An Object Oriented Intermediate Code Representation for the Development of Parallelization Tools

Mark R. Gilder
(518) 276-4857
gilderm@cs.rpi.edu

Mukkai S. Krishnamoorthy
(518) 276-6911
moorthy@cs.rpi.edu

Rensselaer Polytechnic Institute
Department of Computer Science
Troy, New York 12180

Abstract

We show that by developing an intermediate representation, which supports a formalized interface on which to construct parallelization tools, the mapping of the representation onto parallel architectures can be performed quickly and efficiently. A flexible intermediate representation called HICOR (*Hierarchical Intermediate Code Object Representation*) is presented. HICOR is shown to facilitate the exploitation of parallel operations by providing an abstraction layer for performing both parallel transformations and code generation. An object oriented design approach has been employed in the development of HICOR. Source language constructs are transformed into specialized object classes. Inheritance properties provided by the object oriented paradigm are utilized to provide a common interface to each object in the HICOR representation. It is this interface that provides the needed consistency and flexibility in which to construct tools for parallel compiler development that has since been lacking. In particular, we present a tool to perform *Hierarchical Dependency Analysis*. This tool allows program dependencies to be calculated on various grain sizes including: module, function, block, and instruction-level. This approach differs from current methods in that explicit flow graph construction is not necessary. We have implemented a prototype system in C++ which incorporates the methodology described. The target architectures include a Sun 630 MP/4, Sequent Balance 21, and Stardent Titan. The construction of a high-level flexible intermediate representation will provide a strong foundation for future work, eventually leading to a standardized interface in which to develop compiler tools for parallel architectures. The use of object oriented methodologies will play an important role in achieving this goal.

1 Introduction

Widely accepted intermediate code forms such as *quadruples*, *triples*, and *postfix* notations are not sufficient representations for targeting multiprocessor architectures. In particular, useful information that can be quickly attained from high-level source code constructs are often lost during the translation to an intermediate representation thereby complicating the design of parallelization tools. More importantly, current compilers do not easily allow the incorporation of newly proposed tools [3, 4, 5, 10, 13, 16, 21, 24] for lack of a formal interface in terms of the intermediate representation. Hence, the development of general-purpose parallelization tools becomes an expensive proposition. Current research efforts, such as ParaScope at Rice University [14], PTRAN at IBM T. J. Watson Research Center [23], and the Typhoon parallelizing compiler project at Cornell University [18], address this issue but are with respect to a single programming language and are not explicitly based on object oriented principles.

In this paper, we present another approach to accomplishing this for multiple source languages and multiple target architectures. The design employed provides a consistent view of all high-level constructs available at the source code level throughout all phases of the compilation. In particular, the intermediate code generated by the compiler front-end is explicitly represented by objects instantiated from a specialized class hierarchy called HICOR. Each object supports a formal interface mechanism through which internal object information is made available. This added level of abstraction allows flexible parallelization tools to be developed without regard to the underlying structure of the internal representation. Hence, the tool becomes immune to any internal changes made in the intermediate representation. A tool to perform *Hierarchical Dependency Analysis* is presented to demonstrate the flexibility in using this approach.

Each of the system components are shown in Figure 1. ANSI C, Fortran, and Pascal are translated into the HICOR representation. The initial HICOR instance is independent of any target architecture. However, once the target is specified, the HICOR instance is mapped into a suitable view through the use of the *Architectural Filter* object. The filter systematically manipulates the HICOR instance according to the constraints specified by the *Target Architectural Description*. The output of the system is an execution DAG (directed acyclic graph) appropriate for the specified target architecture. Using this DAG machine specific target code may be generated [9]. Current target architectures include the Sequent Balance 21, the Ardent Titan, and the IBM ESA/390. The system was implemented on a SPARC-station 2 and was written in C++. The system is open-ended and interfaces nicely to other packages which either produce C, Pascal, or Fortran as output; or accept C as input. Furthermore, a GUI has been developed for the system which provides the user with a visual representation of the parallelizations performed [20].

The remainder of this paper is organized as follows: Section 2 describes the HICOR intermediate representation; Section 3 describes the implementation of the *Hierarchical Dependency Analyzer*; and in Section 4 we conclude with an evaluation of the object oriented approach, as presented, to construct compiler tools.

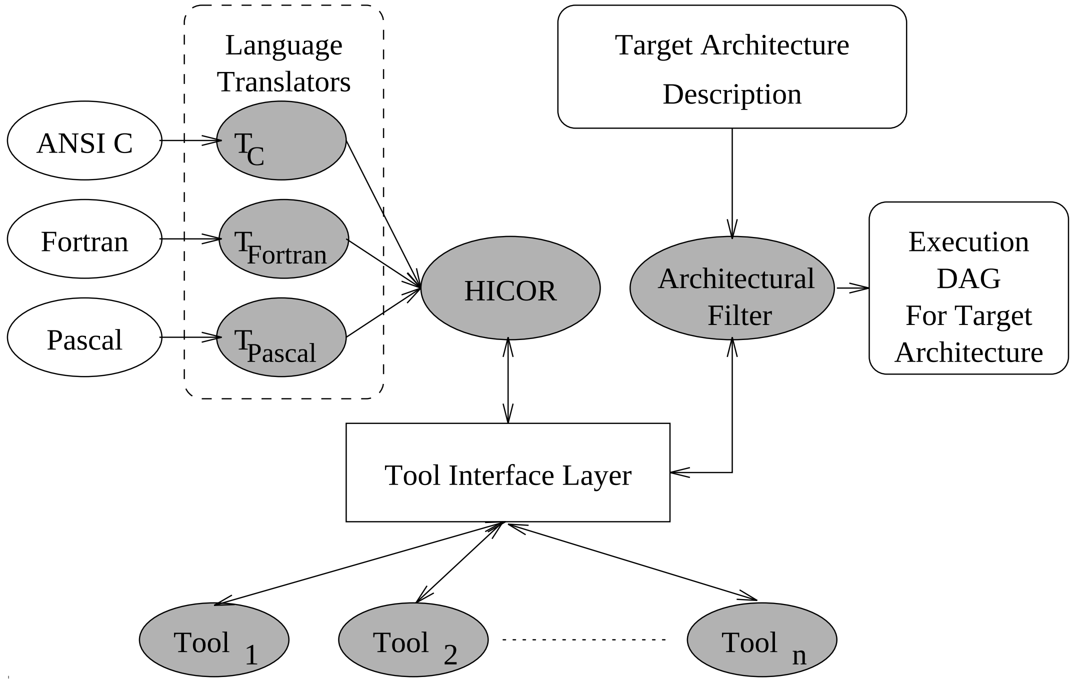


Figure 1: System Components

2 HICOR as an Intermediate Representation

HICOR is an object oriented intermediate representation which provides a hierarchical view of program constructs. Each object in the representation encapsulates the internal intermediate code corresponding to the source construct it represents. Internal object information is provided through a set of common access routines or *methods*. It is these methods that provide the *Tool Interface Layer* on which parallelization tools may be constructed. In most instances a single HICOR object may be decomposed into an ordered collection of smaller, more detailed, objects; for example, a HICOR object which represents a *Function* in some source language, may be viewed as a collection of objects, each representing the immediate basic blocks it consists of. In this respect, the grain size represented by a particular HICOR object is related directly to its position in the hierarchy. Objects occurring at higher levels are more computationally complex than those occurring at lower levels. Therefore, as we descend through the hierarchy, the grain-size represented decreases.

2.1 HICOR Objects

HICOR objects may be classified as either *simple* or *complex*. *Simple Objects* are those objects which cannot be decomposed further and thus represent the finest grain of computation available. *Complex Objects* are those objects which may be viewed as a collection of computationally smaller HICOR objects. The order of objects in this collection is important as it represents the control flow in the original source program. Each *complex* object uses the notion of an *ordered set* to maintain the necessary ordering information.

All objects are derived from the HICOR *Base* class. This class specifies the functionality requirements which are shared by all HICOR objects. Figures 2 and 3 illustrate the relationship between each of the HICOR object classes. In particular, Figure 2 contains object classes representing single program statements (*simple objects*) whereas Figure 3 contains object classes used to represent high-level source constructs (*complex objects*) typically made up of more than one statement.

A subset of the *instance variables* and *instance methods* defined for the base class are listed in Table 1 and Table 2, respectively. The methods listed are common to all HICOR objects; hence, tools may manipulate them without regard to the objects particular type. The *ordered set* object returned by the `ParentOf()` and `GetDescendants()` methods has been implemented as a binary search tree.

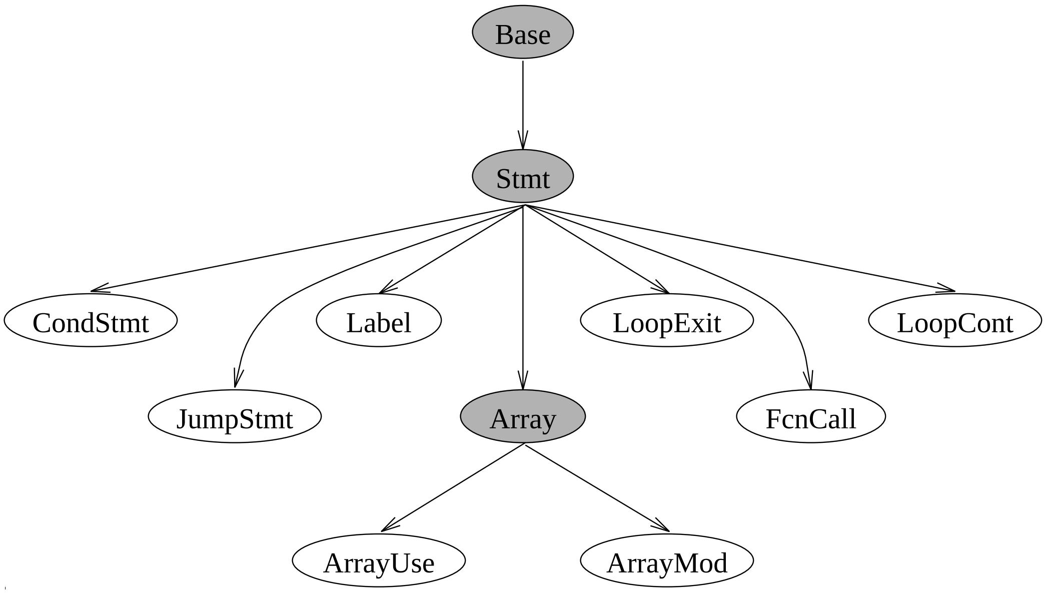


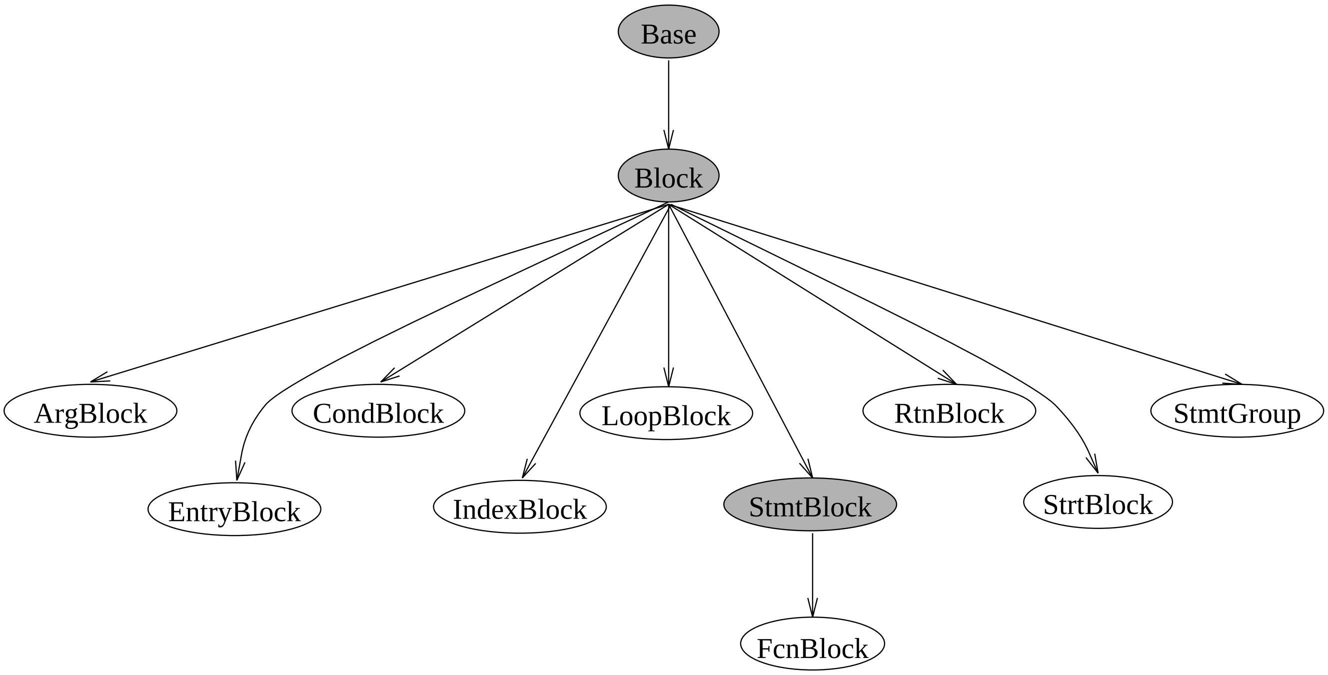
Figure 2: HICOR Statement Objects

Instance Variables	**Description**
`Cost`	The estimated cost of executing all statements represented by the object.
`ModUseVars`	An ordered set of all variables modified and/or used by the object.
`ObjNumber` (oid)	A unique integer identifier for the object.
`ObjType`	An integer indicating the object type.
`Prev`	A link to the previous object in the same layer.
`Next`	A link to the next object in the same layer.
`Child`	A link to the first child object in the next layer.
`Parent`	A link to the parent object in the previous layer.

Table 1: Base Class Instance Variables

Instance Methods	**Description**
`GetNumber()`	Returns the object's identifier, i.e., `ObjNumber()`.
`GetType()`	Returns the object's type, i.e., ObjType.
`GetParent()`	Returns the object's parent.
`ParentOf()`	Returns an *ordered set* containing the immediate children of the object.
`GetDescendants()`	Returns an *ordered set* containing all descendants of the object.
`ComputeCost()`	Computes the estimated cost of executing the object on a specified target machine.
`BuildVarSets()`	Constructs the set of variables modified and/or used by the object.
`IsMod(`**var**`)`	Returns **TRUE** if the object *modifies* variable **var**, **FALSE** otherwise.
`IsUse(`**var**`)`	Returns **TRUE** if the object *uses* variable **var**, **FALSE** otherwise.
`IsModBeforeUse(`**var**`)`	Returns **TRUE** if the object *modifies* variable **var** before it *uses* it, **FALSE** otherwise.
`GetModUseVarSet()`	Returns the set constructed by `BuildVarSets()`.
`GenCode()`	Generates target specific code for the object.
`DispSource()`	Displays the source code corresponding to the object.

Table 2: Base Class Instance Methods



Note: Shaded classes indicate that the class is used to derive specialized classes.

Figure 3: HICOR Block Objects

2.2 HICOR Example

Figure 4 contains a C-function called MinMax which computes the minimum and maximum values of the input array a. The first few levels of the HICOR representation are provided in Figure 5. The numbers to the left of each object indicate the implicit execution order of the original serial version and were obtained by invoking the `GetNumber()` method on each object. These numbers also correspond directly to selected components of the C-program which are labeled by $Obj_{<num>}$ in Figure 4. Dashed arcs provide a link to the immediate children of an object. For example, the immediate children of object 64 consists of all objects at level $i+1$. This allows us to view the MinMax function as single object, i.e., object 64, or as a collection of the sub-objects which comprise it. We emphasize that the tool interface remains consistent no matter which view is chosen. The initial execution order of objects at any hierarchical level is indicated by solid arcs.

The *LoopBlock* object represents iterative constructs found in the source program and may have up to four children. Each child contains the code to be executed for each logical component of a loop, i.e., *initialization*, *comparison*, *loop-body*, and *continuation* (see Figure 5 objects 33, 48, and 63). The *CondBlock* object represents statements which may be executed conditionally and may have up to three children. The first child computes a Boolean expression $\mathcal{B}$. The second child, if present, represents the statements to be executed if $\mathcal{B}$ is true. Similarly, the third child represents statements to be executed if $\mathcal{B}$ is false (see Figure 5 objects 31, 46, 61). All 3-address computations are represented by a *Stmt* object. The *ArrayUse* and *ArrayMod* objects, which are derived from the *Stmt* object, are used to represent array accesses and modifications, respectively. The *StmtGroup* object represents a block consisting of more than one *Stmt* object. If necessary, the type of any object may be retrieved by invoking the `GetType()` method. For example, if `GetType()` were invoked on object 48, a unique integer value identifying the object as a *LoopBlock* would be returned.

Our representation uniquely allows tools to capture information at any level of granularity in a consistent manner. For example, the fact that object 19 modifies arrays b and c, and also uses array a, can be retrieved at any object that implicitly contains it, i.e., objects 31, 33, and 64.

2.3 Code Generation

The generation of code is implicitly handled by each HICOR object. This is accomplished through the `GenCode()` method. Each object implements this method specifically for the type of code it represents. For instance, a *LoopBlock* object is responsible for mapping iterative constructs onto the chosen architecture. The `GenCode()` method takes an *architecture specifier* object as a parameter. This object provides all architectural constraints and code translations for a specific machine. For example, to generate Sun 630 MP/4 code for the MinMax() function, we simply invoke the `GenCode()` method on object 64 with the appropriate *architecture specifier* object. The details of the architecture specifier object may be found in [9]. Our purpose here is to demonstrate the simplicity of using an object oriented approach to design compilation tools. The tool is no longer burdened with the task

```
Obj_64          void MinMax(int a[N]) { /* N is even */
                    int  i, j, b[N/2], c[N/2], min, max;

Obj_0               j = 0;
Obj_33              for (i = 0; i < N; i = i + 2)
Obj_31                  if (a[i] > a[i + 1]) {
Obj_19                      b[j] = a[i];
                            c[j++] = a[i + 1];
                        }
                        else  {
Obj_30                      b[j] = a[i + 1];
                            c[j++] = a[i];
                        }

Obj_36,37           max= b[0];
Obj_48              for (i = 1; i < N/2; i++)
Obj_46                  if (b[i] > max)
Obj_45                      max = b[i];

Obj_51,52           min= c[0];
Obj_63              for (i = 1; i < N/2; i++)
Obj_61                  if (c[i] < min)
Obj_60                      min = c[i];
                }
```

Figure 4: C-Program to Find the Minimum and Maximum Values of Array a

of manipulating the low-level internals of each object but, instead, becomes responsible for delegating commands to be carried out by the objects themselves. Another advantage to this approach is that new object types can be added to the HICOR representation by simply defining the required methods defined in Table 2. Tools developed would therefore not be affected by the addition of new HICOR objects.

2.4 Parallelization

In order to parallelize the sequential code represented by a HICOR instance we need to quickly determine what HICOR objects, if any, can be executed concurrently. Since each HICOR object inherently contains information about all of its descendants, the computational complexity involved in locating independent objects is greatly reduced. For instance, the determination as to whether or not any set of HICOR objects can be parallelized can be made by analyzing only those objects in question. For example, to analyze the first level of basic blocks in the MinMax function for parallelism, we need only refer to objects: 0, 33, 36, 37, 48, 51, 52, and 63. The *Hierarchical Dependency Analyzer* presented in Section 3 computes a DAG (directed acyclic graph) object representing data dependencies for any set of HICOR objects. Once constructed the estimated speedup may be calculated [19, 22]. If acceptable, the DAG object would be responsible for generating the target code. Since each HICOR object represented by the DAG is responsible for its own code generation, the DAG object need only generate the necessary control structures needed to create each of the logical processes. The body of the each process is generated by calling the `GenCode()` method on each object it contains. However, if the estimated speedup is not acceptable, the exploitation of parallelism may be directed to lower levels of the hierarchy, i.e., the immediate children of the objects analyzed thus far.

We have shown that the collection of methods defined for each object type provide the formal interface in which to construct parallelization tools. In the next section we demonstrate how these methods are exploited to calculate hierarchical dependencies.

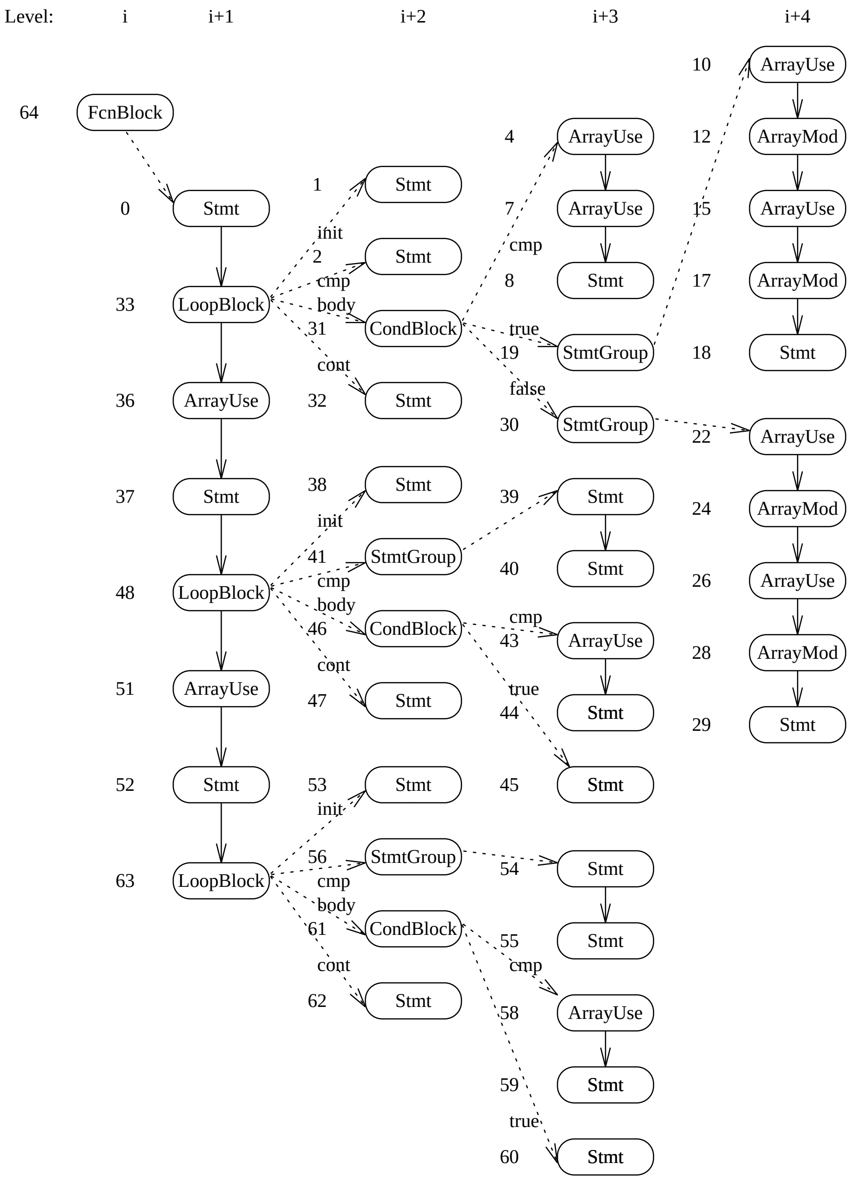


Figure 5: HICOR Representation

3 Hierarchical Dependency Analyzer

By utilizing the HICOR *Tool Interface Layer*, dependency information can be efficiently calculated. The *Hierarchical Dependency Analyzer* (HDA) constructs a DAG object representing the data dependencies that exist between any set of HICOR objects. The HDA tool uses two additional ADT's, each of which is directly implemented as an object class. Only methods necessary for understanding the algorithm are described here.

- The ordered set class *OrdSet* provides all of the basic set operations as well as providing an indexing mechanism. This allows individual elements of a set to be accessed in order. We use the notation $S[i]$ to refer to the i-th element of the ordered set S. We use the notation *OrdSet*$<$ *type* $>$ to define an ordered set containing a particular element type. For example, *OrdSetInt* S, defines S to be an ordered set consisting of integers.
- The directed-acyclic-graph class *Dag* provides the necessary operations needed to construct a dependency DAG. In particular, the method `AddEdge()` adds an edge between the specified vertices. An optional argument may be supplied to indicate the dependency type.

The HDA tool is used as follows:

1. Generate a set of HICOR objects to be analyzed.
2. Pass these objects to the HDA tool.
3. Analyze the returned DAG for dependency information.

For example, to calculate dependency information for the immediate children of object 64, as was discussed in Section 2.4, we first create an ordered set containing its immediate children by calling the `ParentOf()` method on object 64. This set is then passed to the HDA tool, i.e.,

$$D = \textsc{GenerateDependencies}(\texttt{GetObject}(64).\texttt{ParentOf}())$$

The `GetObject()` method is used to retrieve the actual object for a given object identifier. Upon return, the DAG D will contain the dependency information for all of the immediate children of object 64. To generate target code for this representation we simply invoke the `GenCode()` method, i.e.,

$$D.\texttt{GenCode}()$$

We now present the HDA algorithm. For simplicity, we illustrate our algorithm on the HICOR representation shown in Figure 7. In particular, we show how dependencies are calculated for the immediate children of object 4. An ordered set containing HICOR objects 1, 2, and 3 is passed to the GenerateDependencies routine listed in Figure 9. This routine first constructs an ordered set consisting of all variables used and/or modified by the objects in *ObjSet*. Table 3 contains the variable sets returned by `GetModUseVarSet()` for each object in *ObjSet*[1]; hence $V = \{x, y, z\}$. We then call GenParallelDependencies to construct the dependency information. The result of which is stored in D.

The GenParallelDependencies routine listed in Figure 10 traverses the set HICOR objects in *ObjSet* once for each variable $v \in V$. The first *while-loop* sets *SrcObj* to the first object in *ObjSet* which modifies v. In doing so, any objects that used v are placed in *UseSet*. Once this object is found, dependency edges are added from all uses of v, i.e., the elements of *UseSet*, to *SrcObj*. Table 4 displays the values of *UseSet* and *SrcObj* for each of the variables in V at the end of the first *while-loop*. The second *while-loop* continues searching through *ObjSet* for uses of this newly defined v. A variable v is not considered to be *used* by an object if that object *modifies* v before it uses it. The `IsModBeforeUse()` method checks this condition. For each use of v found, we add an edge from the *SrcObj* to the current object that uses v, i.e., *DestObj*. If we encounter a new definition of v we simply set the *SrcObj* to this new definition and continue. In our example, no edges would be added in the first *while-loop* since there are no *uses* of variables in V that are later redefined. However, the second *while-loop* would add edges from $1 \rightarrow 3$ and $2 \rightarrow 3$ for dependencies induced by x and y, respectively.

[1] Even though object 4 is not in *ObjSet* we provide the result to demonstrate the propagation of data throughout the hierarchy.

In Figure 6 we provide the dependency graph generated by this algorithm for the MinMax function presented earlier in Figure 4. In particular, dependency information can be generated very efficiently since only the first level of basic blocks for the corresponding function are analyzed, i.e., objects 0, 33, 36, 37, 48, 51, 52, and 63; hence for this example a dependency graph was produced by analyzing approximately 1/8 of the associated objects. The results clearly indicate that the loops to find both the minimum and maximum values, i.e., objects 63 and 48, respectively, can be executed in parallel once the loop represented by object 33 has finished.

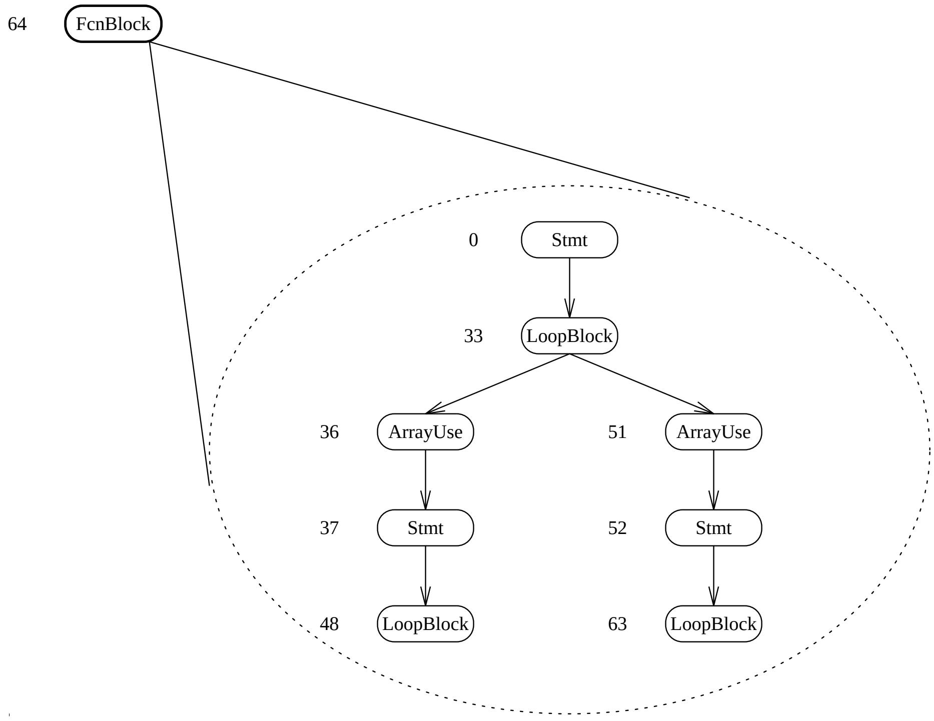


Figure 6: Dependency graph for MinMax function

We should also mention that the method employed here does not limit the scope of data dependence extraction to within *FcnBlock* objects. Interprocedural dependency analysis proceeds in the same manner. For example, any dependencies introduced by the call to f in Figure 8 will be identified by the *FcnCall* object. Therefore, if x were modified by f, `IsMod`(x) would return TRUE when invoked by object 2 and hence induce a dependency arc from $2 \rightarrow 3$.

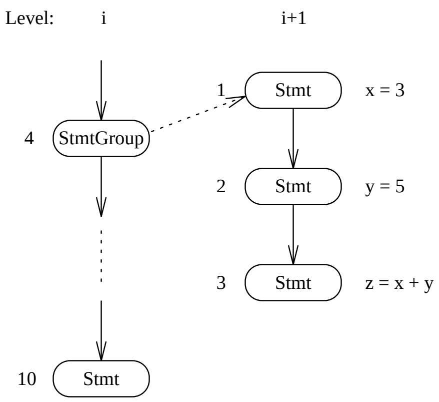


Figure 7: Sample HICOR Representation for HDA Algorithm

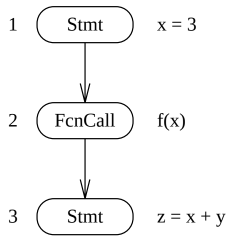


Figure 8: Sample Function Call

HICOR Object Number	**Set of Variables Modified and/or Used**
1	$\{x\}$
2	$\{y\}$
3	$\{x,\ y,\ z\}$
4	$\{x,\ y,\ z\}$

Table 3: HICOR Object Information

Variable Objects	*UseSet*	*SrcObj*
x	$\{\}$	$\{1\}$
y	$\{\}$	$\{2\}$
z	$\{\}$	$\{3\}$

Table 4: Variable Object Information

```
GenerateDependencies(OrdSetBase ObjSet)
  // This routine sets up the initial data required to generate dependency information for
  // the objects specified in ObjSet.
  Dag D
  OrdSetVariable V
  Int i

  // Add all variables used and/or modified by the objects in ObjSet to the variable
  // set V.
  V ← ∅
  Foreach element i of ObjSet do
    V ← ObjSet[i].GetModUseVarSet() ∪ V
  od

  // Generate a dependency Dag suitable for determining which objects of ObjSet may
  // be executed in parallel.
  D ← GenParallelDependencies(V, ObjSet)
  return D
end
```

Figure 9: HDA Driver Routine

```
GenParallelDependencies(OrdSetVariable V, OrdSetBase ObjSet)
  // This routine computes the dependencies necessary for determining which objects of
  //ObjSet may be executed in parallel.
  Dag D
  OrdSetInt UseSet
  Base SrcObj, DestObj
  Int i, j, k
  // For each variable v ∈ V find the first object in ObjSet that modifies v.
  // Then add dependencies arcs from all prior uses of v to this new definition.
  Foreach element i of V do
    UseSet.Clear()
    j ← 1
    While (j < ObjSet.Length()) do
      SrcObj ← ObjSet[j]
      If (SrcObj.IsMod(V[i]) ≠ TRUE ∧ SrcObj.IsUse(V[i]) = TRUE )
        UseSet.Add(ObjSet).GetNumber())
      j ← j + 1
      continue
      // The current variable, i.e., V[i] is modified by ObjSet[j]. Add
      // dependency edges from each element in UseSet to the current object.
      Foreach element k of UseSet do
        D.AddEdge(UseSet[k], SrcObj.GetNumber(),ANTI_DEP)
      od
      break
    od
    // Continue searching forward through ObjSet adding edges to all objects which use
    // this definition of V[i].
    j ← j + 1
    While (j < ObjSet.Length()) do
      DestObj ← ObjSet[j]
      If (DestObj.IsUse(V[i]) = TRUE ∧DestObj.IsModBeforeUse(V[i]) ≠ TRUE )
        D.AddEdge(SrcObj.GetNumber(),DestObj.GetNumber(),FORWARD_DEP)
      j ← j + 1
      If (DestObj.IsMod(V[i]) = TRUE ) do
        SrcObj ← DestObj
        continue
      od
    od
  od return D
end
```

Figure 10: Construction of Parallel-Dependencies

4 Conclusion

We have shown that the implementation of parallelization tools can be greatly simplified by using an object oriented design to construct the intermediate representation. The HICOR representation and associated *Tool Interface Layer* described in Section 2 provide a data encapsulation model with which compiler tools for SISD, MIMD, and SIMD architectures may be constructed; in particular, a tool to perform *Hierarchical Dependency Analysis* was presented. This tool allows program dependencies to be calculated for various levels of granularity. By using the interface layer to HICOR, we have shown that the tool needs no knowledge of the underlying intermediate representation, and therefore cannot be affected by any internal changes. Tools constructed in this manner are inherently simpler since they can manipulate the representation at a much higher level of abstraction. Other tools constructed using this approach are described in [8, 9].

The object oriented approach employed clearly has its advantages over traditional approaches. The HDA algorithm, as we described in Section 3, makes no distinction between interprocedural and intraprocedural analysis. Each type is handled transparently by the tool since the required information needed to compute dependencies is implicitly contained within each object. The object oriented methodology has helped us greatly in the design and implementation of our system.

The success of our system is attributed to the construction of objects and associated methods to encapsulate the underlying intermediate representation. The design allows all high-level constructs available at the source code level to remain consistent throughout all phases of the compilation and therefore simplifies the extraction of data necessary to generate parallel code.

References

[1] A. V. Aho and M. Ganapathi. Efficient tree pattern matching: An aid to code generation. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 334–340, New York, NY, January 1985.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, March 1986.

[3] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, Austin, TX, January 1989.

[4] M. Burke, R. Cytron, J. Ferrante, and Wilson Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, 2(3):71–88, July 1989.

[5] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, TX, January 1989.

[6] R. Duncan. A survey of parallel computer architectures. *IEEE Computer*, pages 5–16, February 1990.

[7] D. Gelernter, A. Nicolau, and D. Padua. *Languages and Compilers for Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[8] M. R. Gilder. An object oriented intermediate code representation for an architecturally independent parallelizing compiler. PhD thesis proposal, Rensselaer Polytechnic Institute, October 1991.

[9] M. R. Gilder. *An Object Oriented Intermediate Code Representation For An Architecturally Independent Parallelizing Compiler.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1993. In preparation.

[10] M. Girkar and C. Polychronopoulos. Compiling issues for supercomputers. *IEEE Computer*, pages 164–173, September 1988.

[11] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proc. of the SIGPLAN '91 Conf. on Programming Language Design and Implementation*, volume 26, pages 15–29, June 1991.

[12] M. Harrold and M. Soffa. Computation of interprocedural definition and use dependencies. In *Proc. 1990 IEEE Int. Conf. on Computing Languages*, pages 297–306, New Orleans, March 1990.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, June 1988.

[14] K. Kennedy, K. S. Mckinley, and C. W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[15] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[16] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(9):1073–1078, September 1989.

[17] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[18] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. Technical Report TR 90-1152, Cornell University, September 1990.

[19] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic, Boston, MA, 1988.

[20] J. R. Punin, M. R. Gilder, and M. S. Krishnamoorthy. A GUI for parallel code generation. In progress.

[21] S. Richardson and M. Ganapathi. Code optimization across procedures. *IEEE Computer*, pages 42–50, February 1989.

[22] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, 1989.

[23] B. K. Szymanski. *Parallel Functional Languages and Compilers.* Addison-Wesley, Reading, MA, 1991.

[24] Y. Tanaka, K. Iwasawa, S. Gotoo, and Yukio Umetani. Compiling techniques for first-order linear recurrences on a vector computer. *IEEE Computer*, pages 174–181, September 1988.

[25] A. E. Terrano, S. M. Dunn, and J. E. Peters. Using an architectural knowledge base to generate code for parallel computers. *Communications of the ACM*, 32(9):1065–1072, September 1989.