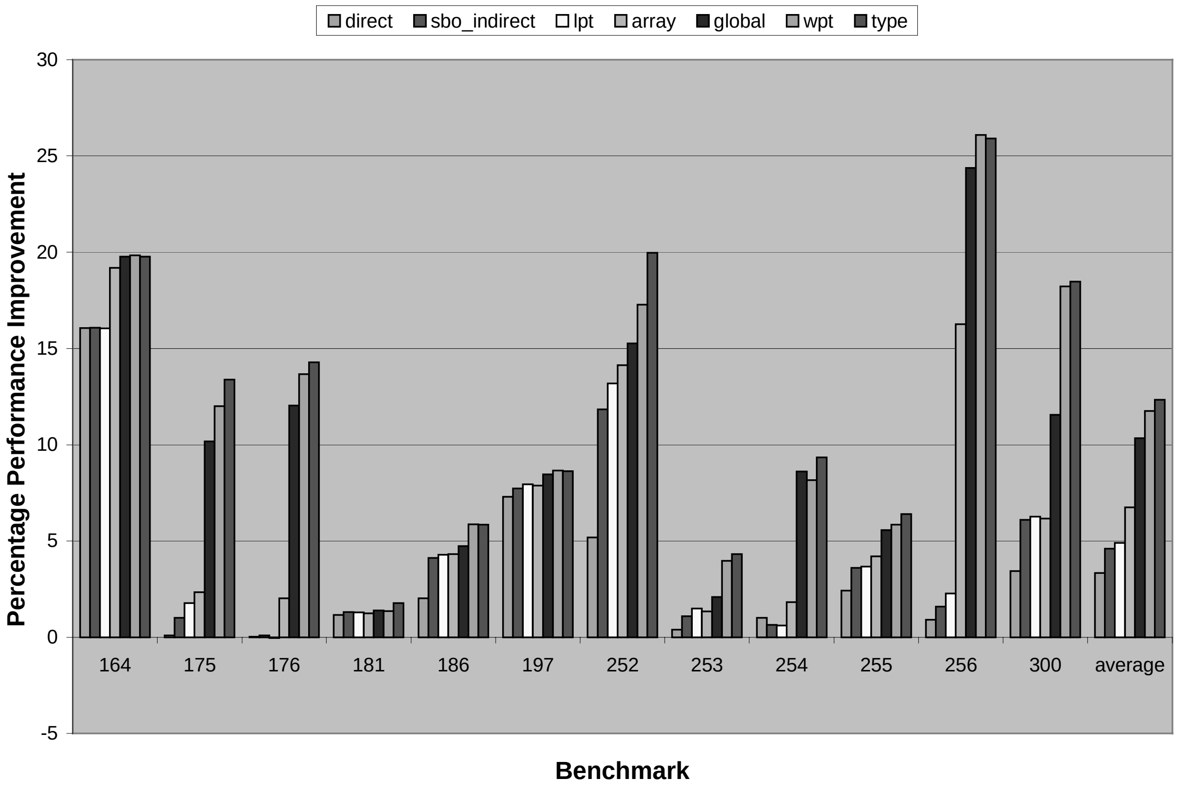

Figure 5 Performance Improvement from each Disambiguation Method

The next method turned on is the use of simple base and offset analysis, and simple rules for indirect references. The simple rules for indirect references pay off for 252.eon. There are many queries with a direct reference to a local automatic variable whose address is taken, but the indirect reference is off an unmodified parameter. In 255.vortex, there are similar cases. In 175.vpr and 186.crafty, the benefit for this method comes from rules for indirect references, as well as from simple base and offset analysis. For 300.twolf, the performance improvement is mostly from simple base and offset analysis.

Local points-to analysis is able to provide small performance gains for 175.vpr, 252.eon, and 256.bzip2. Array data dependence analysis proves to be very important for the performance of 164.gzip and 256.bzip2, both of which include references to different elements of the same array in the frequently executed loops. However, static contribution from this analysis is very small for these benchmarks (Figure 3). This makes sense because the loops containing these memory references contribute very little to the static number of memory references, but contribute greatly to the dynamic number of memory references and to the execution time.

Disambiguation of indirect references from direct references to globals that have not had their address taken helps many programs including 175.vpr, 176.gcc, 254.gap, 256.bzip2, and 300.twolf. 175.vpr, 176.gcc, and 300.twolf all have a healthy mix of indirect references and direct references to globals. These benchmarks also show large percentage of static disambiguation queries resolved by this technique. The frequently executed loops in 256.bzip2 contain indirect writes to an array and direct loads of global scalar variables for the loop bound and array pointer. This method allows these scalar references to be hoisted out of the loop by PRE. This method steals much of the thunder from interprocedural points-to analysis (WPT). Without this method, WPT would normally be required to disambiguate these memory references. Thus it is important to evaluate the performance contribution of WPT in the context of other disambiguation techniques.

WPT provides performance gains for 175.vpr, 176.gcc, 252.eon, 253.perl, 256.bzip2, and especially 300.twolf, consistent with the static query data in Figure 3. The benchmark 181.mcf spends most of its time accessing memory, so despite the fact that a very large fraction of the static queries are resolved by WPT, there is no performance gain.

The final method considered is type-based disambiguation. It provides small gains for 175.vpr, 176.gcc, 254.gap and a larger gain for 252.eon. In 252.eon, the type-based technique proves to be very effective for the routine mrSurfaceList::viewingHit, which is one of the most frequently executed functions in the benchmark.

The astute reader will notice a few cases where increasing the level of disambiguation actually hurts performance slightly, particularly in 254.gap. It is difficult to pinpoint the slight performance loss to a particular optimization. The loss is possibly caused by a complex interaction between different optimizations and requires further investigation.

6. RELATED WORK

In [1], Novack et. al. present a method for preserving algorithm-level and source-level semantics information. Their method defines a hierarchical decomposition of the address space using implicit assertions that reflect the programming language as implemented by the compiler and explicit programmer assertions that reflect the algorithm and the programmer's use of the language. Using this hierarchy, the disambiguator can distinguish between stack and heap, different types, and different variables.

The DISAM approach also preserves source level semantics and decomposes the address space, but it provides access to a richer set of information, including full symbol table information about variables, pointers, types, functions, and memory allocation sites. In our framework, we use the symbolic representation of memory references as a way to interface with analyses such as points-to analysis that are performed while the program is represented at a high level. We have also extended this framework for function call MOD/REF analysis. Our implementation of this technique in a product compiler demonstrates that it is possible with very reasonable cost to maintain high-level information for use by the disambiguator at any compilation phase.

The memory disambiguator in the IMPACT compiler is described in [8]. This compiler performs array data dependence analysis and points-to analysis [16]. It generates memory dependence arcs, called ***sync arcs***, to represent all the memory dependences in the function and maintains these arcs through all of the optimization phases. This requires that potentially $O(N^2)$ arcs be stored in the worst case where N is the number of memory references in the function. In practice the pairs of references that actually have a control flow path between them (and thus require an arc) is much smaller.

In our method, we maintain information about each memory reference (O(N)) plus a data dependence graph containing only arcs for dependences between different elements of the same array in loops. We must store a points-to graph whereas the IMPACT compiler can discard the points-to graph once the sync arcs have been generated. The relative memory usage of the two methods depends on the relative sizes of the information stored in the points-to graph and DISAM tokens verses the sync arcs. We believe that the DISAM approach has a memory usage advantage at lower optimization levels where points-to information is not available, as the number of conservative sync arcs will be higher due to the absence of accurate information. In the DISAM approach, on average, each query is relatively expensive compared to looking up a sync arc. While the compile-time cost of the queries has not been an issue thus far, the results of the most time-consuming queries could be cached to reduce the cost. The total number of queries received by the disambiguator is much larger than the number of unique queries.

The related work on using the results of pointer analysis includes that of Wilson and Lam[13], Ghiya and Hendren[14], and Diwan et. al[15]. Wilson and Lam use points-to information for parallelization of two benchmarks. The speedup is achieved through better disambiguation between pointer-based arrays. Ghiya and Hendren study the benefits of a collection of pointer analyses in the context of three optimizations: loop-invariant removal, location-invariant removal, and common subexpression elimination. In our compiler framework, all three optimizations are subsumed within the PRE and PDSE optimizations. Diwan et. al present the results on redundant load elimination (RLE) for a set of Modula programs, based on information from a type-based alias analysis. Our PRE optimization also subsumes RLE. Pioli and Hind[17] present a thorough study of the efficiency and precision of six context-insensitive pointer analyses, using the metric of average points-to set size at indirect references. Our experimental data indicates that this metric is not always a reliable indicator of the effectiveness of points-to information in resolving disambiguation queries.

Cheng et. al[16] perform a very thorough study of the impact of memory disambiguation on the SPEC CINT92 and CINT95 programs in the context of redundant load/store elimination, loop-invariant location promotion, and load/store scheduling. The memory optimizations are subsumed by our PRE and PDSE transformations, and we also include acyclic scheduling, software pipelining, and other optimizations in our study. Cheng et. al individually show the benefits of disambiguation for scheduling vs. load/store optimization. Their performance improvements are higher than those reported in this paper. Their study uses a different set of programs and a wider simulated processor. Our base case includes modeling of side effects for library function calls and disambiguation of compiler generated references such as spill code. Their base case does not use any disambiguation or side-effect analysis.

7. CONCLUSION

In this paper, we have described a memory disambiguation framework that brings together all the clients and sources of memory disambiguation. We have evaluated the framework using standard benchmarks on an Itanium™-based system. The experimental results show that a broad range of disambiguation methods is necessary to handle the varying characteristics of different programs and provide the highest overall performance. The results also show that it is important to evaluate the performance of additional memory disambiguation techniques such as points-to analysis within a hierarchical framework that implements the simpler disambiguation methods, because these methods steal some of the thunder of the more complex methods. Further, we demonstrated that there is no direct correlation between the effectiveness of memory disambiguation as per the static metrics, and its contribution to overall program performance. This also applies to individual disambiguation methods, when considered in isolation. Finally, we noted that in many cases, loss in accuracy of points-to information occurs due to certain features inherent to the analyzed program, like arbitrary type-casting and user-managed memory allocation, which cannot always be overcome by even applying more sophisticated points-to analyses.

8. ACKNOWLEDGMENTS

We thank the members of the Intel Itanium™ compiler team for their contributions to the compiler technology described in this paper, and the reviewers, whose comments significantly improved the quality of this paper.

9. REFERENCES

[1] Novak, S., Hummel, J., and Nicolau, A. A Simple Mechanism for Improving the Accuracy and Efficiency of Instruction-Level Disambiguation. 8th International Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag, August 1995, 289-303.

[2] Chow, F., Chan, S., Kennedy, R., Liu, S., Lo, R., and Tu, P. A New Algorithm for Partial Redundancy Elimination Based on SSA form. Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997, 273-286.

[3] Bharadwaj, J., Menezes, K., and McKinsey, C. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, 1999, 262-271.

[4] Andersen, L. O. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19)

[5] Rountev, A., and Chandra, S. Off-line Variable Substitution for Scaling Points-to Analysis. Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2000, 47-56.

[6] Fahndrich, M., Foster, J., Su, Z., and Aiken. A. Partial Online Cycle Elimination in Inclusion Constraint Graphs. Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1998, 85-96.

[7] Das, M. Unification-based Pointer Analysis with Directional Assignments. Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2000, 35-46.

[8] Hwu, W., Hank, R., Gallagher, D., Mahlke, S., Lavery, D., Haab, G., Gyllenhaal, J., and August, D. Compiler Technology for Future Microprocessors, Proceedings of the IEEE, December 1995, pp. 1625-1640.

[9] Sharangpani, H., and Arora, K. Itanium Processor Micro-architecture. IEEE Micro, Vol 20, No 5, Sept 2000, 24-43.

[10] Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H., and Zahir, R. Introducing the IA-64 Architecture. IEEE Micro, Vol 20, No 5, Sept/Oct 2000, 12-23.

[11] Bharadwaj, J., Chen, W., Chuang, W., Hoflehner, G., Menezes, K., Muthukumar, K., and Pierce, J. The Intel IA-64 Compiler Code Generator. IEEE Micro, Vol 20, No 5, Sept/Oct 2000, 44-53.

[12] Krishnaiyer, R., Kulkarni, D., Lavery, D., Li, W., Lim, C., Ng, J., and Sehr, D. An Advanced Optimizer for the IA-64 Architecture. IEEE Micro, Vol 20, No 6, Nov 2000, 60-68.

[13] Wilson, R. P., and Lam, M. S. Efficient Context-Sensitive Pointer Analysis for C Programs. Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1995, 1-12.

[14] Ghiya, R., and Hendren, L. Putting Pointer Analysis to Work. Proceedings of ACM SIGPLAN/SIGACT Conference on Principles of Programming Languages, Jan 1998, 121-133.

[15] Diwan, A., McKinley K., and Moss, J. Type-based Alias Analysis. Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1998, 106-117.

[16] Cheng, B., and Hwu, W., Modular Interprocedural Pointer Analysis using Access Paths: Design, Implementation, and Evaluation. Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2000, 57-69.

[17] Pioli, A., and Hind, M. Evaluating the Effectiveness of Pointer Alias Analyses. Science of Computer Programming, Vol. 39 (1) (2001), 31-55.

[18] Ghiya, R., and Hendren, L. Is it a Tree, DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C. Proceedings of ACM SIGPLAN/SIGACT Conference on Principles of Programming Languages, Jan 1996, 1-15.