Designing the Agassiz Compiler for Concurrent Multithreaded Architectures

B. Zheng[1] J.Y. Tsai[2] B.Y. Zang[3] T. Chen[1] B. Huang[3] J.H. Li[3] Y.H. Ding[3]

J. Liang[1] Y.Zhen[1] P.C. Yew[1] C.Q. Zhu[3]

[1] Computer Sci. and Eng. Department, University of Minnesota, MPLS, MN 55108

[2] Hwelett-Packard Company, Cupertino, CA 95014

[3] Institute of Parallel Processing, Fudan University, Shanghai, P.R. China

Abstract

In this paper, we present the overall design of the Agassiz compiler [1]. The Agassiz compiler is an integrated compiler targeting the concurrent multithreaded architectures [12][13]. These architectures can exploit both loop-level and instruction-level parallelism for general-purpose applications (such as those in SPEC benchmarks). They also support various kinds of control and data speculation, runtime data dependence checking, and fast synchronization and communication mechanisms. The Agassiz compiler has a loop-level parallelizing compiler as its front-end and an instruction-level optimizing compiler as its back-end to support such architectures. In this paper, we focus on the IR design of the Agassiz compiler and describe how we support the front-end analyses, various optimization techniques, and source-to-source translation.

1. Introduction

VLSI technology will allow hundreds of millions of transistors on a single chip in a couple of years. Exploiting parallelism at various granularity levels on a single chip for higher performance will soon become a reality, and even a necessity, for both uniprocessors and multiprocessors. As a matter of fact, with the recent introduction of concurrent multithreaded architectures [12][13], the line dividing traditional uniprocessors and multiprocessors for single-chip architectures has gradualy disappeared. A lot of parallel processing technologies, including parallelizing compiler technology, can be leveraged and extended to the more general-purpose applications typified by the industry-standard SPEC benchmark programs. In those applications, the dominant languages are still Fortran and C because of their requirement of high performance.

The traditional parallel processing technology has primarily been focussed on scientific applications dominated by a large number of floating operations and well-structured arrays and loops. To extend this technology to general-purpose applications, we need to face several new challenges which are common in those applications. They include: (1) a large number of integer operations; (2) a large number of pointers and very flexible data structures; (3) many *do-while* loops and *do-across* loops; (4) many loops with a small number of loop iterations, or a small loop body. Many of these challenges cannot be met by the compiler or the processor architecture alone. It requires a careful study and evaluation on the tradeoff between architecture and compiler design. It also requires aggressive exploitation of both instruction-level and loop-level parallelism.

In this paper, we focus on the design of the Agassiz compiler, which is an integrated compiler targeting concurrent multithreaded architectures (such as Multiscalar [12] and Superthreaded processors [13][14]). These architectures support both loop-level (LLP) and instruction-level (ILP) parallelism. They also support various kinds of control and data speculation, run-time data dependence checking for complicated pointers and data structures, and very fast communication and synchronization mechanisms.

The Agassiz compiler has been designed with several goals in mind:

- Support both loop-level and instruction-level parallelization. It includes a front-end loop-level source-to-source parallelizing compiler, and a back-end instruction-level optimizing compiler. Each of those compilers has its own intermediate representations (IR). A mechanism (see Section 2) is provided to integrated both front-end and back-end compilers to produce highly optimized code. This mechanism also allows other front-end and back-end compilers to be integrated into the Agassiz compiler.

- Support multiple languages, such as C and Fortran. It has two levels of intermediate representation (IR) in the front-end: the Raw-IR which is more language-specific, and the Real-IR which is more language independent. They facilitate program analysis and optimization while maintaining useful language-specific information for more efficient analysis and optimization (see Section 2).

- Support large applications, such as SPEC benchmarks, which might have hundreds of procedures and tens of thousands of lines of code. Time- and space-efficient algorithms have to be used to control the total analysis time and the IR size. Furthermore, extensive interprocedural analysis and optimization have to be performed.

- Support end-to-end compilation from the source code to the final target machine code generation for high performance. Potential overhead incurred during the loop parallelization, such as during induction variable optimization and loop transformation, are marked in the IR. This allows such overhead to be removed and the loop cleaned up when it is determined to be unparallelizable in later compiler passes.

- Provide robust and easy-to-use components to support collaborative development effort. Our kernel has been designed with a high-level intermediate representation, an interface to back-end compilers, and layers of functionality. Data structures in the kernel are encapsulated by C++ classes. Any access to these data structures is under the control of a consistency-maintenance mechanism. We also add a built-in pointer-specific error detection mechanism to avoid potential dangling pointers and memory leakage. Layers of functionality make the kernel easy to use for those who work on the compiler.

In this paper, we present the overall design of the Agassiz compiler with an emphasis on (1) the design of its IR to facilitate front-end analyses, optimizations, and source-to-source translation, and (2) its analysis and optimization techniques to parallelize C and Fortran programs for concurrent multithreaded architectures.

The remainder of this paper is organized as follows. In Section 2, we present an overview of the Agassiz compiler. Section 3 addresses some issues in designing our program IR, while Section 4 discusses its implementation issues. We describe the implemented analysis and optimization techniques which are important to support concurrent multithreaded code generation in Section 5. Some related works are discussed in Section 6. Finally, Section 7 draws conclusions and discusses ongoing works.

2. Overview of the Agassiz Compiler

As shown in Figure 1, the Agassiz Compiler currently has six major components: (1) a parser to generate a language-independent intermediate representation (Real-IR) from source; (2) a front-end analyzer; (3) a parallel source code generator targeting concurrent multithreaded processors, or shared-memory multiprocessors; (4) an unparser to generate annotated source from the IR; (5) a high-level information (HLI) generator; and (6) a back-end ILP (instruction-level parallelism) optimizing compiler (based on gcc).

The parser first converts a C or a Fortran program into a language-specific intermediate representation, called Raw-IR. It uses a high-level abstract syntax tree that preserves all the language dependent features in the program. The main purpose of using Raw-IR is to separate the language-dependent parsing and semantic checking from later analyses and transformations. It then tries to eliminate some but not all language specific constructs, such as the side-effect expressions in C programs and the *entry* statements in Fortran programs, and convert the Raw-IR to a more language-independent internal representation, called Real-IR. During the process, it also tries to "normalize" the intraprocedural control flow in the programs to facilitate later analysis. For example, it will convert a code section with a backward *goto* statement into a loop structure; rearrange code segments to reduce the number of *goto* statements; eliminate *entry* statements as mentioned above. It also links global symbols used in different source files.

The front-end analyzer then performs interprocedural pointer analysis and scalar data flow analysis. It also performs scalar optimizations to improve the code, and to prepare for the later parallelization. These optimizations include control-flow and data flow dead code elimination, constant folding, constant propagation, copy propagation, pointer-dereference expression simplification, non-zero-iteration loop recognition, loop invariant code motion, and induction variable substitution. The output of the front-end analyzer is an optimized program with annotated data flow information, which could then be used in later phases.

The parallelizer, which currently is targeted for concurrent multithreaded architectures, performs further analyses to generate parallel codes. The analyses performed in this stage include privatizable scalar analysis, data dependence analysis, and Doall and Doacross loop analysis. It will then generate parallelized code for the target machines. The parallelized code, still in its Real-IR form, can be either fed into the unparser or the HLI generator. The unparser generates an annotated source program. The HLI generator, on the other hand, exports important high-level information, such as alias information, loop-carried data dependence information and interprocedural Ref/Mod information to a file with a standardized format, called HLI (High-Level program Information) [4]. It can then be imported into and used by the back-end ILP optimizing compiler.

The back-end ILP compiler is currently a modified version of gcc. It first parses the annotated source program and generates its internal representation RTL. It then augments its RTL with the imported HLI information. An extensive utility library has been developed to facilitate the use and the maintenance of HLI [4]. It provides utility functions to inquire and to update the associated HLI without the need to know the detail of the HLI format. The analysis and optimization algorithms in the original gcc are modified so that they can operate on the extended RTL and utilize the HLI information. The back-end optimizer finally produces executable codes for the target machines, currently an execution-driven simulator based on the SimpleScalar [3] for the Superthreaded processors and many existing superscalar processors.

3. The Design of IR

The target programming languages as well as the intended compiler analyses and optimizations drive the design of the compiler internal representation. In this section, we focus on issues related to the IR (i.e. Real-IR) design for front-end analyses and optimizations. We also address how our IR supports multiple languages and some software engineering considerations.

3.1 Choosing an Appropriate Intermediate Representation Level

We use abstract syntax trees (AST) to represent the statements and the expressions in a program. The AST representation allows us to represent high-level control structures, such as loops and branches, explicitly by their components. For example, we represent a do-loop by its index, step, lower bound, upper bound, and loop-body. Using AST representation supports structure-based (or syntax-based) analysis techniques as well as loop-oriented parallelization.

We also retain the high-level operators defined in the source language in our IR. An alternative approach is to encode the high-level operators into simpler operators, such as the operators used in machine instructions. For example, SUIF encodes the assignment operator into operators *ldc*, *cpy*, *lod*, *memcpy* and *str*, according to the semantics of the assignment operations. Encoding the assignment operator this way simplifies the machine code generation, but complicates the high-level analyses and optimizations. For example, finding all dereferences to a pointer *p* involves only a search of all expression "**p*" in an IR that preserves high-level operators. However, if the dereference operators are encoded into operators *lod*, *memcpy* and *str*, finding the dereferences to the pointer *p* becomes more complicated. Furthermore, if later analysis determines that the

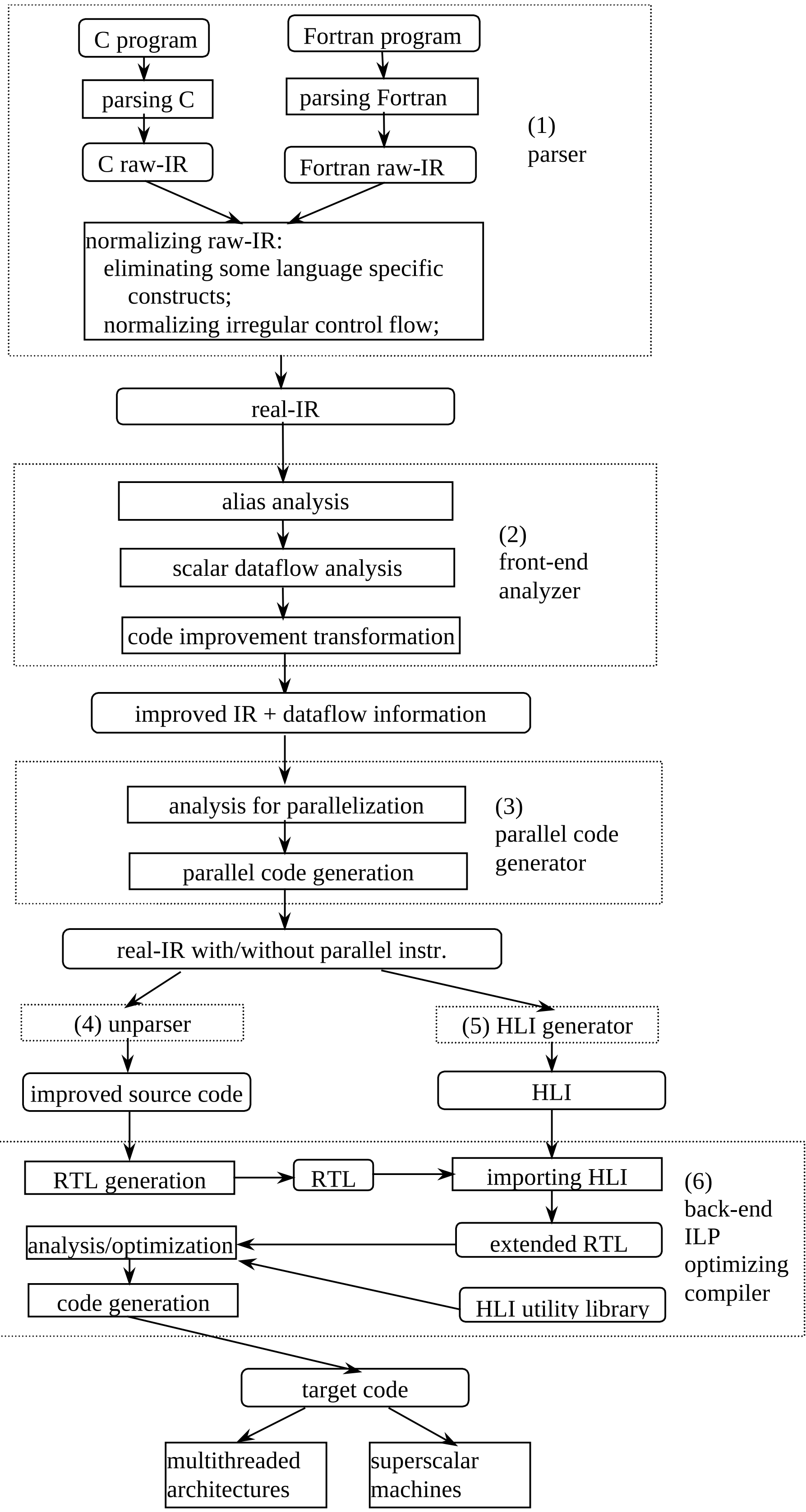

Figure 1: An overview of the Agassiz Compiler

pointer *p* definitely points to *j*, we may want to replace all dereferences to *p* with *j* in the IR. Performing this optimization in an IR retaining the dereference operator is simply a replacement of all "**p*" with *j*. However,performing similar optimization to the statement "*i = *p*" using a lower-level IR, we need to change the whole instruction from "*lod i = p*"' to "*cpy i = j*". This change is not local to the operand "**p*"; it affects the encoding of its higher-level instructions (i = *p).

Source code	SUIF IR
int i, j, *p, *q;	
i = 2;	ldc i , 2
i = j;	cpy i , j
i = *p;	lod i , p
*q = *p;	memcpy q , p
*q = j;	str q1 , j

Figure 2: An Example Program and Its SUIF-1 IR

Our IR retains variable accesses, array and structure references, and complicated array/structure access expressions, such as *s.a[1].f*, without dissecting them into simpler expressions using temporary variables. In doing this, we retain variable type, array size, array dimension, and structure field types information in the IR. Such information is critical for efficient scalar/array data flow analysis, data dependence analysis, and alias analysis.

3.2 Supporting Multiple Languages

Supporting multiple languages, such as C and Fortran, is also an important consideration in the design of our IR. Our IR is expressive enough to capture the details of all the language constructs defined in the target languages. However, to simplify later analyses and optimizations, we avoid putting unnecessary language specific constructs into IR while retaining only critical language specific information. The following C- and Fortran-specific language features are considered:

- Loop structures. The C language provides three loop structures: for-loop, while-loop and do-while-loop. The only loop construct in the Fortran is do-loop. Our real-IR contains three loop structures: Fortran-like do-loop and C-like while-loop and do-while-loop. If a for-loop has an index with a lower bound, an upper bound and a step, and the index is not modified inside the loop, the for-loop is converted into a do-while-loop. Otherwise, it is converted to a while-loop. This transformation preserves the high-level information in the original for-loop.

- Side-effect operators. Our IR does not include any side-effect operators because they may complicate later analyses and optimizations. All expressions with side-effect operators are transformed into equivalent expressions without side-effect operators. This is done in the raw-IR normalization phase.

- Parameter passing modes for procedures. The parameters are passed-in-by-value in C, but passed-in-by-reference in Fortran. Theoretically, it is possible to use pointers as passed-in-by-value parameters to replace the passed-in-by-reference parameters. However, in later analyses, a passed-in-by-value pointer parameter can be treated very differently from a passed-in-by-reference parameter. For instance, in alias analysis, the passed-in-by-reference induced aliases cannot be changed during an execution of a subroutine while the pointer-induced aliases can. Using pointers to replace passed-in-by-reference parameters can cause more pseudo aliases, especially when a less precise alias analysis algorithm is chosen due to efficiency consideration. For this reason, we keep the above two parameter passing modes in our IR.

- Goto statements. Fortran provides *goto*, *compute-goto*, and *assign-goto* statements while C has only *goto* statements. A *compute-goto* statement can be transformed into nested *goto* statements. Thus, we only provide *goto* and *assign-goto* representations in IR. Furthermore, to simplify later analyses and optimizations, there is a control-flow *normalization* phase during the conversion from Raw-IR to Real-IR which eliminates *goto* statements as much possible. It also marks a program segment with *un-normalized goto* statements and *goto* targets as an unstructured program segment. This allows later analyses to apply different algorithms to analyze well-structured program segments and unstructured program segments.

- I/O statements. The I/O statements in Fortran can be very complicated. To avoid handling such I/O statements in the front-end, our real-IR provides a high-level construct to represent the variables read/written in the I/O statements, and mask the detail of these I/O statements in the later analysis.

- Storage overlapping. Fortran has the *common* and *equivalenc*e constructs to specify the storage overlapping among different variables. C has the *union* constructs for the same purpose. Our IR represents these constructs in a uniform way. It use a *(block-name, offset)* pair to represent an element of the above constructs.

- Entry statement. An *entry* statement in Fortran is used to specify an additional entry point of a subroutine. To simply analyses and optimizations, we eliminate *entry* statements by creating a version of the function for each entry statement during the control flow normalization.

- Alternative return. A Fortran subroutine with *alternative return* statements is transformed into an equivalent subroutine without *alternative return* statements. This is done by introducing an additional parameter to the subroutine which indicates a program point in the caller as a return point of the subroutine call. An *if* statement is also added immediately after each call-site to the subroutine. This *if* statement issues a *goto* to a specific return point according to the return value of the additional parameter.

3.3 Supporting Source-to-Source Translation

Supporting source-to-source translation also imposes additional requirement to the IR. First, we want to reduce the machine and other environment dependent features in our generated source program for portability. A Compatible Compiler Preprocessor (CPP) may inject machine and compiler dependent features to a program in three ways: (1) inclusion of system header files; (2) expanding system dependent macros; and (3) including or excluding parts of the program according to system dependent preprocessing directives. To identify the system dependent features included in the system header files, we attach the name of the header file on each line inlined from the header file in IR. This allows the unparser to filter them out and keep them from being included in the generated source program. Second, we want the output source program looks similar to the original source program for readability. Our IR records the file name of each subroutine so that the unparser can output one source file for each original source file.

3.4 Software Engineering Considerations

We implement the IR in C++ because it allows data encapsulation. Layers of functionality can be built on top of each class, thus enable easy access and manipulation of the data structures while maintaining their consistency. However, as in C, the pointer mechanism in C++ can cause two most common errors in such programs: dangling pointers and dangling memory objects. In a large software program which is being implemented and maintained by many people, pointers can become *dangling* when the objects they point to are somehow freed in some other part of the software. A dereference to a dangling pointer can cause an error which is very difficult to detect, especially when the free operation is far from the dereference operator in the program. A dangling memory object is a dynamically allocated memory object which is no longer reachable because there is no pointer points to it anymore. A dangling memory object can waste memory space, which can become a problem when we are dealing with very large application programs. To help avoiding these two errors, we provide high-level classes to manipulate pointer data structures so that the users cannot use unconstrained pointers on IR objects (see Section 4).

Our IR also provides an *Annotation* class (as in SUIF) for extensibility. Users can attach other information to any IR element by adding annotations to it. However, it is the users' responsibility to maintain the consistency of the annotated information.

4. Implementation and Evaluation of IR

In this section, we discuss some implementation issues of our IR. We also describe the utility functions provided in our IR to facilitate program analysis and optimizations. We also present some preliminary experimental data regarding our IR implementation.

4.1 Detecting Pointer-Related Errors

We apply the reference counting technique [10] to detect pointer-related errors such as dangling pointers and memory leakage. This technique requires an additional field, *reference count*, for each object that may be pointed to by pointers. The *reference count* records the number of pointers pointing to the object, thus indicating whether the object is in use or not.

To facilitate this error detection mechanism, we include a *reference count* field in our base class, *AgassizBase*. All other classes that require a *reference count* are derived from this super class. We then design a class template *Pointer<A>*, where parameter *A* is a derived class of *AgassizBase*, to manipulate the reference count. We refer an object in the class *Pointer<A>* as an *agassiz-pointer* and an object of type "*A** **"** as a C++ *raw-pointer*. The only data field encapsulated in the template *Pointer<A>* is *mpoPointer*, which is a raw-pointer of type "*A**". This template keeps track of the reference count for the data structure being pointed to by its member *mpoPointer*, and allocates and deallocates the storage accordingly. More specifically, when a value is assigned to or created through *mpoPointer*, the reference count of its old points-to target is decreased by one while the reference count of its new points-to target is incremented by one. Dynamic objects are allocated via member functions in the template. These dynamic objects are automatically deallocated by the template when the object's reference count becomes zero, thus avoiding memory leakage. Furthermore, an object will not be deleted unless its reference count becomes zero. Not deleting an Agassiz object with a non-zero reference count can avoid dangling pointers

Another class called *GenericPointer* is designed to replace the use of type "void *" in our programs. Type "*void* *" is a frequently used type in C++ because it provides the flexibility to define a pointer that can points to objects of different classes. To provide similar capability as type "*void* *", the class *GenericPointer* encapsulates a raw-pointer to an object of the class *AgassizBase*. It also manipulates the reference count of the *AgassizBase* object it points to as the class template *Pointer<A>* does. Because the data field in *GenericPointer* is a raw-pointer to the class *AgassizBase*, it can point to an object of any class derived from *AgassizBase*.

To allow the above pointer manipulating mechanism to fully control the pointers used in IR, all IR classes must observe the following rules: (1) Agassiz-pointer classes are used to replace C++ raw-pointer types; (2) The class *GenericPointer* is used to replace raw-pointer type "*void* *"; (3) All other classes are derived from the super class *AgassizBase*.

4.2 Discussion

The above reference count approach allows us to effectively detect dangling pointers and memory leakage. However, it also adds some overhead to the system. First, the reference count field and its manipulation add some memory and run-time overhead. Second, calling a member function in our IR may incur a longer call-chain comparing to an approach without such an error detecting mechanism. Third, such an approach instills a stricter programming style which requires some extra programming effort. However, we believe such extra effort have been more than paid off by less debugging time caused by the pointer problems. We now use an example to illustrate the particular programming style needed under this approach.

In general, classes in our real-IR can be divided into two categories: content classes and pointer classes. A content class is a derived class of the class *AgassizBase*. It provides holders for a program entity and is not visible to IR users. Each content class has a corresponding pointer class. The pointer class for a content class *A* is a derived class of the template class *Pointer<A>*. This pointer class defines an interface to access content class A and is visible to IR users. Figure 3 compares an implementation of the class *Symbol* using agassiz pointers with an implementation using regular C++ pointers. As can be seen from the figure, to access the name attribute of a symbol, our approach requires two function calls while the regular approach requires only one. Furthermore, maintaining the consistency between class SymbolContent and class Symbol requires extra programming effort. This is because the class *Symbol* has a similar member function set as in the class *SymbolContent*. A member function in the class *SymbolContent* provides the implementation while its corresponding member function in the class *Symbol* provides an interface to access that implementation. One way to reduce the programming effort is to implement a utility that generates the correponding pointer class for each content class defined in our IR. However, we not yet found it to be a problem since it usually needs to be done only once for each class.

Using agassiz-pointers	**Using C++ raw-pointers**
class SymbolContent : public AgassizBase{	class Symbol {
private: String mesgName;	private: String mesgName;
protected: String mfsgName()	public: String mfsgName()
{return mesgName;}	{ return mesgName;}
};	};
class Symbol : public Pointer<SymbolContent> {	
public: String mfsgName()	
{return mpoPointer->mfsgName();}	
};	
Symbol eoSymbol;	Symbol *eoSymbol;
eoSymbol.mfsgName();	eoSymbol->mfsgName();

Figure 3. Comparing the agassiz-pointer approach with the regular C++ pointer approach

4.3 Utility Functions

Many high-level utility functions are implemented through member functions in the IR classes. They provide programmers with a powerful and easy-to-use environment, and help to enforce consistency among various data structures. These utility functions can be divided into four categories:

(1) Component inquiry. These utility functions provide a compositional view of each class, regardless of how each component is implemented in the class. For instance, the *IfStmt* class has member functions to support the retrieval of the *if-condition*, *then-part* and *else-part* of an *if* statement.

(2) Data structure traversal. We use a set of classes to implement the traversal of data structure collections. The class *IterBase* is a common super class for these classes. It defines a common interface for all traversals, including initializing and resetting a traversal, getting the current node, and moving to the next node.

(3) Program transformation. Program components may be added, deleted, duplicated or moved, during program optimization and transformation. Utilities to support these transformations are built in the IR classes. These utilities enforce the consistency of the IR during transformations. For example, a member function *mfoCopy* is built in the class *Procedure* for procedure cloning. It duplicates the body and the local symbol tables of the procedure. Our IR also allows users to tag the newly modified IR components and undo the modification later according to the specific tag. It allows users to coordinate the program analyses and

transformations among different compiler phases for more efficient output code. For example, induction variable substitution may generate a lot of unnecessary integer operations which could be a major overhead for a small loop if the loop is determined to be not parallelizable later. The ability to undo induction variable substitution creates much more efficient output code.

(4) Debugging assistance. A member function *mfvBrowse* provides a well formated text output of a class components while another member function *mfvUnparse* prints out the C or Fortran source code of the class. These functions can help programmers to keep track of the content of a data structure without knowing its implementation detail. At the program level, these functions can dump the IR of the entire program to a text file or produce a source program.

4.4 Evaluating IR Design

It is not easy to evaluate the design of an IR quantitatively, especially if we want to take its functionality, user support and programmability into consideration. Nevertheless, we present here two quantitative measures to evaluate our IR design. One metric is the time required to generate the IR for a program, including the parsing time and the cleanup time. Another metric is the IR size for a program. Because we take an object-oriented approach using a pointer-error detection mechanism, we want to make sure that such an approach does not substantially increase the size and the time to generate the IR. Also, our cleanup phase may duplicate code segments to *normalize* control-flow, we need to make sure that such normalization does not substantially increase the real-IR size.

We compare our Agassiz Compiler with SUIF-1 on some C and Frotran programs from the SPEC95 benchmark suite. Both compilers are compiled by GCC optimization option –O2. The collected data are shown in Table 1. These data were collected on a 200 MHZ Intel Pentium Pro machine with 128MB main memory running Linux 2.0.30. The time shown in the table is in seconds while the size is in Kbytes.

Programs	Time (sec.)					IR Sizes (KB)		
	SUIF		Agassiz			SUIF	Agassiz	
	Tot. = User + Sys.	User	Tot. = User + Sys.	User = Parsing + Cleanup		IR	Before Cleanup (raw-IR)	After Cleanup (real-IR)
				Parsing	Cleanup			
099.go	303.33	41.35	103.10	15.39	86.57	15,448	21,096	38,884
124.m88ksim	863.03	99.29	104.24	40.14	62.00	11,224	57,832	68,096
129.compress	41.77	6.97	4.43	2.65	1.71	940	2,692	3,108
130.li	136.21	21.26	26.08	8.16	17.52	3,980	12,244	15,672
132.ijpeg	396.19	101.04	100.34	27.87	71.19	11,284	35,168	49,096
147.vortex	968.65	431.12	223.55	70.61	150.36	28,796	75,044	89,372
101.tomcatv	5.26	1.44	1.01	0.15	0.8	392	136	344
102.swim	12.98	1.87	1.96	0.46	1.5	468	296	712
103.su2cor	71.36	5.56	22.92	2.8	20.05	2,040	1,576	6,640
104.hydro2d	56.70	10.69	10.41	2.78	7.53	1,676	1,520	3,432
107.mgrid	18.41	6.26	4.07	0.73	3.32	784	464	1,368
125.turb3d	54.99	8.94	9.36	2.24	7.07	2,116	1,200	2,804
145.fppp	97.18	22.82	207.36	4.28	199.02	3,008	2,068	25,096
146.wave5	254.48	29.32	54.43	12.02	41.10	5,596	6,348	16,588

Table 1. Comparing Agassiz with SUIF-1 in speed and IR sizes (The SUIF-1 IR was generated by executing command "scc –v -suif-link -.sln file-names")**.**

As can be seen from Table 1, the Agassiz IR generation is generally faster than the SUIF IR generation. For all of the programs except the program 145.fppp, the time used in creating SUIF IR (in the second column) is as much as 3 ~ 10 times the time used in creating Agassiz real-IR (in the fourth column). It may be due to two main reasons. First, the interface between SUIF passes are files. A significant portion of the SUIF IR generation time is system time dedicated to I/O. Thus, the SUIF IR generation time could be substantially reduced by modifying its passes so that they communicate via internal data structures, as has been done in current SUIF-2. Second, the Agassiz IR generation time includes the time of the cleanup phase. This cleanup phase not only normalizes the program constructs as SUIF does, but also normalizes the program control flow structures which the SUIF does not perform. Because a program with normalized control flow generally allows more efficient algorithms to be used in later phases, the cleanup time will be amortized by the later analyses. For these reasons, we also show the user portion of the SUIF IR generation time (the third column), and further divide the user portion of the time in the Agassiz IR generation into *parsing time* (the fifth column) and *cleanup time* (the sixth column). As indicated Table 1, the Agassiz parsing time is smaller than the user portion of the SUIF IR generation time for all programs. However, if we compare the user portion of the IR generation time, the Agassiz compiler is slower than SUIF in eight out of 14 programs.

Table 1 also shows that the Agassiz real-IR is usually larger than the SUIF IR, ranging from approximately 1 ~ 8 times larger. Two main reasons account for the larger sizes. First, our pointer-error detection mechanism incurs storage overhead in the data structures. Second, the cleanup phase may increase the IR size by duplicating codes to normalize control flow structures. We are looking into the IR size issue and hope to come up with strategies to reduce the size overhead further.

5. Analyses and Optimizations

In this section, we outline the analysis and optimization techniques currently implemented in the Agassiz compiler. These techniques are primarily for multithreaded code generation and back-end optimization. We also discuss how the IR supports these analyses and optimizations.

5.1 The Hierarchical Interprocedural Data flow Analysis

The interprocedural data flow analysis in our compiler provides *factored use-def* (FUD) chains [16] for all data structures in a program. To collect precise FUD chains, alias analysis is indenspensible. Unlike other approaches which separate the alias analysis from the traditional scalar data flow analysis, we use a hierarchical approach to integrate them into a common unified framework.

We first classify the data structures in a program into two categories according to the possible maximum level of dereference from the data structures [1]:

- Recursive data structures. It is assumed to have an unlimited dereference level for a recursive data structure. In the definition of "*struct S { struct S *next; …} p;*", the variable *p* is a recursive data structure. An expression "*p.next->next->next*" represents a two-level dereference from *p*; A dereference with more levels can be constructed by appending more "->next" to the expression.

- Non-recursive data structures. The maximum dereference level for a non-recursive data structure is a constant value. We call this constant value the *pointer level* of the data structure. For example, a variable s in the definition "*struct { char *c; int v;} *s;*" has a pointer level of two because there can be up to two levels of dereference from the variable (*s->c). On the other hand, a variable *i* defined by "*int i*" is a variable with a zero pointer level and, thus, cannot be dereferenced.

Using the above classification, we can then separate the pointer analysis into two subproblems: (1) analysis for recursive data structures, and (2) analysis for non-recursive data structures. We further divide the analysis for non-recursive data structures into several subproblems. Each subproblem analyzes the data structures of a certain pointer level, and annotates the points-to targets of the data structures to their dereference expressions. Finally, we solve these subproblems by analyzing the data structures with the

[1] In our discussion, if a field of a structure is dereferenced, we say the whole structure is also dereferenced.

highest pointer level, and then use the result to analyze the data structures of the next lower pointer level until we reach the pointer level of zero. This subproblem ordering is based on the observation that (1) a dereference from a data structure *p* with a pointer level *n* results in an access to a data structure *q* pointed to by *p*; (2) without type-casting, the pointer level of *q* is less than *n*.

This hierarchical data flow analysis algorithm handles the pointer-induced aliasing in C and the parameter-induced aliasing in Fortran in a uniform way. Because reference parameters of a subroutine can be replaced by pointers, we set the pointer level of the reference parameters to one. The analysis for pointer level one for a Fortran program propagates the actual parameters to the use of their corresponding formal parameters. This makes the use of actual parameters in the subroutines explicit, and hides the reference parameters to the analysis of pointer level zero.

Our data flow analysis algorithm is interprocedural and context-sensitive. We tag interprocedural data flow values with their call-chains, which are the call-graph paths along which the values are passed into the procedure. We parameterize the maximum length of the call-chains to control the context-sensitivity. This allows us to study its impact on the efficiency and the precision of the alias analysis. A more complete description of our data flow analysis algorithm is available in [17].

Our IR supports the hierarchical data flow analysis in several ways. Our study of eight SPEC95 CINT programs shows that an average of 38.73% pointer variables have initializers defining their initial values [9]. Ignoring these initial values causes unknown points-to values when they are used before explicit assignments assign new values to them, thus reducing the precision of the analysis. For this reason, variable initial values are not neglectable in the data flow analysis. The IR supports the initializer handling with two utilities One is the member function *mfoInitializerToAssignStmts* defined in the class *Variable*. It generates a list of assignment statements, each assigns an initial value to an element of the variable. Another member function *mfoInitializersToAssignStmts* is provided in the class *CompoundStmt*. It generates a list of assignment statements to model the effect of the initializers for all variables defined in the scope introduced by the *CompoundStmt*. Especially, when the *CompoundStm*t is the body of a "*main*" procedure, the assignments also include the initializers for all global variables. These utilities relieve the users from the language-specific detail for initializers.

Besides the initializer support, the IR also provides a good base for the hierarchical data flow analysis. First, it facilitates the grouping of data structures by their pointer levels. Second, not encoding the source language operators makes it straight forward to locate dereference expressions by searching operators "*", "->", and "[]". Finally, the compositional representation of the high-level program constructs make it easy to insert Φ-functions for those constructs, and to use the syntax-directed data flow equations to evaluate the fix-point data flow values.

5.2 Front-end Optimizations

The front-end optimization phase comes after the data flow analysis phase. It improves the effectiveness of the IR and the preciseness of the FUD chains. It also exposes more information for later analyses and optimizations. We first identify the optimizations that are important to later analyses and to multithreaded code generation. These include traditional optimizations such as constant folding, constant propagation, copy propagation, control-flow dead code (or unreachable branch) elimination, data flow dead code (or unused code) elimination, loop invariant code motion and induction variable substitution. They also include two new optimizations: (1) dereference expression simplification; and (2) non-zero-iteration loop identification. Dereference expression simplification is an optimization that replaces a pointer dereference expression with the definite points-to targets of the pointer. This optimization can reduces the number of indirect accesses caused by pointer dereferences, thus reducing the number of memory accesses in the program. Non-zero-iteration loop identification can refine the data flow information by eliminating the bypassing edge from the loop-entry node to the loop-exit node in the control-flow graph for a loop with at least one iteration.

The above optimizations often interact with each other. For instance, control-flow dead code elimination can refine the FUD chains and the points-to information, thus exposing more opportunity for other optimizations such as copy propagation and dereference expression simplification. In general, there are two approaches to implement such a group of optimizations. A *separated approach* implements each optimization independently. Each optimization conducts its own specific data flow analysis, which may be simplier than an integrated data flow analysis that serves all optimizations. This approach also provides the flexibility to allow individual optimizations to be reordered and/or repeated for a better result. An *integrated approach,* on the other hand, performs one unified data flow analysis for all optimizations. Each optimization incrementally updates the data flow information to reflect the refinement of the information as well as to ensure the correctness of the code transformation. Those optimizations are interleaved at the statement level during the data flow graph traversal and communicate each other via incremental data flow information updates. Integrating the optimizations has the advantage of avoiding duplicated data flow analyses. Furthermore, because an integrated approach allows more quick interactions among different optimizations, it may achieve a better result than that of a separated approach given the same number of iterations [5].

We use an integrated approach in our front-end optimizations using an extended version of FUD chains. FUD is a well-defined and compact representation of the use-definition chains [16]. The main difficulty in the FUD chain generation is the placement of Φ-functions, for which some efficient algorithms are available [7]. Some researchers have already noticed that Φ-functions can incorporate control-flow information, thus support incremental update during control-flow dead-code elimination [11][16]. We extend the definition of Φ-function to support its refinement with more accurate alias information.

5.3 Multithreaded Code Generation

Generating efficient code for concurrent multithreaded architectures is one of the main goals in the design of Agassiz compiler. Such architectures can exploit loop-level parallelism and provide hardware support for run-time data dependence checking, and thread execution and synchronization. The multithreaded code is generated in two steps. First, we estimate the *benefit* of parallelization for each loop nest, and select the most profitable loop nests for parallelization. Second, we generate multithreaded code, currently targeted for the superthreaded architecture, to execute the selected loop nests in parallel. In this section, we describe the dependence graph and the algorithm for selecting the appropriate loop nests for parallelization. We also briefly describe how to generate multithreaded code.

5.3.1 Dependence Graph

In general, a dependence graph is a graph that depicts the execution order of its nodes. In our dependence graph, a node can be either a statement or an address calculation for a write operation. We include the address calculation for write operations in our dependence graph because this information is very important in facilitating run-time dependence check between threads in the superthreaded architecture. Edges in our dependence graph depicts control dependence, data dependence and reference dependence between the nodes in the graph. The control dependence edges are added to the graph according to the program control-flow while the data dependence edges are determined by the data dependence test. We also add a reference dependence edge from a node which calculates the address for a write operation to the statement that actually contains the write operation.

In order to *parallelize* C programs, our data dependence test needs to be more comprehensive than a traditional data dependence test. First, our data dependence test needs to handle *while-loops* that may not have an explicit loop index variable. We perform induction variable analysis in the optimization phase and pass the identified index variable to the data dependence test via annotation. Second, our data dependence test needs to handle pointer dereferences. It uses the points-to information provided by the data flow analysis to determine the potential targets of a pointer dereference. Finally, in addition to the traditional data dependence attributes, such as type, distance and direction, our dependence test also returns a reason describing why the test returns a *TRUE* dependence. For example, a *TURE* dependence may be assumed by the test because the points-to information is not precise enough or the subscript expression is non-linear. These reasons can be used in our heuristic algorithms to facilitate possible data speculation.

5.3.2 Selecting Loops for Parallelization

The hardware and the software overhead incurred in parallelizing a loop may overshadow any performance gain from executing the loop in parallel. Most multithreaded architectures allow only one parallel loop level in a nested loop structure as in multiprocessors. Hence, we need to prudently select the appropriate loop level for parallelization.

We first estimate the parallelism of each loop using the loop-level dependence graph. Edges for the dependencies that can be enforced by the hardware are ignored. For example, edges for loop-carried anti-dependencies and output-dependencies can be ignored because the hardware provided in the superthreaded architecture can enforce these dependencies. We also ignore the edges for the speculated dependencies, which are unlikely to happen at run-time. The loop-carried control dependencies, for instance, can be speculated because jumping out of a loop only occurs in the last iteration of the loop. Another example is to speculate on the data dependences which may occur with a low probability (perhaps from profiling information), e.g., data dependences from pointer variables. The largest strongly connected component of the modified dependence graph is then used to approximate the minimum starting delay of the next loop iteration.

The next step is to select the appropriate loops for each nested loop structure for the parallelization. We estimate three kinds of execution times for each loop. *Parallel-execution-tim*e is the time to execute the loop if we parallelize the current loop level and sequentially execute its inner loops. In estimating the *parallel-execution-tim*e, we apply an algorithm similar to the one described in [6] with the minimum starting delay calculated in the first step and an estimated number of iterations. Profiling feedback can provide more accurate information for the number of iterations of each loop to improve this estimation. *Sequential-execution-time*, on the other hand, is the time to execute the loop if we sequentially execute the current loop level and execute its inner loops with their *shortest-execution-times*. The *shortest-execution-time* for the loop is the minimum between its *parallel-execution-time* and its *sequential-execution-time*. We use a dynamic programming algorithm to calculate these three kinds of execution times for all the loops in a nested loop structure. Loops are selected for parallelization so that the smallest *shortest-execution-time* is achieved for the outer loop of each nested loop structure.

5.3.3 Code Generation

To generate parallel code for the superthreaded architecture, we first reorder the statements in a loop into three stages. The first stage computes the basic sequential part of the loop iteration, such as updating the induction variables or loop terminating conditions in a While-loop. The second stage calculates the necessary addresses for dynamic dependence checking, while the third stage performs the main computation in an iteration. We then insert primitives for run-time dependence check and synchronization in the loop. Easy-to-use utility functions are provided in IR to insert these primitives.

An effective and efficient algorithm to generate parallel code will maximize the overlap between loop iterations while enforced all dependencies. However, the algorithm to find an optimal parallization is known to be *NP-hard*. We need to use heuristics to obtain a reasonable solution with an acceptable performance. The discussion on the design of such algorithms is beyond the scope of this paper.

6. Related Work

There exists several infrastructures for research on parallelizing and optimizing compilers. Among these compilers, SUIF [15], Polaris [1] and McCAT [8] are interesting comparisions with our Agassiz compiler.

SUIF is a parallelizing and optimizing compiler for both C and Fortran programs, consisting of a kernel and a toolkit. Unlike our Agassiz compiler which uses separated high-level and low-level IR, the SUIF kernel defines a mixed-level program representation, including both high-level program constructs and low-level machine operations. The SUIF toolkit provides utilities to support research on parallelization of numeric and non-numeric programs, interprocedural analysis, dependence analysis, and MIPS instruction generation.

Polaris is a parallelizing Fortran compiler. It implements optimization techniques, such as automatic detecting of parallelism and distributing of data, that are necessary to transform a given sequential Fortran program into a form that runs efficiently on a target machine. The kernel of Polaris defines a high-level IR which is very similar to our real-IR. However, the implementation of the Polaris IR employs a more sophisticated memory error detecting mechanism, which not only counts the number of pointers pointing to an object but also enforces an explicit ownership on the objects.

McCAT is a compiler/architecture testbed which provides a unified approach to the development and performance analysis of compilation techniques and high-performance architectural features. The core of McCAT is a C compiler providing both a high-level and a low-level IR, and equipped with many analyses and transformations at both IR levels. Similar to our compiler, McCAT also has a phase to convert a program with *goto* statements into an equivalent program without *goto* statements. However, McCAT does not provide any IR representation for *goto* statements. This makes it difficult to represent a program with unrestricted *goto* statements which cannot be converted to an equivalent program without goto statements. Our Agassiz compiler, on the other hand, can represent unrestricted goto statements and mark a program segment with unrestricted *goto* statements and *goto* targets as unstructured program segment. This information can be used to simply later analyses.

7. Conclusions and Current Status

In this paper, we present the overall design of the Agassiz compiler. The Agassiz compiler is an integrated compiler targeting the concurrent multithreaded architectures (such as Multiscalar [12] and Superthreaded processors [13]). It has a loop-level parallelizing compiler as its front-end and an instruction-level optimizing compiler as its back-end to support such architectures. In this paper, we focus on the IR design of the Agassiz compiler, and describe how we support the front-end analyses, various optimization techniques, and source-to-source translation for the parallelization of both C and Fortran programs.

The compiler is currently being developed jointly at the University of Minnesota and at the Fudan University in Shanghai, PRC. A few SPEC95 benchmark programs (mostly C programs) have been successfully run all the way through from the front-end parallelizing compiler to the modified gcc back-end compiler, and simulated on our execution-driven simulator for superthreaded architectures. We are currently evaluating the effectiveness of various analysis and parallelization techniques, and working on ways to improve the back-end compiler to generate more efficient superthreaded code.

Acknowledgments

This work was supported in part by the National Science Foundation under grant nos. MIP-9610379 and CDA-9502979; by the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346, and a gift from the Intel Corporation.

We would like to thank Dan Laney, Youfeng Wu and the anonymous referees for their comments which resulted in numberless improvements and clarifications of the paper.

References

[1] The Agassiz Compiler. http://www.cs.umn.edu/Research/Agassiz/

[2] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. *Polaris: Improving the effectiveness of Parallelizing*

Compilers. Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science 892. K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.) pages 141-154. Springer-Verlag, 1994.

[3] Doug Burger and Todd M. Austin. *The SimpleScalar Tool Set*. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, 1997.

[4] S. Cho, J.-Y. Tsai, Y. Song, B. Zheng, S. J. Schwinn, X. Wang, Q. Zhao, Z. Li, D. J. Lilja, and P.-C. Yew. *High-Level Information - An Approach for Integrating Front-End and Back-End Compilers*. In Proceedings of the International Conference on Parallel Processing, pages 345-355, Auguest 1998.

[5] Cliff Click and Keith D. Cooper. *Combing Analyses Combing Optimizations*. In ACM Transactions on Programming Languages and Systems, Vol.17, No.2, pages 181-196, March 1995.

[6] Ron Cytron. *Limited Processor Scheduling of Doacross Loops*. In Proceedings of the International Conference on Parallel Processing, pages 226-234, Auguest, 1987.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. *Efficiently Computing Static Single Assignment Form and The Control Dependence Graph*. In ACM Transactions on Programming Languages and Systems, pages 451-490, Vol 13, No 4, October,1991.

[8] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. *Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations*. In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, Auguest 1992.

[9] Bo Huang. *Context-Sensitive Interprocedural Pointer Analysis*. PhD Thesis, Computer Science Department, Fudan University, P.R.China, in preparation.

[10]Richard Jones and Rafael Lins. *Garbage Collection*. John Wiley & Sons Ltd, 1996.

[11]Richard C. Johnson. *Efficient Program Analysis Using Dependence Flow Graphs*. Ph.D. Thesis, Computer science, University of Cornell University, 1994.

[12]G. S. Sohi, S. Breach, and T. N. Vijaykumar. *Multiscalar Processors*. In Proceeding of the 22th International Symposium on Computer Architecture (ISCA-22), 1995.

[13]J.-Y. Tsai and P.-C. Yew. *The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation*. In Proceedings of the Int'l Conf. on Parallel Architectures and Compilation Techniques, October 1996.

[14]J.-Y. Tsai. *Integrating Compilation Technology and Processor Architecture for Cost-Effective Concurrent*. Ph.D. Thesis, Computer Science, University of Illinois at Urbana-Champaign, April 1998.

[15]Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*. In SUIF document, http://suif.stanford.edu/suif/suif1/.

[16]Michael R. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[17]Bixia Zheng and Pen-Chung Yew. *Hierarchical Approach to Context-Sensitive Interprocedural Alias Analysis*. Technical Report 99-018, Comuper Science Department, University of Minnesota, April 1999.