# XIL and YIL: The Intermediate Languages of TOBEY

Kevin O'Brien* Kathryn M. O'Brien† Martin Hopkins‡ Arvin Shepherd§ Ron Unrau¶

## Abstract

*Typically, the choice of intermediate representation by a particular compiler implementation seeks to address a specific goal. The intermediate language of the TOBEY compilers,* XIL*, was initially chosen to facilitate the production of highly optimal scalar code, yet, it was easily extended to a higher level form* YIL *in order to support a new suite of optimizations which in most existing compilers are done at the level of source to source translation. In this paper we will discuss those design features of* XIL *that were important factors in the production of optimal scalar code. In addition we will demonstrate how the strength of the* YIL *abstraction lay in its ability to access the underlying low level representation.*

## 1 Introduction

Faced with the task of writing a new compiler for one or more languages, one has no simple rule of thumb to apply in choosing the most appropriate intermediate form. We believe, however, that a set of principles can be enunciated which provides a framework for judging the utility of an intermediate language. Our ideal *il* should be easy to generate and easy to translate into the desired final form. It should allow the functions which operate on it to conveniently and flexibly manipulate it, both for purposes of analysis and transformation. It should be expressive, supporting the representation of a variety of programming idioms and styles, thus allowing it to support a multiplicity of source languages. It should be transparent, making clearly visible the semantics of the underlying program, thereby easing the task of compiling for a variety of architectures. It should be capable of expressing enough detail to facilitate low level optimization, yet of abstracting details which are unimportant to its clients, and it should be amenable to transportation, so that it may be saved, used by other tools and so on.

It is difficult for any *il* to satisfy all of these requirements in every context within a compiler, but we hope to convince the reader that the two *ils* described in this paper, together come very close to satisfying most of them, most of the time.

Over the past thirty years or so, there have been many intermediate languages proposed, to address a wide variety of issues. These languages fall into several broad categories. Chief among them have been *triples* and *quadruples* [11], *Abstract Syntax Trees* [5], and *Virtual Machines* [12, 2]. The notion of machine independence as an attribute of intermediate representation has certainly been around for a long time [3]. More recently, there have been many attempts to define intermediate representations which address problems specific to compilation for parallel or distributed memory machines, or compiling data parallel languages [4]. The intermediate representation of the SUIF compiler appears to nicely solve the scalar/parallel optimization dichotomy [5].

It is not our intention here to provide an exhaustive survey of the field, but merely to assert that the intermediate languages discussed in this paper share some of their properties with both *quadruples* and with *Virtual Machines*, while at the same time dealing with the issue of high vs low level optimization.

### 1.1 Outline

In this paper we will discuss in some depth the intermediate representation of a suite of compilers which generate highly optimized code, by means of the classical optimizations. In addition, we wish to demonstrate the relative ease with which the representation was abstracted to allow the application of a completely different set of higher level loop optimizations which have traditionally been performed at the source language level. The paper will give a brief overview in the next section, of the TOBEY compiler and optimizations. Sections 3 and 4 form the core of the paper; these sections present a detailed look at the two *ils*, *XIL* and *YIL*, with reference to interesting design features, and suitability for the types of optimization being performed. In section 5 we outline some implementation practices and experiences which it is felt underline the preceding discussions. The final two sections discuss future work and provide some concluding remarks.

* email:obrien@watson.ibm.com
† email:kobrien@watson.ibm.com
‡ email:hopkins@watson.ibm.com
§ email:arvin@torolab2.vnet.ibm.com
¶ email:unrau@torolab2.vnet.ibm.com


IR'95 1/95 San Francisco California USA

## 2 The TOBEY compilers

The intermediate languages *XIL* and *YIL* are used by the IBM compilers [1] for the Risc System/6000, *xlf, xlc, xlp* and *xlC++* ( FORTRAN, C, Pascal and C++). Over time compilers have been built for languages from Fortran90 to C++ with code generation for Intel 386, RS/6000, Sparc and S/370.

These compilers share a common back end, the Toronto Optimizing Back End with Yorktown, **TOBEY**, which was inspired by an earlier compiler project for the *801* minicomputer, the *pl.8* compiler, and *XIL* owes several of its major features to the intermediate language of that compiler [16].

The TOBEY optimizer [15] utilizes familiar techniques such as Common Subexpression Elimination (CSE), upward motion of invariant computations from loops, elimination of dead or unused expressions, and Strength Reduction and Reassociation. A host of other transformations are performed, namely, global constant propagation, dead store elimination, global value numbering, procedure inlining, local and global instruction scheduling. As the TOBEY project progressed, it became clear that, although the quality of the generated code met our initial expectations, the rise in popularity of cache-based machines was opening up new opportunities for optimization. Unlike the classical optimizations, these new techniques operated on extended structures within a program, principally nests of loops. These restructuring transformations were more readily performed on an intermediate language which reflected the higher level structure of the code. Out of this insight, *YIL* was forged.

## 3 XIL

### 3.1 Architectural Principles

The chief objectives of the TOBEY project were to produce high quality object code, and support multiple source languages and target architectures. Clearly these goals influenced the design of the intermediate representations in several ways. Ensuring the flexibility to add new languages or target machines, as required, necessitated that the initial design provide clean interfaces. The intermediate representation could not implicitly support peculiarities of language semantics nor could it, at the interface to the front ends, require knowledge of particular features of a specific instruction set architecture.

On the other hand, constructing a highly optimizing compiler inexorably pointed towards an *il* that exposed the low level operations of a typical register-to-register machine. The major gains from the classical optimizations, CSE, Code Motion, Strength Reduction and their ilk, are made at the level of addressing computations [17]. This is reflected in *XIL*, by the requirement that all loads and stores, and expressions necessary to compute addresses are visible from the time the intermediate form is generated by a front end. By exposing these computations, we are able to subject them to the same set of optimizations as are applied to user variables. The main disadvantage of this approach is the increase in volume of the intermediate form of a program, and the concomitant effects on compile time performance.

The ability to reuse or reorder the individual code transformations throughout compilation was also deemed to be a requirement. A clear distinction can be made between intermediate representations which can be stored in strictly

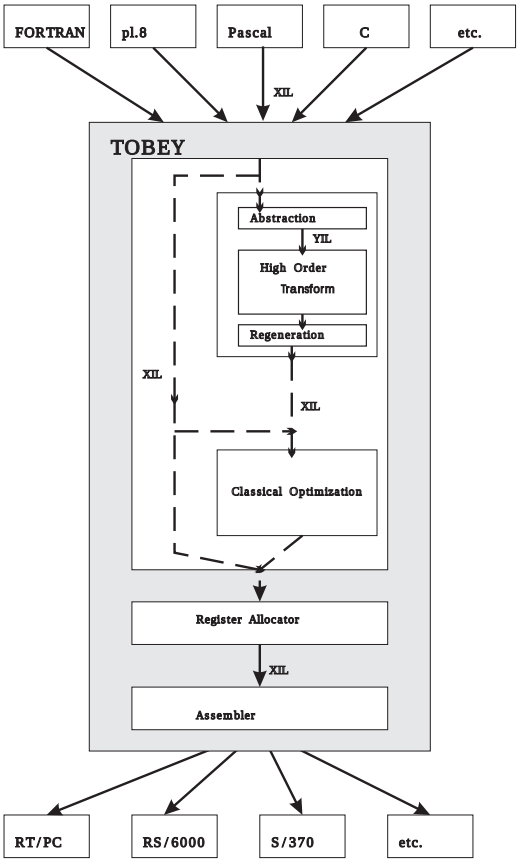


Figure 1: The Tobey Compiler

sequential form, and those wherein linkage between the objects is exposed. The former approach, usually implemented as a file interface, supports a notion of compilation wherein the discrete functions of the compiler form separate programs. Previously such designs were useful for machines with small amounts of real memory. However, this approach did not meet the objectives of an optimizing compiler, since there is no easy way to traverse the sequential text in other than its implicit order. Explicitly linked *ils*, however, lend themselves more easily to a compilation style wherein all the parts of the compiler are bound in a single module. In this approach, the *il* resides in memory, and may be analyzed, transformed and modified by each successive optimization. The objectives of the TOBEY optimizer required that *XIL* belong to the latter category.

## 3.2 Machine Model

*XIL* is a low level intermediate representation of the semantic content of a program. It is, by design, free of source language dependences and thus forms a suitable target for the compilation of a broad range of programming languages. It is not, however, of such a low character that it excessively narrows the range of instruction set architectures to which it can be reasonably translated. When considered interpretively, *XIL* presents a model of a machine with a Load/Store architecture and a number of distinct register sets. These register sets can each contain a conceptually infinite number of symbolic registers. The instructions represented by *XIL* are, by and large, those that would not be thought out of place on any respectable RISC machine, but there are a number of more exotic creatures in the menagerie, those related, for example, to string manipulation. Also, in the early stages of compilation the instructions are more flexible in several ways than those of any real machine. Displacements in addresses can be of any size; addresses can contain as many index registers as desired (Fig 2), and it is possible to specify a multiplier for each such index register (Fig 3); call instructions can have a large number of parameter registers; and instructions can have as many result registers as is convenient (Fig 4). Most of these features are eliminated by a compiler phase called *Macro Expansion*, but they afford great convenience throughout a large part of the optimizer.

```
L    r.i=i(r200,64)
M    r300=r.i,8
L    r.j=j(r200,68)
A    r310=r.j,2
M    r320=r310,400
LFL  fp330=a(r200,r300,r320,30000)
```

Figure 2: Indexed load of *a(i,j+2)*.

## 3.3 Formal Identities

Although it is often fruitful to consider an *XIL* program as a sequence of machine instructions for some abstract machine, as suggested above, there is another and perhaps more significant aspect presented by such a program. *XIL* is structured as a forest of computation trees (Fig 5), where formally

```
L    r.i=i(r200,64)
L    r.j=j(r200,68)
A    r310=r.j,2
LFL  fp340=a(r200,r.i,*4,r310,*400,30000)
```

Figure 3: Indexed load of *a(i,j+2)*., using implied multiply

```
LFLU fp350,gr400=b(gr400,8)
```

Figure 4: Multiple results: `gr400` is incremented

identical computations performed in separate locations are represented by the same node. This alternative way of looking at the program lies at the heart of some of the major optimizations and provides the basis on which the *YIL* abstraction is founded.

The concept of formal identities is important in *XIL*, and not merely as a method of containing the storage requirements of the intermediate text. The entities referred to previously as symbolic registers lie at the heart of this concept. On the interpretive level, a symbolic register can be viewed as the compiler generated temporary for holding intermediate results in the evaluation of an expression. On the other, the computation tree level, the symbolic registers are the names of the formal identities, and as such, act as links in the expression tree. Such a symbolic register is known as a *canonical target*. It is computed in the code generation routines, as the hash value of the inputs on say a load or add instruction, when the front end does not specify the name of the result register. This ensures that a load from a given variable will always produce the same result register; similarly, an add of two given registers will always produce the same result register. This alternative way of looking at the program is key to some of the major optimizations in TOBEY, of which CSE is the most notable example. In addition, it is this view which provides the basis upon which the *YIL* abstraction is founded.

However, the dichotomy engendered by these two faces of a single program (Computation Tree vs interpretive) is a fault line buried in the deepest strata of the representation. As a result the precise definition of an *XIL* program can be complicated. A program which, when naively interpreted may appear correct, can in fact be illegal because it does not respect the rule of canonicity, that any two definitions of a given register must be formally identical.

## 3.4 Representation

A number of data structures form, in aggregate, *XIL*. The procedure is the highest level construct recognized in *XIL*, and each procedure in a compilation is given a slot in the *Procedure Descriptor Table*. This record contains information about the procedure as a whole, such as the size of the stack frame, the size of the register spill area, whether the procedure kills certain global registers and so on. It also contains a pointer to the *Procedure List* for this subroutine.

The sequence of instructions which embodies, in *XIL*, the intent of the source program is represented by a circular doubly linked list, the *Procedure List*. The start of the pro-

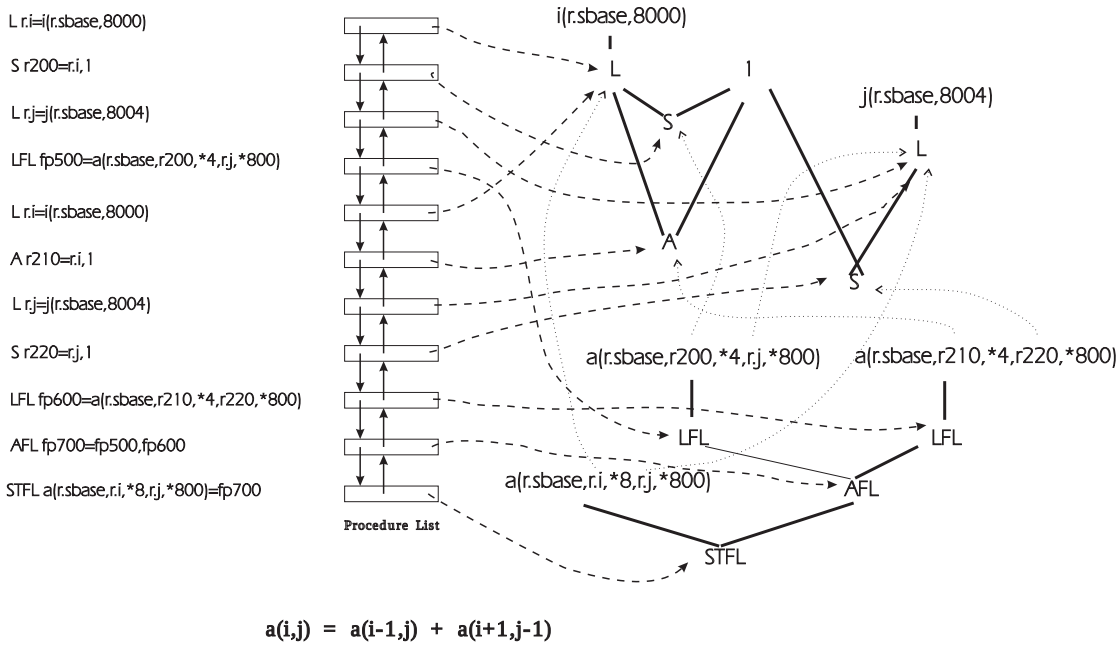


Figure 5: Expression Tree view of *XIL*. The dashed lines show the *CT* links, the dotted lines show the links to formal identities. Note the sharing of subtrees.

cedure is represented by an administrative entry, the header *HDR*. Then follows the procedure instruction *PROC*, the instructions in the body of the routine, and the procedure end instruction *PEND*. The sequence of the operations in the list represents the sequence of their corresponding machine code instructions in the final object. Each entry in the *Procedure List* contains some information peculiar to its particular location, such as the source program line number, and a pointer to another structure, the *Computation Table*, in which the actual operation performed by the instruction is encoded. Any particular encoded instruction may be shared by many elements of the *Procedure List*, as described later.

The *Computation Table*, with its subsidiary data structures, is the hub of *XIL*. It is an array of elements, called *bags*, which represent the opcodes and operands of instructions in the intermediate text. Instructions may only be entered in the *Computation Table* by calling a service routine (ct_hash), and this program ensures that each instruction appears only once in the table. An instruction is represented by a sequence of entries in the *Computation table* ( Fig 6). The first of these entries is the opcode, the remainder are operands. The opcode field encodes the number of operands that follow, and there is a set of rules which govern the ordering of operands within instructions. Each of the operands is self-describing, and represents one of several things: a symbolic register, the name of a variable, an integer value or a long or short floating point value. In general, each of these consists of some data related to the locus of the operand, and a self describing pointer to an auxiliary table. The opcode entry also follows this scheme. The auxiliary tables, the *Symbolic Register Table (SRT)*, the *Opcode Table*, the *Intermediate Language Dictionary (ILD)*, and the *Literal Value Table*, contain less frequently used information about the operand, and for the most part we can omit discussion of this material. However the *Symbolic Register Table* contains one element which is critical to understanding the structure of *XIL*. Each entry in this table points back at the unique instruction in the *Computation Table* which produces it as a result, unless the entry is for a symbolic register representing a dedicated hardware register. Because of this property, the realization of Formal Identities discussed earlier, it is possible to take a symbolic register and treat it as the root of a tree of computations. The leaves of these trees are *ILD* entries, hardware registers, integer or floating values and symbolic registers which are defined by instructions with no inputs.[1] The subroutine ct_hash ensures that each unique instruction receives a new set of result registers, except when the caller explicitly requests otherwise. In this case the generated instruction is referred to as being *non-canonical*.

## 3.5 Support for IL generation

*XIL* code is generated via a procedural interface. The routines of this interface, referred to as the *gen* routines, hide the details of the *il*'s implementation from those doing code generation. Over time, there have been changes to the actual implementation of the *il*. These routines have kept such changes from impacting the compiler front ends.

There are several tasks embedded in the code generation routines. First, there is a plethora of instruction level folding that is performed during code generation. Since the form of the *il* is constant throughout compilation, such folding opportunities are realized starting with initial generation in the front ends and continuing as optimizations make transformations on the *il*. An instruction can fold to a literal or

[1] The last of these essentially represents register temporary storage and occurs only rarely.

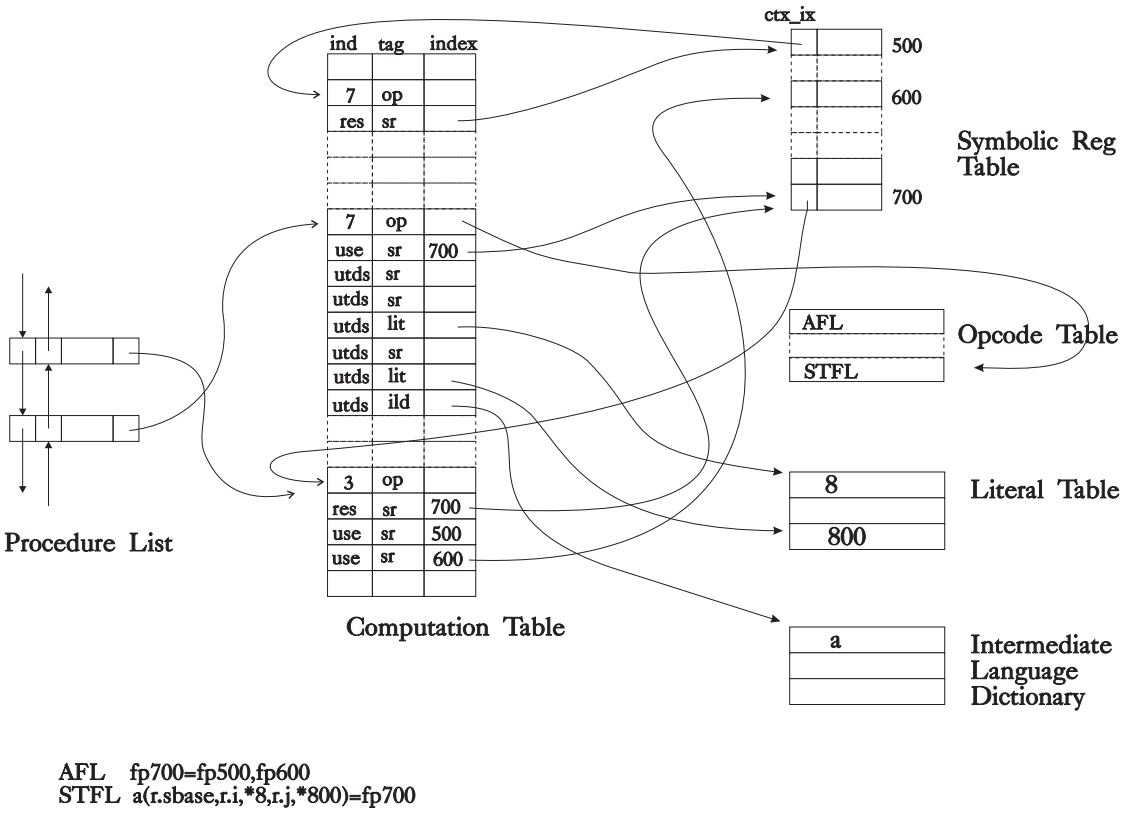


Figure 6: *XIL* Data Structures

to another instruction that is simpler or more efficient.

If an instruction does not fold to a literal, the gen routines will link the instruction into the procedure list, as well as building the computation trees (by calling ct_hash). A symbolic register result is assigned to represent the computation tree. A corollary of this style of code generation, is that the client usually does not specify the result register to the gen routine, although this is possible if needed.

Since registers and various types of literal are all represented as tagged pointers into auxiliary tables, they can each be represented as an integer and be distinguished by their tags. In most contexts, registers and literals can be used interchangeably in instructions.

The clients of the gen interface expect one or more result to be returned by the code gen routines. Since such results are typically used only as inputs into subsequent instructions, most callers need not care if the results returned are registers or literals that were created as a result of folding(Fig 7).

| Return Val | Gen Call |
|---|---|
| iv.14 | t1=gen_r2(op_M,sr_nt_gpr,iv.2,iv.7,...) |
| r.b | t2=gen_mem(op_L,sr_nt_gpr,...,ild.b,...) |
| r300 | t3=gen_r3(op_A,sr_nt_gpr,t1,t2,...) |
| | gen_mem(op_ST,t3,...,ild.a,...) |

.

Figure 7: This sequence of calls will generate code for the statement *a=b+2*7*.The parameter *sr_nt_gpr* indicates that the formal identity should be returned. Notice that the multiply is folded, and the result is transparently used in the call which generates the add.

## 3.6 Suitability for Classical Optimizations

When an optimizing compiler makes some transformation of the compiland, it frequently uncovers opportunities for further optimization that were not previously apparent. Strength Reduction for example, produces dead code. This observation motivates us to ensure that our designs include the ability to run each optimization multiple times, in various orders. The accomplishment of this task is rendered much easier if the *il* remains constant throughout the process. *XIL* is a product of this train of thought. It has the same form throughout the compiler, but changes in content as the work progresses. As generated by a front end, it contains no machine dependant operations, all redundant computation is explicit, and there is a sprinkling of higher level operations. As read by the assembler, there are no operations that require more than one machine instruction and all symbolic registers have been replaced by hardware registers. In between, a great number of fleeting variations have been produced. While this constancy of form greatly simplifies the task of ordering transformations, it should not be thought that it eliminates entirely the dependence of one transfor-

mation on another. There are very many problems in the interaction of optimizations, not all of which have satisfactory solutions.

In illustration of the preceding discussion, consider the following example. The code fragments showing the sequence of transformations are given in Fig 8 through Fig 11. Unneccessary detail has been elided in the later figures. Early in optimization we insert a register copy (LR) after each store. This copy preserves the stored value in the symbolic register which would be defined by a load from the 'stored-into' location. [2] Sometime later, the CSE optimization runs and removes, among other things, any computations which have been made redundant by this insertion (Fig 10). In our example, all the loads from the induction variable i are eliminated. This enables the Store Elimination optimization to remove the stores into the dead variable i(Fig 11). Finally, the register allocator is likely to coalesce the source and target of the register copy, thus eliminating the *LR* instruction altogether. In cases where the inserted copies do not enable CSE, a pass of the Dead Code elimination optimization will remove them. This example illustrates

```
do i=1,10
  a(i)=x
end
```

Figure 8: Source code for a simple loop

```
     ST      i=1
L1:  LABEL
     L       r.i=i
     M       r300=r.i,4
     STFL    a(r200,r300,0)=fpr.x
     L       r.i=i
     A       r310=r.i,1
     ST      i=r310
     L       r.i=i
     C       cr320=r.i,10
     BT      L1,cr320,le
```

Figure 9: Original *XIL* for the loop

several points: optimizations on *XIL* are simple, formal and separate; it is possible to get partial benefits as would be the case if the store had to remain; the mechanism of canonical [3] symbolic registers makes it possible to insert *LR* ops on speculation.

It has been noted [16] that for some optimizations, the greatest opportunities are found in the very low level operations which are the constituents of the source statements. Such things as address computation, loading of data in registers and so forth may be repeated in different statements, or afford other opportunities for optimization. For this reason, *XIL* was designed to expose these low level features. Analogously, high level loop and conditional constructs are represented in *XIL* by simple comparison, conditional branch and label instructions. Information about loop structure and induction variables is obtained from the code by control and data flow analyses.

[2] Of course, this definition of the target register is not formally identical to its original definition, and so a certain amount of care must be taken by routines which expect the program to conform to expression tree semantics.

[3] Satisfying formal identities

```
     ST      i=1
     LR      r.i=1
L1:  LABEL
     M       r300=r.i,4
     STFL    a(r200,r300,0)=fpr.x
     A       r310=r.i,1
     ST      i=r310
     LR      r.i=r310
     C       cr320=r.i,10
     BT      L1,cr320,le
```

Figure 10: *XIL* after inserting copies and *CSE*

```
     LR      r.i=1
L1:  LABEL
     M       r300=r.i,4
     STFL    a(r200,r300,0)=fpr.x
     A       r310=r.i,1
     LR      r.i=r310
     C       cr320=r.i,10
     BT      L1,cr320,le
```

Figure 11: *XIL* after Store Elimination

It is instructive to look at how several of the important optimizations are performed on *XIL*. Dataflow analysis is at the heart of very many optimizing transformations. In the TOBEY compilers the values for which this analysis is done are the Symbolic Registers. Each register within the region of interest, is assigned a position in a bit-vector, and logical operations on these bit-vectors represent the set operations required to solve the dataflow problem at hand. Kill sets for store instructions are also represented by the same type of bit-vector. To compute some interesting set, say the set of *available expressions* used in common sub-expression elimination, is relatively simple. First the inputs to the dataflow equation for this problem, for example the *downward exposed* expressions, are constructed by walking each basic block. The forward and backward links in the procedure list ensure that this is a painless task. At each instruction, the computation table entries are scanned, and every register result is added to the set, while for store instructions, the *kills* are removed. Having computed this, and all the other inputs to the problem, we solve for *available expressions* using interval analysis or iteratively, as appropriate. Given this set, it is straightforward to again walk each basic block and delete instructions whenever all of their results are already available. In the same way, the set *insert* of instructions to be inserted in a block by code motion can be computed. To insert an instruction, given the symbolic register, is trivial in *XIL*. It suffices to link in a new procedure list element and to set its computation table pointer to the canonical definition of the register.

CSE is the prime example of an optimization which re-

lies on the expression tree aspect of *XIL*, since it depends on the fact that formally identical computations are always represented by the same symbolic register. The major exponent of the interpretational, or virtual machine, aspect of the representation is the optimization known as Value Numbering. The core function of Value Numbering is to discover computations in the program which redundantly compute the same value, even when these computations are quite different in form. The program improvement results from replacing the later computation with a copy from the result of the earlier. This process is accomplished by inserting register copy operations into the instruction sequence. Note that this effectively destroys the expression tree interpretation of the program, because it introduces a different path for the computation of a symbolic register than that implied by the canonical definition. Any optimization that recomputed this target canonically, would at best be undoing the results of Value Numbering, and in many cases might cause an incorrect code sequence to be generated. This conflict is the major tension in our compiler, and has never been satisfactorily resolved. All transformations must ensure that they treat such register copies with the circumspection they require, and this leads on occasion to a small loss of performance.

# 4 YIL

## 4.1 The Evolution of YIL

The design of *XIL* was driven by the clear objective of facilitating the production of highly optimal code, by means of classical program optimizations. As work on the optimizer neared completion the desirability of enhancing its capabilities by the addition of techniques designed to exploit locality of reference became evident [7, 14]. Loop transformations such as unrolling, skewing, and tiling used in this type of optimization require a different type of analysis. The low level of representation found in most scalar compilers is usually not suited to such tasks as extracting the data dependence information from array accesses at the level of the individual subscripts. Indeed, the received wisdom in writing such optimizations, is that the program representation used should be at a very high, almost source, level. There are certainly many arguments in favor of this approach, not least that the greater level of detail in a low level intermediate language increases the compile time and space required to perform analysis and transformation. It was however our desire to incorporate these new optimizations into the framework of the existing compiler, which uses *XIL*, as its intermediate representation. It was also expected that the new functionality would be supported for all source language front ends, and so a source level intermediate form, which is difficult to design in a language independent way, was felt to be inappropriate.

These considerations, combined with a desire to reuse existing code in writing the new optimizations, where that was practicable, led us to a design for *YIL* which can best be thought of as an abstraction of *XIL*. Building on the expression tree aspects of *XIL*, we designed a language which could be analyzed and manipulated in terms of high level constructs such as loop statements and explicitly indexed array assignment statements, while not losing the underlying detail as represented by *XIL*. Statement level information, and in particular source level array subscripting information can be very easily derived from the structures with which *XIL* represents a program.

Hence the first phase of the TOBEY locality optimizer performs the task of abstracting, from the *XIL* encoding of the program, this higher level representation which encapsulates precisely the information required for program restructuring through loop transformations. Specifically, the higher level representation, known as *YIL*, represents the program as a statement graph (Fig 12), wherein control flow, loop and array indexing information, and *Static Single Assignment (SSA)* form are all embedded; most pertinent however, is the fact that each assignment statement represented in *YIL* has a pointer to the Computation Table entry for the series of computations which form the lower level representation of the statement. It is this link which is the key to the success of the *YIL* abstraction, in that it provides the ability to exploit or ignore the lower level computation details as appropriate to the type of transformation.

## 4.2 Architectural Principles

Unlike *XIL*, which was designed to expose the minutia of program execution, *YIL* is aimed squarely at the representation of overall structure. This principle is apparent both in the amount of computation represented by the individual statements, and by the level of abstraction represented within the statements. The statements of *YIL*, rather than standing in for primitive operations, such as loads and adds, indicate quite complex tasks, such as the assignment of a complete expression to a variable or the execution of a multidimensional loop. Within a statement making subscripted references the individual subscript expressions are explicitly laid out and, for loop statements, details of the iteration space and any transformations to be applied to it are recorded. Since the formal identities of *XIL* already provided a convenient way of encoding expressions, there was little point in duplicating this mechanism in *YIL*. Each reference to an expression in a *YIL* statement is represented by a reference to the appropriate formal identity. While this approach is sufficient to take care of the arithmetic details of expression evaluation, for the purposes of the derivation of dependence information it is inconvenient to have to parse expression trees. For this reason, all *YIL* statements directly contain information about the uses and defs of variables that they entail. Associated with each such use or def datum is the set of subscript expressions required to fully resolve it. Once again, these subscript expressions are indicated by the appropriate formal identity. Overall the effect is to separate the details of program execution from the structural and dependence elements. The former is required mainly when rewriting the *XIL* version of the compiland, the latter are the essentials of dependence analysis and the restructuring transformations.

## 4.3 Representation

A closer look at *YIL* reveals a program encoded as a doubly linked list of records. The principal statement types are *assign, if, call,* and *loop*. Each of these represents an abstraction of the lower level details of the underlying *XIL* program. In addition, the representation of the *if* statement record in *YIL* (Fig 13), encapsulates the control flow information of the program. Each *if* statement has two or more

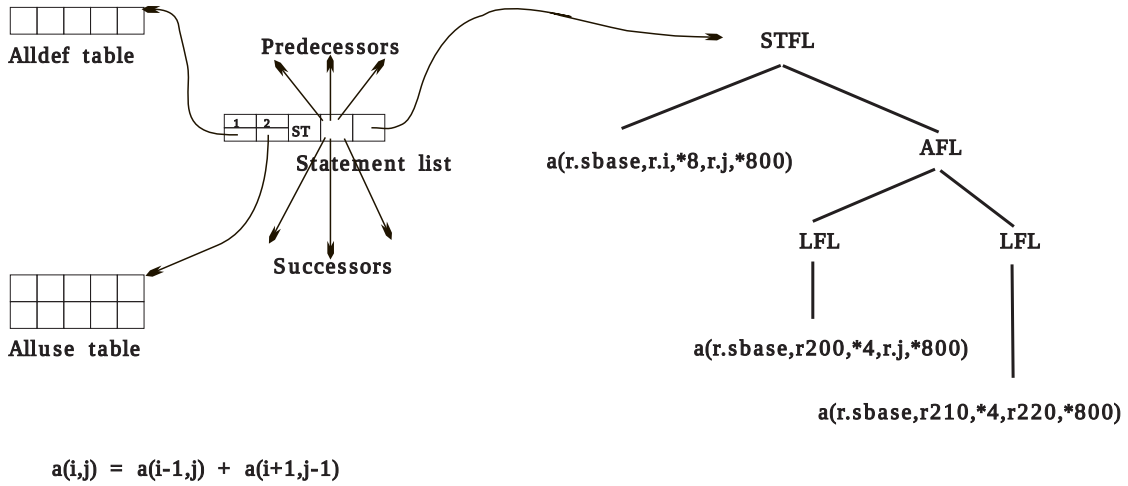


Figure 12: *YIL* Statement Graph Entry

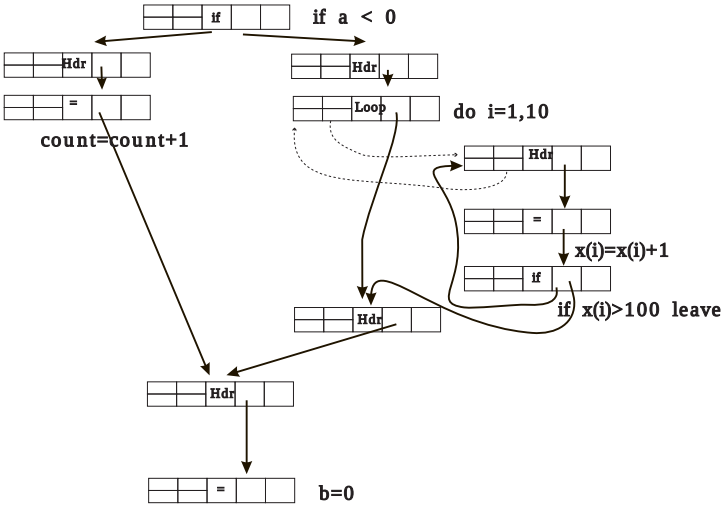


Figure 13: Branching code in *YIL*

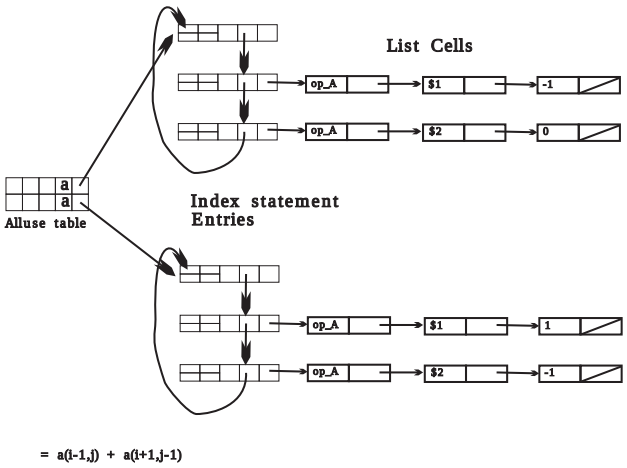


Figure 14: A *YIL* Indexed Assignment Statement

successors such that the *YIL* statements form a statement graph rather than a statement *list*.

A brief look at the *assign* and *loop* statements in *YIL* will serve to demonstrate the functionality and flexibility which it exploits to successfully perform many high level program transformations. The assign statement is a powerful abstraction. It is analogous to the corresponding source level assignment operation, and represents an abstraction of all the low level computation which culminates in committing a value to memory. It also encodes the SSA representation, and records subscript information for those statements which are array references. The most significant fields in the assign statement are the statement type, a left hand side and a right hand side, and an index into the computation table. This points to the underlying *XIL* store instruction, which forms the root of the computation tree for the whole expression.

The left hand and right hand sides allow the incorporation of SSA form into the statement graph [10]. A program in this form has the characteristics that each programmer specified use of a variable is reached by exactly one definition. To ensure that this is so, *phi* statements are inserted at confluence points in the statement graph. Information about the statement's effects on the program's variables is recorded in the statement graph. For each *use* and *def* in a statement, information is recorded in the *alluse* and *alldef* tables (Fig 12). In effect, the statement graph incorporates the *use-def* chains for the program, thus simplifying the analysis task of subsequent components, such as dependence and induction variable analysis. These tables also record the relevant indexing information for array variables (Fig 14).

Just as the assign statement is an abstraction of much lower level detail, so too, the loop statement replaces en-

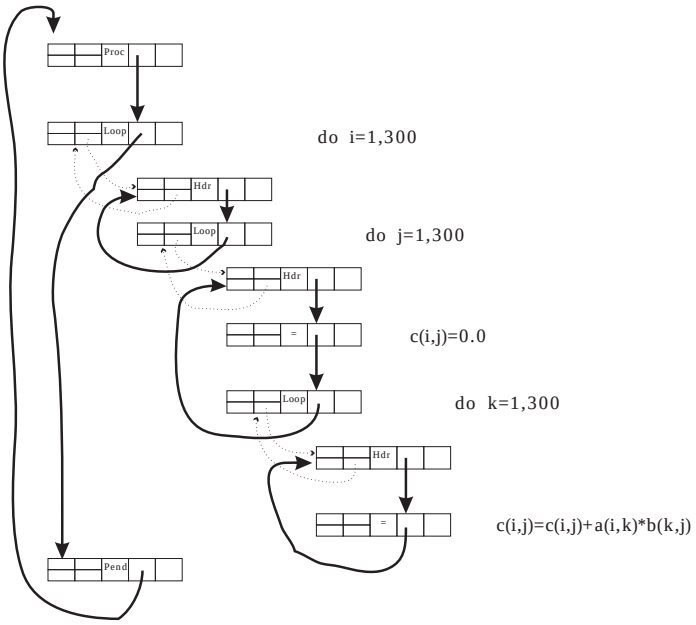


Figure 15: Matrix Multiply in *YIL*

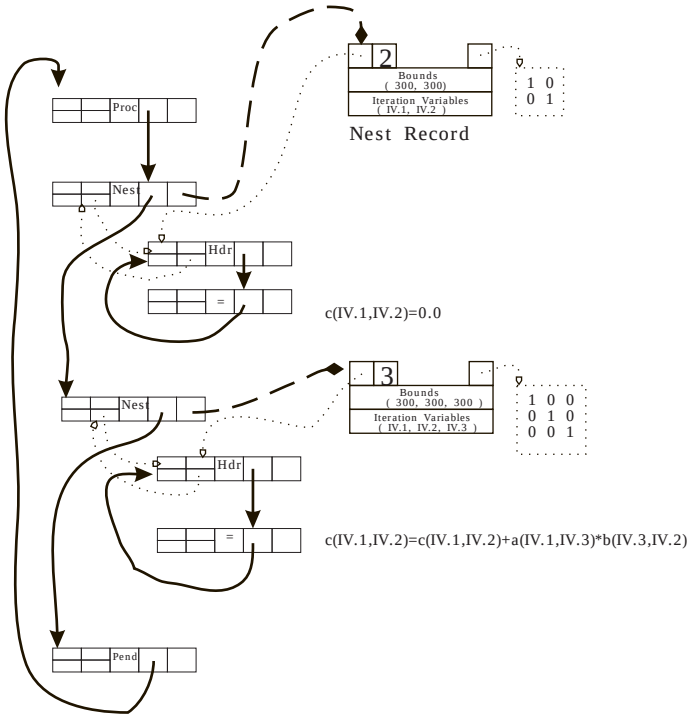


Figure 16: Matrix Multiply after Loop Distribution and Nest Extraction

tirely the body and control elements of an *XIL* loop. Loop bodies are hierarchically linked to their corresponding loop statements, permitting easy traversal both inwards and outwards. This high level loop statement has a single successor, namely the next statement after the loop (Fig 15). Another field in the loop statement record indexes the loop table entry which refers to all the information pertinent to the loop transformations, such as the normalized upper bounds, a vector of normalized induction variables, and the nesting depth (Fig 16). The implementation of the loop optimizations require that the program be represented as a series of either *perfect* or *imperfect* loop nests. To this end *loop distribution* is performed early in the transformation process. After loop distribution and nest analysis, the outermost loop statement refers to a loop *nest*, and the details of this are in a *nest record*. The information in the *nest record* contains in addition to the details of the nest body, the *transformation matrix* to be applied to this particular nest.

## 4.4 Abstraction from *XIL*

The process of abstraction from *XIL* to *YIL* consists of several steps. Firstly, the *XIL* Procedure List is traversed and each store, branch and call type instruction causes the insertion of a statement node into the *YIL* Statement Graph. During this process the flow graph, built over the *XIL* version of the program, is consulted for information on branch targets and for loops. As each statement is added, the computation trees which are rooted in it, are examined to discover the variable names to be entered into the alluse and alldef tables, and the indexing expressions, which give rise to separate subscript statements. Having performed this basic translation, the second phase involves identification of locations where so-called phi and anti-phi functions are to be inserted. Once these locations are determined for each variable, a corresponding function is inserted into *statement graph* at each location. Now the conversion of *YIL* to SSA form is completed by creating a new variable name for each variable at every definition site. This name is used in subsequent references to the variable until the point where the variable is again redefined.

Now induction variable analysis is performed and higher-level loop constructs are created in a form suitable for the analysis that occurs later in the High-Order Transformation phase of the back-end. Finally, the loop table entries are filled in and all the inductive loops of the program are normalized.

## 4.5 Updating *YIL*

The completion of these steps renders the program into a state suitable for the application of dependence analysis and loop distribution which recasts the program as a series of maximally distributed loop nests, annotated with their concomitant dependence information encoded as distance vectors. The way is thus cleared for the loop transformation phase. An obvious side effect of rendering the program into this form, indeed one that is encountered with each phase of the transformation process, is that many components of the *YIL* data structures become out of date. Two obvious examples are the data dependence information and the SSA representation. Indeed, one of the challenges of using SSA is keeping it up to date as the program is transformed. Rather than rebuild all the use-def chains that comprise SSA after each optimization step, we update the chains incrementally.

Incremental rebuilding is possible for transformations that are localized and that do not add control flow (which could require new *phi* function placement).

For example, in the loop normalization step, basic induction variables in the loop body are replaced with their normalized form, $I \Leftarrow (I_c - 1) * bump + init$, where $I_c$ is the loop controlling induction variable, *bump* is the linear increment of $I$, and *init* is the initial value of $I$. Although *bump* and *init* may be arbitrarily complex symbolic expressions, they must be invariant in the loop body, and therefore have a single definition outside the loop that dominates the loop entry. This makes it straight forward to update SSA incrementally as loop normalization proceeds, since all the new uses of *bump* and *init* have well-defined use-def relationships. Further, no extra *phi* functions are needed in the loop body, even if it contains control flow, because the added uses are all invariant in the loop.

This same type of reasoning can be applied to dead code removal and GCP, since these optimizations can only remove uses from the SSA structure. By treating the use-def chains as linked lists of references, dead entries can easily be removed from the middle of the list. As a result of these considerations, the current production version of TOBEY only builds the complete SSA representation once per compilation unit.

## 4.6 Suitability for Restructuring Optimizations

The reordering of the iterations of multi-dimensional loops is the central concern of restructuring compilers. This reordering may be undertaken for several different ends, most commonly the extraction of parallelism from programs or the exploitation of reuse at some level of the memory hierarchy. In recent years a popular approach to this problem has been the use of unimodular transformations, and *YIL* was designed with this method in mind. The *YIL* loop statement has associated with it a unimodular transformation matrix, of the appropriate dimensionality, and a vector containing the loop index variables. The optimization routines of the compiler, after judicious analysis, reflect their decisions in the transformation matrix. It then remains to realize these intentions in the form of a rewritten program. In order to achieve this, it is necessary to modify the loop bounds and any inductive subscript expressions used by the statements in the loop body. The fact that this is a relatively straightforward task in our compiler, assures us that *YIL* is indeed appropriate to its intended use.

How is this task accomplished? In essence, the required changes reduce to the problem of rewriting a set of expressions. The variables in which these expressions are rooted have been transformed in a way that is readily derived from the unimodular matrix. Given each of the original variables and the expression, in terms of the new induction variables, to which it is transformed, we must take the old expression tree and replace each of its roots by the corresponding transformed quantity. By keeping a mapping between the symbolic registers corresponding to the new and the old formal identities, it is simple to walk the old expression tree, and, using the *XIL* gen routines, regenerate each *XIL* instruction by replacing its inputs with the new values. When a complete expression has been processed, the new formal identity can replace the old in the appropriate field of the *YIL* statement it originated from. Essentially the same technique can be used to rewrite the *XIL* version of the program.

## 4.7 Advantages of having *XIL* embedded

From the preceding discussion, it may be inferred that the manner in which *YIL* was derived as a higher level abstraction of *XIL*, was driven by the exigencies of product development. To some extent this is true, but we would argue that in fact *YIL* steers a middle course between the two possible alternatives; namely to perform the high order transformations on the lower level *XIL*, an option which was considered, or to design a totally new, high level intermediate representation.

In fact, many benefits accrue from having the lower level representation available. Firstly, the flexibility exists to perform lower level optimizations as required. In some cases it is beneficial to reorder the *XIL* components of an operation in order to break dependences. We refer to this as *node splitting*. Since in *YIL* we have full access to the underlying loads/stores, complete with a defined order of evaluation, we are able to perform such a transformation by updating the underlying *XIL* representation. Secondly, the ability to reuse existing code is demonstrated in the dependence analyzer implementation. Allusion has been made above to the fact that array index information is stored in the alluse and alldef tables, accessed from a field in the statement graph. In fact the *index* field in this structure (Fig 14) is the head of circular linked list of statement records identical to those in the main body of the statement graph. Hence, every array index statement has a pointer to its underlying computation tree. Given this fact, we were able to exploit the existence of a significant piece of code in the existing TOBEY compiler, which deals with manipulation of array index expressions. Following the approach used in the Reassociation optimization, wherein expression trees are built in order to apply the associative laws of arithmetic, we build a symbolic expression for each array subscript contained within a loop. The expression list consists of the canonical form, in *XIL* representation, of the subscript expression. Typically the expression list for a subscript will consist of an add, of a positive or negative constant value to a register expression which may or may not represent the loop index variable. For ease of reference later on, we annotate those expression lists which do contain induction variables by replacing the canonical register representing the induction variable, with a pointer to the alldef entry for this subscript in the SSA of the program. A pointer to the expression list itself is stored in the node in the statement graph for the index statement. It is these expression lists, then which are manipulated by the dependence tester to form the equations whose solutions provide the distance vector information.

# 5 Experiences

It may be argued that the true test of flexibility in an *il* is the ease with which unanticipated features may be added to the compilers it supports. By this touchstone both *XIL* and *YIL* prove golden. For each of them, we give an example to bolster this assertion.

When *XIL* was first mooted it seemed that 32-bit addressing and 32-bit integer data would be sufficient for computing needs into the distant future. The compilers were designed, and constructed with an implicit understanding that

the *GPRs* of the virtual machine were 32 bits long. When 64-bit machines began to appear (*Mips R4000, PowerPC 620*), it became necessary to consider support for them, and for 64-bit language constructs compiled for 32-bit architectures. This support has been successfully added to *XIL*. When we added 64-bit support, we had two main goals. First, we wanted to minimize the impact on existing code in both the front ends and TOBEY itself. Secondly, we wanted to hide the 64-bit implementation from the front ends so that they need not be aware of the size of gprs during *il* generation. Thus, for them 64-bit data becomes just a language issue. By having front ends always think of registers as 64-bits and adding a few more opcodes to work on these registers, we've largely been able to achieve these goals. Now, as the *il* is being generated, if the actual target is 32-bits, we detect where 64-bit data is actually used and modify the *il* to represent this with 2 symbolic registers. One side benefit of this effort has been to allow us to support 64-bit data in our existing compilers on 32-bit machines.

Our data structures for representing the *il* have served us well in this effort. The way we represent registers did not change. Our integer values are stored in our Literal Value Table which was widened to handle 64-bit values. This abstraction minimized the impact to those few places that actually manipulate such values at compile time (e.g. constant folding).

*YIL* was developed to allow TOBEY to perform memory hierarchy optimizations, principally *tiling*, though the possibility of automatic extraction of parallelism for shared memory machines was entertained. At a later time, there was some interest in having the compilers produce code for a vector architecture. This capability was prototyped in the *YIL* based transformer. Although the extraction of vector parallelism has much in common with the optimizations that *YIL* was designed for, it also requires a more detailed representation of the operations of the machine. Fortuitously, the fact that *XIL* is embedded in *YIL* allows for the required level of detail.

A further example of the useful co-existence of *XIL* and *YIL* is the *predictive commoning* [15] optimization of TOBEY. While initially prototyped in *XIL* with some degree of success, it was felt that this optimization could be more optimistic when applied at the level of *YIL*. Consider the following loop:

```
DO i = 2, 100
   a(i) = b(i-1) + b(i) + b(i+1)
END DO
```

Notice that the reference $b(i+1)$ in iteration $I$ becomes $b(i)$ in iteration $I+1$ and $b(i-1)$ in the iteration after that. Predictive commoning can exploit this reference pattern by transforming the loop as follows:

```
t1 = b(1)                          ! Initialize temps
t2 = b(2)
DO i = 2, 100, 3                   ! Unroll by 3
      t3 = b(i+1)                  ! Load the leader
      a(i) = t1 + t2 + t3
      t1 = b(i+2)
      a(i+1) = t2 + t3 + t1        ! Cycle left by 1
      t2 = b(i+3)
      a(i+2) = t3 + t1 + t2        ! Cycle left again
END DO
```

Note that the original loop has 9 loads every 3 iterations, while the transformed loop has only 3. Note also that the optimization involves both high and low level transformations: at the high level, the loop is unrolled 3 times; at the low level, the references to $b$ are replaced by references to the temps.

Predictive commoning is applied in three steps. In the first step, the *YIL* representation of the loop is used to search for opportunities - *YIL* make this easy because subscripts are represented explicitly. The second step applies cost/benefit analysis to the opportunities discovered in step 1, and determines the unroll factor. Again, *YIL* makes this easy since the unroll factor is simply a parameter in the *nest* record. Finally, the transformation is applied by rewriting the underlying *XIL* of the loop and its body. Here the canonicity property of *XIL* is used to identify the references to $b$ so that they can be replaced efficiently.

## 6 Future Directions

The evolution of *YIL* thus far has demonstrated the feasibility of performing high order transformations at other than the source language level. *YIL* is a high level intermediate representation yet it easily exploits the details contained in a lower level form. The Predictive Commoning optimization described in the previous section, demonstrates this nicely. Moreover, this optimization, which can be done equally well on either *XIL* or *YIL*, exemplifies a key topic for further study in this area, namely to what extent could existing optimizations exploit the type of information available only in *YIL*. In particular, late optimizations such as software-pipelining and scheduling require good dependence information for optimal performance. Unfortunately, by the time these phases run the *XIL* is at a very low level: addressing trees have been linearized, and the canonical symbolic registers may have been replaced by non-canonical hardware registers. As a result, it is difficult to extract detailed dependence information. In contrast, the dependence information acquired from the *YIL* representation of the program is highly precise, since at that time all registers are canonical and subscripting is represented explicitly.

The challenge is to propagate the high-level dependence information into *XIL* so that the late low-level optimizations of TOBEY can take advantage of it. This task is made more challenging by the fact that *YIL* is used early in the compilation phase; the program can look radically different by the time it reaches the scheduler. We have thought of tagging the *XIL* load and store references with the dependence information gleaned from *YIL*, but care must be used to preserve the canonicity of the references. If canonicity is not preserved, many intermediate optimizations may be partially or completely defeated. We have also thought of tagging the loop latches with summary information about the dependences in the loop; this approach is less precise but may still allow some optimizations to benefit.

## 7 Conclusion

The rapid pace of competitive hardware development, has meant that a good compiler must address not only the classical optimizations, but must also be capable of exploiting the architectural features of the most high performance superscalar machines, through techniques such as loop restructuring, instruction scheduling and instruction and data prefetching. We feel that the TOBEY optimizer has maintained

a competitive edge with respect to performance on a variety of target architectures. Moreover, the ability to host a variety of source languages has not been sacrificed; on the contrary, it has been expanded with the addition of C++ to the three already existing. This success is due in no small part to the design of the intermediate representation, *XIL*: it allows for the addition of new front ends with relative ease; it shields those front ends from the specifics of the multiplicity of machine architectures which it targets; it facilitates the application of the classical optimizations in order to produce highly optimal scalar code; and it was easily extended to permit the array and loop analyses which feed the higher level loop optimizations.

## 8 References

## References


[1] *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation.

[2] Wall, D. *Experience with a software defined machine architecture.* In ACM Transactions on Programming Languages and Systems Volume 14, no 3 July 1992

[3] Strong, J., Wegstein, J., Tritter, A., Olsztyn,J., Mock,O. and Steel,T. *The problem of programming communication with changing machines: A proposed solution.* In Communications of the ACM, Aug 1958, pages 12-18 and Sept 1958. pages 9-15

[4] Blelloch, G.E. and Chatterjee, S. *Vcode: a data parallel intermediate language.* In Proceedings of Third symposium on the Frontiers of Massively parallel computation. 1990, pages 471-80

[5] Tjiang, S., Wolf, M., Pieper, K. and Hennessy, J. *Integrating scalar optimization and parallelization* In Languages and Compilers for Parallel Computing Fourth International Workshop 1992, pages 137-51

[6] Wolf, Michael E. and Lam, Monica S. *A Loop Transformation Theory and an Algorithm to Maximize Parallelism.* In Journal of Parallel and Distributed Programming October 1991

[7] Wolf, Michael E. and Lam, Monica S. *A Data Locality Optimizing Algorithm.* In Proceedings of ACM Sigplan Conference Toronto, June 1991

[8] Banerjee, Uptal. *Unimodular Transformations of Double Loops.* In 3rd Workshop on Languages and Compilers for Parallel Computers

[9] Ferrante, Jeanne., Ottenstein, Karl J. and Warren, Joe D. *The Program Dependence Graph and its use in Optimization.* In ACM Trans. on Programming Languages and Systems

[10] Cytron, Ron., Ferrante, Jeanne., Rosen, B K., Wegman, Mark. and Zadeck, F K. *Efficiently computing static single assignment form and the control dependence graph.* In ACM Transactions on Programming Languages and Systems

[11] Aho,Alfred V., Sethi,Ravi. and Ullman,J.D. *Compilers: principles, techniques and tools.* Addison-Wesley

[12] Nori,K.V., Ammann,U., Jensen, K., Nageli, H.H. and Jacobi, Ch. *The PASCAL P compiler: Implementation notes.* In Technical Report 10 ETH Switzerland Revised October 1976

[13] Goff, Gina., Kennedy, Ken. and Tseng, Chau-Wen. *Practical Dependance Testing.* In Proceedings of ACM Sigplan Conference Toronto June 1991

[14] O'Brien,J.K. and O'Brien, Kathryn M. *The Implementation of Locality Optimizations in the RS/6000 FORTRAN Compiler.* In The Proceedings of the 1992 IBM Programming Languages ITL. Toronto, June 1992

[15] O'Brien,J.K. et al *Advanced Compiler Technology for the RISC System/6000 Architecture* In IBM RISC System/6000 Technology, SA23-2619, IBM Corporation.

[16] Auslander, Marc. and Hopkins, M. E. *An Overview of the PL.8 Compiler.* In Proceedings of ACM SIGPLAN Conference Boston, June 1982, pages 22-31

[17] Hopkins, M.E. *Compiling for the RT PC Romp.* In IBM RT Personal Computer Technology SA23-1057