Chapter 5
The RT-Java Compiler

A prototype RT-Java compiler is under development at PELAB, Linköping University. Our first milestone is an optimizing Java-to-Assembler compiler with real-time properties. The input is either Java source or Java byte code, and the output is assembler for the Intel Pentium II processor. Backends producing Java byte code, Sparc assembler, SHARC assembler, and MIPS/Trimedia assembler are planned in cooperation with the University of Karlsruhe, the University of Saarlands, Associated Computer Experts bv. (ACE), Ericsson, and Phillips. The backends for Pentium II and Sparc already exist, but they have to be modified as described in section 5.3 to provide better garbage collection performance. This work will be done within the JOSES (Java and CoSy for Embedded Systems) Esprit project, WITAS (Wallenberg laboratory for research in Information Technology and Autonomous Systems) at Linköping University, and the Dynamic Control and Configuration of Distributed Real Time Applications project supported by NUTEK.

The compilers are developed using CoSy, which is a compiler construction tool from ACE, producing industry-strength optimizing compilers. The tool has been designed and implemented in cooperation with European universities within the COMPARE and PREPARE Esprit projects.

One of the key benefits of CoSy is that the compiler is built using reusable modules such as frontends, backends, and optimizers. All frontend modules can be combined with all backends and optimizers provided they use the same intermediate representation (IR).

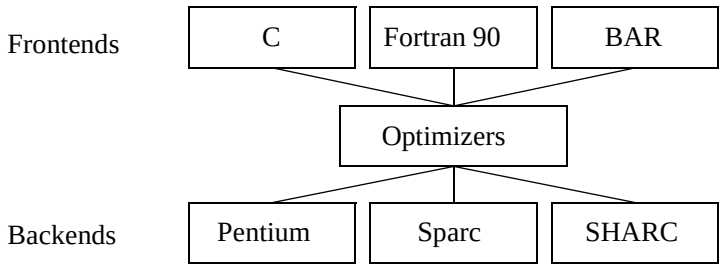

Figure 5.1: CoSy modules

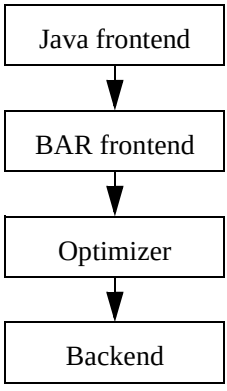

Figure 5.2: The RT-Java compiler (see section 5.4)

5.1 CoSy Compilers

A CoSy compiler is built from modules called engines. The engines communicate via a shared memory called Common Data Pool (CDP). Engines must define the structure of the data they will access in the CDP. These def-

initions are used to generate the Data Manipulation and Control Package (DMCP) library, which engines use to access data in the CDP. The structure of the data in the CDP is defined using the full Structure Definition Language (fSDL). A compiler is constructed by combining engines. This is done using the Engine Description Language (EDL).

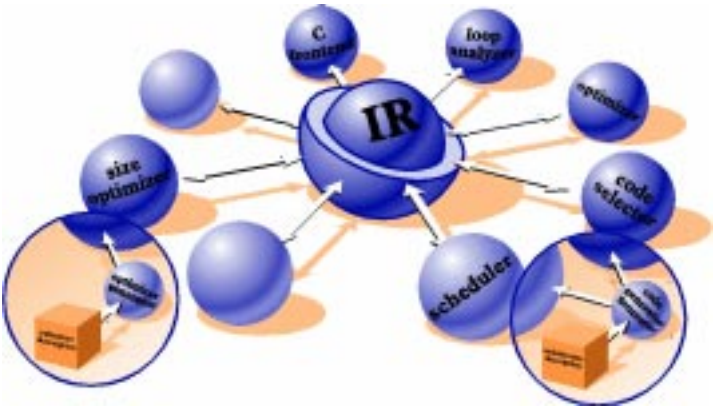

Figure 5.3: CoSy

5.1.1 COSY ENGINES

An engine modifies the data in the CDP. Engines can be programmed in any language that can interface with C, i.e. most modern languages. Engines in a compiler usually include frontends, optimizers, lowerers, and backends. A lowerer is an engine which transforms the internal representation (IR) to a simpler IR-form. This is done to make backends simpler. Lowerers can, for example, transform switch statements into if statements. Engines can also be used to analyze the program and annotate the IR to simplify the work for other engines. An example is the loop analyzer, which finds loop constructs and extracts initialization, condition, advancement, and body information.

The interface of an engine is defined in EDL. Since different engines have different requirements on the IR, engines can specify their own view of the IR. The DMCP library only allows access to the IR which is specified in the fSDL file of the engine.

The backend engines can be produced using a tool called BEG (Back-end Generator). A BEG engine scans the IR graph for patterns which it can translate into assembler. Every possible translation has an associated cost (in time.) The BEG engines select the translation that is cheapest.

5.1.2 THE ENGINE DESCRIPTION LANGUAGE

The engines of a CoSy compiler are glued together using EDL. EDL supports six different ways of combining engines.

pipeline The output of one engine is passed as input to the next.

data-parallel Sometimes data can be divided into sub-graphs. In these cases it is possible to create one engine instance per sub-graph. These instances can be made to work in parallel.

fork Several independent engines can work in parallel.

loop As in a pipeline the data is passed from one engine to the next. In a loop a status engine determines whether the loop should end or continue. If the loop continues, the data is passed from the last engine to the first, otherwise data is passed to the engine following the loop.

speculative Several engines modify the IR graph. The result is passed to a selector engine which determines which result should be used. The results of the other engines is discarded. The output is that of the selected engine.

optimistic This combination is very complex. First one engine is used to generate several potential solutions (graphs) for a specific problem. A status engine determines when enough graphs have been produced and stops the first engine. A third engine selects which graphs are to be considered for further processing. The graphs which pass the third engine are passed on to an instance each of the fourth engine. The instances of the fourth engine work in parallel. Graphs which are rejected by the third engine are discarded. Each instance of the fourth engine processes the graphs. Finally a fifth engine chooses which graph is to be used and discards the rest. The chosen graph is passed on to the next engine.

5.1.3 CCMIR

CoSy does not force engines to use a particular IR, but since supplied engines use the Common COMPARE Medium Intermediate Representation (CCMIR), the use of this IR is recommended. Engines can easily extend the IR without interfering with existing engines. The extension is hidden from engines which do not use them. If another IR is to be used, none of the supplied engines can be used. CCMIR is defined in fSDL as is all data that is communicated between engines.

CCMIR is a graph representation. All CCMIR graphs contain a unit node. Starting from the unit node, all nodes in the graph can be reached using primary edges. The primary edges form a spanning tree of the graph. The fSDL specification specifies which edges are primary.

CCMIR is defined to be language-independent. Currently most imperative constructs are supported. To improve support for object oriented languages, a working group within the JOSES project is designing an OO extension to CCMIR. Other extensions, e.g. a DSP (Digital Signal Processor) extension, already exist.
Bodies of sub-programs are represented by control flow graphs of basic blocks. The code is linear. There are no loop or selection constructs. Loops and selections are represented using conditional and unconditional jumps.

5.1.4 Accessing the Common Data Pool

To access data in the CDP, several methods are supplied. At the lowest level, a set of access functions is generated from the fSDL description. The generated library is called Data Manipulation and Control Package (DMCP). DMCP routines can be used to create and destroy nodes, and to set and get attributes. The functions are named according to a simple schema as described in table 5.1.

Action	**Function name**
Create node of type T	*T*_create(*op*)
Get field F in node of type T	*T*_set_*F*(*node*,*value*)

Action	Function name
Set field F in node of type T	*T*_get_*F*(*node*)

Table 5.1: DMCP naming conventions

To make CDP access more C-like, a C extension, called CoSy-C, is delivered with CoSy. CoSy-C contains constructs for construction and initialization of nodes and attribute access. Attribute access is similar to field access in C-structs. An example is given in figure 5.4.

```
mirEXPR incr('.mirEXPR expr, UnivInt ui) {
  'mirPlus plus;
  'mirIntConst one;

  /* Create and initialize a plus node */
  'plus = mirPlus {
    Group    => TRUE,
    Checked  => TRUE,
    Variance => Variant,
    Strict   => TRUE,
    Modulo   => FALSE
  }

  /* Create and initialize a mirIntConst node */
  'one = mirIntConst {
    Value => ui
  }

  /* Elements can also be accessed like this */
  'plus->Type = 'expr->Type;
  'plus->Left = expr;
  'plus->Right = one;

  return plus;
}
```

Figure 5.4: CoSy-C example

The EMIT library contains functions to help to create and initialize nodes and sub-graphs. EMIT also raises the level of abstractions of the IR by adding labels. By using this library the code is automatically divided into basic blocks. Simplified type creation is another feature of EMIT.

The highest level of abstraction is the stk, which is a stack-based interpreter with an interface similar to the printf function in C. An example is given in figure 5.5.

```
// The statement v = w + 1 could be emitted as
stk(state, "V& V&@ T1i + =",
  v, w, mirLocal_get_Type(v));
```

Figure 5.5: Example of stk code

5.2 BAR - an Interface to CoSy

To simplify frontend development we have designed an ASCII representation of CCMIR called BAR (as in foobar.) A BAR frontend for CoSy has also been developed. The frontend engine is called barre, which is short for BAR reader. Barre parses BAR source and builds the corresponding CCMIR graph. Barre can be combined with any CCMIR engine to create a BAR compiler. Prototype compilers for Pentium II and Sparc have been developed. Both are called barc, short for BAR compiler.

BAR makes it possible to use nearly all front-end construction tools available, e.g. parser generators like yacc. The only requirement is that ASCII files can be produced. Another advantage is that debugging is simplified.

BAR is a prefix notation form of CCMIR. A BAR file contains a sequence of terms. A term can be an operator, a constant, or a list of terms. Operators is the equivalent of nodes in CCMIR. Operators may have attributes, which can be set to values represented by terms. If a value for an attribute is not given, default values can be calculated, e.g. alignment of types. The complete syntax is shown in figure 5.6.

term	::=	*icon* | *rcon* | *scon* | *const*	
		| *op* { *fields* }	
		| [*list*]	
fields	::=	$field_1$ = $term_1$, ..., $field_n$ = $term_n$	$n \geq 0$
list	::=	$term_1$, ..., $term_n$	$n \geq 0$

Figure 5.6: BAR syntax

A complete list of BAR operators can be found in appendix A An example is presented in figure 5.8 where the “Hello world”-program presented in figure 5.7 has been translated into BAR

```
int printf(char *, ...);

int main() {
  printf("Hello World!\n");

  return 0;
}
```

Figure 5.7: Hello World program in C

5.3 Integrating the Garbage Collector

Implementation of a prototype garbage collector can be made using BAR only. The disadvantages of this solution is the lack of control of the produced assembler. If reference count updates can not be done atomically, expensive locks have to be used to prevent erroneous execution. However, if the processor allows atomic increments and decrements, the locks are superfluous. Since the target platform is unknown in the frontend, locks are always needed.

An advantage of reference counting is short locking time. Only reference count updates and list operations (modifying the free list) need locking. Since the locks are very fine-grained, a cheap solution is to disable interrupts during atomic operations. This can not be done in the frontend; other engines have to be garbage- collection-aware.

```
Integer {Name = "int", Size = 32}
Integer {
  Name = "char", Size = 8, Signed = FALSE
}
Pointer {Name = "char*", RefType = "char"}
ProcType {
  Name = "printf", ReturnType = "int",
  Params = [Parameter{Type = "char*"}],
  MoreArgs = TRUE
}

ProcGlobal {
  Linkage = ImportLinkage, Type = "printf",
  Name = "printf"
}

ProcType {
  Name = "main", ReturnType = "int", Params = []
}

ProcGlobal{
  Linkage = ExportLinkage,
  Name = "main",
  Body = [
    Call {
      Proc = "printf",
      Params = [
        CStringConst {Value = "Hello World!\n"}
      ]
    },
    Return {Value = IntConst {Value = 0}}
  ]
}
```

Figure 5.8: Hello World program in BAR

An engine which inserts garbage collection code would provide better performance and flexibility. The garbage collection code should call read- and write-barrier functions where needed. The function calls can later be inlined if appropriate. A major advantage of having an engine inserting

garbage collection code is that the engine can be reused by other frontends. Another advantage is that you do not have to specify any garbage collector algorithm in the frontend. Different algorithms can be implemented in separate engines. To change algorithm, only the engine and the run-time system have to be exchanged.

If efficient code is to be produced, it must be possible to specify that certain instructions should be performed atomically. Thus synchronization has to be added to CCMIR. Backends can then produce processor-specific synchronization if possible. This is the only extension needed to support efficient garbage collection.

5.4 Implementation of the RT-Java Compiler

The RT-Java compiler consists of several modules. The frontend is written in lex, yacc, and RML (Pettersson, 1995). Since it produces Java byte code, any Java-to-byte-code compiler can be used as a replacement. The byte code is then analyzed as described in section 4.4. The memory analyzer adds attributes to the class file. The attributes say which allocation instructions are to allocate objects on the stack. Since the input to this module is Java byte code, pre-compiled code can also be input to the RT-Java compiler. The memory analyzer is currently written in C++. The output is fed into the BAR generator, which translates Java byte code into BAR. The BAR generator, which is implemented in RML, examines the attributes and emits the corresponding allocation instructions. The BAR code is then input to a BAR compiler. Different BAR compilers are used to produce assembler for different platforms There is no working prototype yet, but work is progressing.

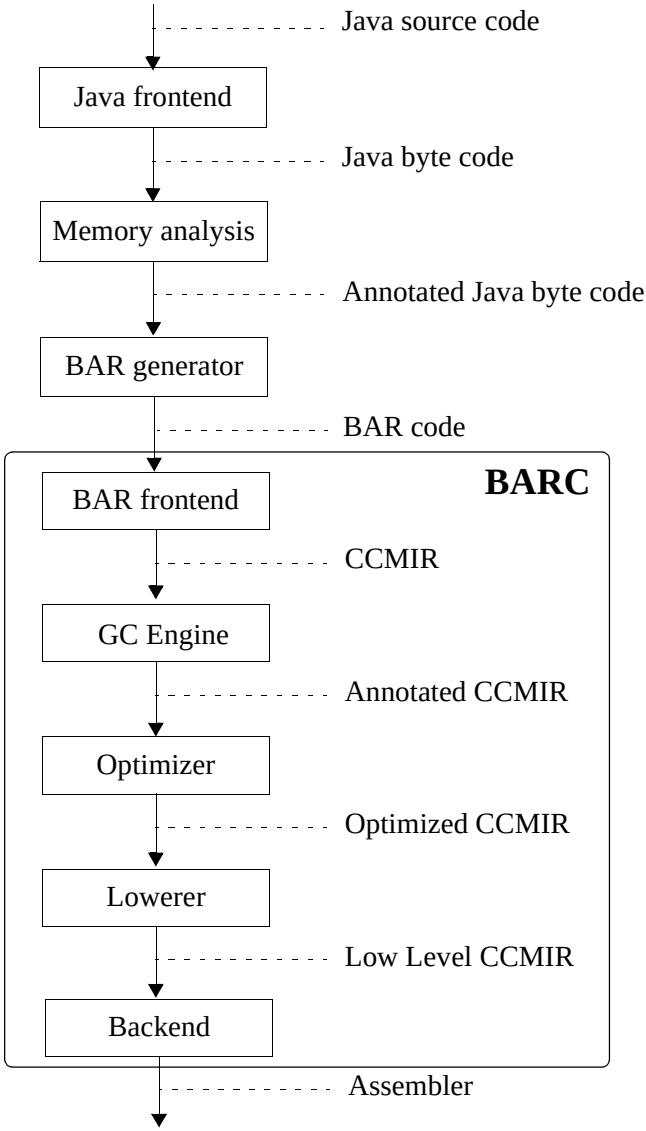

Figure 5.9: The RT-Java compiler