# The Design of the PROMIS Compiler*

Hideki Saito[1], Nicholas Stavrakos[1], Steven Carroll[1],
Constantine Polychronopoulos[1], and Alex Nicolau[2]

[1] Center for Supercomputing Research and Development,
University of Illinois at Urbana-Champaign,
1308 W. Main St., Urbana, IL 61801
{saito, stavrako, scarroll, cdp}@csrd.uiuc.edu
[2] Department of Information and Computer Science,
University of California at Irvine,
Irvine, CA, 92697-3425
nicolau@ics.uci.edu

**Abstract.** PROMIS is a multilingual, parallelizing, and retargetable compiler with an integrated frontend and backend operating on a single unified/universal intermediate representation. This paper describes the organization and the major features of the PROMIS compiler.
PROMIS exploits multiple levels of static and dynamic parallelism, ranging from task- and loop-level parallelism to instruction-level parallelism, based on target architecture description. The frontend and the backend are integrated through a unified internal representation common to the high-level, the low-level, and the instruction-level analyses and transformations. The unified internal representation propagates hard to compute dependence information from the semantic rich frontend through the backend down to the code generator. Based on conditional algebra, the symbolic analyzer provides control sensitive and interprocedural information to the compiler. This information is used by other analysis and transformation passes to achieve highly optimized code. Symbolic analysis also helps statically quantify the effectiveness of transformations. The graphical user interface assists compiler development as well as application performance tuning.

## 1 Introduction

Most systems under design and likely to be built in the future will employ hierarchical organization with many levels of memory hierarchy and parallelism. While these architectures are evolutional, reflecting advances in hardware technology, they pose new challenges in the design of parallelizing compilers.

The PROMIS compiler tackles these challenges through its hierarchical internal representation (IR), the integration of the frontend and the backend, extensive symbolic analysis, and aggressive pointer analysis. The hierarchical

* This work is supported in part by DARPA/NSA grant MDA904-96-C-1472, and in part by a grant from Intel Corporation.

IR provides a natural mapping for exploitation of multi-level memory hierarchy and parallelism. The frontend-backend integration via the unified IR enables the propagation of more information from the frontend to the backend, which in turn helps achieve a synergetic effect on the performance of the generated code[6]. Symbolic analysis not only produces control flow sensitive information to improve the effectiveness of the existing analysis and optimization techniques, but also quantitatively guides program optimizations to resolve many tradeoffs. Pointer analysis uses information provided by symbolic analysis to further refine aliasing information.

The PROMIS compiler is a multilingual, parallelizing, and retargetable compiler with an integrated frontend and backend operating on a single unified/ universal IR (or UIR). Unlike most other compilers, PROMIS exploits multiple levels of static and dynamic parallelism ranging from task- and loop-level parallelism to instruction-level parallelism, based on target architecture description.

Fig. 1 shows the organization of the PROMIS compiler. The core of the compiler is the unified/universal hierarchical representation of the program. Both the frontend and the backend analysis/optimization techniques, driven by the description of the target architecture, manipulate this common UIR. Support for symbolic analysis is an integral part of the UIR, which provides control sensitive information throughout the compilation process. The current implementation of PROMIS supports C, C++, FORTRAN, and Java bytecode as input languages, and can target wide variety of systems, such as CISCs, RISCs, and DSPs.

PROMIS is an on-going research project with many parts of its optimizer and backend still under development. In this paper, we focus on the design aspects of the compiler, while we anticipate to obtain the first performance results in early 1999.

The rest of the paper is organized as follows: Section 2 focuses on the frontend-backend integration. Section 3 describes the PROMIS IR. Issues on analysis and optimization are discussed in Section 4. The PROMIS GUI is introduced in Section 5. Section 6 discusses related work on compiler development. Finally, Section 7 summarizes the paper.

## 2 Motivation for the Frontend-Backend Integration

Conventional compilation frameworks use abstract syntax information (typically represented in the form of source or intermediate code) to connect the frontend and the backend. Examples include restructuring tools (such as Parafrase-2[13], Polaris[3], and KAP[11]) and system vendors' backend compilers. This conventional framework makes the development of the frontend and the backend independent and modular. However, the inability to transfer dependence information from the frontend to the backend results in performance degradation[6].

For example, suppose the backend is a multiprocessor-aware compiler which is capable of dealing with parallel directives (such as `C$DOACROSS` and `C$OPENMP PARALLEL DO`). In this case, the frontend usually augments a parallel loop with a parallel directive and leaves other optimizations to the backend. However, since

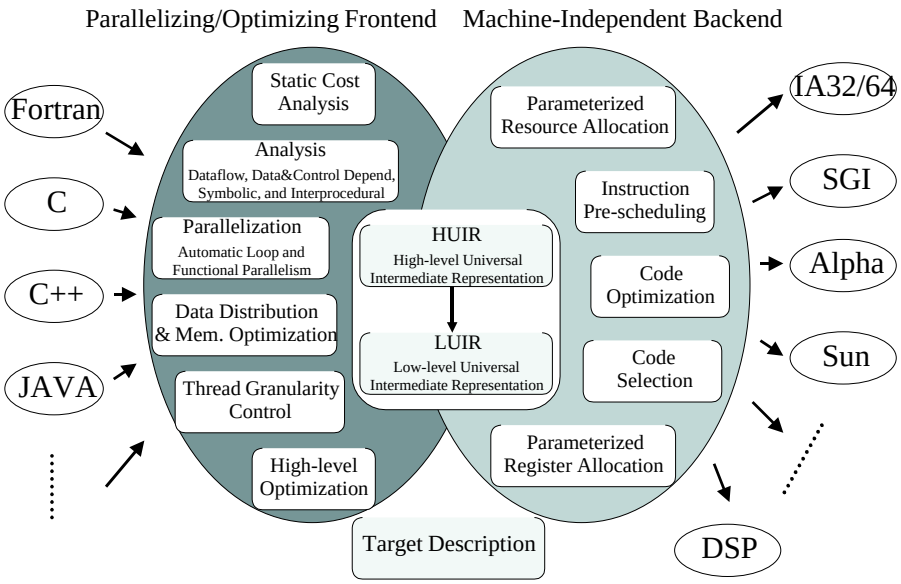


**Fig. 1.** Overview of the PROMIS Compiler

dependence analysis in the backend is usually less accurate than in the frontend (due to loss of array index expressions, data dependence analysis is substituted with memory disambiguation at the level of memory addresses) intra-iteration dependence information gets lost, resulting in lower performance. Backend compilers can still perform extensive dependence analysis in the high-level representation before optimizing in the low-level representation. However, it is simply a waste of compilation time to perform time-consuming dependence analysis both in the frontend and in the backend if such information can be communicated.

In cases where the backend is a uniprocessor-oriented compiler (such as GCC), the situation is even worse. A typical frontend replaces a parallel loop with a call to a runtime library routine (such as DOALL()), creates a function for the loop body, and uses the pointer to this loop body function as an argument to the runtime library call. Since the loop body is now a separate function requiring interprocedural analysis, the backend optimization is unlikely to be as effective.

Furthermore, there are cases where inter-processor parallelism and intra-processor parallelism can be exchanged[6]. A frontend which is independently developed from the backend may package what could best be exploited as ILP

parallelism into iteration level parallelism. This not only leads to lower functional unit utilization, but can also increase the total execution time.

In PROMIS, these problems are tackled by integrating the frontend and the backend via the common IR. The following section describes the PROMIS IR and how it is used to address these problems.

## 3 The PROMIS IR

In the PROMIS compiler, the frontend and the backend operate on the same internal representation, which maintains all vital program structures and provides a robust users' and developers' IR interface. The IR interface makes most transformations and optimizations independent of the implementation details of the IR data structures. The PROMIS IR is capable of dealing with multiple input languages and output ISAs (instruction set architectures). This is achieved through transformations and optimizations viewing and accessing IR structures as semantic entities, not as syntactic constructs. The IR framework consists of the following:

- Symbol Table
- Expression Trees
- Control Flow Edges (CFEs)
- Control Dependence Edges (CDEs)
- Data Dependence Edges (DDEs)
- Hierarchical Task Graphs (HTGs)
  - HTG nodes
  - Hierarchical Control Flow Edges (HCFEs)
  - Hierarchical Control Dependence Edges (HCDEs)
  - Hierarchical Data Dependence Edges (HDDEs)
- Support for the whole program optimization

The Hierarchical Task Graph (HTG) has been successfully used both in a frontend parallelizer[13] and in a backend compiler[12]. In the HTG, hierarchical nodes capture the hierarchy of program statements, and hierarchical dependence edges represent dependence structure between tasks at the corresponding level of hierarchy. Therefore, parallelism can be exploited at each level of the HTG: between statements (or instructions), between blocks of statements, between blocks of blocks of statements, and so on. This flexibility promotes a natural mapping of the parallelism onto the hierarchy of the target architecture.

### 3.1 Multilingual Frontend

Unlike previous attempts at multilingual IR (such as UNCOL), PROMIS does not try to accommodate all programming languages. Instead, PROMIS aims at generating high performance code for the mainstream imperative programming languages, such as C, C++, and FORTRAN. The current version of the

PROMIS IR represents a subset of the union of the language features of C++, FORTRAN, and Java. Performance critical features are directly supported, and thus represented in the PROMIS IR. Mere syntax sugar is still supported but must be converted during the IR construction process. For example, virtual function calls are directly supported, while some of the operators, such as, comma, increment, and decrement are converted.

PROMIS translates stack-based Java bytecode into register-based statements and applies language independent analyses and optimizations. Two major challenges in optimizing Java are exceptions and type inference. In PROMIS, both of these challenges are tackled by symbolic analysis.

### 3.2 Frontend-Backend Integration

Enhanced support for integrated compilation in PROMIS is enabled by the UIR, which propagates vital dependence information obtained in the frontend to the backend.

The PROMIS IR has three distinctive levels of representation: high-level (HUIR), low-level (LUIR), and instruction-level (IUIR). Although the UIR can be at any arbitrary sub-level between the HUIR and the LUIR during the course of the IR lowering process, the focus of the current development effort is given to the three major levels. In the PROMIS IR, statements are represented as HTG nodes.

The abstract syntax trees from the parser can have arbitrarily complex expression trees. During the construction of the HUIR, expression trees are normalized to have a single side effect per statement. Function calls and assignments to pointer dereferences are identified and isolated as separate statements.

During IR lowering (from HUIR to LUIR), complex expression trees are broken down to collections of simple expression trees, each of which is similar to quadruples. Data dependence information is maintained and propagated throughout the lowering process. Therefore, the PROMIS backend utilizes the same quality of dependence information as the frontend, unlike conventional compilers.

Fig. 2(a) shows a HUIR representation of the statement `a[i] = b * c`. At the leaf-level of the HTG, there is an `AssignStmt` node corresponding to this assignment statement. The associated expression tree gives the semantics of the statement. DDEs connect the source and the destination expressions of data dependence for this expression tree. HDDEs connect the source and the destination HTG nodes, summarizing detailed data dependence information provided by the DDEs. Fig. 2(b) is the LUIR corresponding to Fig. 2(a). In this example, IR lowering is performed for register-register type architectures. The PROMIS IR represents virtual registers in almost the same way as any other temporary variables, which is reflected in the way addresses of the virtual registers are taken. During the lowering process, local dependence information is generated and non-local dependence information is updated to reflect the lowering.

In addition to providing detailed dependence information to the backend, the UIR also enables sharing of compiler passes between the frontend and the

backend. For example, tools such as the available expression analyzer and the constant propagator can work on the HUIR dealing with complex expressions, on the LUIR dealing with simple expressions, and the IUIR dealing with simple expressions, opcodes, and side effects. The ability to raise the IR from IUIR to LUIR and from LUIR to HUIR (again, without loss dependence information) is unique to the PROMIS IR. Since the LUIR is a proper subset of the HUIR, high-level analysis and transformation techniques can be seamlessly applied to the LUIR. Raising from the IUIR to the LUIR is simply performed by removing the opcode and transforming the node to a set of single side-effect nodes.

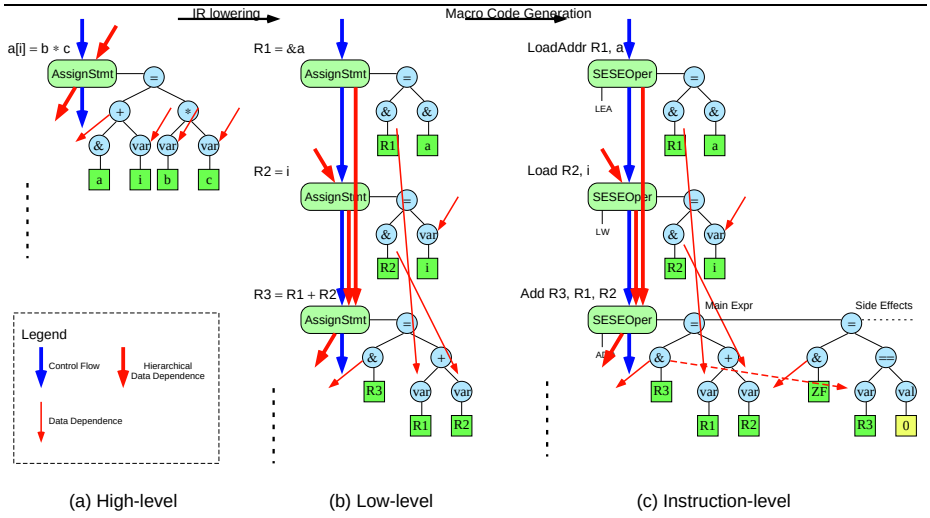


**Fig. 2.** High-level, low-level, and instruction-level IR

### 3.3 Multitarget Backend

Macro code generation on the PROMIS IR converts the LUIR into the IUIR. This involves the conversion of generic simple expression trees to a restricted set of simple expression trees, the assignment of a macro opcode to each of the converted expression trees, and expression tree construction for the side-effects of the opcodes. As in IR lowering, macro code generation maintains and propagates dependence information. It also generates dependence information for the side effects so that they can be handled in a uniform manner. The target-level backend optimizer operates on the IUIR, and eventually all macro opcodes are replaced by actual opcodes of the target. The target system information is automatically

or manually generated from the target architecture description, which is common to the compiler and the simulator.[1]

Fig. 2(c) shows the IUIR of Fig. 2(b) for a pseudo instruction set architecture. During the macro code generation process, dependences to/from side effects and the transformed main expressions are generated and updated, respectively. The first instruction in Fig. 2(c) corresponds to the first statement in Fig. 2(b). The instruction is still an assignment statement representing `R1 = &a`. However, the HTG node is changed from `AssignStmt` to `SESEOper` (Single-Entry Single-Exit operator) in order to attach an opcode `LEA` and side-effects (in this case, none). The third instruction `ADD` has side effects, of which the zero-flag (`ZF`) assignment is presented. `ZF` is assigned based on the result of the addition. Therefore there is a data dependence (within the instruction, shown as a dashed line) from the assignment to `R3` and the use of its value. In this example, there are dependence arcs from `R3` and `ZF` to elsewhere, indicating that the values are used later.

### 3.4 Support for Symbolic Analysis

As will be shown in the next section, symbolic analysis plays a dominant role within PROMIS. To increase the efficiency of the symbolic interpreter the IR has been extended in two ways.

First, variable versioning has been implemented directly into the IR. Scalar variables and scalar fields of a structure can be versioned. Second, conditional algebra operators have been included in the IR. The conditional operator $\tau(e)$ returns 1 if $e \neq 0$ and 0 otherwise. With this simple operator, control sensitive information can be encoded into the expressions of the IR. For example, encoding the situation where $X_3$ is dependent on the outcome of a branch with condition $C_1$ would yield: $X_3 = X_1\tau(C_1) + X_2\tau(!C_1)$. In this expression $X_3$ gets the value $X_1$ if $C_1$ is true, else it gets the value $X_2$.

### 3.5 IR Extensibility

The core IR is designed to provide the basic functionality that is required by the majority of passes in the compiler. In addition to this core functionality many additional data structures are used during the compilation process (e.g. connectivity matrix, dominator tree, etc). Although these data structures are useful in many compiler passes, they are transient and not a necessary part of the IR; rather they are data structures built upon the core IR. Allowing these transient data structures to be placed within the IR would clutter the IR unnecessarily. Another problem is maintaining them across multiple passes. It may not be possible (or extremely difficult) to maintain them across passes that were developed before the addition of such transient data structures, and thus not aware of them. Development of a new pass would also be difficult if the pass has to maintain transient data structures it does not use.

[1] A VLIW simulator developed at UCI is used to quantitatively evaluate various transformations and optimizations during the development phase.

To alleviate both these problems PROMIS provides an API called External Data Structure Interface (EDSI). EDSI allows compiler developers to register data with each HTG node (e.g. each node can contain the immediate predecessor and successors of a dominator tree). In addition, a data structure can register a call back function to be called during certain IR events (e.g. control flow arc removal/insertion, node addition/removal, etc). These call back functions allow the data structures to perform the necessary tasks to maintain their consistency with the IR.

# 4 Analysis and Optimization

## 4.1 Symbolic Analysis

Ever since the benefits of symbolic analysis were first demonstrated for compilers[13], many commercial and research compilers have adopted the use of symbolic analysis. The number of analysis and transformation techniques using symbolic analysis has increased greatly, due to the symbolic analysis capabilities of modern compilers. In light of this, support for symbolic analysis has been integrated within the internal representation of PROMIS. This integration provides a mechanism for extending the symbolic analysis framework, thus allowing new analysis/transformation techniques to be easily added into the framework.

The symbolic analysis framework uses a symbolic kernel that allows symbolic expressions to be handled in a manner similar to numeric values. Symbolic expressions consist of either scalar variables, scalar fields of structures, and/or arrays. Symbolic expression types include integers, floating point, and complex. Because the values a variable can possess may be dependent on the control flow of the program, control sensitive values of a variable are encoded within a symbolic expression. Control sensitive value extraction and symbolic expression simplification are also performed by the symbolic kernel.

In PROMIS, symbolic analysis is performed via symbolic interpretation. Values (or ranges of values) for each variable are maintained by the interpreter in environments. These environments are propagated to each statement. Each statement is interpreted, and its side effects are computed. These side effects are applied to the incoming environment of a statement, resulting in new versions for the affected variables. Successive application of these side effects simulates the execution of the program. Pointer analysis is performed during interpretation. This tight integration between symbolic and pointer analysis allows for efficient information flow to occur between the two passes.

Fig. 3 shows a section of code along with the corresponding interpreted HTG. Interpretation begins with a new environment initialized with the formal parameters. The first node assigns 10 to the variable `x`. Since `x` is yet to be versioned, the new version 1 is assigned to `x`, and $x_1$ is added to the symbolic environment. For the next node, `y = a`, the interpreter searches for the current version of the variable `a` in the current environment and finds $a_1$. The variable `y` in this node also needs a version, and it becomes $y_1$, just like the variable `x` in the previous node. The variable $y_1$ is added to the symbolic environment.

The next node is a branch statement. The conditional expression of the branch is evaluated and then two child environments are created (corresponding to the true and false paths of the branch). Variable lookup requests, when they cannot be satisfied by these child environments, are forwarded to their parent environments. In addition, control flow tags are assigned for each child environment. A control flow tag corresponds to the condition that must be satisfied in order for a section of code to execute.

The true and false portions of the `IF-THEN-ELSE` structure are evaluated. As control flow converges, the two incoming environments into the `RETURN` statement must be merged. In this example, the variables $z_1$ and $z_2$ are merged into the new variable $z_3$. Finally, the expression to be returned gets versioned to $z_3$.

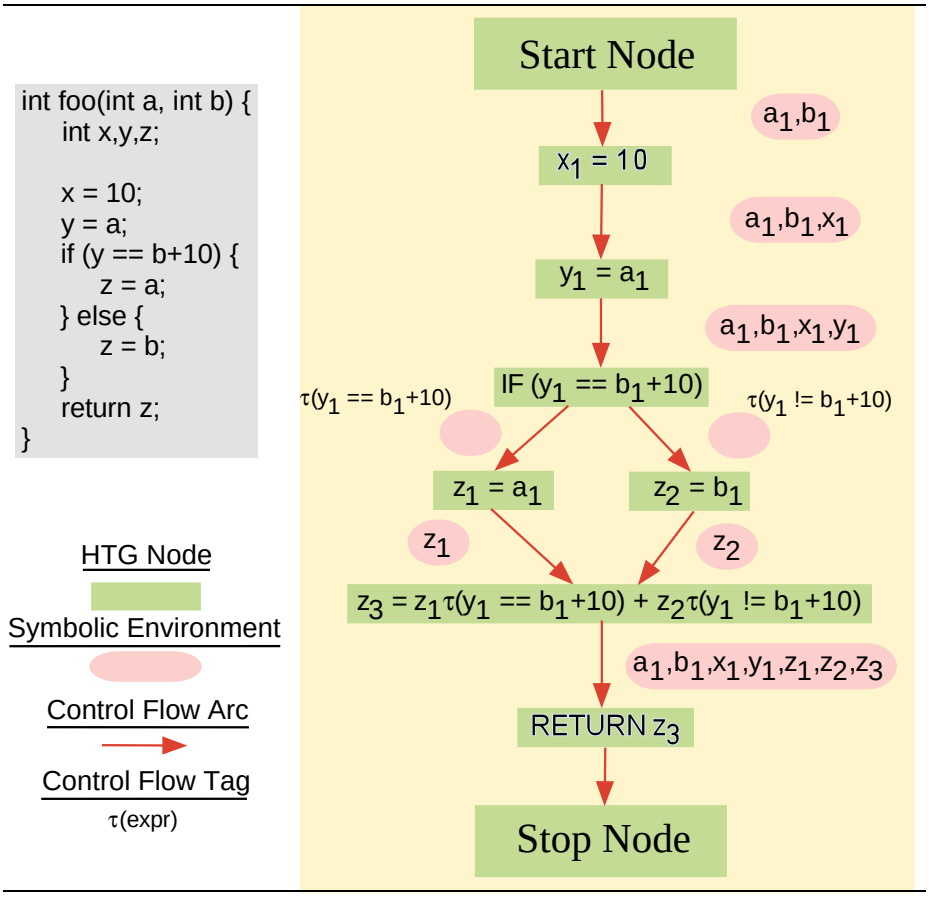


**Fig. 3.** Symbolic Interpretation Example

Interprocedural analysis seamlessly integrates into the symbolic analysis framework. When a function call is encountered by the interpreter, its side effects are

calculated and applied to the incoming environment, like any other expression. Once calculated, the side effects of a function call can be saved for subsequent interpretations or discarded to alleviate the memory footprint of the compiler. This method of handling function calls eliminates the need for special case function call handling in many analysis and transformation techniques. The only caveat is that function calls are interpreted for a specific alias pattern. Aliasing between parameters and global variables, which are used within the function call, must be properly identified and handled.

Alias information improves the accuracy of other analysis techniques, the effectiveness of optimizations, and the compilation time. Alias information is first gathered during IR construction. Static aliases (e.g. Fortran EQUIVALENCE and C/C++ unions) are analyzed, and their alias patterns are saved. Formal parameter aliases are then analyzed iteratively before symbolic interpretation. Although not exact, this iterative process eliminates many possible alias patterns. Symbolic interpretation is then applied to the program. The interpreter utilizes this alias information and points-to information collected during interpretation.

### 4.2 High-Level Parallelization and Optimization

Similar to most parallelizing compilers, PROMIS will include a number of classical analysis and transformation techniques. In PROMIS however, these techniques will be implemented within the symbolic analysis framework. This allows classical optimizations to exploit the full power of symbolic analysis. Several optimizations have been reengineered within the symbolic analysis framework, such as strength reduction, static performance analysis, induction variable elimination[9], symbolic dependence analysis[4], and array privatization[15]. Other techniques need not be reengineered to benefit from symbolic analysis. These optimizations, which include constant propagation, dead code elimination, and available expression analysis, benefit from the control sensitive information provided by symbolic analysis. The application of these techniques can be controlled by an integrated symbolic optimizer, which determines the ordering of the analysis and optimization techniques for each segment of code.

A quantitative measure of the synergetic effect of the combination of symbolic and pointer analysis is a major goal of the PROMIS project. Symbolic analysis will benefit from the disambiguation power of pointer analysis. Likewise, pointer analysis will benefit from the control sensitive value information of pointer expressions provided to it by symbolic analysis.

### 4.3 Instruction-Level Parallelization and Optimization

The PROMIS backend is divided into machine independent and machine dependent phases. The former works on the LUIR, while the latter works on the IUIR. As in the frontend, symbolic information plays an important role throughout the backend.

The machine independent phase includes classical optimizations, such as, common subexpression elimination, copy propagation, and strength reduction. The conversion from the LUIR to IUIR involves instruction selection and preliminary code scheduling. The mutation scheduler[12] performs instruction mutation, instruction scheduling, register allocation, loop unrolling, and code compaction on the IUIR. The machine dependent phase derives target specific information from the target machine description, and therefore the optimizer code itself is target independent. The PROMIS backend can also be guided by the results of an architectural simulator, which shares the target machine description with the compiler.

Unlike other backend compilers, PROMIS does not perform memory disambiguation because data dependence information from the frontend is available in the backend.

## 5 Graphical User Interface

Graphical user interfaces (GUIs) have become a necessity for developers and users of any compiler. The PROMIS GUI aids in compiler development and user program optimization; both of these tasks benefit greatly from the graphical representation of information.

For compiler development, PROMIS provides both textual and graphical views of the IR (Fig. 4). Since the PROMIS IR is hierarchical, both views (textual and graphical) are also hierarchical. Users can expand or collapse a compound node to display more or less information of the node. This is useful to prevent unnecessary information from drowning out the needed information that the compiler developer is seeking.

The IR can be viewed in two modes: online and offline. An offline view take a simplified snapshot of the IR, and save it in a separate data structure. This allows compilation to continue while the compiler developer explores the IR. Offline views are particularly useful to compare the IR before and after an operation. Online views require the compilation of a program to stop (or pause). While the compilation is paused, the IR can be inspected and modified (e.g. removal of superfluous data dependence arcs). Compilation continues when the view is closed. The offline view can be saved to disk and retrieved for comparison to later compiler runs.

The GUI also provides a mechanism to set breakpoints at specific points in the compilation of a program. This functionality allows compiler developers to dynamically pause compilation and perform an offline or online view of the IR.

The PROMIS status bar informs the compiler user of the current phase of the compiler. It also gives feedback to the compiler developer as to the time spent in each module of the compiler. The compiler developer can use this information to identify performance bottlenecks in the compiler.

External data structures can be added to the compiler to implement new analysis and transformation techniques. It would be helpful if these external data structures used the GUI in a similar manner as the IR. To promote the use

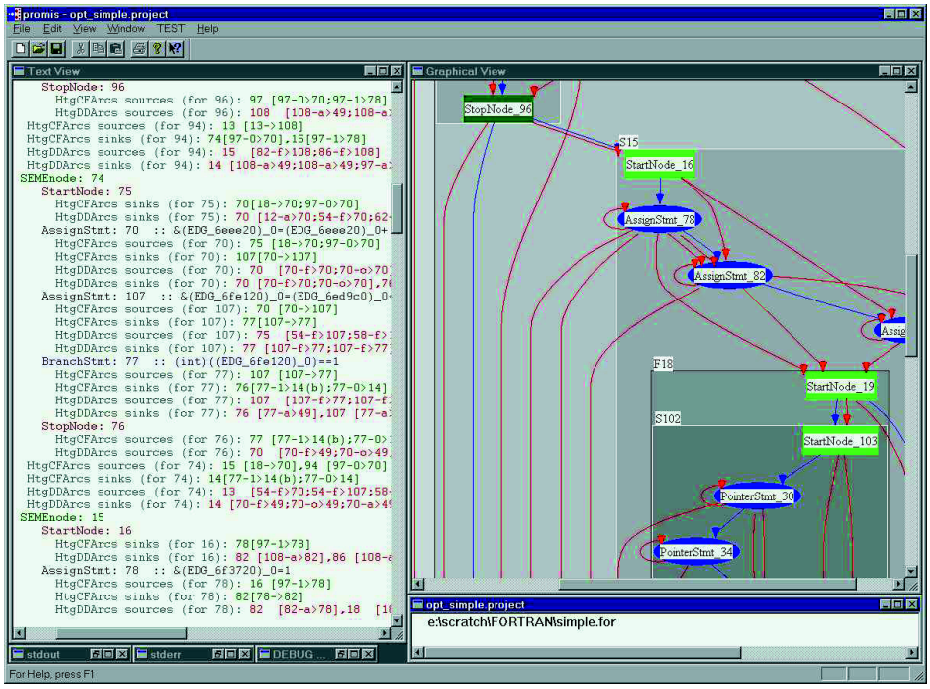


**Fig. 4.** Screen Capture of the PROMIS Compiler

of this common interface, an API has been developed for compiler developers. By defining some functions in the new external data structure, which the GUI can call, the graphical display for the new data structure will be integrated within the existing GUI. Also, the API allows new PROMIS developers to quickly use the power of the GUI without having to spend time learning GUI programming.

Application programmers using PROMIS will be able to give and receive information about the program under compilation. Programmers will be able to receive information, such as profiling information, which they can then use to optimize time consuming portions of the code. Programmers will also be able to give information to the compiler to aid compilation. This information will include dependence arc removal, dead code identification, and value (or range of values) specification for variables.

## 6 Related Work

PROMIS is the successor of the Parafrase-2 Compiler[13] and the EVE Compiler[12]. The PROMIS Proof-Of-Concept (POC) Prototype[6] is the combination of these two compilers. The POC compiler uses semantics retention assertions to propagate data dependence information from Parafrase-2 (frontend) to

EVE (backend). Experimental results on the POC compiler indicate that propagating high-level data dependence information to the backend leads to higher performance and underscore the significance of tradeoffs between inter-processor and intra-processor parallelism[5]. The unified PROMIS IR propagates all dependence information computed at the frontend to the backend, and static/dynamic granularity control is used to achieve better parallelism tradeoffs.

Another compiler effort aiming at similar goals is the National Compiler Infrastructure[14]. The infrastructure is based on the intermediate program format called SUIF[10], and analysis and optimization modules which operate on SUIF. These modules communicate using intermediate output files. SUIF is based on the abstract syntax information of the program, and data dependence information can be represented in the form of annotations[16]. The SUIF compiler system aims at independent development of compiler modules while PROMIS compiler employs an integrated design approach. Zephyr[1] is the other component of the National Compiler Infrastructure. Zephyr is a toolkit that generates a compiler from the input language specification, the target machine description, and a library of analyses and transformations.

Other compiler research projects includes Polaris[3] (parallelizing frontend), IMPACT[7], Massively Scalar Compiler Project[8], and Trimaran[2] (optimizing backend).

## 7 Summary

As computer systems adopt more complex architectures with multiple levels of parallelism and deep memory hierarchies, code generation and optimization becomes an even more challenging problem. With the proliferation of parallel architectures, automatic or user-guided parallelization becomes relevant for high-end PCs to supercomputers. In this paper we presented the PROMIS compiler system, which encompasses automatic parallelization and optimization at all granularity levels, and in particular at the loop and instruction level. Based on the preliminary results obtained from the Proof-of-Concept prototype, we believe that our unique approach to full integration of the frontend and the backend through a common IR, together with aggressive pointer and symbolic analysis will amount to significant performance improvements over that achieved by separate parallelizers and ILP code generators using equally powerful algorithms. Moreover, our design approach does not compromise retargetability and it further facilitates the ability to compile different imperative languages using the same compiler.

PROMIS is an on-going research project with many parts of its optimizer and backend still under development. In this paper, we focused on the design aspects of the compiler. The first performance results on the PROMIS compiler are anticipated to become available in early 1999.

## Ackknowledgements

The authors are grateful to Peter Grun, Ashok Halambi, and Nick Savoiu of University of California at Irvine for their work on the trailblazing part of the backend and pointer analysis contribution to PROMIS. We would also like to thank other members of CSRD who contributed to PROMIS.

## References

1. Zephyr: Tools for a national compiler infrastructure. http://www.cs.virginia.edu/zephyr.
2. Trimaran: An infrastructure for research in instruction-level parallelism. http://www.trimaran.org, September 1998.
3. William Blume et al. Effective automatic parallelization with polaris. *International Journal of Parallel Programming*, May 1995.
4. William J. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization.* PhD thesis, University of Illinois at Urbana-Champaign, 1995. Also available as CSRD Technical Report No.1433.
5. Carrie Brownhill, Alex Nicolau, Steve Novack, and Constantine Polychronopoulos. Achieving multi-level parallelization. In *In Proceedings of the International Symposium on High Performance Computing (ISHPC)*, 1997.
6. Carrie Brownhill, Alex Nicolau, Steve Novack, and Constantine Polychronopoulos. The PROMIS compiler prototype. In *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1997.
7. Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen mei Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 266–275, 1991.
8. Massively Scalar Compiler Group. Massively scalar compiler group. http://softlib.rice.edu/MSCP/MSCP.html.
9. Mohammad R. Haghighat. *Symbolic Analysis for Parallelizing Compilers.* PhD thesis, University of Illinois at Urbana-Champaign, 1994. Also available as a CSRD Technical Report.
10. Mary Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.
11. Kuck and Inc. Associates. Kuck and Associates, Inc. Home Page. http://www.kai.com.
12. Steve Noback. *The EVE Mutation Scheduling Compiler: Adaptive Code Generation for Advanced Microprocessors.* PhD thesis, University of California at Irvine, 1997.
13. Constantine D. Polychronopoulos, Milind Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1):45–72, 1989.
14. The National Compiler Infrastructure Project. The national compiler infrastructure project. http://www-suif.stanford.edu/suif/NCI, January 1998. Also at http://www.cs.virginia.edu/nci.
15. Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, 1995. Also available as CSRD Technical Report No.1432.

16. Robert Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. Technical report, Computer Systems Laboratory, Stanford University.