A Simple Mechanism for Improving the Accuracy and Efficiency of Instruction-level Disambiguation*

Steven Novack, Joseph Hummel, and Alexandru Nicolau

Department of Information and Computer Science
University of California, Irvine, CA 92717

Abstract. Compilers typically treat source-level and instruction-level issues as independent phases of compilation so that much of the information that might be available to source-level systems of transformations about the semantics of the high-level language and its implementation, as well as the algorithm in some cases, is generally "lost in the translation", making it unavailable to instruction-level systems of transformations. While this separation of concerns reduces the complexity of the compiler and facilitates porting the compiler to different source/target combinations, it also forces the costly recomputation at the instruction-level of much of the information that was already available to the higher level, and in many cases introduces spurious dependencies that can not be eliminated by instruction-level analysis alone.
In this paper we present a simple mechanism for retaining algorithm and source language semantics information for use in instruction-level disambiguation, while retaining the ease of porting the compiler to different source/target combinations, and without noticeably increasing the complexity of the compiler. Simulation results show that our mechanism provides a significant increase in both performance, resulting from more accurate higher level data dependence information, and efficiency, as instruction-level dependence analysis no longer needs to recompute information that was already available at the higher level.

1 Introduction

In this paper we present a mechanism for using algorithm-level and source-level (high-level language) semantics information to simplify and improve memory reference disambiguation (anti-aliasing) at the instruction-level, thus enabling improved instruction-level parallelization by removing spurious data dependencies[3]. This mechanism is motivated by the observation that ambiguities that constrain instruction-level parallelization often result, in practice, from the loss of important semantics information when translating a program from higher to lower levels of abstraction: when the programmer translates an abstract algorithm into high-level source code, any information about algorithm-level semantics that can not be expressed directly in the source language are typically lost, and then, when the compiler "front-end" translates the source code into an instruction level (e.g. 3-addr[1]) intermediate representation, there is usually further loss of algorithm-level, as well as source-level semantics information that can not be expressed directly within the instruction-level representation. For instance, algorithm level semantics may imply that two different pointer variables always refer to disjoint memory spaces; the definition of a source language may state that separately defined arrays within the same scope use disjoint pieces of memory; a particular source

* To appear in Languages and Compilers for Parallel Computing, Springer-Verlag LNCS Series, 1995. This work was supported in part by ONR grant N00014-93-1-1348.

language implementation may imply that different kinds of variables are allocated on the heap, while others are allocated on the stack; and a programmer may specify that variables of certain data types may never share memory with variables of certain other data types, even though the source language itself allows for it. While in principle these kinds of higher level semantics information could be passed through from higher to lower levels, in practice, very little, if any, of this kind of information is available for use in instruction level disambiguation. In fact, the typical approach to compilation is to completely separate front-end (algorithm-/source-level) and back-end (instruction-level) issues in order to manage the complexity of the compilation problem and to facilitate the porting of the compiler to different source/target combinations. The unfortunate consequence of such a complete separation is that *all* memory references at the instruction level that represent accesses to variables specified in the source, as well as those introduced by the compiler for temporaries, spill code, and passing parameters on the stack become potentially ambiguous at the instruction level. Thus the instruction-level parallelization engine is forced to make overly conservative assumptions regarding memory reference interdependencies or must resort to potentially very expensive analysis techniques to determine these interdependencies — a redundant and inherently incomplete exercise since in many cases we are at best re-computing dependencies that were already known at the higher level, and at worst we are forced to make overly conservative decisions anyway as some information can not be determined via instruction-level analysis, but still might have been available at the algorithm or source level.

For example, consider a typical 3-addr representation for a LOAD, "(LOAD $r1 $r2 1234)", which copies into register r1 the value stored at the memory location addressed by the sum of the contents of register r2 and the constant 1234. Without considering the context in which this LOAD exists we must assume that it could access any location in memory, since r2 might contain any value. Instruction-level disambiguation techniques typically attempt to remove this ambiguity by deriving symbolic expressions (e.g. by folding up along def-use chains leading from r2) that represent the possible locations that each LOAD or STORE might access, usually as a function of one or more irreducible variables (e.g. loop induction variables, indirections through memory, or program inputs), and then estimating solutions to systems of these equations in order to determine whether memory accesses sometimes, always, or never interfere with one another. More sophisticated symbolic analysis techniques like [8] might then be applied to further refine disambiguation. Techniques based on symbolic analysis can be fairly accurate when dealing with memory references whose access expressions are expressed in terms of the same unknowns, or can be reduced to unknowns with a known relationship (e.g. for references to different parts of the same array); however, when the expressions are too complicated for the particular technique or are expressed in terms of different unknowns with complex relationships, the instruction level disambiguator must conservatively assume that the references conflict. Whether the instruction-level disambiguator correctly determines the dependence or not, the time spent trying to disambiguate the reference is completely wasted if the dependence had been known at the algorithm or source level; if in addition, the disambiguator conservatively reported a conflict when none actually existed at the higher level, then the degree of parallelism, as well as the efficiency of parallelization, are unnecessarily, and sometimes severely, diminished.

While the need for using algorithm-level and/or source-level semantics information for coarse grain transformations and parallelization at the source level has been well established [14, 23, 2, 12, 11, 9], little attention has focused on retaining higher level semantics for use in instruction-level transformations and parallelization. For instance, source-level dependence analysis and programmer assertions regarding pointer interdependencies are commonly used to increase the degree of coarse-grain parallelism by enabling various loop transformations like loop interchange and strip mining [23]; however, by the time source-level data structures have been mapped to memory locations and source-level references have been converted to sequences of machine instructions for accessing that memory, the correspondence between the actual memory LOAD and STORE operations and the source-level references that they represent, as well as the dependence information that related the different source-level references is lost.

Our goal in this paper is to provide a mechanism for retaining high-level semantics for use in instruction level parallelization that both improves and simplifies memory reference disambiguation, without noticeably increasing the complexity of compilation or sacrificing the ease of porting the compiler to different source/target combinations. The mechanism we propose is based on defining a hierarchical decomposition of the address space of the program wherein all instruction level memory references (e.g. LOADs and STOREs) are associated with one or more objects in the hierarchy. References associated with disjoint objects in the hierarchy are considered independent (and references to different parts of the same object may still be determined to be independent via traditional disambiguation techniques). The hierarchical decomposition is defined by explicit and implicit assertions that embody higher level semantics: algorithm-level semantics, as explicitly specified by an analysis tool or a programmer, and source-level semantics, as specified implicitly by the definition of the source language and its implementation in a particular context, and explicitly by certain restrictions on the use of the language enforced by the programmer (including those that represent algorithm-level semantics). For example, one approach to creating the hierarchy that has worked for us when using C as the source language is to begin with default assertions provided by the compiler writer about the semantics of C and its implementation by a particular compiler (e.g., GCC), and then to refine the hierarchy using assertions about programmer enforced restrictions on the use of C and algorithm level semantics provided by an applications programmer and/or high-level analysis tool.[2] These assertions are made initially about each source-level alias, which may be an actual source-code variable or a temporary introduced by the compiler, and then are inherited by any instruction-level memory references generated to access the aliased data object.

For example, Figure 1 shows how a semantics retention hierarchy might be used to parallelize a piece of C code at the instruction level for an idealized VLIW architecture with 3 homogeneous, unicycle functional units. In this example, the arrays A, B, and C are auto variables and are therefore allocated on the stack and N is a pointer into heap memory. The 3-addr statements labelled 1-4 are the instruction-level representation of the first line of C code in the body of the loop, 5-8 represent the second line, and 9-10 represent the last line (loop control code is omitted here for conciseness). The semantics hierarchy shown was initially created based on assertions about the source-code that represent the abovementioned facts: A, B, and C are variables allocated on the stack, and whatever N points to is allocated on the heap. The 3-addr statement

[2] Depending on the language, algorithm, and user sophistication, some or all of this information may not be supplied. We are not advocating the use of the techniques proposed in this paper by all users and for all codes, but rather we are merely trying to show that implementation and use of this approach are not too hard and can have very significant benefits if used appropriately.

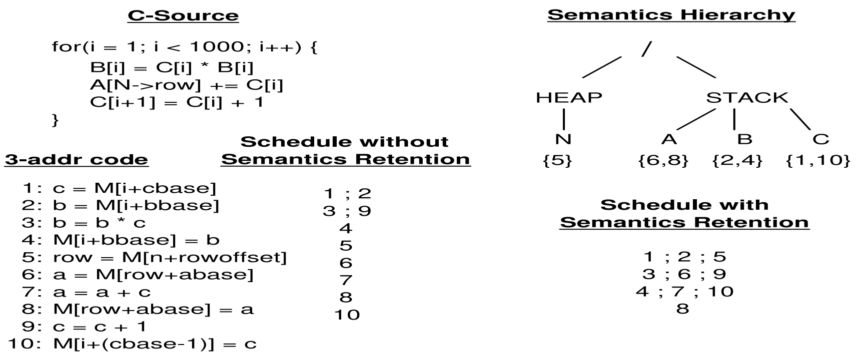

Fig. 1. Using a simple semantics retention hierarchy.

5 inherits the associations of N, statements 6 and 8 inherit those of A, statements 2 and 4 inherit those of B, and statements 1 and 10 inherit those of C. Ignoring the facts represented by the semantics hierarchy, the LOAD on line 5 must be assumed to conflict with all STOREs, and the STORE on line 8 must be assumed to conflict with all STOREs and LOADs.[3] The best schedule that can be obtained for the target architecture given these ambiguities, labelled "Schedule without" in the figure, is twice as long as the schedule achieved ("Schedule with") when the scheduler is able to examine the semantics hierarchy for conflicts.

Note that our use of assertions differs significantly from the typical "statement-based" approach in which assertions describe explicit dependence relations (especially independence) between individual source-level statements or variables.[4] Statement-based assertions have two main drawbacks. The first is the interspersing of numerous, point-specific assertions throughout the program, often resulting in sloppy and potentially error-prone code. The second is that, whether at the source-level or the instruction-level, the point-specific dependence information described by statement-based assertions generally needs to be updated when the code is changed by optimizing and parallelizing program transformations. One of the main advantages of our approach is that, rather than provide "point-specific" assertions that describe dependencies between individual statements or variables, our assertions describe, at a high level, how a program's data is structured, without any reference to the specific statements used to create, modify, or access this structure. Thus, our assertions are valid across most code optimizations and transformations (i.e., as long as the semantics of the data structures remain the same), without the costly overhead of updating asserted dependencies in response to changes in the code.

The remainder of this paper is organized as follows. Section 2 relates our work to other techniques that, to one extent or another, make use of higher level semantics during instruction-level parallelization. In Section 3 we describe our simple semantics retention mechanism that could easily be integrated into most compilers and we illus-

---

[3] Note that better symbolic analysis will not help — the problem is not with `i`, but rather is related to not knowing the values of `N` or `N->row`.

[4] Given the appropriate mechanism, statement-based assertions could also be retained at the instruction-level as explicit dependence assertions between individual 3-addr operations; however, in practice, they are typically used exclusively at the source-level.

trate the importance of using semantics retention in instruction level parallelization, not only to increase the degree of parallelism but also to decrease the overall cost of disambiguation. Finally in Section 4 we provide a brief outline of how semantics retention is integrated within a parallelizing compiler being developed at UCI and provide some results indicating its effectiveness.

2 Related Work

Compilers have generally taken one of three approaches when attacking the problem of memory reference disambiguation at the instruction-level. The most obvious (and conservative) approach is for the compiler to do nothing and simply assume that all memory references potentially conflict. The compiler may disambiguate a few simple references by checking for some special cases (e.g. a load of $SP + 4$ followed by a store into $SP + 8$), but applies no general technique of memory disambiguation. The *gcc* compiler falls into this category, which is unfortunate given its wide use both as a C compiler and as a back-end for other high-level languages. For instance, the GNAT project, which developed an Ada 9x compiler using *gcc*'s back-end is both an excellent motivator and candidate for semantics retention as proposed here, given its use of *gcc* as a back-end while the front-end is dealing with Ada 9x, a language with a stricter type system, parameter specifications, and pure functions.

A second approach is for the compiler to make all source-level information directly available at the instruction-level. In particular, the data structures created by the compiler during parsing and source-level analysis—which maintain, for example, def-use and alias information—are accessible from the instruction-level. This is typically done by having each instruction point back to the statement from which it originated, thereby providing access to the relevant source-level information. The advantage of course is that the instruction-level now has access to more accurate program information, thus enabling more accurate memory disambiguation. However, the disadvantages are two-fold: (1) the separation of concerns between the front-end and back-end of the compiler has been severely compromised, and (2) code movement at the instruction-level may force the costly recomputation of information at the source-level. For example, sophisticated techniques are available for computing alias information in the presence of pointers [16, 6]. Collecting such alias information can be expensive, and the resulting information is *program-point* specific. Yet the movement of a single instruction may alter not only its alias information but also that of other instructions, thereby requiring alias recomputation.

The third and last approach is the one taken by most optimizing compilers. In this case the compiler performs at least some intraprocedural disambiguation analysis at the instruction-level, but algorithm and source-level dependence information, if present, is discarded during the translation from source-level to instruction-level. The advantage of this approach is that the back-end is fairly easy to build and maintain due to its total separation from the front-end, but also retains some ability to perform memory disambiguation. The disadvantage is that the back-end may be repeating analysis previously performed at a higher level. Worse, some information simply cannot be deduced at the instruction level (or at least is too expensive to compute), such as information about types, parameters, and the algorithm.

Our approach attempts to retain the three separate advantages of each of the above-mentioned approaches — namely efficiency, separation of concerns, and retention of source-level (and algorithm level) information for increased accuracy — without incurring the disadvantages of any. We achieve this by allowing the front-end to communicate higher level, memory reference information to the back-end by associating each reference with a portion of a hierarchical decomposition of the program's address space. This information is not program-point specific, can be efficiently tested to detect potential memory conflicts, and is sufficiently concise and high-level to ensure a reasonable separation of concerns during compiler development. The information being communicated is similar in spirit to that collected by Emami et. al. [6] (and later extended by [7]), which is used to build a memory hierarchy based on pointers into the STACK and HEAP. Our approach however is not program-point specific, and goes a step further in supporting a general framework for all memory references, scalars and pointers alike. A language-based approach of similar vein was also proposed by Lucassen in [17], however, in this case the hierarchy is completely user-defined and at the granularity of entire data structures.

3 Semantics Retention

Semantics retention refers to retaining semantics from higher to lower levels of program abstraction. From the highest level of abstraction, the algorithm-level, we would think of retaining that information that is either known or can be inferred from the algorithm, but that can not typically be expressed directly in the next lower level of abstraction, the source-level. For instance, in C there is no way to specify that two pointer variables always refer to disjoint pieces of memory, however, this fact may be obvious, or determinable via sophisticated analysis engines, at the algorithm-level. Similarly, from the source-level of abstraction, we would think of retaining information at the instruction-level that is known or can be inferred about the source language, from its definition, from its implementation, or even from programmer enforced restrictions on its use (including those restrictions that represent algorithm-level semantics). For instance, in C, auto variables are typically allocated on the stack, while static variables and memory allocated by malloc are allocated on the heap, so any pointer that is known to refer to the heap is known to be independent of any auto variable; however, little if any of this information is typically expressed within the framework of an instruction level representation of the program. For instance, with a flat, unsegmented memory space, there is no reason other than the definition and/or implementation of the source language to assume that stack and heap space are disjoint. Even when it is known that the heap and stack are disjoint, instruction level representations rarely distinguish between LOADs/STOREs to the heap versus LOADs/STOREs to the stack; moreover, even when this distinction can be inferred through analysis, doing so is redundant (and potentially very time consuming) since the information was readily available at a higher level of abstraction.

In this section we will describe a simple mechanism for retaining algorithm-level and source-level semantics for use in instruction-level parallelization. This description consists of two parts. First, we will define the hierarchical decomposition of the program's address space, the levels of which will ultimately be associated with the potentially ambiguous instruction level memory references of the program, allowing for many of the ambiguities to be resolved based on the properties of the hierarchical decomposition, without the potentially very expensive analysis that might otherwise be required. Then, we will describe how algorithm-level and source-level semantics are used to guide the decomposition and to bind memory references to the objects in the decomposition.

3.1 Hierarchical Decomposition: Definition

The semantics retention mechanism is based on a hierarchical decomposition of the program's address space in which successively lower layers in the hierarchy represent a more and more refined partitioning of the program's address space. Within this framework, references to data objects that were known to be independent at the algorithm-level or source-level are associated with disjoint objects within the hierarchy. Given this information it is often possible to disambiguate instruction-level LOADs and STOREs simply by examining their associated objects within the hierarchical decomposition — when the objects are disjoint, the references are known to be independent. Thus, the hierarchy allows the instruction-level to benefit from much of the same sorts of dependence information that is often available to source-level systems of transformations, and vastly decreases the overall cost of disambiguation by allowing powerful, but often expensive, instruction-level analysis techniques to be employed only when strictly necessary (i.e. for references to possibly different parts of the same object). More formally, a hierarchical decomposition of the address space of a program is a tree of nodes, wherein each node N is associated with a set S_N having the following properties:

- $S_I \subseteq S_N$ for each successor I of N
- reference $\mathrm{A} \in S_N \Rightarrow \mathrm{B} \in S_N \ \forall\ \mathrm{B}$ s.t. $\mathrm{SPACE}(\mathrm{A}) \bigcap \mathrm{SPACE}(\mathrm{B}) \neq \emptyset$.
- if N is the root of the tree, then reference $\mathrm{A} \in S_N \ \forall\ \mathrm{A}$.

where a reference is either an explicit reference for a variable declared in the source or for a temporary introduced by the compiler, and $\mathrm{SPACE}(\mathrm{A})$ is the set of all addresses that A might access. Note that for any reference A, $\mathrm{SPACE}(\mathrm{A})$ never actually needs to be known, but rather, only whether or not it might intersect with $\mathrm{SPACE}(\mathrm{B})$ for some other reference B, which can always be answered conservatively in the affirmative, but in most cases can be answered far more precisely according to the available algorithm-level and source-level semantics.

Figure 1 shows an example decomposition for a piece of code. Notice that since membership at a leaf node implies membership at all nodes leading back from the leaf to the root of the hierarchy, we only show the S_{LEAF} sets associated with each leaf node LEAF.

It follows directly from the definition of the hierarchical decomposition that for any two memory references, A and B, A is independent of B if there exists a node N in the hierarchy such that $\mathrm{A} \in S_N$ and $\mathrm{B} \notin S_N$. In other words, even though A and B may refer to the same object higher in the hierarchy, at a sufficiently low level, the references are to disjoint objects in the hierarchy. It remains now to show how such a hierarchy can be constructed that embodies a useful amount of algorithm-level and source-level semantics of the program.

3.2 Hierarchical Decomposition: Instantiation

Any technique that satisfies the definition of the hierarchical decomposition as defined above can be used to decompose the address space. In this section we will describe one such technique that begins with a general structuring of the hierarchy that embodies the semantics of the C language as implemented within the GCC compiler, plus a few common restrictions on the usage of C that a programmer would likely adhere to,[5] and then ends with a discussion of how a high level dependence analysis tool is used to refine this hierarchy to include algorithm-level semantics specific to a particular program.

[5] Of course, the relevant refinements to the hierarchy would only be used when the programmer does indeed adhere to these restrictions.

Ultimately we want the hierarchical decomposition to associate instruction level memory references with different objects in the hierarchy, based on source-level assertions about algorithm-level and source-level semantics; however, at the source-level, instruction-level memory references are not yet known. To get around this apparent contradiction, we divide the creation of the hierarchy into two phases. During the first phase, the hierarchy is built in its entirety based on source-level assertions about *source-level* aliases and the data objects they represent, and then in the second phase, while generating the instruction-level representation of the program, each memory reference created to access the data aliased by the source-level alias X is simply added to each level in the hierarchy that X belongs to (i.e. the memory reference inherits the associations of X). For the purposes of this paper we define an alias to be any source-level variable, temporary introduced by the compiler, or a component of an aggregate variable (e.g. a field of a `struct` in C).

The first phase of decomposition proceeds as follows. We start with a trivial hierarchy consisting only of a root node containing all source-level aliases (i.e. all aliases in the program are contained within S_{ROOT}). Then, using a simple assertion mechanism we refine this initial hierarchy by adding new nodes and/or new aliases to existing nodes to represent newly acquired higher level semantics information about the program. These assertions take the form: ASSERT(ALIAS, PATH) where ALIAS is a source-level alias and PATH has the form: $/\mathrm{NODE}_1/\mathrm{NODE}_2/\ldots/\mathrm{NODE}_M$, $M \geq 1$ and represents a path of nodes from the root, denoted "/", to the node NODE_M. The function of ASSERT is to traverse the hierarchy, following in order the nodes on PATH while adding ALIAS to each node visited; when visiting NODE_I, if NODE_{I+1} is not a successor of NODE_I, then NODE_{I+1} is created and made a successor of NODE_I. In this fashion, a single ASSERT can completely specify one way in which an alias fits within the hierarchy. The entities making the assertions, such as a compiler writer, application programmer, and/or high-level dependence-analysis tool would be responsible for ensuring that the resulting hierarchical decomposition is correct with respect to the definition given in Section 3.1.

One approach to creating the hierarchy that has worked for us when using C as the source language[6] is to begin with default assertions provided by a compiler writer about the semantics of C and its implementation by a particular compiler (e.g. GCC), and then to refine the hierarchy using assertions about programmer enforced restrictions on the use of C and algorithm-level semantics provided by an application programmer and/or high-level analysis tool.

Figure 2 shows a basic structure for the semantics hierarchy based on a few trivial but fundamental facts about the C language as implemented by GCC (the other information in the figure will be described shortly) . The first layer in the hierarchy, which distinguishes between HEAP and STACK variables, is based on the facts that each data object is allocated on either the stack or the heap, and that auto variables are always allocated on the stack while static variables and memory returned by malloc are always allocated on the heap. The next layer in the hierarchy, which distinguishes between objects of different types, is based on the fact that a variable of type "pointer to T" will indeed usually be required to point only to objects of type T.[7] Note that unless the application programmer is known to follow this rule strictly, then no assertion

[6] C is merely used to provide a concrete example — any high-level language would be equally amenable to semantics retention along the lines described here.

[7] The ability in C to freely cast pointers of one type to pointers to another, means that a pointer to type T, could in fact point to an object of type T1.

is made about any particular pointer variable, i.e. the semantics of the source language alone can only imply that all pointers are associated with all nodes in the hierarchy; however, as we will see shortly, the programmer and/or a dependence analysis tool will usually be able to make specific assertions about individual pointer variables, or collections of pointer variables, that do follow this rule (or even more precise restrictions) so that those pointer variables will be associated only with paths for which the appropriate "type" node is present. Below we will show examples of using this simple "type" assertion, as well as other, more algorithm specific assertions to create a hierarchy that greatly improves disambiguation of pointer references at the instruction level.

While the first two levels in the hierarchy represent partitions that can be made for virtually all C programs, the next level in the hierarchy, which shows a different node for each variable, represents the first level that is program specific. The partitioning at this level is based on the fact that different variables in C do not share memory with one another. In fact, the existence of this level highlights one of the main problems with the conventional approach of completely separating front-end from back-end issues in compilation. At the instruction level, every LOAD and STORE is potentially ambiguous since there is typically no way to know *a priori* what it is used for. The conventional approach for dealing with this is to try to determine interdependencies between LOADs and STORE's (and other memory references for non-RISC targets) via analytical (symbolic) disambiguation techniques. Due to imperfections in these techniques and/or cost vs. performance trade-offs used to artificially constrain the amount of work done, many LOADs and STOREs that would be clearly independent if looked at in the context of the high-level references that they represent (e.g. LOADs and STOREs to different data structures), may be conservatively determined to be dependent by conventional instruction-level disambiguation techniques. Even for those dependencies that conventional analysis techniques correctly determine, the time taken to do so is completely wasted when considered in relation to the fact that most of these dependencies were already known at the higher level.

The example shown in Figure 2 illustrates how even this simple base structure for the hierarchy can by itself eliminate many potential ambiguities and much of the cost of disambiguation at the instruction level. The ASSERT's shown create a hierarchical decomposition that represents the facts that NODE and CTRL each point to objects of different types and furthermore that CNT, DATA, and NEXT are disjoint parts of *NODE (i.e. the object pointed to by node) and VEC is a field of *CTRL. These assertions can be made manually by the programmer or automatically by the compiler if the programmer is known to follow the rule that all pointers are to objects of the correct type,[8] then, when translating to the instruction-level representation, 3-addr statements 1 and 3 inherit the associations of CNT, 4 inherits those of DATA, 5 and 7 inherit those of VEC, and 8 inherits those of NEXT. For the sake of simplicity we again assume an idealized 3-wide VLIW target architecture with homogeneous, unicycle functional units. If the compiler does not exploit the facts represented by the semantics hierarchy, then the compiler must make the conservative assumption that the pointers NODE and CTRL could point anywhere in memory and therefore that the LOAD represented by the 3-addr statement 5 must be conservatively assumed to depend on the STORE in statement 3. In this case, the best schedule that can be obtained for the selected target architecture, shown under the "Schedule without" heading in the figure, takes 6 cycles. If on the other hand, the compiler exploits the properties of the hierarchical decomposition, then the 3-cycle schedule, shown under the "Schedule with" heading,

[8] This information would typically be imparted via a command line argument to the compiler.

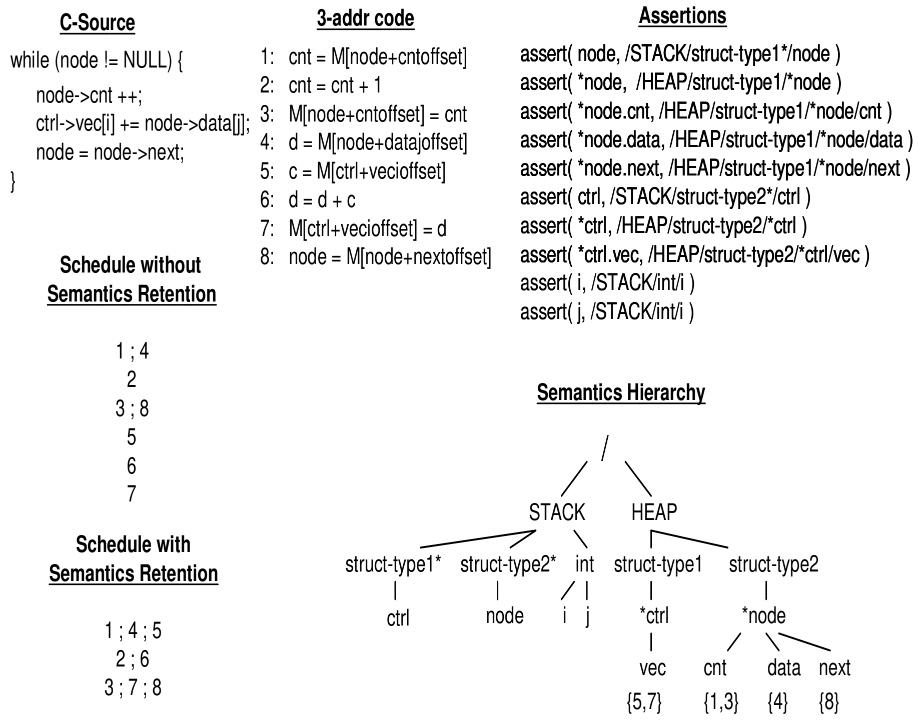

Fig. 2. Eliminating ambiguities.

can be achieved. In this case, the hierarchy indicates that NODE and CTRL can not possibly point to the same object and therefore statement 5 could be scheduled in parallel with statement 1. In addition, the hierarchy also implies that statement 4 is independent of statement 3. While this fact might be determinable via analytical instruction-level disambiguation techniques (and in fact, notice that we did assume that the references were correctly disambiguated for the "No Semantics Retention" schedule), the hierarchy obviates the need for performing such analysis for these two references; at the very least this represents improved compilation time efficiency and in cases where the analysis would fail to disambiguate the references, it would often yield improved parallelization as well. For instance, if the range of possible values for J, the variable used to index `node->data`, is unknown at the instruction level, and the compiler lays out data so that CNT follows DATA, then any purely analytical instruction-level disambiguation technique would still have to assume that statement 4 depends on statement 3 since, in this case, there is no way to distinguish between the end of DATA and the beginning of CNT, even though at the source-level the references are clearly independent according the semantics of the source language.

The above example shows how simply retaining some trivial semantics information about the C language can significantly improve instruction-level parallelization. The above example also shows that this is insufficient to yield maximum parallelism — if, as it seems, the "while" loop traverses an acyclic linked list of nodes, it should be possible to pipeline the loop so that successive iterations are executed in parallel; however, without some algorithm-level semantics information about what the loop is doing, the memory references to NODE from successive iterations are conservatively assumed to be ambiguous and thus a loop carried dependence would exist between statements 3 and 1 that would prevent loop pipelining from increasing the degree of parallelism within the loop.

While we have shown how to eliminate many of the ambiguities introduced when translating from a source-level to an instruction-level representation, we have yet to deal with the semantics loss that occurs when translating from the algorithm-level to the source-level (and from there to the instruction-level). Below we show how source-level dependence analysis can further refine the hierarchical decomposition to retain much of the important algorithm-level semantics that would otherwise be lost.

Algorithmic Semantic Retention – (Complex) Data Structures The optimization of programs involving dynamic, pointer-based data structures poses a difficult problem for compilers. The compiler needs both an accurate dependence test and information about the properties of the data structure. For example, the iterations of the loop shown in Figure 2 are provably independent if the compiler knows that `next` is acyclic. At the very least, analysis must be done at the source-level if there is any hope of discovering this information about the data structure. Given the very limited success of current fully automatic analysis techniques, it has been proposed that the user should provide much of the algorithmic information to the compiler. One such approach is the **ADDS** language [11] for abstractly describing data structures. **ADDS** allows the programmer to provide algorithm-level information about data structures in a structured, concise manner, in a way that will not significantly increase the complexity of the coding task as it only requires the programmer to make explicit those properties of the data structure that should be obvious to anyone who understands the algorithm. This information can easily be integrated into the approach proposed here by using it to further refine the program's memory reference hierarchy.

```
typedef struct llnode [N] {
  int cnt;
  float data[100];
  struct llnode *next forward along N;
} list;
```

(a) Supplying the property of "acyclicness".

```
while (...) {
  node->cnt++;
  ctrl->vec[i] + = node->data[j];
  node1 = node->next;
  if (...) break;
  node1->cnt++;
  ctrl->vec[i] + = node1->data[j];
  node2 = node1->next;
  if (...) break;
  node2->cnt++;
  ctrl->vec[i] + = node2->data[j];
  node = node2->next;
}
```

(b) Thus allowing more code to be exposed for parallelization at the instruction-level.

Fig. 3. Using **ADDS** to provide algorithm level semantics.

Thus, in the example shown in Figure 2, the user can use **ADDS** to provide a more informative type declaration, supplying the necessary properties to the compiler — namely that of "acyclicness". The revised declaration is shown in Figure 3a: the keyword **forward** conveys the fact that the `next` field is traversing the specified *dimension* `N` in an acyclic manner. Analysis of the loop would then reveal that the iterations are independent if for any node p, it can be shown that the path $p.next^+$ does not lead back to p. This is easily proven given the fact that `next` is acyclic. To exploit this information at the instruction-level, the compiler could, for instance, unroll the loop from

Figure 2 a few iterations at the source-level to expose more operations to be scheduled in parallel at the instruction-level.[9] This is what is shown in Figure 3b. In this case, each version of NODE from successive iterations is assigned a new compiler-generated variable, NODE_I, while the other, loop invariant, variables CTRL,[10] I, and J, remain the same across all iterations. The compiler would then encode the fact, derived from the **ADDS** declaration, that each NODE_I is independent of the others, by generating new ASSERT's that have the cumulative effect of creating new copies of the sub-trees rooted at NODE and *NODE for each NODE_I version. Given this new refinement to the hierarchy, the degree of parallelism available to the instruction level scheduler is now limited only by the length of the linked list and the rate at which the **`next`** fields can be traversed.

One of the advantages of dynamic data structures is their ability to be composed in complex ways, allowing more complicated relationships to be expressed. The challenge of course is for the compiler to perform accurate analysis even in the presence of such complexities. Furthermore, as indicated by the previous example, it can be critical for the results of this analysis to be available to all stages of compilation, not just to systems of source-level transformations. For example, a *Bipartite graph* is essentially two linked-lists, where each node contains a list of connections (pointers) to some of the nodes in the other linked-list. This data structure simply allows nodes in one list to be related to those in another, and since it is composed of linked-lists, the previous analysis and semantics retention techniques should apply. The difference now is that the data structure is no longer a simple tree-like structure, but a DAG.

```
typedef struct bgnode [N][E] {
  float coeff, value;
  struct bgnode *next forward along N;
  struct bgnode *dests[max] forward along E;
} bigraph
where ∀ p and i, p.next* <> p.next*dests[i] ;
```

(a) **ADDS** declaration for the bipartite graph.

```
p = head;
while (p ≠ NULL) {
   for (i=0; i<max; i++) {
     d = p->dests[i];
     p->value += p->coeff * d->value; /* L */
   }
   p = p->next;
}
```

(b) The main computational loop.

Fig. 4. A more complex example.

Bipartite graphs are useful in real applications, as for example, in the simulation of the propagation of electromagnetic waves through objects in 3D [4]. The graph's linked-lists are composed of electric (E) nodes and magnetic (M) nodes. The most important properties of this data structure are that (1) the sets of E and M nodes are distinct, and (2) the data structure is acyclic. This is extremely difficult to deduce automatically, and yet is known to the programmer as a consequence of the algorithm. These properties can however be conveyed to the compiler using a combination of the approaches discussed in [11, 13], as shown in Figure 4a. The data structure has two dimensions, `N` and `E`, which denote traversal along the nodes and traversal along edges to the destination nodes, respectively. These fields are acyclic, which is conveyed by the **forward** keyword. The **where** clause supplies an axiom based on regular expressions

[9] In practice, we tend to rely on instruction-level loop pipelining techniques to expose new operations; however, in the interest of brevity we will focus here on source-level unrolling — the issues involved for refining the hierarchy at the instruction-level are essentially the same, but require a bit more terminology to describe.

[10] Recall that we already know that CTRL is independent of any value that NODE might have.

[13] to state that the sets of nodes are distinct: the set of nodes S reachable by traversing `next` (0 or more times) is disjoint from the set of nodes reached by starting from those in S and traversing along `dest[i]` exactly once.

Figure 4b is the main computational loop taken from the simulation program. The outer loop updates each E (or M) node based on the values in the M (or E) nodes to which it is connected. The outer loop iterates across the linked-list of E or M nodes, while the inner loop performs a sum operation using the values in its destination nodes. The inner loop is inherently sequential given the sum computation performed by statement `L`. However, at first glance there also appears to be a loop-carried dependence across the outer loop: unless the relationship between `d` and `p` is known, it appears that one iteration of the outer loop is writing a value (`p->value`) that is being read by some later iteration (`d->value`).

Given that the sets of E and M nodes are disjoint, our system can prove that the iterations of the outer loop are in fact independent. This would allow, for example, the while loop to be parallelized at the source-level. However, there are additional optimization opportunities at the instruction-level, which will be missed unless this information is retained. In particular, consider the inner loop once again. Unless the relationship between `d` and `p` is known, the load of `d->value` in iteration $i+1$ cannot proceed until the store of `p->value` in iteration i is completed, thus forcing sequentiality between successive iterations of the loop. We know at the algorithm-level however that `d` and `p` refer to disjoint sets of nodes, and thus the load of `d->value` and the computation of `p->coeff*d->value` for iteration $i+c$ can proceed independently of the store `p->value` in iteration i. The end result is an improved instruction schedule which exploits this parallelism. The information which enables this transformation is supplied at the algorithm-level, calculated at the source-level, and easily retained at the instruction-level via ASSERT(*p, /HEAP/bigraph/*p) and ASSERT(*d, /HEAP/bigraph/*d).

4 Implementation and Results

This section provides a brief overview of how the semantics retention hierarchy is integrated within a retargetable fine-grain parallelizing compiler being developed at the University of California, Irvine,[11] and presents results that illustrate some of the advantages of semantics retention. This compiler is divided into a frond-end, based on the GNU C compiler (GCC), and a retargetable back-end responsible for performing instruction-level parallelization. GCC is responsible for translating C into an optimized 3-addr representation that becomes the input to the back-end. Higher-level semantics are communicated to the back-end via ASSERT statements embedded within the 3-addr code. These ASSERT statements may be generated by GCC itself or simply passed along from the source-level — source-level ASSERT's are implemented as macros that use the GCC ASM statement to pass the information along from source to the internal representations of GCC (i.e. TREE and RTX code), and finally to the 3-addr representation which will be read by the back-end (the 3-addr representation is based on the MIPS instruction-set).

Note that, while the selection of ASSERT's by the front-end depends entirely on algorithm level and source-level semantics, the semantics retention hierarchy constructed by the back-end is based entirely on the ASSERT's themselves, without any other information about the algorithm, the source language, or the front-end itself. Thus, given

[11] Details about the functionality of this compiler can be found in [19, 20, 22, 21].

this "3-addr + ASSERT's" representation, the back-end is able to construct the semantics retention hierarchy and parallelize code at the instruction-level with fairly precise algorithm level and source-level semantics that improve both the efficiency and degree of parallelization, without sacrificing the principal advantages of the traditional separation of front-end and back-end concerns, namely complexity management and the ease of porting the compiler to different source/target combinations.

	Neglecting Semantics			Retaining Semantics		
bench	speedup	disamb cost	efficiency	speedup	disamb cost	efficiency
SPARSE	1.34	0.25	5.36	4.85	0.11	44.09
EM3D	2.25	0.03	75.00	4.87	0.04	121.75
1DPP	2.18	1.46	1.49	4.35	0.43	10.12
OCTREE	3.50	0.61	5.74	6.44	0.30	21.47
CAXPY	2.39	0.11	21.73	5.90	0.05	118.00
Avg	2.33	0.49	4.74	5.28	0.19	28.40

Table 1. Neglecting vs. Retaining Higher-Level Semantics

Table 1 compares the results of compilation with and without semantics retention for a few codes actually used in some real-world applications. EM3D is the bipartite graph example discussed in Section 3.2 used for simulating the propagation of electromagnetic waves through objects in 3D [4]; SPARSE is code for scaling a sparse matrix abstract data type implemented using orthogonal linked lists [15]; 1DPP is a one-dimensional particle pusher implemented using array indirections [18]; OCTREE traverses threads in a 3-dimensional tree representation of a scene, mapping colors from RGB to CMY[10]; and finally, CAXPY is the linpack routine[5], implemented in C, for multiplying a vector by a constant and adding it to another vector (the potential instruction-level ambiguity here comes from the fact that vectors (i.e., arrays) are passed as pointers in C). The target architecture used for these results is a VLIW architecture with 32 registers and can issue two operations per cycle to any of the following functional units: 2 ALU units (which handle integer addition, subtraction, and logical operations), 2 SHIFT units (shift operations), 2 FALU units (floating point addition, subtraction), 2 MUL units (integer and floating point multiply), 1 DIV unit (floating point division), 2 MEM units (memory load and store), and, finally, 1 BRANCH unit (conditional branches). All functional units are pipelined. The ALU and SHIFT units take 3 cycles to execute an operation, the FALU and MUL units take 5 cycles, the DIV unit takes 15 cycles, the MEM units take 4 cycles for cache reads and 1 cycle for cache writes (cache misses stall the processor), and the BRANCH unit takes 4 cycles. These functional unit kinds and latencies are roughly the same as those of the Motorola 88110 Superscalar.

Speedup is the ratio of sequential to parallel cycles observed during simulation, *disamb(iguation) cost* is the total time (in seconds) spent trying to compute dependencies between memory accesses, and *efficiency* is the ratio of speedup to disambiguation cost. Notice that the speedup using semantics retention is uniformly better by more than 2.4 times on average. Similarly, even though semantics retention results in increased code motion and parallelization, the disambiguation cost is still generally lower with semantics retention by an average of 40%. Finally, the efficiency numbers show that, even when the disambiguation cost increases slightly as a result of semantics retention (as for EM3D), the time spent compiling with semantics retention is consistently better utilized, by an average efficiency improvement of 416%.

References

1. A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.
2. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *TOPLAS*, 9(4), 1987.
3. U. Banerjee. *Dependence analysis for supercomputing.* Kluwer Academic Press, Boston, MA, 1988.
4. D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of Supercomputing 1993*, 1993.
5. J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users's Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1978.
6. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, 1994.
7. R. Ghiya. Disambiguating heap references. Technical Report Masters Thesis, School of Computer Science, McGill University, October 1994.
8. M. Girkar and C.D. Polychronopoulos. A general framework for program optimization and scheduling. *TOPLAS*, 1995. Preprint.
9. M. Haghighat and C. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Lang. and Compilers for Par. Comp.*, volume 757 of *LNCS Series*. Springer-Verlag, 1992.
10. D. Hearn and M. Pauline Baker. *Computer Graphics.* Prentice-Hall, 1986.
11. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *PLDI*, 1992.
12. L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Computing*, 1(1), 1990.
13. J. Hummel, L. Hendren, and A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *IPPS*, 1994.
14. D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Fourth International Computer Software and Applications Conference*, 1980.
15. K. Kundert. Sparse matrix techniques. In A. Ruehli, editor, *Circuit Analysis, Simulation and Design.* Elsevier Science Publishers B.V. (North-Holland), 1986.
16. W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *PLDI*, 1992.
17. J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* PhD thesis, MIT, 1987.
18. F.H. McMahon. The livermore fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.
19. S. Novack and A. Nicolau. An efficient global resource constrained technique for exploiting instruction level parallelism. In *ICPP*, St. Charles, IL, August 1992.
20. S. Novack and A. Nicolau. Vista: The visual interface for scheduling transformations and analysis. In *Lang. and Compilers for Par. Comp.*, volume 768 of *LNCS Series.* Springer-Verlag, 1993.
21. S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Lang. and Compilers for Par. Comp.*, volume 892 of *LNCS Series.* Springer-Verlag, 1994.
22. S. Novack and A. Nicolau. A hierarchical approach to instruction-level parallelization. *International Journal of Parallel Programming*, 23(1), February 1995.
23. D. Padua and M. Wolfe. Advanced compiler optimization for supercomputers. *CACM*, 29(12), December 1986.

This article was processed using the LaTeX macro package with LLNCS style