An optimizing C-compiler for the PMS500 processor using the Lcc front end

van Loon, M.R.

Award date:
1995

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Digital Information Systems

Master's Thesis:

An optimizing C-compiler for the PMS500 processor using the Lcc front end

M.R. van Loon

Coach : Ir. A.G.M. Geurts, Drs. C.M. Moerman, H.J.M. Joosten
Supervisor : Prof. Ir. M.P.J. Stevens
Period : July 1994 - February 1995

Table of Contents

List of Tables

List of Figures

List of Algorithms

1 Abstract

To be able to build a complete, optimizing, C-compiler for the PMS500 microprocessor core, a project was started to gather the information and knowledge necessary to build such a compiler and to implement a base from which the compiler could be completed without too much difficulty, i.e. in just a few man-months. An investigation has been held to select a tool to simplify this task. A large number of compiler building toolkits and retargetable compilers have been examined and compared to select the most appropriate candidate. This investigation resulted in the selection of the Lcc retargetable C-compiler to be used as C-front end from which the final compiler could be developed. After thorough examination of the limitations and features of both the target processor and Lcc, a number of optimizations, promising the largest gain in the areas of code elimination and execution time minimalization, were selected. The selected optimizations include loop-optimizations (loop-invariant code detection, induction variable detection), optimizations to eliminate code (dead code elimination through copy- and constant propagation and folding, common subexpression elimination), improving register allocation and, providing they don't interfere with the previously mentioned optimizations, code substitution for faster execution (reduction in strength). The currently selected optimizations are basically target-processor independent (and are strictly speaking part of the front end), but since optimizations that have no effect on code size or execution speed of programs running on the PMS500 are left out, the choice of optimization algorithms can be said to be 'target-processor dependent'. Subsequently, preparations were made to implement these optimizations. The need for different types of data flow analysis have been investigated and methods and algorithms to implement these types of analysis have been provided to deal with the different aspects of the C-language and the Lcc toolkit. These algorithms include a number of data flow analysis types as well as algorithms to build a data flow graph or to handle the effect of pointers in C. Most algorithms have been implemented yielding a base from which most of the selected optimizations can be implemented within the timespan mentioned above. The optimizations for which it is currently possible to write an implementation include induction variable detection, detection of loop-invariant code and the resulting code hoisting, constant -propagation and -folding and dead code elimination, and sufficient information is available from data flow analysis to perform reasonable register allocation by simulated execution. Loop detection, register allocation and code selection as well as the actual implementation of the different optimizations must be done to complete the compiler.

2 Introduction

This is a report on a project performed at Pijnenburg micro-electronics & software b.v. in Vught, in order to acquire the degree of Master of Science from the Eindhoven University of Technology. Pijnenburg has developed a microprocessor core, the PMS500, for which a C-compiler was to be developed. Particularly, since building a complete C-compiler from scratch was recognized to be too large a task to finish in a reasonable amount of time for a graduate project, the objective was to establish a firm base, in the form of documentation and implementation, from which an optimizing C compiler could be built. The largest part of the work consist of providing a basis for optimization.

Modern compilers can be divided into two main parts, the front end and the back end. The front end embodies the source-language dependent actions, while the back end takes care of target language specifics. In most cases the back end also provides solutions for target system requirements and/or optimizations. For higher level programming language compilers, the front end accepts source language programs and translates them into a machine-independent form like intermediate code (IR) or abstract syntax trees (AST's). Subsequently, the back end takes this intermediate representation and adds the machine dependent information to generate the assembly. In most practical compilers, both intermediate code and final assembly are subject to optimization phases.

The front end itself comprises three blocks:

- The scanner, converting source code to tokens
- The parser, combining sequences of tokens to find the syntactic structure of sentences in the source language, often resulting in an AST or a preliminary IR
- The constrainer, doing semantic analysis such as storing symbols and ensuring their correct use, resulting in a decorated AST or final IR.

Machine-independent optimizations, such as common subexpression elimination and copy propagation are also considered part of the front end.

The back end comprises of the code generator and the machine dependent optimization. The code generator performs tasks like outputting the actual assembly instructions, and assigning storage space for symbols (memory or registers). Optimizing can include instruction scheduling (to take advantage of some processor's pipeline), register reallocation (to reduce memory access) or peephole optimization (replacing sequences of instructions with specialised or more suitable instruction sequences).

Existing code generators show two ways to translate the IR or AST to assembly. The first approach is to have the front end generate intermediate code for a virtual machine that resembles the target machine in architecture and instruction set. The back end then translates this intermediate code by means of a simple mapping to the target assembly. This is the simplest way to generate code but this approach usually results in inefficient assembly.

Another way to generate code is to have the front end generate an AST or some IR that is largely independent of the source- and target language or target machine. Assembly instructions are represented by short sequences of intermediate code or AST subtrees. The code generator then substitutes parts of the AST or IR with matching instructions. The generator is also able to translate sequences of IR instructions or subtrees of the AST into semantically identical sequences to match assembly instruction sequences. There are generally two ways to do this substitution:

- Using 'Attribute Grammars (AG's)': The code generator parses the IR in the same way as the front end parses the original source, using abstract grammars. Production rules in this grammar can be attributed with actions to output pieces of assembly, assign values to symbols and so on.
- With 'Tree Pattern Matching' or 'Tree rewriting': Tree pattern matchers try to cover the original AST with subtrees corresponding to assembly instructions, until an assembly instruction sequence is found covering the complete AST. Register allocation is usually delayed until a full match is found and is then done by means of a graph colouring algorithm. 'Tree rewriters' substitute subtrees of the original intermediate tree with nodes representing machine instructions, rather than match them against subtrees.

3 Survey of compiler generating utilities

A sensible way to build a compiler nowadays is to generate it, at least partially, automatically. Much research effort has been put into creating all kinds of compilers from standard descriptions of source- and destination languages. As a result of this, many ready-made compiler generating utilities exist. Because of the popularity of the C language, the possibility exists that some of these utilities can be used or are intended to be used specifically as C-compiler generators. The front end of the compiler, comprising scanning, lexical analysis and syntactic analysis is practically the same for every C-compiler (due to the presence of an ANSI/ISO standard for the C-language). Using existing utilities can therefore keep us from reinventing the wheel, thus saving time and hopefully providing an amount of code that has already been debugged.

A survey of existing compiler generating utilities was made to find the most suitable 'starter kit'. A query was started at the USENET 'comp.compilers' newsgroup, yielding a list of publicly available utilities. Checking references to articles using the 'Science Citation Index' did not result in any utilities besides those already known from the USENET query, but it did bring up a large number of related articles, some of which are ([6, 9]). Simultaneously Paul Jansen at Philips Eindhoven conducted a query for compiler compilers [2], also using USENET, showing that Yacc, Flex, ELI, Cocktail and PCCTS were the most frequently used compiler compilers. Retargetable compilers such as Lcc or Gcc were not included in this query, however. Finally, the Eindhoven University libraries provided interesting reading material ([14, 25]), including an earlier conducted survey on attribute grammars listing over 33 compiler compilers using attribute grammars [13].

Following the survey, some of the most promising options were tried (when available) to verify the different qualities found in the survey. Inspected utilities were ELI, Cocktail, Lcc and Gcc. ELI proved to be a large system but thoroughly documented. A C scanner/parser was included with the package. Cocktail was dropped when it became known that BEG would not be available. Lcc had the smallest size of all packages, and it was not very difficult to see that simple compilers could easily be built using the Lcc front end. Documentation of the Lcc package was limited. However, the new version of Lcc (due September 1994) would be followed by the release of a book about Lcc, expected to come out near the end of 1994. Finally, inspection of existing Gcc compilers showed them to be high quality compilers, using a large amount of resources. Documentation was present in on-line form but was not as complete as the ELI documentation. The front end- back end interface, using Gnu's Register Transfer Language (RTL) was more complex than the Lcc interface (using a tightly coupled system in which the front end and the back end make use of each other's functions), but would be more flexible to use.

To choose the best possible utility, a list of criteria was made. Every utility was checked against this list to decide to which level it was fit to be used. Appendix I shows the list of criteria. These criteria cannot, however, be used to compare utilities point-by-point because of the different nature of some utilities. There are generally three ways to generate a compiler, each resulting in a different type of utility:

- Using a compiler construction toolkit
- Using a retargetable compiler
- Writing it completely from scratch

In case of a compiler toolkit, programs to deal with reside on three levels:

- The toolkit level: the actual utilities. These programs are finished and will only have to be compiled into executables for the machine the compiler will be developed on. System requirements (memory usage, original platform the code was written for), documentation, ease of use have to be considered.
- The compiler level: the resulting compiler or the programs comprising the compiler. These programs have to be compiled into executables for the user platform, which is DOS in our case. Important factors are size of the compiler, compiling speed, debugging capabilities, ANSI conformance and the possibility to generate code for machines with varying register size.
- The target code level: These are the programs written for the PMS500 that are to be compiled by the new compiler. Code size and code speed have to be considered.

When using a retargetable compiler, two levels of code exist:

- The compiler level: Programs that comprise the compiler front end, and that implement an interface to the machine-dependent back end. The previously mentioned considerations still hold, in addition to the documentation of the interface, complexity of said interface, portability of the front end code (what platform was the original front end developed for and how much time does it take to translate it to the DOS platform?)
- The target code level.

Writing a completely new compiler means scanning, lexical- syntactic- and semantical analysis and -checking must be implemented from scratch. As this task was estimated to take up approximately three man-years of time, it was decided not to take this approach.

Appendix II lists compiler toolkits and retargetable compilers considered. Possible advantages, disadvantages and expected problems are also included.

Practical considerations influencing the decision process were cost of a package, availability, copyright restrictions and support. Looking only at the compiler tool kits, Cocktail, PQCC and ACK seemed the most promising as these where the only toolkits with a specific code generator generator. Most toolkits provided similar utilities and only differed in ease of use or completeness, with ELI being the most complete and general, and PCCTS being a very user-friendly but less sophisticated package. Toolkits mentioned in [13] range from 'simpler than PCCTS' to 'comparable with ELI or Cocktail' (including Cocktail itself), but since these toolkits were not reported to be in use and were older than the above toolkits, they where not pursued any further. Further investigation showed that the PQCC project had been abandoned some time ago and satisfying results were never reached.

From the retargetable compilers, Lcc and Gcc seemed the most promising. Lcc for it's ease of use, Gcc for its wide support and the fact that it was known to have been ported to many systems and applications. Archelon would be very well suited to our needs but the copyright restrictions and price prohibited its use. Information on ACC never arrived (Only one reference to its use was found. The information promised never arrived and as it took too much time to get other information, ACC was dropped.). CCG was not to be sold.

Considering the fact that the target processor has a fairly straightforward instruction set and (from a compiler's point of view) a relatively simple architecture (No pipeline, almost every instruction takes one clock cycle), Lcc seemed the best alternative. Compiler generating toolkits would generate front ends that would be larger and slower than the Lcc front end, and Lcc offered an easy way to generate debuggable output. Other points were the fact that ACK was quite expensive, whereas Lcc was free, and the fact that the latest version of Cocktail no longer incorporated its code generator generator (BEG). BEG was at that time in use for an Esprit project and the policy towards selling BEG was not yet clear. The author offered to speed up the decision process, but even then this would take too long so Cocktail lost its main advantage. Another problem was the fact that the C front end, which was reported as written for Cocktail, would not come with the package. This meant either writing a new C description or using a publicly available but incomplete description.

Gcc was (and still is) on of the best optimizing compilers available, and is also able to generate debuggable code. Taking the above points into consideration, the difference between Lcc's (less optimized) and GCC's code would be marginal. GCC, however, is much more difficult to retarget than Lcc, and would also result in a larger, slower compiler due to the built-in options and functionality that would never be used. However, a version of GCC ported to DOS (called DJGPP, from DJ Delorie's DOS port of GPP, the GCC compiler including C++) was initially used as development compiler to build the Lcc compiler.

4 Description of the target processor

This chapter is an extract of [17].
The PMS500 processor contains a 16-bit RISC processor core centered around a dual-ported windowed register file. It uses separate program and data memory spaces. The system architecture of the core is shown in figure 1. The controller is register based. Registers are divided in two groups; the general purpose working registers organised in a register file, and the device registers. Table 1 shows the available registers.

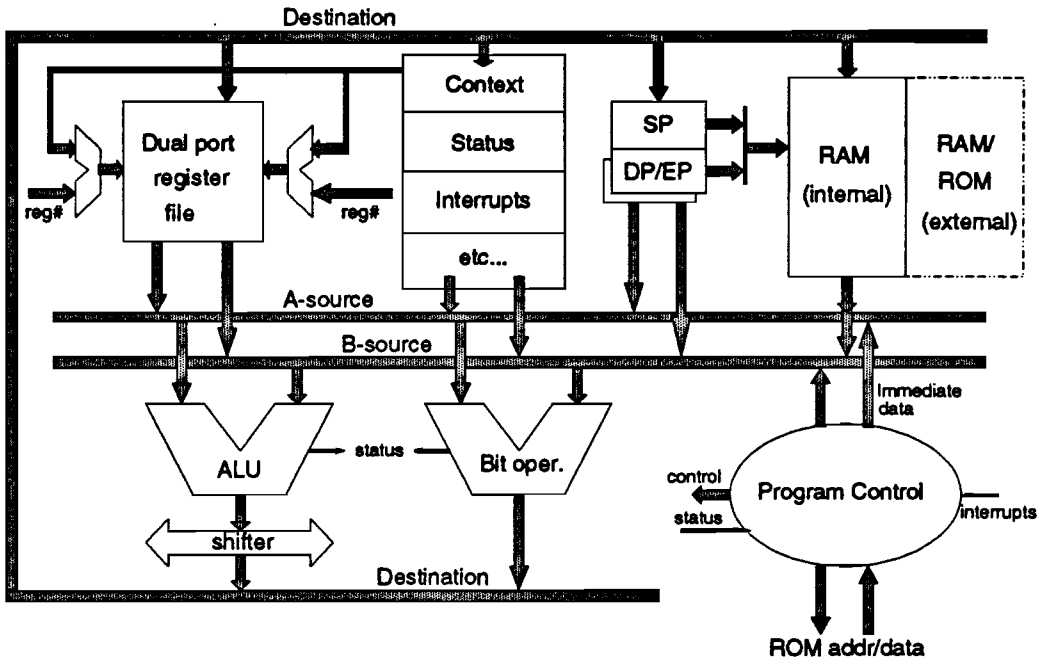

Figure 1 PMS500 processor architecture

The PMS500 is intended for integration with custom specific circuits. It can easily be extended with off-chip customized I/O and other devices, such as A/D and D/A converters, external memory controllers or parallel/serial ports using UARTs.

Program space, data space and I/O space are strictly separated. These three area's can be accessed simultanuously so execution speed is increased.

Name	Access	Description
		General purpose registers
A0..A7	R/W	General purpose arithmetic registers. These registers are a subset of the complete register file, as selected by the current position of the sliding context window (See 4.1).
		Device registers
MODE	R/W	Mode and I/O register bank select.
STAT	R/W	Arithmetic condition codes.
IRQE	R/W	Interrupt enable bits.
IRQS	R	Interrupt status bits. Indicates pending interrupts.
CNTX	R/W	Context register. Determines which register window of the general purpose register bank is visible. (See 4.1)
PC	R/W	Program counter. Accessible for e.g. indirect or calculated jumps and for creating relocateable code.
MULDIV	R/W	Intermediate register used for multiply and division steps.
		Data pointers
SP	R/W	RAM stack pointer. Used to select a specific RAM location, and to stack PC for subroutines/IRQ's.
[SP]	R/W	The RAM contents as selected by the stack pointer.
[SP++]	R/W	The stack pointer can be post-incremented or
[--SP]	R/W	pre-decremented automatically.
DP	R/W	RAM stack pointer. Used to select a specific RAM location.
[DP]	R/W	The RAM contents as selected by the DP data pointer.
[DP++]	R/W	The data pointer can be post-incremented or
[--DP]	R/W	pre-decremented automatically.
EP	R/W	RAM stack pointer. Used to select a specific RAM location.
[EP]	R/W	The RAM contents as selected by the EP data pointer.
[EP++]	R/W	The extra pointer can be post-incremented or
[--EP]	R/W	pre-decremented automatically.
		I/O Registers
IO0..IO3	R/W	Defined by I/O of specific implementation.

Table 1 Available registers

4.1 The CONTEXT switching scheme

The general purpose registers are treated as a window on some sort of stack, the base address of which is contained in the CNTX register. The context space contains a total of 64 registers, of which 8 can be accessed directly. One of the intended usages of this register file by the designers was as follows: Each routine can reserve its own local register set by decrementing the number of words it needs. Any general register above the created local ones is shared with its calling routine, thus enabling parameter passing between them (see figure 2).

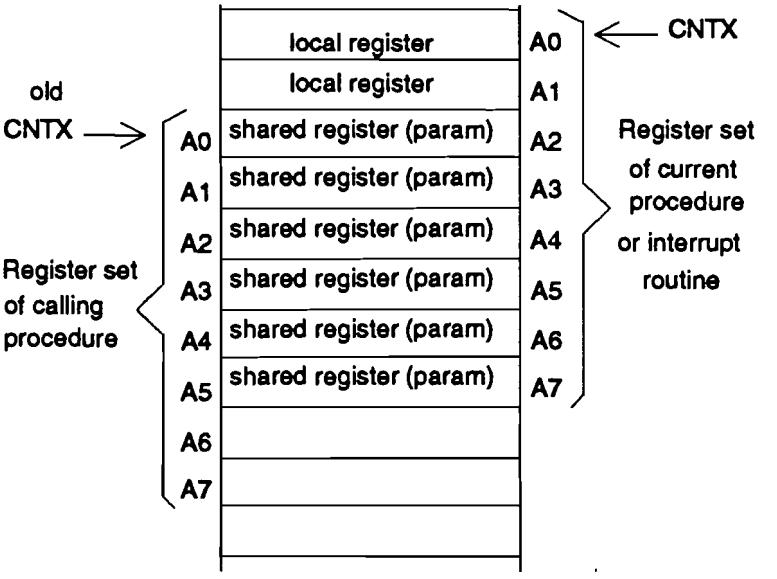

Figure 2 Context switching

The processor automatically generates an interrupt (number 4) when CNTX becomes less than 0 (overflow) or if it reaches value 57 or higher (underflow). For a more specific description of the register file and the interrupt routines, the reader is referred to [17].

4.2 The SP, DP and EP pointers

The stack pointer SP, data pointer DP and extra pointer EP are the only ways in which data in the data space RAM and program space ROM can be accessed. The addressing scheme provides indexed and auto in/decrement modes for all pointers. The SP is used as hardware stack for pushing the program counter PC and STAT register, when a subroutine or interrupt is activated.

Depending on the actual external RAM, only one read or write access may be done in a single instruction. This means read/modify/write instructions such as add (`ADD [SP], #1`) and bit set are not allowed for this type of RAM. Due to the single RAM address bus, instructions like `MOV [DP++], [EP++]` are never allowed. The internal RAM does allow a single cycle read/modify/write operation on a single location.

4.3 The status register

The status register reflects the status from the last arithmetic/logic instruction. MOVes and control flow instructions do not alter the status register bits. The status register is automatically saved when entering an interrupt routine and stored on return from interrupt. When STAT is used as destination register, the actual value written is the result of the ALU operation, not the value of the flags.

4.4 The mode register and I/O

In the PMS500 instruction encoding, 4 I/O addresses are direct accessible. The MODE register is intended as a 'bank' register for the I/O address space. By (externally) implementing this mode register, extra address bits can be added to extend the I/O address range.

4.5 IRQ registers and Interrupts

The PMS500 has 7 interrupt inputs: IRQ0..IRQ3 and IRQ6 are external interrupts. IRQ4 is activated when the CNTX space overflows or underflows. IRQ5 is reserved for a built-in trace mode, which enables single

step execution for easy debugging.

Two registers control the IRQ response of the PMS500:
- The IRQS (status) register shows a status bit for all currently active interrupts. Up to five external events may cause interruption. Only bit 0..6 are used, bit 7 is always cleared. Bit 5 (trace interrupt) is always set.
- The IRQE (enable) register contains individual interrupt enable bits for each interrupt. Clearing these bits disables the corresponding interrupts. Only bit 0..6 are used, bit 7 is always cleared.

Once an interrupt is detected, the processor will:
- stack the status register on [--SP]
- stack the program counter on [--SP], i.e. stack the address of the instruction to be executed after RETI.
- adjust the status register to reflect the current interrupt level, thereby disabling all interrupts of the same or lower priority
- decrement CNTX by 2 (freeing 2 local registers)
- jump to an address, specified by the interrupt number.

Return from interrupt is as follows:
- increment CNTX by 2
- restore PC from [SP++]
- restore status from [SP++]
- enable interrupts of same or lower priority

4.6 The PC register

Data can be moved into the Program Counter to enable calculated jumps (via the `JMP <reg>` instruction or other instructions using the PC as destination) or to use jump tables located in ROM (via the `JMPC` instruction). Using PC as an explicit destination (as in `ADD PC, #1`) will take one extra instruction cycle.

4.7 The PMS500 instruction set

The PMS500 instruction set contains three mayor instruction groups:
- Control flow instructions
- Data transfer instructions
- Arithmetic/Logic instructions.

Appendix V list the instruction set. Note the following points:
- An assembler is present that selects the appropriate `MOV` combination when moving immediate data into a register (i.e. the data5 or data8 and possibly an extra move to HIGH)
- Instructions for moving to and from ROM take extra cycles, and writing to ROM takes extra (external) hardware
- Pushing the stack pointer on stack will first decrement `SP` and then push the decremented value
- Arithmetic and logic instructions operating on immediate data can include only 5 bits of data. However the assembler can generate code to take into account larger data constants.
- Multiply- and divide instructions perform only part of the calculation. A full multiplication (or division) takes 16 steps to complete (see [17] for a full explanation).

5 Building a dumb compiler with Lcc

To get acquainted with the Lcc package and code generation interface, a dumb compiler was built. No assumptions were made with respect to the way in which 'good' code could be generated or what 'good code' should look like. This approach was chosen to gain a better insight in Lcc's code generation interface and the assumptions Lcc makes about the target processor. The design of the real compiler would be based on information found during this recognition phase, thus making it less likely that design decisions would collide with Lcc's assumptions or requirements in a later stage. A better understanding of the code generation interface would also make it easier to take advantage of Lcc's features to simplify 'good' code generation.

5.1 A brief description of the Lcc code generation interface

For better understanding of the following chapters, a brief description of the Lcc code generation interface is added. For a full description, the reader is referred to [1].

The Lcc front end and back end are closely coupled. This means that the front end calls functions from the back end, and vice versa. Both ends share two datastructures: the symbol table and the directed acyclic graph nodes (DAG nodes). The symbol table stores information on name and place of variables, constants and labels. DAG nodes store information on the program flow and program semantics.

The front end provides the following information using the symbol table entries:
- The front end's name for the symbol
- Scope level of the symbol (Global, local, label, constant etc.)
- Storage class of the symbol (Static, register, auto etc.)
- Its type
- In case of a constant symbol, its value or location
- In case of a label, its number
- Additional information, such as whether the symbol is defined, generated, or addressed, or if it's a temporary or a structure parameter.

The back end can annotate these symbol table entries to its own liking with information such as offset from stack, heap address or back end name.

The dag nodes provide the following information:
- The Lcc-opcode for this node (appendix IV lists all available opcodes)
- Number of references to this node's result
- Links to symbols used by this node and/or the kids of this node (nodes that compute values needed by this node's computation)

The back end can annotate these nodes with information like register number or symbol to store the result of the node's computation, or the back end can do the first optimization phase on the DAG. The front end passes DAGs in execution order, sometimes bundling various DAGs in case they share common subexpressions, and forests containing DAGs to set up and execute jump- and switch statements.

The front end manages four logical segments being the code segment, the bss segment (uninitialized variables), the data segments (initialized variables) and the lit segment, containing constants. Code- and literal segments can be mapped onto read-only memory, data and bss segments must be mapped on read- and writable memory. These segments can be declared to the back end in random order, so it may be possible that references to (for instance) labels in a segment occur before they have actually been declared.

When compiling a source program, the front end first announces global symbols and symbols to be exported or imported, such as function names and externally defined variables. The front end will switch to the appropriate segment before announcing symbols belonging to that particular segment. If no global symbols are announced in one segment this segment may be declared after the code segment.

Generating program code is done in the following way:

The front end first completely consumes a function before calling the back end. The back end then gets the opportunity to initialize the annotation phase. Control is then returned to the front end that in its turn repeatedly calls the back end for every DAG forest in the function, so the back end can annotate the DAG. After that, the front end returns control to the back end to initialize the code generating phase. Control is passed back again to the front end that passes the annotated forests in sequence to the back end to emit the final code. Finally, the back end may round up the code generation phase and the front end continues to read the next function from the source file.

5.2 Description of the dumb compiler

5.2.1 Assumptions

Because the sole purpose of building the compiler was to determine Lcc's assumptions, features and shortcomings, the following (simplifying) assumptions were made:

- No effort was put into correct representation of different variable types. The front end provides ample opportunity to correctly implement this functionality as can be seen from the table in appendix IV For the dumb compiler, it is assumed that the value of a basic variable type fits into one PMS500 word.
- No effort was put into dynamic memory allocation. Locals and function parameters reside on the stack, which is assumed to be infinitely large. Globals and statics are assumed to be initialized by a (nonexistent) linker.
- Possible optimization of the DAGs was not investigated.
- A0, A1 and A2 are reserved for use by the compiler to pass function return values and copy blocks to stack (A0), hold the base address of the current stack frame (A1) or to hold the address of temporaries to which register values are spilled (A2).
- To free registers at function entry, 8 is added to CNTX. CNTX is restored at function exit. The register file is also assumed to be infinite.
- Certain functions (multiply, divide and modulus) that take a sequence of assembly instructions are not expanded.
- Functions returning structures are not supported

Figure 3 shows the layout of the stack frame used. During the first code generation phase, maximum local offset and maximum argument offset are calculated. At function entry, the stack pointer is decreased according to the sum of these values to declare the necessary stack space. This approach enables the use of the PUSH and POP commands without having to keep track of the location of locals or arguments relative to the stack pointer.

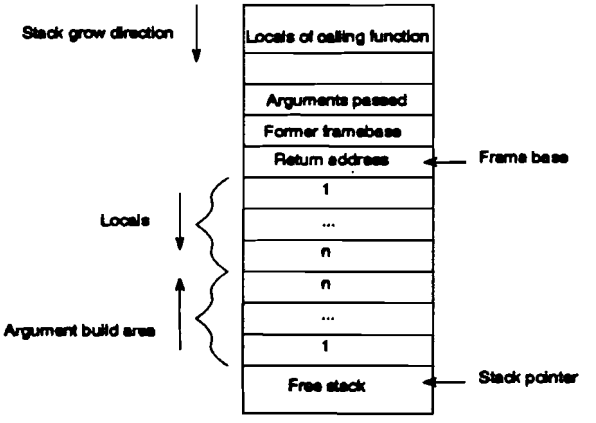

Figure 3 Stack frame layout of the dumb compiler

Register allocation was taken from the example VAX code generator that came with Lcc. Only small changes were necessary to make it suitable for the PMS500 code generator. Registers are allocated on the fly, and the register allocation algorithm does not keep track of values of symbols already present in a register; All symbols are fetched from memory the moment their values are needed, except values used more than once per DAG forest. Register variables are not supported by the register allocator.

5.2.2 Problems, possible solutions and useful information provided by Lcc

Problems or possible problems were encountered in the following area's:

Segment management:
Lcc manages four logical segments, including a read only and a read/write data segment. String literals, for instance, might be declared inside the read-only segment. It cannot, however, be computed at compile time whether a pointer dereference accesses read only or read/write space. String literals can therefore not be mapped onto PMS500 codespace since the compiler is unable to determine if it should use the MOV or the MOVC instruction. Besides that, Lcc generates the code to initialize variables inside the data- and literal segments. Strictly speaking this means that this is code that will not be executed by the PMS500 processor but should be interpreted by the PMS500 assembler and linker. One or both modules should therefore be able to initialize memory for the compiler, for example by generating initialization routines in the startup code. If the compiled program is to be able to initialize all variables by itself, then the linker should first call the module's initialization routines before calling 'main'.

Register allocation:
Because the front end passes the DAGs to the back end in execution order and at most one forest at a time, it is difficult to allocate registers on a more global level. If this is to be implemented, extra information containing symbol lifetime and storage location must be added during the first phase of code generation. This information can be used to choose which variables should be allocated to registers rather than to memory. It will also be necessary to combine forests into basic blocks and basic blocks with each other in the back end to get a complete view of variable lifetime and usage. Chapter 8.3 explains the concept of basic blocks.

Instruction layout:
Lcc assumes for arithmetic instructions the presence of three operands for diadic, and two operands for monadic instructions. An ADD instruction, for instance, adds the values of two registers and stores the result in a third, whereas the PMS500 ADD instruction adds the values of two registers and stores the result in one of the original registers. It might therefore take one extra cycle for the PMS500 processor to move the value of one of the original registers into the third, assigned by the front end, for every arithmetic instruction.

Overhead caused by calculating the address of locals and arguments:
The dumb code generator has to calculate the address of locals and arguments every time the value of such a variable is used. This results in nearly 20% of the generated code consisting of address calculations. This is partly a result of the choice to store every local variable or argument on stack and the percentage might be lowered by choosing another storage method. However, the addition of an instruction to calculate this address in one instruction could achieve an easy gain in speed and code size. To find out if such an instruction would indeed cause a fundamental improvement in code size and execution time, a small investigation was held (chapter 6).

Lcc provides information on the number of times the result of a node is used after it has been calculated, and whether its address is taken. This is useful when deciding if a symbol can be assigned to a register rather than to a memory location. A relation between the symbol (in the symbol table) and the node (in a DAG forest) has to be calculated and stored for this purpose.

5.2.3 Possible optimizations

Building the dumb compiler, several points were found on which the generated code could be improved without too much trouble. At this point, optimization means those transformations on the assembly code that result in less and/or faster executing code. Lcc does some local optimizations such as constant folding and eliminating conditional jumps with a constant condition. The code that cannot be reached via these eliminated jumps is still generated, though.

Optimizations than can easily be implemented:
- Every node for which the value can be calculated at compile time need not be emitted but can be substituted directly wherever the value is used (this equals tree pattern matching, in which subtrees of the AST are matched against subtrees representing instructions). This goes for:
  - Addresses of labels,
  - Constant values or expressions,

    - Offsets to locals and arguments.
- Keeping track of values in registers, value lifetime and distance to next use can aid in:
    - Assigning frequently used symbols to registers,
    - Minimizing references to memory,
    - Generating better spill code.

Optimizations that will need a more fundamental change in approach:

- Building and linking of basic blocks. This will make it possible to:
    - Allocate registers globally,
    - Eliminate dead code,
    - and perform many other types of global optimizations.
- Changing the way locals and arguments are stored and passed. This could mean, for instance, that addresses of symbols no longer have to be calculated as an offset from stack but can be accessed directly in memory. To enable this approach, dynamic memory allocation has to be provided by the startup code or host operating system (if available), or a protocol has to be invented so the compiler can allocate memory by itself. Argument passing could make use of the context register file that could speed up the process, but would introduce the problem of reindexing all registers in use.
- Loop optimizations, such as invariant code movement, can be implemented.

5.2.4 Summary

Lcc does not make assumptions about the type of processor it will generate assembly for, nor about the kind of environment in which it will run, that provide any real problems concerning the PMS500 processor. The way in which symbols are declared implicitly assumes the presence of an assembler, for instance because the generation of code for variable initialization is left to the assembler. Since the existing PMS500 assembler did not recognize multiple data segments, a solution had to be found to initialize data in other than the code segment. But the PMS500, being an embedded processor, allowes for multiple types of external ram, so the use of segments or other ways to discriminate between these memory areas had to be added to the assembler. If these different types of external memory are to be discriminated by means of different assembly instructions, however, then the properties of the C-language make it impossible to effectively use these different kinds of memory. Discrimination between segments using different address ranges in the same numbering space, can be supported by the compiler (to the extend of the four segments managed by Lcc), but the assembler must make sure that labels declared in one of the segments indeed refer to addresses inside the correct address range. All address ranges must then be accessible through one machine instruction (instead of using either `MOV` or `MOVC`)

Various types of code improvements can be added without fundamentally changing the structure of the dumb compiler but to be able to add global optimizations, the DAG forests Lcc sends to the back end have to be rejoined, meaning that some of the work done by Lcc has to be undone.

6 Investigation for useful additions to the PMS500 instruction set

To be able to make suggestions about extensions to the PMS500 instruction set, a small investigation was held to determine the effect of some extensions on code size and program execution time. Investigated additions were the possibility to add a constant to a register before moving its value into another register (like `MOV A0, A1+3`), and to access the value of a memory location addressed by an address pointer plus a constant offset (such as `MOV A0, [DP+2]`, which moves the value at the memory address designated by `DP` added with two into `A0`). These instructions were considered because it is inevitable that a C compiler uses offsets from a known address to designate local variables; the addresses of these locals have to be calculated at run time while the compiler needs to have a scheme to designate these locals as well (compile time). Optimization can focus on minimizing these address calculations, but since values have to be assigned to variables at ANSI C's agreement points, a substantial amount of code will be dedicated to calculating the addresses of locals. (Agreement points are points in the source code at which the values of variables as stored in the real machine have to be identical to the values of variables as if the code was run on the abstract machine defined by ANSI C).

Source module	# of lines	# of usages	% of usage
dag	10,127	1,243	12.27%
decl	15,159	1,761	11.62%
enode	8,915	1,246	13.98%
error	1,054	112	10.63%
expr	16,735	2,040	12.19%
init	6,267	792	12.64%
input	1,239	46	3.71%
lex	7,477	402	5.38%
main	928	85	9.16%
output	1,494	132	8.84%
profio	2,917	344	11.79%
simp	13,883	1,850	13.33%
stmt	9,033	1,129	12.50%
string	1,668	138	8.27%
sym	3,634	404	11.12%
tree	3,347	359	10.73%
types	11,536	1,431	12.40%

Table 2 Usage statistics of MOV Ax, Ay+c instruction

To gain insight in the amount of space and time the addition of one of the above instructions would save, the dumb compiler was modified to assume the presence of a type `MOV A0, A1+x` instruction. The number of times the instruction was used was counted and compared to the total number of code lines in the module. Table 2 shows the results for the modules comprising the Lcc front end. On the average, 10% of the code consist of this new instruction. This means that program code will be 10% larger without this instruction as it expands to a `MOV` and an `ADD` instruction in the current instruction set. Execution time will increase by about 10%, as the effect of loading the HIGH register when dealing with large constants has to be taken into account.

Checking the usage of the `MOV A0, [DP+x]`-type instruction was not possible in this way due to the structure of the dumb compiler. A similar usage count was performed on code for the INTEL 386 processor, that possesses this instruction type. This showed that an average of 20% of the code consisted of indirections on a register value plus an offset, for the same set of modules. The modules were compiled both optimized and non-optimized. This can be seen as an indication that if the processor provides the instruction, it will be used a lot. It does not indicate, however, the gain in code size or execution time compared to code lacking this instruction type, although every time a local variable has to be referenced, this instruction can save up to two 'ordinary' instructions.

Considerations whether these types of instructions belong in the instruction set of a RISC processor or if these instructions impose problems on the processor design (data path length etc.) were left to the designers of the processor (of course). It is not possible to form a reliable conclusion based on the results acquired by using the dumb compiler and an Intel 386 compiler, besides the fact that the extension would be easy to have in the eyes of a compiler writer. It may well be possible that a compiler designed to minimize these types of calculations could do perfectly well without the extra instructions. Further investigation into this subject was not considered to be an objective for this project and the investigation was abandoned.

7 Structure of the final compiler

Based on the information gathered while building the dumb compiler, a number of decisions were made concerning the implementation of the real compiler. These decisions affect the function call interface, register usage and allocation, stack frame layout and memory usage. Below is the list of points the compiler will have to conform to:

- The compiler assumes library functions or assembler macro's for the following actions:
  - Multiply, divide and modulus. These operations take a large number of instructions to implement, and inline expansion is not always desirable.
  - Dynamic memory allocation. This was not primarily found to be a compiler problem and leaving this functionality to library routines enables multiple allocation schemes for different memory types. The decision to use stack as dynamic storage however makes this assumption redundant.
  - Floating point operations. These functions also take more than one PMS500 instruction to be implemented. It also enables the programmer to choose different floating point libraries (if provided).
  - Shift and rotate over multiple bits. As the PMS500 isn't equipped with a barrel shifter, these functions also take a number of instructions to implement.
- The compiler will calculate an upper bound for relative jump distances and decide accordingly which jump construct will be used. Peephole optimization can then be used to collapse long jumps to relative jumps if possible. Calculating this upper bound is an easier task for the compiler than for the assembler.
- The compiler will not recognize different memory types the user might add to the PMS500 core besides the code- and data memory. This effectively means that all datapointers used by the compiler are assumed to address the dataspace in which the stack resides (or to which SP points). DP will be used as framepointer (see below), EP will be used for block copy/move instructions.
- DP will be used as framepointer. Addresses of locals and argument will be calculated this way:

```
ADD DP, offset
MOV Ax, [DP]
SUB DP, offset
```

This approach enables merging of successive SUB- and ADD instructions when calculating multiple addresses and keeps the general purpose registers free for other uses.

- The context-stack will not be used by the compiler. CNTX is reserved for use by interrupt routines. This leaves eight available registers: A0-A7.
- Locals and arguments will be allocated on the stack. Globals and static variables will be declared to the assembler which will concern itself with the actual storage allocation.
- The callee (in stead of the caller) saves and restores the registers it uses; it also saves and restores the caller's framepointer. This choice enables assembly writers to interface with the compiler-generated code and save/restore only the necessary number of registers.
- A stack frame will be similar to the stack frame used by the dumb compiler, with the exception of the argument build area. The dumb compiler reserved enough stack space at function entry to be able to handle all function calls from this function, which means declaring enough stack space to handle the function with the largest number of arguments. The real compiler will push arguments on stack and pop them again at function exit, thus making more efficient use of stack space.
- The caller removes arguments from stack. In case of library functions with a variable list of arguments, the callee doesn't even know how much of the stack should be freed.
- ANSI dictates the following lower bounds for type sizes (in bits): CHAR: 8; INT: 16; LONG: 32; FLOAT: 32. Lcc assumes INT and LONG types of equal length. This would mean that INT=LONG=32 bits, even for the 16 bit PMS500. This poses a serious problem. Separating INT's and LONG's, aliasing LONG's and DOUBLE's or replacing SHORT's for LONG's will take a considerable amount of redesign of the Lcc front end. For now, INT=LONG=16 bits is assumed for the 16 bit PMS, invalidating the compiler as ANSI conforming. The 32-bits PMS500 compiler can safely assume INT=LONG=32 bits.
- Registers will be allocated with 'Function scope'. Allocation schemes will be investigated starting with graph colouring.
- The compiler will perform global (function level) optimization, optimizing primarily for program size.

Most decisions only affect the last steps of the code generation process, when the code is actually emitted.

Register allocation and global optimization, however, induce a number of analysis steps preliminary to the emitting stage. Both register allocation and optimization require data flow analysis to decide which values to store in registers, which variables can be substituted by constants and so forth. The compiler back end will therefore perform the following operations:

- Data flow graph construction. Necessary to perform data flow analysis.
- Data flow analysis.
- Global (function level) and local (basic block level) optimization using the results of data flow analysis.
- Global register allocation.
- Code selection and emitting.
- Peephole optimization.

To some extent, local optimizations such as strength reduction and common subexpression elimination are performed by the front end (always concerning a few code-trees at a time but not complete basic blocks) but might be extended to cover complete basic blocks or even functions. The compiler parts will be constructed in the following order: first data flow graph construction and data flow analysis, because this step is obligatory for both register allocation and optimization. At that point a number of optimizations should be investigated for implementation cost, computation cost and optimizing effect. After the desired optimizations have been chosen, the methods for implementing these optimization have to be put on paper to make it possible for other programmers to implement the desired algorithms. Subsequently the register allocation algorithm will be chosen and implemented. Finally the emitting stage of the compiler will be constructed. This completes the first version of the compiler and marks the first point in time on which correct machine code can be generated by the compiler. It is at that point possible to add local, global and peephole optimization to the compiler.

7.1 Possible optimizations

[18, 5, 12, 14, 15, 23 and 25] list a number of optimizations and data flow analysis techniques to aid in optimization that can be performed during or after code generation. These optimizations generally aim to reduce the amount of machine code produced by the compiler and at the same time minimize the execution time of the produced program. Often these aims interfere with each other. Most of the optimizing techniques only work or work best if the data flow graph of the program is reducible (See [7] for information on reducible flow graphs). It is easily seen, however, that data flow graphs derived from C programs need not be reducible per se, so a number of techniques cannot be implemented or will have greater complexity when implemented for a C compiler. The following paragraphs list the optimizations that are considered useful for the PMS500 compiler and are therefore candidates for implementation.

Because the PMS500 was intended to be used as a processor core that could easily be embedded in specific applications, and the amount of memory in these systems is generally fairly small, it was decided that optimization of program size would prevail over minimizing the execution time of target programs. Based on this decision, the most important optimization techniques are:

- Loop optimizations
- Code elimination
- Code substitution

Other optimizations such as instruction scheduling or code selection using tree rewriting yield a relatively low optimizing performance because the PMS500 has no visible pipeline, executes every instruction in one clock cycle (apart from a few instructions that need the HIGH register such as load immediate with large constants) and has a fairly orthogonal instruction set (i.e. functionality of instructions doesn't overlap making it unlikely that a sequence of instructions can be substituted for one, more complex, instruction).

Local optimizations affect only basic blocks. Even when Lcc performs a number of optimizations on code trees, local optimization can find a number of optimizations that can be performed parallel to global optimization, such as (local) common subexpression elimination, strength reduction or copy propagations. Local optimizations are easier to perform since it is not necessary to do data flow analysis.

Loop optimization algorithms try to reduce the number of instructions contained in the body of the loop, or the number of times the loop body is executed. Since most programs spent 90% of their execution time inside loops, these optimizations provide a large gain in execution time minimalization while at the same time decreasing the program size (excluding the loop unrolling algorithm).

Code that will never be executed (dead code) can be eliminated and calculations occurring more than once and yielding the same result (common subexpressions) can be substituted with a temporary so the expression needs to be evaluated only once, its result can be used many times and every other occurrence of the subexpression can be eliminated. These optimization techniques can be used to reduce the size of the target program and, to a lesser amount, decrease the execution time of the program.

7.2 Data flow analysis

Most optimization problems require data flow analysis, so let us first, for a better understanding, establish what data flow analysis means. The different types of data flow analysis must also be identified. [18] defines data flow analysis as "the transmission of useful relationships from all parts of the program to the places where the information can be of use". These 'useful relationships' include relations between the occurrence and the usage of a variable definition or the availability of (sub)expressions at any point in the program.

Data flow analysis can be done in two directions. Forward data flow analysis takes information from certain points in the program and tries to propagate the information through the program to points were it might be used. Backward analysis does the exact opposite; it recognizes points that use some kind of information and tries to trace points in the program where the information might have been generated. The information propagation for both types of analysis can be done based on confluence or divergence. For instance, analysis based on confluence initially assumes that nothing is valid, but if, at some point in the program, the possibility exists that information may become valid, it is propagated until it is absolutely sure it becomes invalid. Analysis based on divergence initially assumes that anything is valid but if at some point in the program the chance exists that the information might become invalid, propagation of this information beyond that point is stopped and is only started again if it is absolutely sure the information becomes valid again.

An example of forward flow analysis based on confluence is the analysis of 'reaching definitions'. The aim of this analysis is to establish which definitions of variables reach uses of certain variables (e.g. in the expression x=a+b, x is defined and a and b are used). This type of analysis is can be used to decide wether a used variable can be substituted for a constant value (constant propagation). Such a substitution can only be made if it is absolutely sure the variable can only have one particular value at that point in the program. So if there exist different paths from the start of the program to the point in question, and each path contains a different definition of the variable, the variable cannot be substituted. If no definitions occur on both paths, the variable can also not be substituted (note that this generally means a programming error). So it is clear that the initial condition is 'no variable is defined', and every point y in the program that might define a variable x adds a 'x is defined at y' to the information heap. Only if it is absolutely certain that a variable is defined at some point, all other definitions of the same variable prior to this point are removed from the heap.

7.3 Loop optimization

As stated, loop optimizations provide an easy way to speed up the program with relatively little effort. To be able to perform loop optimizations, the following information has to be gathered from the source program:

- The Data Flow Graph (DFG) of the program has to be constructed. (See chapter 8.3 for more information on DFG's)
- Loops in the DFG must be discovered. A loop is a collection of nodes that are connected in such a way that from every node in the loop, walking the connections, any other node (including itself) can be reached. To make optimization possible, a loop must also have one entry node, the only node through

which any node inside the loop can be reached from outside the loop.
- The flow of information through the program must be analysed. (Data Flow Analysis or DFA). DFA makes it possible to detect computations that yield the same result every time the body of the loop is executed, regardless of the conditions changed by multiple execution of the loop body. These computations can be moved out of the loop. It can also provide information about the number of times a loop will be executed.

Detecting loop invariant expressions makes use of usage-definition information, derived from the 'reaching definitions' information described in the previous chapter, and can be implemented using an algorithm that iterates over the loop. Detection of loops can be done using an iterative algorithm for data flow graphs in general and using depth-first ordering or dominance relations if the flow graph is reducible. In C, however, the possibility exists that flow graphs are not reducible.

Loops provide a number of sources for optimizations. Summarizing the possibilities, we have (from [18, 14] and [25]):
- Movement of loop-invariant computations out of the loop-body (Code Motion).
- Induction variable elimination.
- Loop unrolling.
- Loop jamming.

Especially code motion and induction variable elimination promise large gain in execution speed with little programming- and computation effort. Loop-invariant computations can easily be detected if usage-definition information has been calculated (see chapter 8). Even if, for instance, only two expressions can be moved outside the loop, the gain will be substantial as the loop body executes repeatedly. Induction variable elimination tries to detect variables that increase or decrease linearly with the loop counter. These variables can be introduced by the programmer, if for instance an expression `j=c*i+d` inside the loop body exists, with `i` the loop counter and `c` and `d` constants. `I` and `j` are both induction variables. Induction variables can also result from array index calculation, if the programmer uses the loop counter in expressions like `A[i]=B[i]`. To calculate the actual memory address, `i` will be added to a pointer, making the result of this addition an induction variable. Loop invariant parts of these computations can then be moved outside the loop and their results used directly, test for conditional branches can be simplified, and expressions can be reduced in strength (an assignment of the type `x=c*i` can be substituted for an initialisation $x=c*i_0$ outside the loop and an addition `x=x+d` with `d` equal to `c` times the amount that `i` gets increased every time the loop body is executed. To detect induction variables, information about loop-invariant expressions must be calculated and the expressions inside the loop body must be scanned. Data flow information is also necessary.

Loop unrolling and loop jamming try to reduce the overhead from the loop entry- or exit tests. Unrolling a loop (replicating the body of the loop) avoids one test per replication every time the loop is iterated. Loops can be jammed (merging bodies of two loops in one loop) if both loops get executed the same number of times and the data flow through both loops doesn't interfere. Jamming two loops clearly saves the tests of one loop. Evidently, loop jamming can seldom be performed, while the costs in programming- and compilation-time are substantial (Data flow analysis and loop invariant expression detection are necessary, besides the actual loop analysis to detect if and how loops can be unrolled or jammed). Loop unrolling is a tradeoff between code speed and code size. Not unrolling a loop yields the smallest code which was considered more important than faster code.

7.4 Code elimination and code substitution

Code can generally be eliminated if it is never executed, if its result is never used or if the result of the code can be calculated by simpler code sequences or as part of already existing code. Code sequences that are candidates for elimination can result from the following actions:
- Constant propagation.
- Copy propagation.
- Global or local common subexpression elimination.
- Data flow analysis in general.

Constant- and copy propagation and common subexpression elimination both require data flow analysis, but forward data flow analysis by itself may show the presence of expressions whose results are never used and can thus be eliminated. Propagating constants means that variables whose value at some point in the program is known, independent of the state of the program, the value can be used directly in the expression instead of fetching the value runtime from memory. Common subexpression elimination is the substitution of (sub)expressions by a temporary variable used to store the result of the (sub)expression, so the (sub)expression has to be evaluated only once.

Constant propagation can lead to even more code elimination. Consider the situation that a constant can be propagated until it reaches the test of some conditional branch. If this test becomes a constant expression, one of the branch targets will never be executed by this expression. This again may lead to the situation that the unused branch becomes dead, and can be eliminated completely. Thus a test and possibly even a complete instruction sequence can be eliminated.

Copy propagation considers expressions of type y=x, in which the value of variable x is copied into variable y. Whenever y is used after such an expression, x can be substituted. If y is a temporary variable, copy propagation may eliminate all subsequent references to y, and the expression y=x can be eliminated, saving code as well as run-time memory usage. Copy propagation may also free extra registers.

We have already seen that the result of data flow analysis is used so often that incorporating it into the final compiler is necessary to perform any optimization at all. Constant propagation can easily be implemented using only this information; to add copy propagation it is necessary to identify the copying expressions and to use data flow information to decide what variables may be substituted. Common subexpression elimination needs information on availability of expressions. Finding available expressions is a forward dataflow analysis problem.

8 Data Flow Analysis

Prior to the execution of the actual data flow analysis algorithms, a number of things have to be set up. The codegraph has to be partitioned into basic blocks, which will be interconnected to form the data flow graph. Every code tree needs to be examined to find out what its effect is on the data flow. For every basic block, the data from the code trees has to be gathered concerning the information generated inside the basic block (referred to as the GEN set), and information that gets killed inside the basic block (the KILL set). Note that different types of data flow analysis (DFA) require different interpretations of these GEN and KILL blocks; chapter 8 elaborates on this notion. From these GEN and KILL sets, data flow analysis calculates the information entering a basic block (IN set) and leaving the basic block (the OUT set). Every type of DFA calculates different IN and OUT sets, as do the same types of DFA concerning different types of information.

8.1 Collecting reference- and definition information

Reference- and definition information (ref-def information) gives, per code tree, information on the variables (symbols) used in this tree, and the variable or symbol defined by this tree. If a code tree represents an assignment, the definition information equals the symbol at the left side of the assignment-operator and the reference-information is the set of symbols occurring at the right side of the assignment-operator. If the code tree represents a jump statement, reference- and definition information both equal the target label.

Ref-def information can be used to determine the targets of jumps, branches, and calls. It also is the starting point for different types of global data flow analysis such as alias-analysis and reaching definitions, described in chapters 8.4 and 8.5. The information collected will always be some pointer to a symbol in the front end's symbol table, coupled with a number representing the number of times the symbol is 'dereferenced'. A symbol can be dereferenced by means of the INDIR and ASGN operators (appendix IV), corresponding to the C-operators '*' and '='. Dereferencing a symbol is equal to taking the value of the memory location pointed to by the symbol. This can be done more than once. To represent such a symbol pointer/dereference level pair, the following conventions is followed: Let S be a symbol in the symbol table, and N be a (non-negative) integer. Let * be the dereferencing operator and & be the 'address of' operator (similar to their meaning in the C-language). Then the tuple (S,0) represents the (run-time) address in memory of the symbol, &S. (S,N) equals *(S,N-1), the value acquired by fetching the contents of the memory location denoted by (S, N-1).
Note that (S,1) denotes the 'actual' value of S and that the presence of tuple (S,N) stipulates the presence of pointers denoted by tuples (S,n), $0<n<N$. So following this notation, an assignment of the form a=*c+1 can be written as (a,1)=(c,2)+('1',1). Note that the Lcc front end creates symbols for every constant used, so the constant 1 used in the assignment results in a symbol '1' in the symbol table.

Every node of a code tree can now be said to have a 'points to' tuple and a set of 'uses' tuples. The set of 'uses' tuples represent the values used to calculate the value of the node. The 'points to' tuple, only defined for nodes operating on pointers, is used to extract the symbol used in the code tree that supplies the original address; symbols used to calculate offsets from this address are added to the uses-set. Only the ASGN nodes, whose sole occurrence may be as the root of a code tree, define an extra tuple 'defs', representing the memory location defined by the assignment (in the above assignment, a would be represented by the 'defs' tuple as (a,1); if a were a pointer (in which case *c must also be a pointer), then the 'points to' tuple would hold (c,2); finally the 'uses' set would be {(c,1),(c,2);('1',1)}).

Code tree and ref-def information for a[1]=b[i]+c;

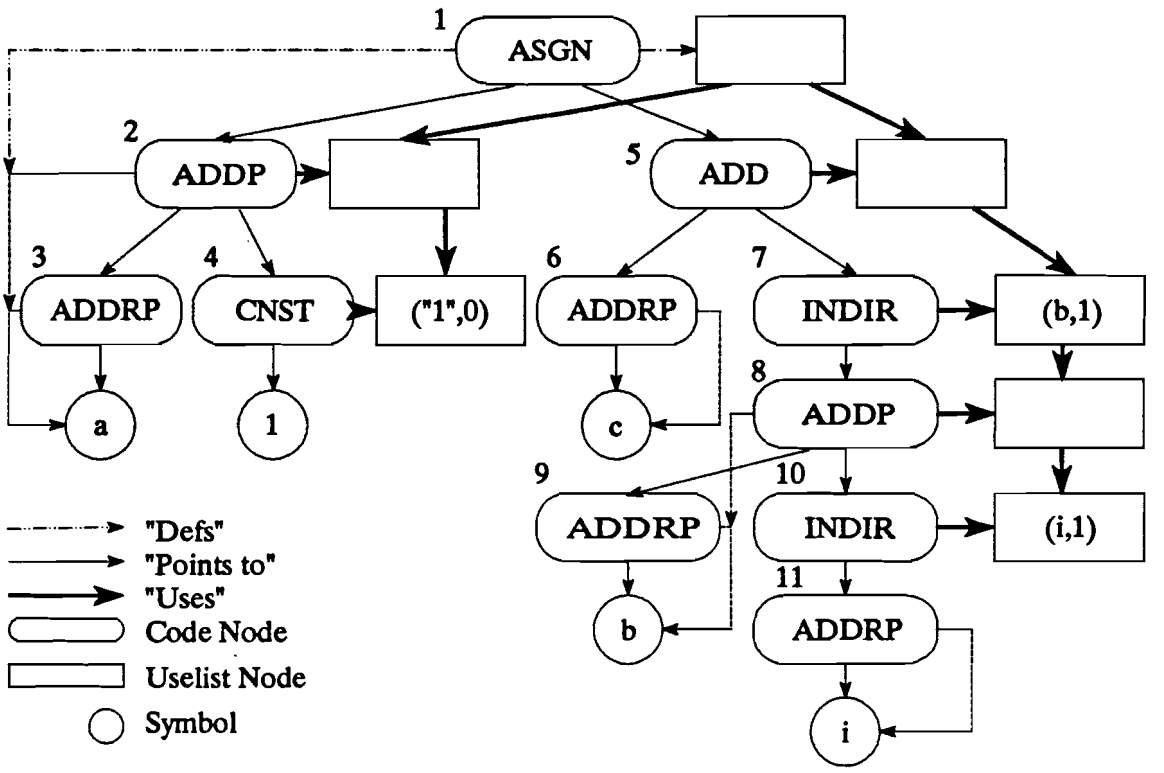

Figure 4 A code tree and its corresponding ref-def information.

To collect the ref-def information, the code trees are traversed in a depth-first order and the information of a node is synthesized from the information of the kids of the node. The leaf nodes ADDRGP, ADDRLP, ADDRFP and CNSTP initialize the 'points to' tuples. Only the CNST nodes initialize 'uses' sets since calculating the address of a symbol is not considered to be a usage of the symbol's value. The 'uses' sets are implemented as linked lists. Figure 4 shows a code tree annotated with its ref-def information. . It can be seen that the points-to information of the ADDP node (8) is copied from the underlying ADDRP node (9) so at the preceding INDIR node (7) it is known that an item in array b is accessed.

Uses-nodes have two pointers to other uses-nodes and one pointer to a 'points-to' tuple. A uses-node is added if a codenode can introduce a new 'points-to' tuple (INDIR, ADDP etc.) or if the codenode can be the root of two codetrees with each a corresponding uses-tree (ADD, ASGN etc.). The new uses-node then joins both uses-trees. Nodes 1, 5 and 6 have such a uses-node, even though nodes 5 and 6 have only one uses-tree.

The advantage of organizing the usagelist in this fashion is that parts of trees that are referenced more than once (common subexpressions) need not be evaluated again and cost no extra memory. If, for instance, a code forest contains two trees that both make use of the subexpression b[1], it is sufficient to copy the uses- and points-to pointers of the upper INDIR node (7) in figure 4. The necessary information for various data flow analysis problems can now be collected easily at the root of the tree: The symbol defined by the assignment is known, the symbols used by the assignment are known, and if the assignment had been to a pointer, the symbol pointed to had also been known (in that case, the right kid of the ASGN node would have it's points-to information initialized. Nodes 1 and 5 would than have the P-qualifier, and the points-to information of node 6 would reach node 1).

Finally some remarks must be made concerning pointers to unknown locations. It is possible that somewhere inside the code tree a CVIP (convert integer to pointer) node exists. From that node upward (in the direction of the root of the tree) nodes are pointing to or using a location somewhere in memory, not associated with any known symbol. These types of pointers, the UNKNOWN pointers, represented by the notation (?,n), impose particular restrictions on data flow analysis and subsequent optimization algorithms. Dereferencing an unknown pointer always introduces the danger of an undetected definition of a symbol. The following paragraphs will specify how these types of pointers are handled.

8.2 Determining usage- and definition points

After establishing what variables are defined and used by certain assignments, the locations of these definition- and usage points can be coupled to the symbol representing the variable. Doing this simplifies the implementation of the algorithms introduced in the following paragraphs. Note that determining the usage- and definition points of a variable is not the same as calculating definition-usage or usage-definition information! DU- and UD-chaining couple the list of definitions reaching a certain usage point or vice versa to that usage point or definition point, respectively. Usage-definition determination couples nodes defining or using a symbol to that symbol. It can be achieved by simply traversing the code-trees and storing the information on the fly.

Definitions and usage of the UNKNOWN symbol are not stored. Because the definition of an unknown location through the UNKNOWN symbol can never kill another definition of such a location (how will it ever be known if both locations are identical?), it is not necessary to store this information.

8.3 Data Flow Graph Construction

The Lcc front end passes code nodes or DAG (Directed Acyclic Graph) nodes to the back end. These nodes are passed in forests and contain trees affecting data as well as program flow. To perform data flow analysis, these forest have to be combined into basic blocks. Basic blocks are sequences of statements which may be entered only at the beginning and when entered are executed in sequence without stopping or branching except at the end of the block. The program flow (DAG)nodes must be used to derive the interconnection of the basic blocks to make up the Data Flow Graph (DFG). The DFG has a directed edge from node A to node B if there is a conditional or unconditional jump from the last statement of A to the first statement of B or if A and B follow each other directly in the program and A doesn't end with an unconditional jump. One node, the initial node, is the block whose first statement is also the first statement of the program.

Every algorithm handling global optimization or data flow analysis assumes the presence of the DFG. This means the data structures used to represent the DFG will be travelled and referenced very often and it is vital to the speed and memory usage of the compiler that this structure is given careful thought. Deciding on the representation of the DFG shall therefore be postponed until the various uses of the DFG have been established.

Constructing the DFG comes down to recognizing the basic blocks in the source code and linking these blocks according to the program flow information. To recognize the basic blocks, the following set of rules can be used:

- A basic block begins:
  - At the start of every procedure.
  - At the target of any branch.
  - Immediately after any branch
- A basic block ends:
  - Before the start of the next basic block, or
  - At the end of the procedure.

Starts of basic blocks can be identified by searching for labels; every label is the target of a jump or a branch Ends of basic blocks are signalled by (un)conditional branches and jumps. Detecting the end of a basic block implicitly signals the start of a new block. Multiple labels immediately following each other can be bundled

to signify the start of one basic block and multiple jumps immediately following each other can be bundled to signify the end of a block. This is especially the case with branch tables resulting from C's `switch` statement. Note that these rules do not mention the CALL instruction. Since this instruction jumps to unknown locations but always returns and continues execution with the next instruction, CALL instructions will not divide basic blocks. The final optimizing algorithms must recognize these instructions and decide what to do with global variables or local variables that have their address taken. These variables may be changed inside the called procedure!

Virtually all information necessary to create the DFG can be collected from the Lcc front end directly, except for the branch tables. These are translated into code sequences to calculate an address inside a jump table, or into separate test- and branch sequences, depending on the density of the branch table (see [1] for details). The branch table is then generated at the end of the procedure, so after the code was emitted. This means that, while traversing the basic block, the targets of the branches are not known and the DFG cannot be completed. To fix this problem, the front end was adapted to call a 'defbranch' procedure every time a branch table is used in the program.

The algorithm used to construct the DFG can now be given (algorithm 1). It basically bundles sequences of codetrees into basic blocks, while at the same time interconnecting those blocks using the program flow information. For branches to blocks that do not yet exist a backpatching strategy is used. The 'active label' administration combines multiple labels directly following each other into one access point, i.e. one basic block can start with code to define more than one label. The following example of C-code might produce the data flow graph of figure 5.

```
a := 1;
for (i:=1; i<10; i++)
  a++;
if(a=10)
  b:=1;
else
  b:=2;
```

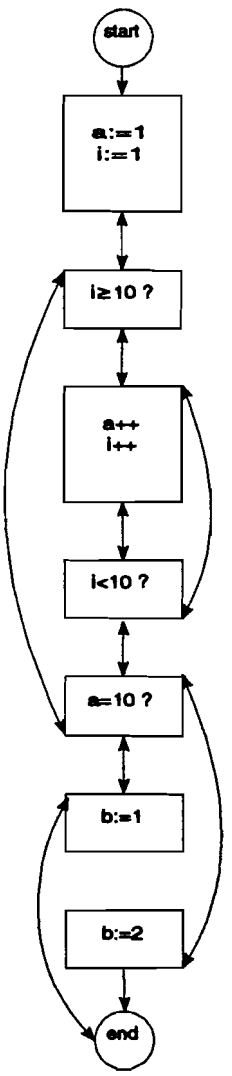

Figure 5 Example of a data flow graph

```
/* Codetrees are sorted in execution order. All labels and branch targets known. */

Dfg:={1 Empty Basic Block};

FOR every tree in the code forest DO
{
  IF tree is a label:
  {
    IF   A) there is currently no valid Basic Block OR
         B) the current BB is not empty
    {
      create new Basic Block
      IF there is currently a valid Basic Block
          create an edge between the new and the current Basic Block
      make new block the current Basic Block
    }
    add label to the list of active labels
    IF label has been referenced by previous code-nodes
    {
      backpatch those references by creating an edge between the referencing and the
      current BB
    }
  } ELSE IF tree is an unconditional jump:
  { /* If at this point there is no valid BB, we have detected dead code*/
    IF the jump target is a label
    {
      IF the label has been associated with a Basic Block
        create an edge between the target BB and the current BB
      ELSE
        Mark this BB as 'referencing the target label'
      IF current BB contains no code
        move any labels from 'active label list' to current BB
      make 'current BB designator' invalid
    }
    ELSE
    { /* If the jump target is not a label, then it is a result of a branch table
      calculation */
      move any labels from 'active label list' to current BB
    } ELSE IF the code tree is a conditional jump
    { /* Again, no valid BB means dead code */
      act as if the code tree is a conditional jump to a label but leave current BB valid.
    } ELSE IF the code tree is a return statement
    {
      IF current BB contains no code
        move any labels from 'active label list' to current BB
      make 'current BB designator' invalid
    } ELSE
    {
      IF there are edges leaving this BB
        create new BB and make current
      assign code to current BB
    }
  }/* The next part of the algorithm needs the 'defbranch' function */
  IF the current codetree is followed by a branch table
  {
    FOR every label in the branch table DO
    {
      IF the label has been associated with a Basic Block
        create an edge between the target BB and the current BB
      ELSE
        Mark this BB as 'referencing the target label'
      disable 'current BB designator'
    }
  }
}
```

Algorithm 1 Construction of the Data Flow Graph

8.4 Alias analysis

If the ref-def information has been set up and the DFG has been constructed, alias-analysis can be performed. The aim of this analysis is to establish what every pointer in the program can point to during the execution of the program. To see why alias analysis is of vital importance to data flow analysis, consider the following situation. With the code sequence:

```
int a, *p;

a=1;
p=&a;
*p=2;
```

Straightforward data flow analysis would recognize the variables a, p and *p, without noting that *p and a are aliased. This could, in a later stage, lead to the substitution of '1' for a, because a wasn't (visibly) redefined. It is therefore necessary to discover what pointers can point to if we want to be able to perform optimization without changing the functionality of the program.

Using the tuple notation of the previous paragraphs, the problem can be defined as follows:
If a tuple (a,n) is used or defined, find the set Q≡{(q,1)|(q,1) is aliased with (a,n)} to determine what physical values (not pointers!), represented by the tuples in Q, are used or defined. To be able to find this set Q, introduce sets IN and OUT for every node in the DFG. IN and OUT consist of tuples (p,q), p and q symbols, denoting the fact that (p,1) points to (q,1). Call these tuples aliases (strictly speaking, p and q are not aliases of each other as p only points to q. But to prevent confusion with the 'points-to' information during ref-def collection, the term aliases is used. In the following paragraphs these tuples will be denoting aliases, however). Note that only tuples of level 1 are used, because these represent the actual value of the pointer. Now, let IN be the set of aliases that exist at some point in the program where the set Q is needed. It is clear that if the IN-set is known, the set Q can be calculated for a tuple (a,n) by starting at {(a,1)} and n-1 times substituting any tuple (a,1) by another tuple (b,1) for every alias (a,b) in IN. Algorithm 2 does just that.

```
/* IN: set of tuples (a,b) with (a,1) pointing to (b,1) */
/* p : symbol to find set of aliases for */
/* n : number of times p was dereferenced */

/* Returns the set of aliases Q of (p,n) */

Q:={(p,1)};
T:=∅;

FOR i=1 to n-1 DO
{
  FOR every (a,1)∈Q DO
  {
    FOR every (a,b)∈IN DO
    {
      T:=T ⋃ {(b,1)}
    }
  }
  Q:=T;
  T:=∅
}
```

Algorithm 2 Calculating the set of aliases

Note that algorithm 2 does not discriminate between normal symbols and the UNKNOWN symbol. This implicitly states that the UNKNOWN pointer will be handled as any other symbol, or specifically, any other symbol of aggregate type. To see why this is allowed, consider the following: All variables reside somewhere in memory. The location of most variables in memory can be traced, either because they're only accessed directly or through dereferencing traceable pointers. The variables accessed by dereferencing unknown pointers reside, by the assumption of the previous paragraph, somewhere else in memory. The memory can therefore be partitioned in an allocated part (where the traceable variables reside) an in an unallocated part (the part of memory not allocated by the compiler). This unallocated part can be seen as a giant array to which all unknown pointers point. This notion, however, will not become important until the point where we explicitly want to determine what symbols are aliased with a dereferenced pointer. Until that time it suffices

to assume that for every pointer assignment to a symbol of aggregate type a tuple is added to the IN-set (without deleting other information concerning this pointer present) to signify that the aggregate symbol points to the assigned symbol. Dereferencing a pointer to a symbol of aggregate type can therefore result in a large number of aliased symbols (e.g. if an array of pointers to chars is completely initialized with pointers, then dereferencing an element of this array results in a set Q containing all characters pointed to by every element in the array). It is easy to see that this construction also works for 'the array spanning all other memory'.

We still need to calculate the IN sets. This is, like usage-definition chaining, a forward data flow problem based on convergence. [18], and a number of other books and publications that all reference [18], provide a method to solve this kind problem without assuming reducibility of the flow graph. The algorithm is based on the idea that information gets defined at some point and then propagates through the flow graph until it gets killed again. This means that a basic block in the flow graph:

- Can generate information,
- Can kill information,
- Can leave information unchanged.

Every basic block in the DFG can therefore be associated with an IN-set (the set of information reaching the basic block), an OUT-set (the set of information leaving the basic block) and a TRANS function, used to calculate the effect of an instruction I on an information set such as IN or OUT. The notation S1=TRANS(S2, i) is used to signify that S1 is the information set acquired by applying instruction i to set S2. For a basic block with a sequence of instructions $I=\{i_1..i_n\}$, S1=TRANS(S2, I) means the sequential application of TRANS to itself: $TRANS(S2,I)\equiv TRANS(TRANS(...(TRANS(S2,i_1),...),i_{n-1}),i_n)$. Subsequently a set of data flow equations for a sequence of instructions BB in a basic block is defined:

$OUT[n]=TRANS(IN[n],BB)$
$IN[n] = \{OUT[p] \mid p \text{ a predecessor of } n\}$

With these equations, algorithm 3 propagates the information through the DFG. This leaves the construction of the TRANS function. [18] lists a number of rules to handle pointer information in alias analysis, but assumes that pointers cannot point to pointers. This is clearly not true in C, where pointers can point to almost everything, including memory locations that cannot be traced to addresses of variables. If such pointers are dereferenced, it is impossible to know what variables are changed. Due to this effect, the existence of these so-called 'unknown' pointers introduces serious restrictions for optimization.
Closer investigation of unknown pointers reveals that they can only occur if a non-pointer type is explicitly cast to a pointer, as in `p=(char *)i`, with `i` an integer. Unknown-pointers are therefore always the result of a deliberate decision of the programmer to manually assign a variable to a memory location. Because of this, the compiler will assume the programmer isn't creating aliases for any regular variables using such casts. Thus, an instruction sequence

```
int i,j,*p;

i=(int)&j;
p=(int *)i;
```

after which `p` will normally point to `j`, introduces the possibility that the optimizing algorithms generate incorrect code!

```
/* Assume depth-first ordering for the DFG. */
/* N is the number of nodes in the DFG. */
/* n_i is the DFG node with depth-first number i.

/* Initialize: */
FOR every node of the DFG in depth-first order DO
{
  IN[i]:=∅;
  OUT[i]:=TRANS(∅, n_i);
}
CHANGE:=True;
WHILE CHANGE DO
{
  CHANGE:=False;
  FOR i := 1 to N DO
  {
    NEWIN:=∅;
    FOR all predecessors p of n_i DO
      NEWIN:= NEWIN ⋃ OUT[p];
    IF IN[n_i]≠ NEWIN THEN
    {
      IN[n_i]:=NEWIN;
      OUT[n_i]:=TRANS(IN[n_i],n_i);
      CHANGE:=True;
    }
  }
}
```

Algorithm 3 Solving the general forward dataflow problems

To implement the TRANS function, the following rules apply:
For an assignment (p,n):=(q,m), assume the sets P and Q to be the sets of symbols (x,1) that might be aliased by (p,n) and (q,m+1) (Note that the set of aliases Q of (q,m+1) denotes the set of symbols that (q,m) might point to), respectively (as calculated by algorithm 2). Then:

- Every symbol (x,1) in P can point to every symbol (y,1) in Q: add tuples (x,y) to IN.
- If the alias-information will be used for data flow analysis based on confluence: If P={(p,1)} then remove every tuple (p,a) from IN if (a,1)∉Q.
- If the information will be used for data flow analysis based on divergence: remove every (p,a) in IN with (p,1)∈P and (a,1)∉Q.

The distinction between confluence- and divergence is necessary to ensure that the corresponding data flow analysis is performed correctly. For data flow analysis based on confluence, the most important aspect is to find those assignments that define exactly one symbol. If the possibility exists that another symbol might be defined as well, certain optimizations may not be performed. In this case, if a variable is dereferenced it is vital that every possible alias is found. Had we been interested in data flow analysis based on divergence, only those cases in which it is absolutely sure what variable is accessed are of interest. The effect on the TRANS function is to add every alias that might arise and delete only those that are certain to get killed by the current assignment.

Algorithm 4 shows the case for alias analysis based on confluence. Now, using algorithms 2 and 4, algorithm 3 can be used for alias analysis suited for data flow analysis based on confluence.

```
/* Let (P,n) := (B,m), n>0. m>=0, be the assignment under consideration.
Let IN be the set  of pointer tuples { X->Y | X=(X,1) points to Y=(Y,1)},
valid at this point. Assume the presence of algorithm 2 in the form of a
function Aliases taking as parameters a set of pointer tuples and a
pointer, and returning a set of aliases of the pointer in the form of
(X,1). */

Q := Aliases(In, (P,n));
A := Aliases(In, (B, m+1));

For every (C,1) in Q
  For every (D,1) in A
    In := In ⋃ {C->D};

If |Q| == 1 Then
  If (C,1)∈Q not an array
    For every (D,1) in A
      In := In - {C->D};
```

Algorithm 4 The TRANS function

8.5 Reaching definitions

After the algorithms of the preceding paragraphs have been applied, all necessary information to calculate what definitions reach a basic block is present. The reaching definitions information is used to perform various optimizations listed in the previous chapter. The problem of reaching definitions is, like alias analysis, a forward data flow problem based on confluence. To see this, note that we want to establish the set of definitions that *can* reach a certain basic block, not the set of definitions that *do* reach a block, hence confluence. Since we start with a definition and try to propagate it as far as it goes before it gets killed, it is a forward flow problem.

A slightly altered version of algorithm 3 can be used to calculate the reaching definitions information. Specifically, since basic blocks are considered instead of single code trees, the TRANS function of algorithm 3 can be substituted by sets GEN and KILL, containing the set of definitions generated or killed inside the basic block, respectively. Again, [18, page 433] supplies the algorithm. The corresponding data flow equations become:

OUT[BB] := (IN[BB] - KILL[BB]) ⋃ GEN[BB]
IN[BB] := ⋃ { OUT[P] | P a predecessor of BB }

The problem is the calculation of the GEN- and KILL sets. [18] provides an algorithm to do this barring the presence of pointers. In the presence of pointers, establishing the GEN and KILL sets is an altogether different problem. In particular, determining what variables are defined by an assignment and hence what other definitions are killed by that assignment depends on the presence of pointers and the information on what they point to.

The following list shows what pointers can point to in C:
- single variables (as in p=&a, a not an array),
- arrays (e.g. the a in a[i]),
- other pointers of equal type (from normal pointer assignments),
- other pointers of different type (as in p=(int *)a, a char pointer and p int pointer),
- unknown memory locations.

Pointers to simple symbols provide possible sources for definition elimination. Casts of pointers to other types of pointers introduce no extra complexity as it is the original symbol (the a in the 4th item of the

previous list) that is stored as target; had this been a symbol of aggregated type, it would still be known. Pointers to aggregated symbols require special care, as has been explained in paragraph 8.4. It is not known whether a pointer into an array points to element x or to element y, so even if only one definition of this pointer is live it cannot be established if the new definition access the same element as the one live definition. Dereferencing pointers to aggregate types can therefore never result in the killing of another definition. It is possible, though, to kill definitions if, through multiple dereference of a pointer, a simple symbol is reached, even if an array is passed while dereferencing. This statement is based on the notion that, if a programmer dereferences an array element, this element had to be defined prior to the dereference. If an element has been defined, its definition is live at any subsequent point in the program, so also at the point under consideration. (Note that if no definitions are live, the program is dereferencing an uninitialized element and can end up altering any location in memory. This is considered to be a programming error.). Or, in other words, if only one element in the array of pointers has been initialized, and the program dereferences an (unknown) element in the array, it is assumed that the initialized element was dereferenced.

The rules to determine the GEN- and KILL sets while walking the assignments in the basic block now become:

- The current definition is added to the GEN-set.
- If a symbol (p,n), n>1, is defined, establish what the symbol points to (find a set Q containing only symbols of type (q,1) that can be reached by n-1 times dereferencing (p,1)).
- Every symbol in Q, except (?,1) is marked to be defined by the current assignment.
- Determine if this definition kills any other definitions. A definition kills another definition if:
  - The current definition is not an assignment to the (?,1) symbol, and
  - both assignments define a simple symbol directly (not through a pointer dereference), where a 'simple symbol' is a symbol with type other then aggregated (arrays, structs etc.), or
  - the definition to be killed originates from a direct assignment to a simple symbol, and the set Q contains only one simple symbol (s,1)≠(?,1).
- Killed definitions are added to the KILL-set.

Algorithm 5 shows these rules in programmable form.

```
/* Calculate the effect of assignment D:(P,n):=(A,m), n>1, m>=0 on the
set of definitions generated (GEN) and killed (KILL) by this basic block
BB.
Assume In and Out to be this block's sets to signify the incoming and
outgoing sets of aliases. Out should be initialized by algorithm 2. En
passant, add this definition point to symbols defined through pointers.
*/

/* Q: set of tuples of the form (s,1) */

GEN := GEN ⋃ {D};
Q := Aliases(Out, (P,n));

For every (q,1)∈Q do
  If (q,1)≠(?,1)
    Mark D as definition point for q;

If |Q|=1
  If  ((q,1)∈Q)≠(?,1) ^ (q is not an aggregate type)
    For every definition point DP of q do
      If DP directly defines (q,1)
        KILL := KILL ⋃ {DP};
  If n=1
    KILL := KILL - {D};

GEN := GEN - KILL;
```

Algorithm 5 Calculating the GEN- and KILL sets

Algorithm 5 makes use of the alias information calculated by algorithm 3, so it is important to execute the reaching-definitions algorithm after the alias analysis algorithm. However, the initialization phase of both routines can be merged. During the initialization phase of the alias-analysis algorithm, a call to algorithm 5 can be inserted after the call to TRANS. This eliminates one initialization loop, but also provides a better initialization! If the initialization for reaching definitions is done after alias-analysis is completed, both GEN and KILL sets will be calculated according to the alias information of a complete basic block. Consequently, instructions inside a basic block might be prevented from killing other definitions because some pointer, defined by an instruction occurring later in the same basic block, introduces a blocking alias (the condition $|Q|=1$ in algorithm 5 is not satisfied). When the initialization is performed immediately after the alias information of the current instruction has been calculated, the same definition can kill other definitions because the information from the offending pointer definition hasn't been calculated yet. An example of this situation is given with the following code:

```
int a, b, c, *p;

for(c=0; c<10; c++)
{
  a:=1;
  p:=&a;
  *p=2;
  if(c<5)
    p:=&b;
}
```

Intuitively, we know that the definition `a:=1` never reaches the point directly following the loop body and should therefore not occur in the GEN set of the BB consisting of the first three instructions inside the loop body. What happens if we first calculate the alias-information, is that we know from alias analysis that at the start of the loop body (which is the start of the BB ending at the `if`-statement), p can point to either a or b. So the last instruction of this basic block, `*p:=2`, does not kill `a:=1` because `*p` aliases both a and b. Therefore the GEN-set will include the (redundant) definition. If, however, the GEN-set is calculated directly after the alias information of an instruction has been calculated, it isn't know yet that because of the `if`-statement p might point to b (remember that we now have access to the alias information inside the BB while otherwise we only have the collective information of the complete BB), and the definition `a:=1` is killed immediately and never occurs in the GEN-set, only in the KILL-set.

8.6 Finding cycles in the DFG

To perform loop optimization, loops have to be found first. Or, to be precise, the loops in the flow graph that have exactly one entry- and one exit point must be detected. Using the depth-first ordering present, the problem is equivalent to finding the natural loops associated with the retreating edges in the graph. If an edge originates in a node S with depth first number s and ends in a node E with depth first number e, then the edge is advancing if $e > s$ and retreating if $s \geq e$. Retreating edges that do not create a loop are called cross edges. Note that this definition differs from that of [18, p. 452]!

A possible way to detect the loops in the DFG is to traverse the graph depth-first and, on detection of a back edge, backtrack until the target node of the back-edge is encountered, marking all nodes passed on the way as belonging to the same loop. To detect back edges, note the following: assume a depth-first traversal of the graph has propagated to a node n. If a successor s of n has already been visited by the depth-first search then the edge $n \rightarrow s$ is a back edge if and only if a path from the initial node to n passes through s. During a depth first search, a node can be assigned one of three states: unvisited, visited and completed. A node marked 'visited' is the root of a subtree currently under investigation by the depth first search and containing nodes marked 'unvisited'; a node marked 'completed' has already been left by the depth first algorithm and is the root of a subtree with all it's nodes marked 'completed'. Thus, a back edge is an edge encountered during depth-first search pointing to a node marked 'visited'.

To collect the nodes belonging to a loop, the following strategy can be used:
Let L_n be a set of nodes to form the loop numbered n. Assign states 'open' and 'closed' to L; If a loop is open, it is currently being constructed. The 'closed' state signifies the nodes composing the loop have been

identified and marked accordingly; the set L_n can not increase in size. During depth-first, at node *n*, for every successor *s*:
If a back edge *n*->*s* is encountered, with *s* status 'visited': open a new loop L_n. Mark *s* as 'starting L_n.
If a back edge *n*->*s* is encountered, with *s* status 'completed' and *s* belonging to but not starting a closed loop L_x (btw: a node belonging to a closed loop always has status 'completed' since a loop is only closed at the point that the starting node gets 'completed'.): L_x is a loop with more than one entry point.
If *s* has status unvisited, recurse to *s*. On return of the recursion, mark *n* as belonging to L_x iff *s* belongs to L_x and *s* is not marked as 'starting L_x'.
If all successors have been processed, check if *n* belongs to a loop that one of its successors doesn't belong to. For every such loop, mark *n* as the loop exit. If the loop already has a (different) exit point, the loop has multiple exits and is not a candidate for optimization.

The above strategy can be combined with the depth-first numbering algorithm. The 'unvisited' status can be checked using the visited bit already used by that algorithm; the distinction between 'visited' and 'completed' can be made by checking if the node already has a depth-first number assigned to it. (visited->no dfn, completed->dfn assigned).

Algorithm 6 shows the solution in a more structured form.

```
/* DFN(n) is the depth-first number of node n.DFN(n)==0 means n has not
yet  been assigned a depth-first number.
  N is the number of nodes in the DFG.
  L is a set of nodes composing a loop; functions Open and Close resp.
open   and close the loop as stated previously. */

Procedure depth_first(n)
  int i=0;

  mark n visited;
  FOR every successor s of n DO
    IF s visited and DFN(s)==0
      Open L_i;
      Mark s as starting L_i;
      i := i+1;
    ELSE
      IF s marked 'completed' AND s belongs to a loop L_x AND L_x closed
        L_x is a multi-entry loop;
      ELSE
        depth_first(s);

  IF n belongs to L_n AND any successor s of n not in L_n
    IF L_n already has an exit
      L_n is a multi-exit loop;
    ELSE
      mark n as exit of L_n

Return
```

Algorithm 6 Detecting loops in the DFG.

Note that algorithm 6 detects the so called 'natural loop' of a back edge. Natural loops consist of a back edge *n*->*s* and the set of nodes (and the edges connecting them), reachable through *s*, that can reach *n* without passing through *s* again.For example, consider figure 6a. From the set of nodes {1,2,3} and {2,3,4}, only {2,3,4} will be marked as comprising a loop since {1,2,3} has two entrypoints, nodes 1 and 2 respectively. However, the loop consisting of nodes {1,2,3,4}, with node 1 acting as loop entry and exit, will never be recognized as such since it is not a natural loop of one of the back edges 3->1 and 4->2. Likewise nodes 1,2 and 3 of the graph shown in figure 6b and nodes 1 to 4 of figure 6c will not be marked as comprising a loop. Extending loop detection to include the recognition of these constructs requires two new actions. Detecting the loops in graphs a and b of figure 6 requires that a slightly altered version of algorithm 6 is applied repeatedly to the graph. This new algorithm is not capable of 'opening' new loops but detects the

fact that the head of an edge points to a node belonging to a loop not (yet) incorporating the node at the tail of the edge. Subsequently the tail-node will be added to the loop, unless the head node was the start of the loop. This algorithm must be applied to the graph, expanding the loops, until no more expansion can take place. It can easily be checked that this strategy will lead to the detection of the (hidden) loops in graphs a and b of figure 6

Graph c of figure 6 represents a situation that introduces another type of problem. When multiple back edges point to the same node, it is possible to construct more loops then there are back edges. In fact, with n back edges pointing to one node it is theoretically possible to construct such a graph that in total $\sum n$, from 1 to n,loops can be formed by making every possible combination of loops. Depending on the loop entry- and exit nodes most of these combinations will not conform to the one-entry-one-exit criterium that loops of interest must satisfy, but since exits and entries can occur at random positions it is still neccesary to be able to detect all possible loops (try which loops should be detected if the exit point of graph c is positioned at different nodes). Therefore algorithm 6 should be extended with the following functionality: If a back edge is encountered pointing to (a) loop entry point(s) (i.e. a node that was the target of a previously visited back edge), make a copy of every loop the node belongs to and mark the tail of the back edge as member of these copies. With copying a loop is meant that every node marked as belonging to the loop will also be marked as belonging to the new loop. Marking the tail of the back edges ensures that nodes encountered while backtracking from the recursion will also be marked as members of the new loops. Applying this extention to graph c results in the following actions:
The algorithm recurses, depth first, from node 1 through nodes 2 and 3 and encounters the back edge 3->1. Nodes 1 and 3 are marked as belonging to loop 1. The algorithm then backtracks to node 2, marking that node also as member of loop 1. It subsequently recurses to node 4 and encounters the back edge 4->1, pointing to the entry node of loop 1. Therefore loop 1 is copied (nodes 1,2 and 3 are marked as belonging to loop 2) and loop 3 is opened. Node 1 is added to loop 3, node 4 is marked as beloning to loops 2 and 3. The algorithm then backtracks back to the node 1, en passant marking node 2 as belonging to loops 2 and 3 as well. Thus, on completion of the algorithm three loops have been marked: loop 1 comprising of nodes 1,2 and 3, loop 2 spanning the complete graph and loop 3 made up by nodes 1,2 and 4. Depending on the exit point(s) of the graph some loops may get eliminated: with node 1 as exit loops 1 and 3 are eliminated, exiting at node 2 keeps all loops, and exiting at nodes 3 or 4 respectively eliminates loops 1 or 3.

Note that applying both extensions at the same time may result in the detection of a loop more than once, possibly as a result of the order in which the graph is traversed; graph b provides an example of this situation. (the order in which the exits of node 2 are chosen for recursion determines wether an extra third loop is created). This is due to the fact that as a result of the multiple iterations of the first extension the same back edge can be used to close more than one loop, causing the second extension to make a redundant copy of one of the loops.

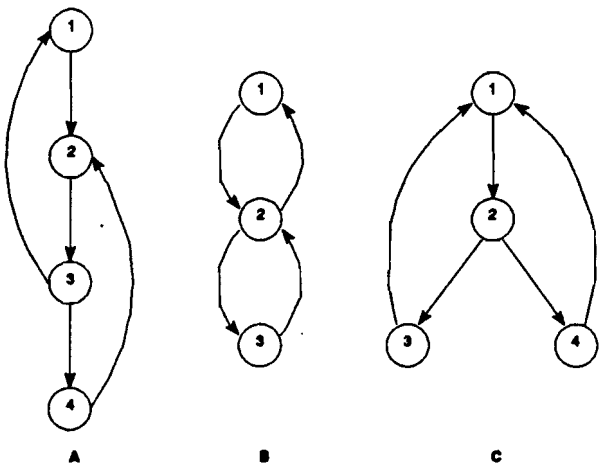

Figure 6 Example of complex loops

8.7 Making use of the calculated data flow information

After having calculated the reaching-definitions information, what can be done with it? [18] suggests the calculation of usage-definition-chains. These can then be used to perform a number of optimizations. UD-chains are chains of definitions of variables that reach a certain usage of these variables. If these chains are to be stored for every used variable, a large amount of memory may be necessary. Fortunately it is not necessary to calculate these UD-chains due to the way the reference- and definition information is stored. If we want to find out what definitions reach a certain usage point for a specific variable, it is sufficient to match the definition points of that variable, accessible via the symbol denoting the variable, with the reaching definition information of the previous paragraph. Note that if we want to match the usage points of some variable to a specific definition point (DU-chaining), both the reaching definitions and the alias-analysis algorithms have to be rewritten for use with backward-flow analysis, as DU-chaining is a backward flow problem based on confluence. However, backward flow versions of these algorithms are largely identical to the (current) forward flow versions, since only the data flow equations change slightly.

It is evident that with the information on definition points, usage points and reaching definitions, constant folding can be performed. Simply check if, at some usage point, one and only one definition of the variable reaches and defines the variable with a constant, and a substitution can be made. The substitution eliminates a usage point of the substituted constant, which in turn may lead to code elimination if the constant is no longer used.

Another, more sophisticated optimisation is detection of loop-invariant computations. Besides the information already present, loops in the DFG must be detected (specifically: loops with one entry- and one exit point). Finding loops in the data flow graph can be simplified by using the depth first ordering present for the DFG but still requires an extra iterating routine. But, given a loop, we can find loop invariant computations by marking those computations whose used variables are either constant or have their definitions outside the loop, or are defined by exactly one other invariant computation inside the loop (note that this statement again introduces the necessity to iterate over a loop, as marking one computation as invariant introduces possible new invariant computations). Invariant computations of course give rise to code motion.

Other optimization, such as copy propagation, calculation of available subexpressions and the resulting strength reduction and global common subexpression elimination, or induction variable elimination require other types of data flow analysis. Copy propagation and available expression calculation need forward flow analysis based on divergence. Finding live variables, used to detect and eliminate induction variables, is, like du-chaining, backward flow analysis based on confluence.

8.8 Forward data flow analysis based on divergence

The previous paragraph concluded with the notion that different types of data flow analysis are necessary to perform other optimization. Specifically, to enable copy propagation and available expression calculation (in turn enabling global common subexpression elimination), forward data flow analysis based on divergence is needed. Confluence-based analysis isn't suitable because the optimization mentioned above can *only* be applied if it is absolutely certain that a certain variable or expression is equal to exactly one other variable or expression, reaching the execution point via *every possible* path through the flow graph. Remember that confluence based analysis relied on the notion that certain optimization could *not* be applied if some information reached an execution point via *any* path through the DFG.

The data flow equations reflect this difference between 'any' path and 'all' paths by changing the operator used to calculate the new Insets from $\bigcup$ to $\bigcap$. So they become:

OUT[BB] := TRANS(BB, IN)
IN[BB] := $\bigcap$(IN[P] | P a predecessor of BB).

Algorithm 3 can easily be changed into a 'divergence based' algorithm by changing these equations. Algorithm 2 and algorithm 5 need a little more consideration. Both algorithms were designed for use by a confluence based algorithm and we need to assess what the effect of pointers and aliases amounts to. The following list summarizes the relevant aspects:

- An assignment to a dereferenced pointer defines every variable pointed to to guarantee the 'one and only one' relationship.
- An assignment to a dereferenced pointer kills all existing definitions of all variables pointed to (even variables of aggregate type) so the condition that 'it must be absolutely certain some information reaches an execution point through all paths' is satisfied.
- An assignment to a pointer creates a points-to relation for every known alias of the pointer at that execution point.
- An assignment to a pointer only kills a points-to relation if the pointer is not aliased, or is not of aggregate type.

So the actual alias calculation (algorithm 2) can remain the same, but the calculation of the GEN and KILL sets has to be changed to accommodate the first two points. Finally, the TRANS function must be changed to conform to the last two points.

8.9 Backward data flow analysis

Finally, backward data flow analysis can be applied to enable live variable information and the resulting induction variable elimination. This information could also be used to identify those definitions that will never be used, or establish if the storage space of a variable can be used by another variable.

Backward flow analysis requires the following data flow equations:

IN[BB] := TRANS(BB, OUT[BB])
OUT[BB] := $\bigcup$(IN[P} | P a successor of BB)

for confluence based analysis. To find the data flow equations for divergence based analysis, simply substitute the union operator with the intersection operator, as with forward flow analysis.

Effects of pointer assignments for confluence- and divergence based analysis are identical with the cases for forward flow analysis, with the notion that for every occurrence of 'definition', the word 'usage' should be substituted.

9 Register allocation

Register allocation is an important part of compiler design, especially so in a risc processor. The aim of register allocation is to keep as many computation results as possible in registers to minimize access to memory. The dumb compiler allocated registers on the fly and only for single code trees. It is evident that this approach leads to unnecessary store and reload instructions. Allocating registers using knowledge of liveness and future use of certain values greatly improves execution time and code size of a program. An elegant method to achieve good register allocations using this information is register allocation using graph colouring, described in [21] and [22]. Graph colouring however has the disadvantage that it does not perform allocation of registers needed while calculating (complex) expressions. Extending graph colouring to include register allocation in expressions will take a large amount of memory to store subexpressions and will also enlarge the (already substantial) interference graph described in paragraph 9.2. A simpler method is simulated execution (used by the dumb compiler to allocate registers inside expressions), treating the available registers as a kind of stack and using DFA to decide which values can be kept in registers, what values can be killed and what registers should be spilled. This method does allocate registers inside expressions, but doesn't (provably) allocate registers optimally, in contrast to graph colouring. This type of register allocation is described in [18], [25] and [14].

9.1 Defining live variables for register allocation

Information on live variables will be used to determine whether the value contained by a variable can be stored in a machine register instead of memory by checking if two values are simultaneously live. Assigning a value to a register is of course only possible if the register is not assigned to another variable or if the register's value is destroyed as a side effect of an instruction somewhere in the execution path between the definition and the usage of the value. As the PMS500 instruction set does not incorporate an instruction that destroys the contents of a (general purpose) register as side effect, it suffices to state that a variable is live if it is defined on a path from the start of a program to the current execution point, and there exists at least one path to a usage point of this variable that doesn't redefine its value.

The above paragraph implicitly mentions the relation between a variable and the value assigned to that variable. The most convenient entity that can be used to gather information on the liveness of a value assigned to a variable is the assignment itself, or the definition as it is called. Definition- and usage points of variables have already been identified and, noting the fact that 'definitions' and 'values assigned to variables' are essentially the same, we can couple the liveness of variables to the liveness of definitions. With this notion, a definition is live at some execution point if it reaches some usage point (using the reaching definitions information described in chapter 8.1), and there exists some path through the DFG from the definition point through the execution point and to the usage point of the defined variable that doesn't contain another definition point of said variable.

[21] furthermore limits the notion of interference to 'two values interfere if either one of them is live at the definition point of the other'. The rationale for this limitation is the simpler interference graph and the possible reduction of the chromatic number of the interference graph. It enables two values, defined in separate branches of an if-statement, to be assigned to a single register, while otherwise they would fall under the first definition of interference at the point directly after both branches join again, inhibiting the assignment of both values to one register. It can easily be seen that, in the absence of a loop containing the if-statement, both values can in fact be assigned to the same register. Had the if-statement been part of a loop the possibility arises that the definition in one of the branches stays live until the other branch is passed so both values have to be assigned to different registers.

It has already been stated that live variable analysis is a backward flow problem based on confluence. However, that statement was based on the fact that live variables will, for example, be used to decide whether a variable has to be stored into a register at all. In that case it must be discovered which uses 'reach' what definitions to delete those definitions that have no reaching uses of the defined variables. Live variable analysis then simply propagates usage information upwards through the DFG, entering all branches

until it reaches another usage point, much in the same way as reaching definition information is propagated downward through the graph. The notion of live variables for register allocation, however, inhibits this simple approach; if there is no path from the starting node of the DFG, through the definition point, to the usage of the variable, the variable is not live. Thus, if we encounter a point where two branches rejoin while walking the DFG bottom up, it may well be possible that inside one of the branches the variable is not live by the above definition. So another type of 'live variable analysis' must be found.

Note that a usage point only makes those (used) definitions live that indeed reach this point (definitions that do not reach a certain usage point can of course never be used by that point). Also note that two definitions can only interfere if one of them reaches the other. By using the original strategy to propagate live variable information through the graph, combined with the previous notions, we can conclude that two definitions interfere if they are live (using the notion in this paragraph) at the same time and if the definition of one reaches the definition of the other. Otherwise, both definitions do not interfere and the defined variable can be assigned to the same register.

It will be clear by now that the algorithm used to find live variables is identical to the algorithm used to find the reaching definitions except that for live variables the DFG traversal is reversed. Therefore an algorithm to calculate live variable information is omitted.

9.2 Graph Colouring

The idea is to build an interference graph of the function. The interference graph is a graph with a node for every computation in the procedure and an edge between two nodes if, at any time in the function, both results are live at the same time. Such results are said to interfere with each other. A result is live if it is possibly going to be used by an instruction following the current execution point. To this graph is added the clique of machine registers. (a clique is a fully connected graph). By colouring the interference graph, a register allocation can be achieved, if the number of colours used does not exceed the number of machine registers. Colouring a graph is the assignment to every node in a graph of a colour such that no two adjacent nodes have the same colour, using as little colours as possible. The minimum number of colours needed to colour the graph is called the chromatic number of the graph. An n-colouring of a graph is a colouring using n different colours. Result can then be assigned to the register with the same colour. So, for the PMS500, an 8-colouring must be reached to provide register allocation. If an interference graph is not 8-colourable, spill code has to be added to reduce the chromatic number of the graph. Spill code must split some node of the offending clique in order to break the clique in two or more separate cliques with a lesser number of nodes. (It is easily seen that a clique containing n nodes has chromatic number n).

The graph colouring problem was proved to be NP-complete ([11] on NP-completeness, [10] on the proof of NP-completeness of the graph colouring problem). The heuristic used by [21] was shown to provide practical register allocation, i.e. the execution time for actual programs stays within reasonable bounds. It must be noted, however, that solving an NP-complete problem can take an exponential amount of time! For these cases, the programmer should always be able to activate another register allocation scheme, such as the on-the-fly allocation of the dumb compiler.

Another problem that arises with register allocation via graph colouring is the fact that, while calculating an expression, subexpressions have to be stored into registers. The graph colouring approach described in the previous paragraphs doesn't provide for these allocations, while the solution to the problem may not interfere with the graph colouring algorithm. The possibility exists, however, that the calculation of the expression takes more registers than are available at that execution point. At that point, spill code has to be introduced, in its turn affecting the effectiveness of the colouring algorithm. A solution to this problem is to identify every extra register necessary for calculating the expression and introducing new variables, and thus new definitions, for those registers so graph colouring can be enhanced to allocate every register, including temporaries. This approach can be effective if it is used in combination with global common subexpression elimination, which must calculate these subexpressions as well. Lcc already provides for local common subexpressions. Identification of these subexpressions and adding the definitions of the temporaries assigned to these expressions to the definition universe should be done before data flow analysis to include

subexpressions in reaching definitions- and live variable calculations.

For graphs that have a chromatic number higher than the number of available machine registers, spill code must be introduced. After inserting spill code, the interference graph has to be rebuilt, and the graph colouring algorithm must again be invoked, until a colouring is found using no more colours than machine registers available. The decision where to insert the spill code must be founded on information of future usage of the spilled variable and the amount of execution time the spilling adds to the program. Evidently, the decision to insert spill code can only be reached using an heuristic, adding another limit to the effectiveness of graph colouring. Especially with the small number of available registers with the PMS500 the chance that spill code must be introduced frequently is always present, making graph colouring less applicable for implementation.

9.3 Simulated execution

Register allocation by simulated execution walks the DFG in approximate execution order, assigning registers to (sub)expressions on the fly. Every time a register is needed, a demand is made upon the register pool to supply an unused register. If all registers are used, spill code has to be inserted.

This type of register allocation, used by the dumb compiler to assign registers in single DAG trees, can be extended to perform reasonable allocation if information gathered from live variable analysis and reaching definitions is used, saving compile-time computation time and memory usage because it is not necessary to build an interference graph. Specifically, register allocation for small loops can yield highly efficient register usage and even outer loops (loops containing other loops) can be taken into account, albeit with more programming effort. With the observation that control flow resides inside program loops for 90% of the time, putting a little extra effort into register allocation in loops can make simulated execution perform almost as well on register allocation as graph colouring, with much less programming effort yielding a smaller and faster compiler because of the effects described above.

The decision to insert spill code uses, as with graph colouring, an heuristic (in fact, the simulated execution algorithm is itself an heuristic) to decide which register to spill. Factors that can be taken into account are usage frequency of the value stored in the register (provided as an approximation by the front end), the distance before the value is used, and the fact that the register is or is not in use as an induction variable. Induction variables can be found using the results of the live variable analysis and loop detection algorithms previously mentioned. Values belonging to variables that are no longer live are never spilled, of course, while values belonging to variables of volatile type are never kept in registers but are always stored to memory.

The weak point of register allocation by simulated execution is it's inherent 'local-ness'. While graph colouring tries to allocate registers as efficiently as possible considering complete functions, the result of register allocation by simulated execution can be greatly influenced by the local order in which statements are executed. The decision to a value in register A at a certain execution point may lead to an extra register copy if further down in the code the same register has been used to store a different variable, for example if two blocks have an exit to a third block, the first blocks allocate different values to register A and both values are needed inside the last block. One of the values has then to be copied into another register. Had both values originally been assigned to different register, this copy could have been avoided (barring other conflicting allocations). It is therefore critical to use as much global data flow information as possible since that information tells what the future and the past of the value underhand may be, enabling the register allocator to predict a situation as described above and allocating a value to a 'better' register.

Comparing the problems and features of both types of register allocation, the graph colouring scheme appears to introduce more problems and will therefore take more time to implement than the simulated execution scheme. Weighing these problems against the expected 'quality' of the resulting allocations, it was concluded that register allocation by simulated execution provided the better solution. Specifically, the far greater complexity introduced by graph colouring will only result in a marginally better allocation due to the small number of available registers and hence the expected amount of spill code that must be introduced, severely crippling the graph colouring algorithm.

10 Implementation

10.1 Data structures

10.1.1 Reference-definition information

A number of data structures is used by the back end. First, the reference- and definition information. Chapter 8.1 sketched the outline of the data structure used to store the relation between the DAG provided by the front end and the ref-def information added by the back end. Every DAG node structure has an extension named Xnode, used by the back end to annotate the DAG. Consequently, the Xnode structure contains a `uses` pointer to access the `Uses` structures, used in the implemention to represent the square blocks shown in figure 4, and `defs` and `points` pointers to access members of the list of `Pointer` structures, used to represent the pointers used ina function. Clearly `Uses` structures contain three fields (looking at figure 4), being the pointers to the left- and the right kid in the structure, respectively, and a pointer to a member of the `Pointer` structure to propagate pointer information through the tree.

While collecting ref-def information, a list of `pointer` structures is built. To avoid confusion, the following notation is used: `pointers` are the (PMS500's) compiler representations for pointers used in the program to be compiled; 'pointers' are variables in the (host) compiler-code used to reference other data structures. Thus, `pointers` are data structures while pointers are parts of compiler source code. Every `pointer` structure represents a separate pointer in the same form it appears in the function, i.e. a pointer to the symbol in question and an integer to signify the derefencing level of the pointer. A `pointer` structure can be thought of as a tuple (s,l) as used in chapter 8.4. The `pointer`-list is used to distinguish between the various occurrences of one pointer with different dereference-levels. If a pointer is encountered the list is traversed to see if a `pointer` already exists with identical symbol name and dereference-level. If so, the list already contains a structure for this pointer and the structure in question is referenced. Otherwise, a new structure is added. This way every `pointer` occurs exactly once for every deference-level it is used with, and comparing if two `pointers` are equal can simply be done by comparing the addresses at which the structures are stored (i.e. comparing the values of the variables pointing to the `pointer` structures). The `pointer` list is a linked list terminated by a NULL pointer.

Runtime memory usage

Every `uses` structure takes up 12 bytes of memory (3 pointers each consisting of 4 bytes - the current host compiler's pointer size). One of these pointers points to a `pointer` structure which consumes 9 bytes of memory (two pointers and a `Boolean` type defined as `char`). Assuming the worst case situation that every dag node needs a new `uses` structure and every symbol declared by the front end is used by the back end, the total number of bytes used by the reference-definition information for programs yielding N codenodes and S symbols is $12N+9S$. Normally, however, a number of codenodes make up a local common subexpression and can be used more than once. Equally, normal programs reference certain variables a number of times, and every time such a reference occurs there is no new memory allocated; the same `pointer` structure is referenced.

10.1.2 The data flow graph

Since it is known what the various uses of the data flow graph are, a data representation can be chosen. While building the DFG the information about the interconnection of the basic blocks is incomplete and the total number of basic blocks is unknown. To store the basic blocks, a linked list data structure is used, linking the basic blocks in the order in which they are emitted by the front end. While performing data flow analysis, either the predecessors or the successors of the basic blocks must be inspected quite often, making the presence of a double link between basic blocks that can follow each other in the program flow plausible. Finally, while performing DFA the basic blocks are best traversed in order of their depth-first numbering. Therefore a separate array containing an entry for a pointer to every basic block is created, with pointers to basic blocks in order of the depth-first numbering, to eliminate the recursion necessary for a depth first

traversal of the DFG.

To store the list of exit- and entry points of a basic block, as well as the list of DAGS contained in the basic block, the `List`-datastructure provided by the front end is used. This data structure consists of two pointers: a pointer to the next element in the list, and a pointer to another structure containing the data. These `List` elements are managed by the front end and are re-used when a new function is processed, so unnecessary allocation and freeing of memory is prevented. The `List` datastructure is cyclic, i.e. while walking the list starting with a certain element, the same element is eventually reached again. For the DAG-list, however, the pointer to the list always points to the first DAG in the basic block and the execution order of the DAGs is preserved. Furthermore the DFG-nodes contain a depth-first number and a flag to mark nodes visited, finished, or according to other states necessary for DFG-analysis.

Finally, every DFG node possesses an `In` and an `Out` pointer. Both pointers point to the head of a list of `Alias` structures, and will be used to store the aliases that enter and leave the basic block represented by this DFG node, respectively. An `Alias` structure contains a pointer to another `Alias` structure to make a single linked list. It also contains two pointers to `Pointer` structures to represent the fact that `Pointer a` points to or is aliased with (depending on the usage of the list) `Pointer b`. To preserve memory and to speed up the alias analysis process, these lists could also be represented by bitfields. However, the bit field can only be allocated and initialized after the initial alias analysis has been performed since it is only then that the exact number of aliases in the program is known: aliases formed by assigning the value of a pointer to another pointer introduce new aliases while performing alias analysis.

Runtime memory usage

Each DFG-node consists of 36 bytes (3 integers and 6 pointers). Two of these pointers point to the entry- and exit point lists of the node. Depending on the interconnectivity of the DFG, these lists vary in size. Every extra entry- or exit point yields two pointers or 8 extra bytes; every exit point has a corresponding entry point in the target node, making the DFG doubly linked. Depending on the complexity of the source program these lists and the number of DFG nodes may grow, but there will always be less DFG nodes than DAG nodes; subsequently, there are always less than $2*2*(D!)$ interconnections for a DFG with D nodes. Of course, a program yielding a fully connected data flow graph either consist of only a very small number of nodes or is written to yield a pathetic flow graph.

10.1.3 Extensions to codenodes

Figure 7 shows the declaration of the Xnode structure, the back end's extension to codenodes.

```
typedef struct {
        char *expr;             /* For expression reconstruction
                                   convenient for debugging */
        int id, lev;            /* Node id and nesting level */
        int argoffset;          /* Max stackoffset for arguments */
        unsigned char rmask;/* Mask of regs set by this node */
        int visited;            /* Flag if node has been processed */
        Usagelist uses;         /* List of symbols this node uses */
        Pointer points;         /* Symbol this node (might)
                                   point to. For ASGN nodes, the uses and
                                   points are used for data flow analysis;
                                   these lists are inherited from the
                                   kids-nodes. For nodes without P qualifier,
                                   points is zero. */
        Pointer defs;           /* For ASGN-nodes: pointer to symbol that
                                   is defined by this node */
        int di;                 /* Position of this node in Def-array
                                   (in case of ASGN-node) */
        Node next;              /* next node on output list */
} Xnode;
```

Figure 7 The Xnode structure

Besides the structure entries mentioned in paragraph 10.1.1, the code DAG nodes are extended with a number of other fields. Not all of these fields are used in the present state of the compiler, but are inherited from their usage by the dumb compiler.

The `expr` pointer is a pointer to a string containing the reconstructed expression represented by the code tree of which the current DAG node is the root. This string can be printed during development and debugging stages to verify compiler functionality. Entries `argoffset`, `rmask` and `lev` are used by the dumb compiler to allocate registers to nodes and to calculate the memory address at which the value of the node (for those nodes that represent a value to be stored) resides. `Argoffset` is used to calculate the maximum offset from the stack base necessary to store all locals used by the current code tree, `rmask` is a mask to store the registers in use at the moment the node is to be emitted and `lev` is currently only used by printing routines to identify root nodes. The `id` entry is simply used to number the codenodes that define or use values, on the one hand to be able to identify them while debugging, on the other hand to provide a reference to the node's position in the bitfields described in the following paragraph.

Runtime memory usage

Codenode extensions consist of 41 bytes and contains only pointers to structures and lists described and allocated elsewhere, such as the front end's symbol table, the `uses` lists or the `pointer` lists. Extensions are automatically allocated by the front end and therefore consume an amount of memory linear to the number of codenodes produced by the front end.

10.1.4 Extensions to symbol table entries

```
typedef struct {
 char *name;        /* name for back end */
 int offset;        /* locals: frame offset */
 Boolean unknown;   /* Signals the possibility (unknown=True) that */
                    /* this symbol was assigned an unknown value */
                    /* for instance globals after a function call, */
                    /* addressed vars after a function call, or aliased */
                    /* symbols after a definition of an UNKOWN symbol. */
                    /* (Currently unused) */
                    /* Labels: */
 DFG  def;          /* Pointer to DFG node that defined label */
 List ref;          /* List of pointers to DFG nodes that reference
                       this label */
                    /* Variables + labels: */
 List defpoints;    /* List of DAG nodes that define this symbol */
} Xsymbol;
```

Figure 8 The Xsymbol structure

As with DAG nodes, the symbol table entries are extended with a number of entries combined it the Xnodes structure (figure 8). Name is a character pointer to find the name of the symbol used by the back end. Offset is the stack frame offset necessary to find the run-time value of the symbol on stack. Unknown is a flag added for future use by the optimizer to signal the fact that the symbol's value must be fetched from memory, even if it is still marked as residing in a register. This is necessary to be able to handle the effects of procedure calls and assignments to untraceable pointers.

To build the DFG, every symbol of type LABEL has its def pointer initialized to point to the DAG node that defined the symbol; this pointer can then be used to update entry- and exit lists of DFG nodes. Ref and defpoints are both pointers to List datastructures, each list member representing a usage- or a definition point for the symbol, respectively.

Runtime memory usage

The symbol table extension, designated by the Xsymbol structure, comprises of 18 bytes per symbol table entry. All pointers reference structures belonging to lists or sets already described, so no extra memory has to be allocated except for the 18 original bytes. Therefore, besides the memory allocated by the front end for every symbol, the back end uses 18 bytes per symbol.

10.1.5 Lists for aliases

The aliaslists, consisting of alias structures, form sets of aliases at entry- and exit points of the DFG nodes. An alias structure contains two pointers into the pointer list to identify resp. the pointer p and the aliased symbol a, which in turn may be a pointer; however every used symbol occurs in the pointer lists to enable the handling of other types of variables that are cast into pointers.

Runtime memory usage

For every alias present at DFG entry, 12 bytes are used to store this information; also for every alias present at DFG exit. An alias that doesn't get killed inside an inner basic block therefore occurs in at least four lists: one of the exit lists of one of the preceding basic blocks, the entry- and exit list of the current block and the entry lists of every descendant of the current block. The size of these alias-lists depends heavily on the source program, although pointers that obtain the possibility to point to any symbol in the symbol table do not actually create all these aliases. One (common) symbol is used for these pointers. Clearly a lot of memory is used to store the alias information as the number of aliases grows. A lot of memory can be saved if the alias information is stored using bitvector-like structures, where only a number of bits is needed to

store an alias-relationship. The current storage method is only suitable for small source program functions using an equally small number of pointers resulting in little aliases.

10.1.6 Bitfields for data flow analysis

To perform data flow analysis, a number of set-like operations have to be performed on the 'universe' of code-trees making up a definition. To reduce memory usage and computation time, the DAG nodes defining or using a value are assigned an id-number. Subsequently, an amount of memory is allocated to represent every data set used, with every set containing at least one bit for every node with an id-number assigned to it. The set of nodes having a valid id is the definition universe. Since memory can only be allocated in chunks of 32 bits (the integer size of the host compiler), superfluous bits are always allocated, since every set used is made up of these chunks of memory. Sets are always related to DFG nodes: every DFG node possesses its own set of bitfields. For example, to perform the reaching definitions data flow analysis, every DFG node has sets IN, OUT, GEN and KILL associated with it, addressed by multiplying the number of sets (4) with the size of the sets (the number of definitions in the definition universe divided by the integer size of the host compiler, rounded upwards) and adding n times the setsize to reach the individual sets (for example, `n=0` for addressing IN, `n=2` for OUT etc.). The resulting value is the offset from the memory address at which the first element of the bitfield is located.

To address the bit corresponding to the DAG node with `id=id` belonging to a certain DFG node, the address of the desired set is taken (using the method described previously); from that address, the offset into the set is calculated, in machine words: id/(size of integers); the remainder of this division is used to address the desired bit.

Runtime memory usage

To translate the bit vectors back to the actual definitions a list is compiled containing every definition in the program. This list is of the `List` type taken from and managed by the front end. Therefore every definition adds two extra pointers or 8 bytes to the total memory usage. From the previous paragraphs it is clear that the actual set of bit vectors (for each type of analysis) takes up a number of bytes corresponding to four times the number of nodes in the DFG multiplied by the number of number of integers needed to represent the definition universe. So, for reaching definitions and live variables, two equally sized bitfields of these length are necessary. Considering the fact that the last of these integers (one per set per DFG node) may contain a number of unused bits, it can be fruitful to combine a bitvector representation for the alias lists with the bitvectors for data flow analysis.

10.2 Algorithm complexity

Complexity of algorithms is measures by estimating the number of times the implementation passes its internal loops, increased by the number of calls to other function implementations multiplied by their respective complexities. Calls to library functions, such as allocating amounts of memory, are not taken into consideration. Only those algorithms that take more than one pass over some sort of data set are discussed. Calculating ref-def information for instance is linear to the amount of DAG nodes in the complete function and will not be discussed. Omitted algorithms can therefore be assumed to be at most linear to the number of DAG nodes comprising the function.

10.2.1 Building the DFG

Walking the DAG trees to combine them to basic blocks takes time linear to the number of DAG trees comprising the function. Updating the entry- and exit points of the DFG is linear with the number of interconnections in the DFG. Assuming that the front end only introduces labels if they are indeed used, i.e. for every label there exists a jump referencing that label, the minimum number of interconnections is equal to the number of labels in the program; the maximum number is the faculty of the number of labels. For DFG's with a large number of nodes, the latter case of course only represents pathetic flow graphs originating from pathetic programs.

10.2.2 Alias analysis

Because the administration necessary to access and update the alias lists introduces an important part of the complexity of certain algorithms, these functions are discussed first. Before adding an alias to the alias list of a certain basic block (or DFG node), the list is checked to find whether the alias is not already on the list. Adding an alias to a list of length N therefore introduces a complexity of order $O(N)$. Likewise, removing an alias from the list or checking if an alias appears on the list adds a complexity of the same order. Merging two lists, length M and N respectively, therefore requires time proportional to $O(M \times N)$. The calculation of the complexity for manipulating the `Pointer` is analogous to alias lists, with identical results. In fact, the manipulation of any linked list requiring an existence check in the program introduces the above complexities.

Initially, the alias analysis algorithm makes a pass over the DFG to initialize the alias sets. For every node, the alias information is calculated by the `alias_trans` function. This function inspects every definition in the basic block and calculates new aliases to be added to or removed from the block's alias list. First, it invokes the `aliases` function twice to calculate the alias lists of the defined symbol and the symbol pointed to. For a pointer of level n, the `aliases` function simulates n-1 times dereferencing the pointer under investigation by building a set of aliases of the same symbol with a dereferencing level one less than the current and again dereferencing these the next iteration step (recall the paragraphs on alias analysis in the data flow analysis chapter). Every iteration, the complete current alias list of the DFG is checked against the temporary alias list and depending on this check new aliases are added to a new temporary list of aliases. This new list is returned to `alias_trans`, and the old temporary list is freed. The upper bound for the number of pointer dereferences is 12 (dictated by ANSI-C); in the hypothetical case that A pointers of level twelve, all aliased with each other, have to be processed, the maximal number of checks and added pointers amounts to $12A^2 \times O(A)$ (twelve times checking A aliases from the DFG node against A identical temporary aliases and for every check one pointer added to a list of maximum length A), giving the `aliases` algorithm a complexity of order $O(A^3)$.

Subsequently, `alias_trans` adds a combination of every pointer of both lists to the (new) DFG alias list, again adding a complexity of $A^2 \times O(A)$ to the function. The total complexity of `alias_trans` therefore amounts to the order of $O(A^3)$ in the presence of a set of A cyclic, fully closed aliases.

After initialization, `alias_analysis` starts the iteration process. For every iteration, the DFG is traversed in depth first order. For every DFG node, the OUT-sets of its predecessors are merged into the new IN set of the DFG, and if this amounts to a different IN set, `alias_trans` is again invoked to calculate the new OUT set. The upper bound on the number of iterations is the depth of the DFG, i.e. the largest number of retreating edges on a non-cyclic path in the DFG; this is assumed to be a program complexity parameter and has been shown to be small (smaller than three) for practical programs, although a DFG with a depth equal to its number of nodes is theoretically possible. Therefore, for a program yielding a DFG with N nodes and a depth D, creating a total of A aliases, the complexity becomes of order $O(N \times D \times A^3)$, or $O(N^2 \times A^3)$ as absolute upper bound. The quadratic order in N can not be avoided unless a check is added for reducibility of the DFG, in which case a complexity of $M\log(N)$ can be achieved for reducible graphs (see [3]); the third order in A can be brought down if the linked list representation of the aliases is exchanged for another representation such as a hash table, a sorted linked list of a bitfield representation (but in that case the total number of possible aliases must be established beforehand). When using hash tables, the most sensible thing to do is to investigate whether the hash table used by Lcc's front end to store the symbol tables can be used.

10.2.3 Reaching definitions

After completing the alias analysis, the reaching definitions information is calculated in much the same way as the alias information. The initialization for the iteration, performed by `calc_genset`, requires one call to `aliases`, resulting in a set of aliases for the symbol to be defined (a set with cardinality 1 if the symbol is not a pointer). For every alias, a definition point is added to the list of definition points for the aliased symbol. Then, iff the set of aliases has a cardinality of one, all other definitions defining the current symbol directly (not indirectly) are killed. To do this, the defpoints list of the current symbol is traversed. In this

sequence of events, again the most expensive part is the call to `aliases`, with a complexity of $O(A^3)$ against a complexity of A times walking a defpoints list with maximum length equal to the total number of definitions in the DFG (which is not even an absolute upper bound on the number of aliases in the program; that would be A^2 with every pointer definition adding an alias with every other defined pointer or addressed symbol).

After initialization the iteration stage of the algorithm is entered. The iteration process is identical to the process used for alias analysis. For reaching definitions, however, a bit field has been initialized to represent all definitions. The set operations therefore reduce in complexity to one operation, such as bitwise AND, OR and NOT operations. The complexity of the iteration stage then becomes equal to the complexity of the alias analysis iteration stage without the set administration complexity, $O(n^2)$ for a DFG of n nodes. Remember that for practical programs the depth of the flow graph is small and the typical complexity for these algorithms will be near linear.

10.2.4 Live variable analysis

The live variable analysis function uses both the alias- and the reaching definition information. As stated, the iteration part of the algorithm is nearly identical to the reaching definitions and alias analysis algorithms and, using the same bitfield representation as reaching definitions, again introduces a complexity of $O(n^2)$ for a DFG comprising n basic blocks. The initialization phase of live variable analysis traverses the DFG once in reverse depth first order, and inspects every DAG in the basic blocks in reverse execution order (remember live variable analysis is a backward flow problem). Unlike reaching definitions, however, live variable analysis not only inspects the definitions itself but also the `uses` information collected during the reference-definition stage of the back end. Since every used and every defined symbol necessitates a call to `aliases` and for every symbol in the resulting set of aliases its respective list of definition points must be traversed, the initialization phase of live variable analysis takes a substantial amount of computing time. For a procedure comprising of D definitions, U uses, creating A aliases, the complexity is bounded by the following formula: $(D+U)(O(A^3)+AD)$, or in words the total number of symbols under consideration, multiplied by the complexity introduced by calls to `aliases` and the complexity of walking the `defpoints` lists which can never exceed the total number of aliases times the total number of definitions. Typically, U will be two or three times as large as D, and A can, worst case, have a quadratic relation with D (a new definition n can at most introduce n-1 new aliases, iff every definition indeed aliases the previous.)

10.3 Using the data flow information and implementing optimization algorithms

The data flow information currently available can be used to optimize register allocation and loops. It can also be used to eliminate redundant code produced by either the codewriter or the compiler front end. Redundant code can result from dead code intentionally introduced by the code writer to enable different compilation options, or dead code resulting from optimizations like copy propagation, common subexpressions or unused definitions.

Using the available information, the following points should be taken into consideration:

- Data flow analysis handles the CALL as if it were a normal, in line, instruction. This means that global variables and locals that have their address taken (all addressed locals reachable through dereferencing one of the parameters or global variables, not just those used as parameter!), may have their values changed after returning to the caller. In fact even the alias information can be affected by a function call, influencing not only the current basic block but the alias information in the complete DFG. That notion makes alias analysis and, as a result thereof, data analysis for pointers or depending on pointer information virtually impossible in the presence of function calls. It is left to the designer of the optimizing routines to establish a number of assumptions to enable this type of analysis in the presence of function calls, or to ignore globals and addressed variables in the optimization process. Note that the information on locals that do not have their addresses taken doesn't lose its validity over function calls. One option to overcome the previous problem is to store information on modified variables globally so the effect of a function call can be calculated. This solution however requires a fundamental change in

both the front- and the back end of the compiler.

- Volatile type variables may never be kept in registers.
- The available information is valid at a block boundary. For optimization of some DAG inside the blocks, it may be necessary to calculate the effect of preceding or succeeding DAG's on the value of the information. For instance, one might be tempted to conclude that a definition that doesn't occur in the block's OUT set for live variables (i.e. its value will never be used by any other block) can be eliminated altogether. However, one of the succeeding instructions inside the same block may use that value and in that case, eliminating the definition leads to an incorrect program. A similar case can be made for reaching definitions and alias analysis. If we want to know the set of aliases existing at the execution point of a single DAG, the block's In-set must be taken and the TRANS and TRANS1 functions should be used to calculate the effect of the preceding DAG's on that set.
- The effect of C's special `setjmp` and `longjmp` keywords for interprocedural jumps has not been investigated.
- Moving or eliminating blocks of code may change the structure of the DFG. Be aware of the possibility that the data flow analysis has to be redone!

Having mentioned the above limitations and keeping them in mind, the following notions can help in writing optimizing routines:

- How to find what variables are really accessed by a pointer dereference?

Providing the above restrictions do not apply, the `aliases` function can be used to find (the set of) symbol(s) reached by dereferencing the pointer. It should always be remembered that variables of aggregate type (arrays, structures, or unions) and the unknown symbol require special care in handling: if, after dereferencing, such a type is reached, it is still impossible to determine the exact memory location that will be accessed. Therefore, even if only one definition of such type reaches a certain execution point, and the value resulting from that definition still resides in one of the registers, it is not correct to eliminate the memory fetching operation and use the register instead.

- How to check whether the value of a variable, last defined outside the current basic block, can be taken from a register instead of fetching it from memory?

First note that this is possible if either there is only one definition of that variable reaching the current block, or if all reaching definitions are stored in the same register. Checking what definitions reach the current block can be done by walking the list of definition points, `x.defpoints`, attached to the symbol representing the variable, and checking the bits in the bitfield `IN` of the block, corresponding to those definitions. The bit number to be checked is the number stored in the `x.di` field of the defining DAG node (which was in turn extracted from the defpoint list). Every definition that has its corresponding bit set reaches the current basic block and verifying if those definition are stored in the same register can only be done by the register allocator. From this follows the notion that it may be beneficial to investigate the possibility of assigning the results of every definition point of one variable to the same register. Note that if the symbol in question is a dereferenced pointer, only a call to `aliases` can reveal the variables that are actually referenced. If exactly one symbol, not of aggregated type or unknown, was aliased and the alias information is valid (mind the previously mentioned restrictions!), the value may be taken from a register.

- How to decide if a definition is live at a certain execution point?

First, check the L_IN and L_OUT sets of the basic block the execution point resides in. If both sets have the bit corresponding to the definition in question set, the definition is live throughout the complete block. If one (or both) bits are not set, then at least during part of the basic block the definition is not live. With both bits not set and the corresponding bit in L_USE not set the definition is not live in this block; if the bit in L_USE is set, then the corresponding bit in L_DEF must also be set otherwise the dataflow analysis stage of the compiler contains a bug. The definition is then live in the part of the basic block enclosed by the definition and the last usage point in the basic block. This situation typically arises with temporary variables. With the bit in L_OUT set and L_IN cleared, the bit in L_DEF must also be set and the definition will be live from its execution point onward. Finally, the bit in L_IN set and in L_OUT cleared requires the existence of a usage point in the basic block (marking the bit in L_USE), which is also the last execution point at which the value resulting from the definition is used Note that it may still be possible that the basic block in question contains more than one usage point of the value under investigation: don't stop looking at the first occurrence of a usage point! Also note that to find a usage point it may be necessary to simulate execution to establish the possible aliases created inside the block. A similar procedure can be used to

compute if a definition reaches a certain usage point inside a basic block, of course allowing for the reverse order of the dataflow analysis.

It may be clear by now that the current implementation of data flow analysis was meant to support global (function level) optimization, and all information is valid at basic block boundaries. For local optimization the information gathered by data flow analysis can be used but has to be extended to be valid inside a specific execution point inside a basic block. This statement it true for alias analysis, reaching definitions and live variable analysis alike. To see how information at block boundaries can be propagated into the block it can be helpful to look at the initialization functions for every type of data flow analysis: `alias_trans` for alias analysis, `calc_genset` (called from `alias_analysis`) for reaching definitions, and `live_init` for live analysis.

10.4 Calling trees

To provide insight in the program flow inside the compiler, this chapter will elaborate on the calling hierarchy of the back end. Implementation names will be used; appendix VI lists all names with a short description of the functionality. Calling trees will be shown related to the different stages in the compilation process, i.e. first the building of the DFG, followed by alias analysis, reaching definitions, and live analysis. Routines to access and maintain the data structures are not shown.

The implementation of the reference- and definition information collection and the building of the data flow graph consists of the functions shown in figure 9. Gen is the function the front end calls to annotate the DAG's and is called every time the front end completes a forest of (relating) DAG's. First, `number` assigns a numbering to the nodes of the DAG's that are mainly used to help the programmer (me) to identify and reconstruct DAG trees from debugging information. `refdef` takes care of the collection of the reference- and definition information by calling `trans` which in turn calls the recursive `transl` to walk the DAG's. Again, `find_pointer` is used to keep pointer tuples unique. This is also the place where pointers in the code are initially detected and stored. Finally `update_dfg` performs the bundling of DAGs into basic blocks and updates the interconnection between the basic blocks.
Figure 10 shows that the alias analysis algorithm has been placed within the initialization of the reaching definitions implementation. This was done to stress the relation between the alias analysis implementation and the reaching definitions algorithm: reaching definitions is a form of analysis based on confluence, therefore needing an implementation of alias analysis that was also based on confluence. The implementation parts have the following functions: `depth_first` performs the depth first ordering of the data flow graph. The depth first order is subsequently stored in an array (by `ud_chain`). Next the definition universe is built by `add_definitions`. After those necessary initialisations alias analysis can be executed by calling `alias_analysis`. `alias_trans` is called to initialize the analysis and to calculate the effect of codetrees on sets of aliases. `calc_genset` pre-initializes the GEN and KILL sets used by the reaching definitions algorithm; it has already been stated (chapter 8.5) why it can be placed here. `aliases` simulates pointer dereferencing and `find_pointer` is called to ensure uniqueness of pointer tuples, enabling the comparing or two tuples by comparing the (C) pointers to their datastructures. Finally, `iterate` performs the actual iteration of the data sets over the data flow graph as described by algorithm 3.

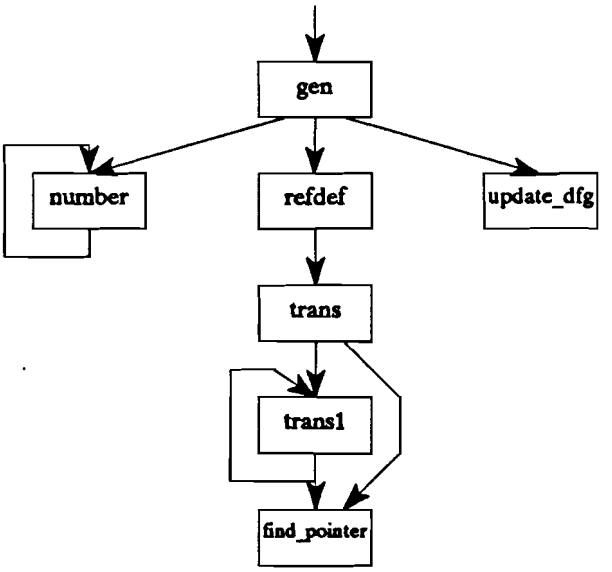

Figure 9 Calling tree for collecting reference-definition information and data flow graph construction

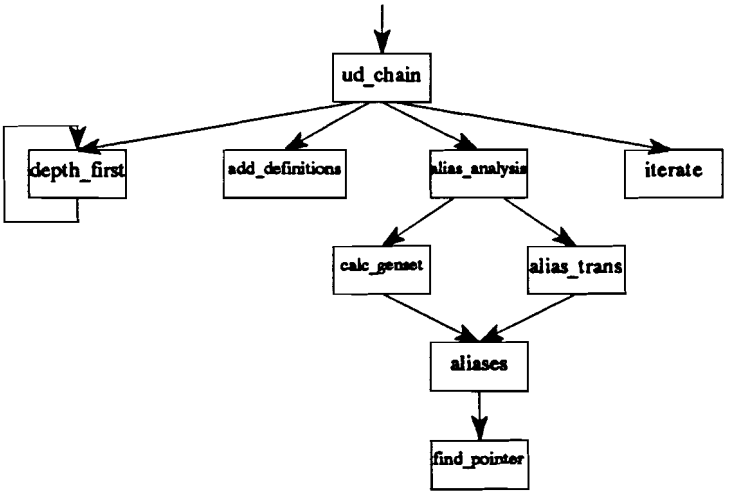

Figure 10 Calling tree for reaching definitions

Figure 11 shows the last of the three important calling trees of the back end, the live variable analysis tree. `live_analysis` (called by `function`), first initializes the various sets by calling `live_init` and subsequently performs the iteration. The chapter on data flow analysis explained the mechanism of this iteration. `Live_init` requires quite complex inspections of both the alias- and the reaching definitions information to establish what usage points must be marked and what definition points finally can kill. This, of course, is what `live_markuses` and `live_killdefs` are for. As always when working with pointers a call to `aliases` is necessary at different points to find what exactly gets defined or killed, and `aliases` still calls `find_pointer`, as it did before.

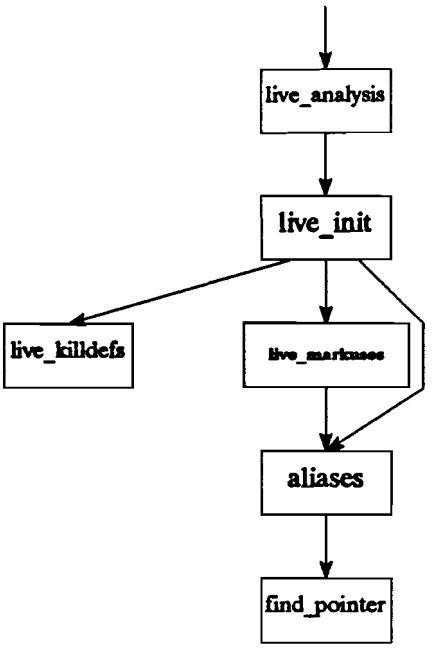

Figure 11 Calling tree for live analysis

10.5 Modifications to the front end

The front end was modified at a few points to enable certain wishes or requirements of the back end. These modifications can easily be found in the front end by scanning the source text for the pair of conditional inclusion statements `#ifdef pms_500...#endif` and `#ifndef pms_500 ... #endif`. Their effect and the reason for the modification is listed below:

- In `string.c`: functionality was added to use the front-end's linked list datastructure. Elements of this data type are managed by the front end in an efficient way to reduce the number of memory allocations and releases. Added functions were `free_list` to return a complete list to the heap, and `remove_from_list` to return single elements, though this function is currently not in use by the back end.
- In `simp.c` (function `simplify`): the 'simplification' to add a new temporary named 'a+c' to replace a sequence 'get the address of a and add (or subtract) the value c' in case c is a known constant of type (unsigned) integer was excluded because this led to the situation that the variable `a[1]` was not considered to be an element of the array a, leading to incorrect conclusions on behalf of the data flow algorithms.
- In `dag.c` (function `gencode`): an extra call to the back end function `defbranch` was added to enable the construction of the data flow graph in the presence of branch tables.
- Finally, the original driver uses a feature of the Unix operating system not supported by DOS, the pipe construction. This construction is used to transfer data between different modules of the compiler. The driver has been modified to use temporarie files instead of the unsupported pipe functions.

All modification take effect if the sources are compiled with `pms_500` defined, otherwise the original front end is generated. The modifications have to be enabled to use the PMS500 back end.

10.6 Validation and libraries

After the compiler has been implemented it will have to be validated. It has to be tested at a number of points, such as:

- Conformance to the ANSI C standard
- Correct compilation of arithmetic expressions (including checks for guard- and sticky bits, overflow bits, carry bits and so on)
- Generation of correct PMS500 code in relation to register allocation, stack usage, memory allocation and usage and so on. It is especially important to check all optimization types for actions that change program results.
- Stability of the compiler and of the compiled programs (related to the previous point)
- Sufficient, correct and usable (readable) documentation
- Correctness and usefulness of debug information

It is evident that this list is not exhaustive. Except for the first two items, all tests can or must be taken with specific tests written for this particular compiler, although the Lcc distribution comes with a number of programs that can be used to check (manually) whether translation of a number of basic C constructs proceeds correctly. These tests include programs for testing array and structure handling, spill code insertion and register allocation and recursion. Testing ANSI C conformance can be done using specialized test suites that are commercially or publicly available. While looking for compiler toolkits, a small number of validation suites were encountered:

- ACE C Validation Suite - 50000 lines of source over 600 programs perform over 2000 tests. The suite is priced at 15500 dutch guilders. It is available via:
  - ACE Associated Computer Experts bv — Phone: +31 20 646461
  - van Eeghenstraat 100 — Telex: 11702 (ace nl)
  - 1071 GL Amsterdam — Fax: +31 20 750389
  - The Netherlands
- C Validation suite from MetaWare, $2,000
  - 903 Pacific Ave, Suite 201
  - Santa Cruz, CA 95060
  - (408) 429-6382
- The Plum Hall Validation Suite for C $10,000 (This is the suite used to validate the Lcc front end)
  - Plum Hall
  - 1 Spruce Ave.
  - Cardiff, NJ 08232
  - (609) 927-3770
- The PERENNIAL Validation Suite for C Compiler Validation
  - PERENNIAL
  - 4677 Old Ironsides Drive, Suite 450
  - Santa Clara, CA 95054
  - (408) 727-2255
- C Compiler Torture Test - Checks a compiler against K&R. $20
  - The Austin Code Works
  - 11100 Leafwood Lane
  - Austin, TX 78750-3409
  - (512) 258-0885
- HCR offers a C Test Suite in various forms (50,000 to 350,000 tests)
  - HCR Corporation — Phone: (416) 922-1937
  - 130 Bloor Street West — Telex: 06-218072 HCR TOR
  - Suite 1001 — Fax: (416) 922-8397
  - Toronto, Ontario
  - Canada

This list is by no means intended to be complete and is an extract of a list that was originally compiled by Tom Wood at Data General. It was made available as an article posted to the USENET comp.compilers newsgroup.

To finally complete the compiler package the item of libraries has to be considered. To conform to the ANSI C-standard for a freestanding implementation, the features listed by the standard headers `<float.h>`, `<limits.h>`, `<stdarg.h>` and `<stddef.h>` must be provided. The contents of these headers can be found in [24]. The Gnu public libraries might be used to help implementing these libraries, although these libraries are protected by an extended version of the Gnu public licence.

11 Conclusions

An investigation was held to establish how an ANSI-C compiler could be built in a relatively short period of time. First a number of compiler building toolkits, retargetable compilers and compiler compilers were inspected to help with this task. It was expected that by using a toolkit, retargetable compiler or compiler compiler the amount of work necessary to develop a front end could be greatly reduced. The selected retargetable compiler, Lcc, lived up to these expectations, providing a complete front end thus reducing the task of writing a compiler to constructing a back end and, in this case, a number of optimizing routines.

While constructing a first version of the back end, problems, possibilities and options introduced by either the front end or the processor were discovered and investigated. The main conclusion resulting from this phase of the investigation was the need for a data flow graph to provide insight in the control flow of the source program, and a number of optimizations expected to yield the best result for the Lcc-PMS500 combination. Effective optimizations included loop-optimizations such as loop-invariant code detection and movement and induction variable detection, code elimination optimizations like dead code elimination through constant folding, copy propagation or common subexpression detection and data flow analysis for use by register allocation. Whenever possible these optimizations would be combined with optimizations for execution time, providing these don't interfere with the optimizations for code elimination. As a side effect, the first version of the compiler showed a large number of MOV instructions resulting from address calculations for variables either residing on the stack or as member of an aggregate variable type, thus needing a base address and an offset from that base. A small investigation was held to find if an extension of the PMS500 instruction set with a `MOV <reg>, address+offset` type instruction could be beneficial. The result of this experiment point to an approximate code reduction of 10%.

In order to perform the optimizations mentioned above, the source program had to be subjected to data flow analysis to provide insight in the data flow of the source program. Of the types of data flow analysis possible, the types necessary to be able to implement the optimizations mentioned above are primarily forward and backward analysis based on confluence, including the so called 'reaching definitions' and 'live variable analysis' analysis. To enable these types of data flow analysis, a graph had to be build to gather information on definition and reference points of variables (ref-def information) and a solution had to be found to handle the possibilities of the versatile pointer type in C. Also, loops in the data flow graph must be detected. To allocate registers the technique of register allocation via graph colouring has been investigated, as well as register allocation by simulated execution. The latter was shown to be the better choice for this situation.

For the final compiler, the data flow analysis types 'reaching definitions' and 'live variable analysis' have been implemented, as well as reference-definition information and alias analysis to handle pointers. An algorithm is provided for implementing loop detection. With the information gathered by these algorithms (provided loop detection will be implemented), possible optimization routines that can be implemented include induction variable detection, detection of loop-invariant code and the resulting movement of this code out of the loop, constant propagation and folding and dead code elimination. To acquire a working compiler, register allocation and code selection have to be implemented. After that, the optimizations previously mentioned can be implemented to provide better code. To reach a production version of the compiler it must be tested and finally a minimum set of libraries must be provided.

12 Bibliography

1 A Code Generation Interface for ANSI C, Christofer W. Fraser, David R. Hanson
Research Report CS-TR-270-90 , 1992

2 A Compiler Generator Usage Inquiry, Paul Jansen
Philips Research Information and Software Technology, Eindhoven, the Netherlands No. RWB-510-re-94009 , 1994

3 A Fast and Usually Linear Algorithm for Global Flow Analysis (172-202), Susan L. Graham, Mark Wegman
Journal of the American Association for Computing Machinery Vol. 23, No. 1 , 1976

4 A New Strategy for Code Generation - the general purpose optimizing compiler (29-37), W. Harrison
Proceedings of the 4th ACM Symposium on Principles of Programming Languages , 1977

5 A Unified Approach to Global Program Optimization (5-15), G.A. Kildall
Proceedings of the ACM Symposium on Principles of Programming Languages , 1973

6 Automatic Derivation of Code Generators from Machine Descriptions (173-190), R.G.G. Cattel
ACM Transactions on Programming Languages and Systems Vol. 2, No. 2 , 1980

7 Characterizations of Reducible Flow Graphs (367-375), M.S. Hecht and J.D. Ullman
Journal of the Association for Computing Machinery no. 21 , 1974

8 C Handboek, Brian Kernighan, Dennis M. Ritchie
Prentice Hall Academic Service 2nd Edition , 1990
ISBN 90-6233-488-1

9 Code Generation Using Tree Matching and Dynamic Programming (491-516), Alfred V. Aho, Steven W.K. Tjiang
ACM Transactions on Programming Languages and Systems Vol. 11, No. 4 , 1989

10 Computers and Intractability: a guide to the theory of NP-completeness (191), Michael R. Garey and David S. Johnson
Freeman, New York , 1979
ISBN 0-7167-1045-5

11 Elements of the theory of computation (349-356), Harry R. Lewis, Christos H. Papadimitriou
Prentice-Hall International Editions , 1981
ISBN 0-13-273426-5

12 Global Data Flow Analysis and Iterative Algorithms (158-171), John B. Kam and Jeffrey D. Ullman
Journal of the Association for Computing Machinery Vol. 23, No. 1 , 1976

13 Lecture Notes in Computer Science, vol. 323: Attribute Grammars: definition, systems, and bibliography, A Survey on Attribute Grammars in Three Parts., P. Deransart, M. Jourdan, B. Lorho
Springer-Verlag, Berlin Heidelberg. part II: Review of Existing Systems. , 1988
ISBN 3-540-50056-1

14 Introduction to Compiler Construction, Thomas W. Parsons
W.H. Freemand and Company, New York, U.S.A. , 1992
ISBN 0-7167-8261-8

15 Node Listing techniques applied to Data Flow Analysis (10-21), K.W. Kennedy
Proceedings of the 2nd ACM Conference on Principles of Programming Languages , 1975

16 Node Listing for Reducible Flow Graphs (286-299), A.V. Aho
Journal of Computer and System Sciences no. 13 , 1976

17 PMS500 programmers manual (unpublished)

18 Principles of Compiler Design, Alfred V. Aho and Jeffrey D. Ullman
Addison-Wesley Publishing Comnpany, Reading Massachusetts, U.S.A. Third printing , 1979
ISBN 0-201-00022-9

19 Proceedings of the Third International Workshop on Compiler Compilers (CC '90), Schwerin, France, October 1990, D. Hammer (Ed.)
Lecture Notes in Computer Science, Springer Verlag, Berlin Heidelberg no. 477 , 1991
ISBN 3-540-53669

20 Rationale for American National Standard for Information Systems - Programming Language - C (9-70)

21 Register Allocation via Coloring (47-57), Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra,

John Cocke, Martin E. Hopkins, Peter Markstein
Computer Languages no. 6 , 1981
22 Register Allocation & Spilling via Graph Coloring (98-105), G.J. Chaitin
SIGPLAN 82 Symposium on Compiler Construction , 1982
23 Some Topics in Code Optimization (76-102), Christofer Earnest
Journal of the Association for Computing Machinery Vol. 21, No. 1 , 1974
24 The Annotated ANSI-C Standard, Herbert Schildt
Osborne McGraw-Hill, California, U.S.A. , 1990
ISBN 0-07-881952-0
25 The Art of Compiler Design, Theory and Practice, Thomas Pittman and James Peters
Prentice-Hall International, Inc, London , 1992
ISBN 0-13-046160-1

I. List of criteria

A division is made between utilities or code used to generate the compiler front end, and utilities or code used to generate the compiler back end.

Criteria considering the complete compiler tool box or the retargetable compiler front end:

- Transparancy: can somebody without knowledge of building compilers generate a new compiler?
- Sufficient documentation?
- Does the system generate the complete compiler or do some parts have to be generated manually?
- What are the resource requirements? (Memory, disk space, processor power)
- Is the system still supported by the author?
- What is the state of development? (Finished, under construction)
- Are there restrictions on commercial use?
- Is the system in use by other people? If so, how many and what is their experience?
- How much time does it take to generate a new compiler?
- What is the status of the C front end? Are there alternative front ends available?
- Can the sources (both of the system and the generated compiler) be ported to a DOS platform, and how long will this take?
- What is the availability of the sources?

Criteria concerning the resulting compiler or retargetable compiler back end:

- What is the complexity of the processor specific information?
- What is the quality of the generated compiler?
  - Optimized for size and speed
  - Compilation speed
  - Conformance to design goals:
    - Accept ANSI C code
    - Output assembly code listing for the PMS500 processor core
    - Pass a C-compiler test suite. (A decision has to be made on the suite to be used)
    - Debugable assembly code
    - Optimize primarily for size
    - Generate code for 16 and 32 bit architectures, and possibly 64 bit.

II. List of compiler generating utilities

A. Compiler Tool Kits

1. Amsterdam Compiler Kit (ACK)

Authors: Vrije Universiteit Amsterdam,
Andrew S. Tanenbaum, Hans van Staveren, Ed G. Keizer, Johan W. Stevenson
Cost: $995,- for universities,
$9995,- for commercial use (prices per april 1992)

Documentation
A Practical Toolkit for Making Portable Compilers
Andrew S. Tanenbaum, Hans van Staveren, Ed G. Keizer, Johan W. Stevenson
report no. IR-74, october 1981, Vrije Universiteit Amsterdam

Description of a Machine Architecture for Use With Block Structured Languages
Andrew S. Tanenbaum, Hans van Staveren, Ed G. Keizer, Johan W. Stevenson
report nr. IR-81, august 1993, Vrije Universiteit Amsterdam

Ease of use
The ACK information sheet reports four phases to generate a new compiler:

- Writing a back end table for the processor. This may take 2-3 months
- Writing the machine-dependent part of an assembler for the processor. This may take several days. Since an assembler for the PMS500 already exists, this part can be skipped.
- Writing a system-call interface. This may again take a couple of days. (Only if the PMS500 will posess system calls, of course)
- Writing a conversion program, converting the machine-independent object format that ACK produces into a binary for the machine at hand.

Several development utilities are provided: Test suites for back-ends, intermediate code assembler- and interpreter, libraries written in processor independent form etc.

Description of the front end
A C front end is part of the package. Conformance to ANSI-C unknown. Front ends for Modula 2, Pascal, Basic and Occam.

Description of the back end
The user has to generate a table to map machine independent instructions generated by the front end to (sequences of) machine-specific instructions. Peephole and global optimization are performed on the intermediate code; table-driven peephole optimization is performed on the machinecode. Back-ends for DEC PDP-11, DEC VAX, Motorola 68000, Motorola 68020, Intel 8080, Intel 8086, Zilog Z80, Zilog Z8000 and NS 16032 will be supplied.

Resources
ACK has been tested on the following machines:
VAX, SUN workstations, PDP-11, AT-compatibles running Microsoft Xenix and an M68000-system running Unix Sytem V.0.

Restrictions
Unknown

Possible problems
The IR used is stack-directed; somewhere during code-generation variables will have to be assigned to registers and memory locations. The back-end driver table will probably be used to make this assignment.

If this is the case, register allocation may be ACK's weak spot.

Pro's
ACK is a completely developed system. Multiple front- and back ends are available; these make clear that generating compilers for different architecture widths is no problem. Ack is commercially in use by various companies. ACK comes with a lot of utilities.

Con's
Cost, possibly weak on register allocation.

Info on the compiler generated by ACK
The compiler will be larger and slower than handwritten compilers, but debugging options and optimization will be included so the generated compiler will be fairly complete. Register allocation will be a weak spot.

Info on the code for the PMS500
Code will be compact and fast due to the number of optimizations performed by the compiler, though handwritten codegenerators could take more advantage of the PMS500's specifics.

Why/why not
ACK is expensive. ACK's weak register allocation poses a problem because register allocation is exactly the area for which the PMS500 needs special care.

2. ELI

Authors	Compiler Tools Group, Electrical and Computer Engineering Department University of Colorado Boulder,CO, USA.
Cost	-

Documentation
Full documentation is supplied with the package; also available via anonymous ftp from the University of Colorado

Ease of use
The ELI toolkit is a very extensive set of tools but does not contain a code generator as such. ELI generates AST's (as do most other toolkits) and includes utilities to process these trees, thus helping to generate code generators. ELI is highly integrated and contains an online help function. ELI is the most flexible toolkit which means that it can generate compilers for a wide range of applications. Generating a specific compiler with ELI however will take more time than with a dedicated, less flexible tool.

Description of the front end
Specifications to generate a C-code parser are supplied as example. ANSI conformance is unknown and the parser description probably doesn't include semantical analysis.

Description of the back end
There is no explicit back end; it must be generated using ELI's tree walking tools. Optimalization will have to be included manually.

Resources
The complete system takes about 30MB of harddisk space on a UNIX platform. It has been tested on the following systems:
SUN workstation (SunOS 4.1.x), Sun workstation (Solaris, SunOS 5.2 sun4m sparc)
DECstation 5000 (Ultrix 4.2), IBM RS/6000 (AIX version 3, release 2)
HP9000/370 (HP-UX version B, release B.09.00)

HP9000/175 (HP-UX version E, release A.09.01)
SGI (IRIX system V release 4.0.5F)
LINUX (Linux 0.99.12 Slackware-Release 1.0.3 i486)

Restrictions
-

Possible problems
Translating the AST to code, optimizing. Compilers generated by ELI will probably be larger and slower than handwritten ones. It will take quite some time to get to know ELI. ELI works on UNIX platforms; the generated compiler will have to be ported to a DOS platform.

Pro's
Flexible, free, (in the end) user friendly.

Con's
No code generator, large.

Info on the compiler generated by ELI
Compilers generated by toolboxes in general will be larger and slower than handwritten ones. Because a code generator has to be built, specific parts of the code generation process can be stressed. This can be an advantage with the PMS500's relatively straightforward instruction set and architecture, but this goes for all utilities without a back end generator.

Information on the code for the PMS500
Quality of the code will depend on the amount of work put into the compiler. Creating good code will take more time using ELI then toolkits with back end generators.

Why/why not
ELI is large. It will take a lot of time to generate an acceptable compiler using ELI. ELI offers unneccesary flexibility.

3. The GMD Tool Box (Cocktail)

Authors	Josef Grosch, Helmut Emmelmann
Cost	Back end (BEG) costs $2500,- for commercial, $250,- for non-commercial use.

Documentation
Every part of the toolbox is seperately documented, the back end even extensively:

BEG- Generator for Back Ends
H. Emmelmann, F.-W. Schröer & R. Landwehr
SIGPLAN '89 PLDI conf., Portland
SIGPLAN Notices 24, 7 (july 1989), pp. 227-237

A Tool Box for Compiler Construction
Report No. 20, Jan. 21, 1990
Gesellshcaft Für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
(Vincenz-Prieβnitz-Str. 1
D-7500 Karlsruhe)

BEG - a Back End Generator, User Manual
same address

Ease of use

As with every compiler construction toolkit, the compiler is generated in phases: from a description of the source language to an intermediate form a scanner, a parser and a semantic analyser have to be generated. GMD even uses some extra representations. For each of these representations, a regular expression, attribute grammar of other kind of source-target mapping must be supplied. The modules will then be generated automatically. After that a mapping from intermediate code to target language must be supplied, from which a code generator generator will be derived. All generated modules together make up the compiler. A (standard) grammar to generate a C front end is available but is known to have bugs. The back end generator generates good code generators.

Description of the front end
The compiler is made by combining the previously mentioned modules. This will result in a large compiler generating good code, though not as good as code produced by a hand written code generator, even if the techniques for code generation used by the new back end are largely the same as handwritten ones. Debugging options will be minimal.

Resources
BMG was developed on a SUN workstation but a port to PC exists. Availability of this port is unknown.

Restrictions
The toolbox with the exception of BEG is freely available. Since the author of BEG took the program to an Esprit project it is unclear wether it still is commercially available.

Pro's
Toolkits provide flexibility. Cocktail is comparable to ACK as a toolkit but the resulting compiler from BEG will probably generate better code.

Con's
Incomplete front end, back end possibly unavailable. No easy way of debugging the final (PMS500) code, whereas ACK provides EM code debugging.

Info on the compiler generated by Cocktail
Again, automatically generated compilers will be slower and larger than handwritten ones, although tests provided by the authors indicated that Cocktail-generated compilers where relatively small and fast. Dynamic memory usage was on the high side.

Info on the generated code for the PMS500
The codegenerater created by BEG handles register allocation as wel as instruction selection. The AST can also be transformed to generate better code. This results in code comparable to code generated by hand-written compilers and even better than code generated by native SUN-compilers (from tests supplied by the authors).

Why/why not
The C grammar is incomplete and it will take a lot of time to finish it. BEG may no longer be commercially available.

4. Purdue Compiler Construction Tool Set (PCCTS)

Authors School of Electrical Engineering, Purdue university, West Lafayette
Cost -

Documentation
The PCCTS reference manual comes with the distribution; there's a USENET newsgroup especially for PCCTS that is in frequent use.

Ease of use
PCCTS comprises of two tools, antlr and dlg. These are lex- and yacc like scanner/parser generators,

extended to handle attribute grammars. Another tool in frequent use with PCCTS is sorcerer, a tool to walk AST's and emit code or actions along the way. PCCTS uses the same approach as toolkits like ELI and Cocktail to generate compilers, but is less sophisticated and doesn't have a specific code generator generator. Learning to use PCCTS would be easier than ELI or Cocktail, and the functionality may suffice for our needs.

Description of the front end
As for all lex- and yacc like tools a grammar for accepting C exists but will probably need some work.

Description of the back end
N/A

Resources
?

Restrictions
-

Possible problems
It will take some work to generate a front end, and the complete back end has to be constructed. This gives a lot of control over the final PMS500 code quality but will also take a lot of time.

Pro's
Easier to learn than ELI. PCCTS is actively supported by it's authors so advice and bug corrections are easily available. PCCTS contains that part of ELI's functionality that is of use for generating a compiler for the PMS500.

Con's
It will take a lot of time to generate a compiler. Generated compiler and PMS500 code will be comparable to ELI results.

Why/why not
PCCTS is easier to use then ELI, while the neccesary functionality is present. PCCTS is actively supported. It will take a lot of time to generate a quality compiler.

B. Retargetable compilers

1. Lcc

Authors: Christofer W. Fraser,
AT&T Bell Laboratories, 600 Mountain Avenue
Murray Hill, NJ 07974

David R. Hanson,
Department of Computer Science, Princeton University
Princeton, NJ 08544

Cost
Front end is free. Some back-ends are available, though not for commercial use.

Documentation
A Retargetable Compiler for Ansi-C
Christofer W. Fraser, David R. Hanson
Research Report CS-TR-303-91

A Code Generation Interface for Ansi C

Christofer W. Fraser, David R. Hanson,
Research Report CS-TR-270-90

(Both available via anonymous FTP from Princeton University)

Ease of use
Lcc users report that a dumb compiler could be generated in a few days. Lcc supports standard VAX and SUN debugging tables, and supplies some extra facilities for debugging and profiling.

Description of the front end
Lcc was designed to use ANSI C. No other front ends possible (lcc is the front end). No global optimization. Elimination of common subexpressions and constant folding.

Description of the back end
The back end will have to be supplied by the user. The interface is relatively easy. Back ends available for VAX, MIPS, Motorola 68020+68881 FP coprocessor. These are reported to generate better code than native compilers but not as good as Gcc with optimization.

Resources
Lcc was developed to run on unix systems. It needs a third-party preprocessor like the GNU gpp preprocessor.

Restrictions
The Lcc front end may be used freely but charging money for distributing is prohibited. Charging money for personally developed back-ends is allowed. Using the front end to write, for instance, a C++ compiler is prohibited.

Possible problems
The fact that the complete back end has to be written introduces problems like register allocation and machine dependent optimization.

Pro's
Front end ready, correct and fast. Ease of use. Resulting compiler small.

Con's
Impossible to modify the compiler to accept other source languages. Complete back end has to be written.

Porting problems

Lcc uses the host's arithmic to fold constant expressions. This results in overflows when the host's native integer size is smaller than the target's integers. To run on a DOS platform, a DOS extender and 32 bit compiler will be neccesary. Generating code for a 64 bit version of the PMS500 will be a nontrivial problem. Since Lcc was written for a UNIX platform, code rewriting will be neccesary to compensate for non supported system calls such as fork and pipe.

Info on the resulting compiler
The front end is handcoded and so very fast. It includes debugging options. Quality of the codegenerator largely depends on the time invested in the project, although the interface between front- and back end will surely impose some restrictions.

Info on the generated code for the PMS500
The front end performs a few optimizations on the AST, but since code optimizations and register allocation have to be provided by the programmer, code quality is a function of invested time and designer ability.

Why/why not
Complete front end including semantic analysis. Easy to learn as the first version of a compiler should be

available in just a few weeks (according to Lcc users). Lcc is still supported and a 386 back end and an Lcc book are expected in a couple of months. The Lcc front end is free but restricted to the supplied copyright notice. Resulting compilers will be small. Generating compilers with integers larger than 32 bits (or better: larger than the integer size of the host Lcc is running on) is problematic.

2. GCC

Authors Free Software Foundation
Cost None.

Documentation
Extensive online documentation, readable with a special info-reader. There's info on what GCC is and does, a description of RTL (GCC' intermediate representation) and how to port GCC or generate new back-ends for it. USENET has special newsgroups devoted to GCC. The compiler is actively under construction. Updates and bug fixes appear regularly. In various locations all over the world people have ported or are porting GCC to generate code for a wide variety of processor architectures.

Ease of use
Gcc is very large and it will take a lot of time to learn all it's features. GCC has loads of options and utilities with new ones appearing all the time. Making a back end for a new compiler turns out to be quite a chore. It takes some time to get to know GCC and to be able to create a new back end.

Description of the front end
GCC accepts various C dialect including ANSI C. Using the -pedantic options causes GCC to complain about every little deviation from the standard, but different options provide a fairly extensive superset of the ANSI C standard.

Description of the back end
Creating a new back end is not a trivial task. In the end, however, GCC probably generates the best possible code.

Resources
GCC can run on almost all UNIX platforms as well as most conventional microcomputer platforms as DOS, Atari, Amiga or Acorn, altough a large amount of disk space, lots of memory and a powerfull processor are required for smooth operation.

Restrictions
See the Gnu General Public License.

Possible problems
GCC might cause a problem when generating code for 16 bit architectures. It has been done before, however, for a DSP with an architecture that resembled a common RISC structure.

Pro's
GCC is free; if a working version can be constructed then there are a variety of possibilities for upgrades and utilities in the future, as well as a large group of users for questions and problems.

Con's
Huge resource requirements. Writing a back end is not a trivial task.

Porting problems
GCC is up and running for a DOS platform. 64 bit ints can be generated using GCC's Long Long int's, however this is not ANSI compatable. Generating code with int's of 16 bit poses a problem.

Info on the resulting compiler

GCC requires a large amount of system resources: the compiler is large and not very fast. However, you get a lot of extra options and a great optimizing compiler, including different front ends to accept different languages and a large set of utilities for debugging and profiling.

Info on the code for the PMS500
As GCC is one of the best optimizing compilers, generated code will be compact an/or fast.

Why/whu not
Writing a code generator for GCC is not a trivial task. GCC is very large and incorporates a lot of unneccesary functionality. A less optimizing compiler will be able to generate comparable code for the PMS500.

3. Archelon User Retargetable Development Tools II

Authors Archelon Inc.
460 Forestlawn Road
Waterloo, Ontario N2K2J6
Tel. (519)746-7925

Cost DOS version: US$3495,-

Documentation
For full documentation: see Archelon's information folder. This is a new system; they couldn't generate code for 64 bit int's yet but if we really wanted to, they could add it to their actions list for medio '95. In the same period a debugger will be developed. For approx. US$50.000,- they could even build the complete compiler for us.

Ease of use
Processor dependent information has to be supplied in text files, being:

- The Compiler Information File, for information on registers, operand types, instruction formats, instructions, code tables and the mapping from IR to the code table. This will take up approximately 2000 lines of code.
- The Machine Definition file: for a mapping from assembly to object code (for the assembler); between 1000 and 3000 lines.
- The Replacement Rule file for the peephole optimizer
- The Microcode Definition file, unused for this project.

The complete package includes a C preprocessor, an ANSI C compiler, a peephole optimizer, a code convertor/compactor (for parallel/pipelined processors), a microcode assembler, a linker and an object librarian.

It's a flexible system and perfectly suited for generating the PMS500 compiler. Almost every problem that could be solved without knowing anything about the processor has been solved, using a minimum number of assumptions.

The systems comes complete with a Users Guide, Reference Manuals, one year of support and P&P. Extra copies against 40% reduction.

Description of the front end
The package is designed to comply to ANSI C norms with extensions to supply the compiler with information on how to generate better code. Even inline assembly is a possibility. Extensions include global register variables, fast implementations of the 'switch' statement, inline function expansion, hardware loop counter control, use of built-in or direct assembly code, user-specified register usage, use of special registers for argument passing, multiple address spaces and symbolic debug tables.

Description of the back end
All processor dependent information is supplied using textfiles. Estimated number of lines range between 2000 and 5000. Optimalizations include constant folding, global common subexpression elimination in

extended conditional regions, register allocation by graph coloring, peephole optimization.

Resources
Binaries for the following systems are available:
DOS, Unixware on Intel processors, SUN solaris on Intel and Sparc workstations, HP-UX/HP-PA

Restrictions
This system is sold per package or per site, and cannot be handed out to other users. The system doesn't generate a compiler: it is the compiler which the user can retarget to suit his needs. Compiled sources are free but the compiler itself cannot be distributed.

Possible problems
Restrictions, cost.

Pro's
It's the complete package for our needs.

Con's
Parts of the system are still under development.

Porting problems
None. A DOS version is available.

Information on the resulting compiler
The system is the compiler. There's no indication on size, speed and memory usage of the system.

Information on the code for the PMS500
This system can handle architectures with a much greater complexity than the PMS500. A number of optimalizations are performed and the compiler can even be supplied with compiling directives to squize the last bit of perfomance out of the code.

Why/why not
The copyright restrictions pose the biggest problem, next to the cost and the fact that the system is still under construction. Besides that, this system can do more than is needed for this project.

III. Summary of discarded tools

Lex, Yacc, Bison, Flex, Ox, Muskox: All parser generators and lexical analysers derived from Lex and Yacc, based on attribute grammars.

Production Quality Compiler Compiler Project: abandoned several years ago, and never yielded the result people expected from it.
ACC: Never received information
PCC: The portable compiler, the program that started it all.
CCG: Used internally by Harris Computer systems Division and not for sale
In: 'Lecture notes in Computer science', no. 323,
"attribute grammars"
Pierre Deransart, Martin Jourdan, Bernard Lorho,
This work describes a large number of compiler compilers based on attribute grammars. These compiler compilers are comparable to toolkits like PCCTS and Cocktail (also listed), but older.

Twig, Codegen, Burg, MIMOLA, Pagode: codegenerators of the same type, walk AST's. Can be used to help generate a back end for toolkits or retargetable compilers; a version of the Burg code generator was used to generate the x86 back end for Lcc.

IV. List of Lcc opcodes

Opcode	Type Suffix	Description
ADDRF	P	address of a parameter
ADDRG	P	address of a global
ADDRL	P	address of a local
CNST	CSIUPFD	constant
BCOM	U	bitwise complement
CVC	IU	convert from char
CVD	IF	convert from double
CVF	D	convert from float
CVI	CSUD	convert from int
CVP	U	convert from pointer
CVS	IU	convert from short
CVU	CSIP	convert from unsigned
INDIR	CSIPFDB	fetch
NEG	IFD	negation
ADD	IUPFD	addition
BAND	U	bitwise AND
BOR	U	bitwise OR
BXOR	U	bitwise XOR
DIV	IUFD	division
LSH	IU	left shift
MOD	IU	modulus
MUL	IUFD	multiplication
RSH	IU	right shift
SUB	IUPFD	subtraction
ASGN	CSIPFDB	assignment
EQ	IFD	jump if equal
GE	IUFD	jump if greater than or equal
GT	IUFD	jump if greater than
LE	IUFD	jump if less than or equal
LT	IUFD	jump if less than
NE	IFD	jump if not equal
ARG	IPFDB	argument
CALL	IFDBV	function call
RET	IFDV	return from function
JUMP	V	unconditional jump
LABEL	V	label definition

Table 3 List of Lcc opcodes

suffix meanings:

C	constant	P	pointer
S	short	F	float
I	integer	D	double
U	unsigned	B	block (struct, union, array)
		V	void

So 'ADDI' means integer addition and ASSGNB means assignment of one block to another (by value, not by reference).

V. The PMS500 instruction set

Mnemonic		Description
The PMS500 control flow instructions		
JMP	<addr>	Jump to address <addr>
JMP	<reg>	Jump to address in reg
JMPC	<reg>	Jump via table in code space, PC:=rom[<reg>]
JSR	<addr>	Jump to subroutine at address <addr>
Bxx	<addr>	Branch conditionally to address. Max. displacement is -127..+128. xx or <cc>
BRA	<cc>,<addr>	represents the condition to be tested
BSxx	<addr>	Branch conditionally to subroutine at address <addr>. Max. displacement is -127..+128.
BSR	<cc>,<addr>	xx or <cc> represents the condition to be tested
RET	<cc>	Conditional return from subroutine. <cc> is optional
RETI	<cc>	Conditional return from interrupt. <cc> is optional
NOP		No operation (BRN $+1)
The PMS500 data transfer instructions		
Register to Register Transfer		
MOV	<drg>,<srg>	Transfer data form <srg> to <drg>
Move bits immediate data to register		
CLR	<reg>	Transfer constant data to <reg>. For constants that need more than 8 bits to store the
MOV	<reg>, #data5	constant has to be split in an 11- and a 5 bit part and the actual transfer consists of a
MOV	<reg>, #data8	move of the 8 bit part into HIGH immediately followed by a move of th 5 bit part to the
MOV	HIGH, #data11	regsiter. The full 16 bits will be written to the register
Move data from/to code space (program memory space)		
MOVC	<drg>,<srg>	Transfer indexed data from program memory space to <drg>
STRC	<drg>,<srg>	Store data from register in code (program memory) space pointed by <srg>. This instruction requires extra hardware
Move data from/to stack		
PUSH	<srg>	Push register onto stack
POP	<drg>	Pop register from stack
PMS500 arithmetic instructions		
Arithmetic Dyadic Instructions		
ADD	<drg>,<srg>	Add <srg> to <drg>
	<drg>,#data5	Add immediate data to drg
ADDC	<drg>,<srg>	Add with carry
	<drg>,#data5	
SUB	<drg>,<srg>	Subtract
	<drg>, #data5	
SUBC	<drg>,<srg>	Subtract with carry
	<drg>,#data5	
RSUB	<drg>,<srg>	Reverse subtract: <drg> := <srg> - <drg>
	<drg>,#data5	
CMP	<drg>,<srg>	Compare (flags set according to <drg>-<srg>
	<drg>,#data5	

Mnemonic		Description
Arithmetic Monadic instructions		
BSWAP	<drg>	Byte swap within reg
INC	<drg>	increment (ADD #1)
DEC	<drg>	Decrement (SUB, #1)
NEG	<drg>	Negate (RSUB #0)
Bitwise Logical Dyadic Instructions		
AND	<drg>,<srg>	Bitwise Logical AND
	<drg>,#data5	
OR	<drg>,<srg>	Bitwise Logical OR
	<drg>,#data5	
XOR	<drg>,<srg>	Bitwise Exclusive OR
	<drg>,#data5	
Bitwise Logical Monadic Instructions		
COMPL	<drg>	Complement (as XOR #-1) (2-word instruction)
Bit Manipulation Instructions		
BTST	<drg>,<srg>	Bit test (logical AND) <drg> not altered
	<drg>,#data5	
BSET	<drg>,<srg>	Bit test and set
	<drg>,#data5	
BCLR	<drg>,<srg>	Bit test and clear
	<drg>,#data5	
Shift Instructions		
LSR	<drg>	Logic shift right
LSL	<drg>	Logic shift left
ROR	<drg>	Rotate right
ROL	<drg>	Rotate left
RCR	<drg>	Rotate right through carry
RCL	<drg>	Rotate left through carry
ASR	<drg>	Arithmetic shift right
ASL	<drg>	Arithmetic shift left
Multiply/Divide steps		
UMUL	<drg>,<srg>	Unsigned multiply step
SDIV	<drg>,<srg>	Unsigned division startup
UDIV	<drg>,<srg>	Unsigned division step
LDIV	<drg>,<srg>	Unsigned division last step

Table 4 List of PMS500 opcodes

VI. Function declarations

Appendix VII lists the global variables and definitions of the program. The source files `config.h` and `c.h` contain all other definitions.

add_definitions - collect every definition in the dfg and append to list.
```
static void add_definitions(DFG dg);
```

add_list - add x to l if not already included
```
static List add_list(Generic x, List l);
```

address - initialize q for addressing expression p+n
```
void address(Symbol q, Symbol p, int n);
```

alias_add - add node with pointers p and b to list l if not already on it
```
static Aliaslist alias_add(Aliaslist l, Pointer p, Pointer b);
```

alias_analysis - Establish what every pointer can point to at any point in dfg. Create separate entries for P and *P if *P also a pointer. Annotate dfg-nodes with list of live aliases
```
static void alias_analysis(DFG dg);
```

alias_free - append nodes of l to list of free nodes.
```
static void alias_free(Aliaslist l);
```

alias_member - return True if p in l, else return False
```
static Boolean alias_member(List l, Pointer p);
```

alias_merge - add copy of every node of s not in *t to *t. Return True if nodes were copied.
```
static Boolean alias_merge(Aliaslist s, Aliaslist *t);
```

alias_remove - remove node (p,x) with pointer p and any x from list l except when x in n.
```
static Aliaslist alias_remove(Aliaslist l, List n, Pointer p);
```

alias_trans - calculate effect of assignment to pointer n->x.def
```
static Aliaslist alias_trans(Aliaslist in, Node n);
```

aliases - calculate the list of symbols that might be accessed when p is dereferenced n times.
```
static List aliases(Aliaslist IN, Symbol p, int n);
```

asmcode - emit assembly language specified by asm
```
void asmcode(char *str, Symbol argv[]);
```

blockbeg - begin a compound statement
```
void blockbeg(Env *e);
```

blockend - end a compound statement
```
void blockend(Env *e);
```

calc_genset - calculate effect on KILL- and GEN sets by codenode p
```
static void calc_genset(DFG dg, Node p);
```

clear - clear bit n in bitset s
```
static void clear(Bitfield s, int n);
```

clear_globals - make sure linked lists attached to s are freed-called from function
```
static void clear_globals(Symbol s, Generic d);
```

daglist_append - append an item to the doubly-linked Daglist
```
static Daglist daglist_append(Node n, Daglist l);
```

defaddress - define an address. BEWARE: this function may be called in dataspace (defining a pointer) or in codespace (defining a branch table)!
```
void defaddress(Symbol p);
```

defbranch - update current dfg node with default and branch table labels BEWARE: this function is specific for the PMS-500 compiler! it was added to make the construction of the dfg from gencode possible, so the (global) codelist doesn't need to be walked (the global codelist was meant to be used by the front end only). Called directly after gen has processed the code for the switch statement.
```
void defbranch(Swcode *s);
```

defconst - define a constant
```
void defconst(int ty, Value v);
```

defstring - emit a string constant
```
void defstring(int len, char *s);
```

defsymbol - define a symbol: initialize p->x
```
void defsymbol(Symbol p);
```

depth_first - Depth-first traversal of the DFG, assigning depth-first numbers and detecting back edges.
```
static void depth_first(DFG dg);
```

emit - emit the dags on list p
```
void emit(Node p);
```

export - export a symbol
```
void export(Symbol p);
```

find_pointer - find the pointer-struct P for symbol p
```
static Pointer find_pointer(Symbol p, int lev);
```

function - generate code for a function codehead points to codegraph for this function. Offsets etc. are reset. First, dag nodes are annotated, ASGN nodes in particular, for data flow analysis. Next, registers are allocated (using dfa), and finally the assembly is written.
```
void function(Symbol f, Symbol caller[], Symbol callee[], int ncalls);
```

gen - annotate and linearize dags on list p; return pointer to new list
```
Node gen(Node p);
```

genl - annotate *p and append to head of list
```
static void genl(Node p, int lev);
```

global - emit code to define a global variable
```
void global(Symbol p);
```

import - import a symbol
```
void import(Symbol p);
```

iterate - propagate ud-information through the data flow graph. Return True if changes have been detected, false if not.
```
static Boolean iterate();
```

live_analysis - Live variable analysis
```
static void live_analysis(DFG dfg);
```

live_init - Calculate L_DEF and L_USE sets for live variable analysis
```
static void live_init(DFG dfg);
```

live_killdefs - Kill liveness of every direct definition of s reaching dg by resetting the corresponding bit in L_USE and setting the bit in L_DEF;
```
static void live_killdefs(DFG dg, Symbol s);
```

live_markuses - recursively walks the annotated DAG and marks reaching definitions of used symbols as live.
```
static void live_markuses(DFG dg, Usagelist u);
```

local - local variable
```
void local(Symbol p);
```

number - number nodes in list p
```
static void number(Node p);
```

prepend - prepend s to Symlist; return pointer to new list N.B. function 'append' in use by front end
```
static Usagelist prepend(Usagelist l, Usagelist r, Pointer p);
```

progbeg - beginning of program
```
void progbeg(int argc, char *argv[]);
```

progend - finalize program
```
void progend(void);
```

ralloc - perform register allocation
```
static void ralloc(void);
```

refdef - Collect reference, definition and usage information from forest
```
static void refdef(Node p);
```

segment - emit code to change segment
```
void segment(int x);
```

set - set bit n in bitfield s
```
static void set(Bitfield s, int n);
```

space - emit code to allocate x bytes
```
void space(int x);
```

stab functions to emit symbol table information

stabblock
```
void stabblock(int a,int b,Symbol *c);
```

stabend - finalize stab output
```
void stabend(Coordinate *cp, Symbol p, Coordinate **cpp, Symbol *sp, Symbol
*stab);
```

stabfend -
```
void stabfend(Symbol a,int b);
```

stabinit -
```
void stabinit(char *a, int b, char *c[]);
```

stabline - emit line number information for source coordinate *cp
```
void stabline(Coordinate *cp);
```

stabsym -
```
void stabsym(Symbol a);
```

stabtype -
```
void stabtype(Symbol a);
```

trans - collect definition, usage, and reference information from DAG-nodes (data transfer information)
```
static void trans(Node n);
```

trans1 - recursive extension of trans
```
static void trans1(Node n);
```

ud_chain - Calculate the usage-definition chains of the data flow graph
```
static void ud_chain(DFG dg);
```

unite - Perform a bitwise OR of a and b into a. Return True if a is changed, False if not.
```
static Boolean unite(Bitfield a, Bitfield b);
```

update_dfg - add basic blocks and data flow information from forest to data flow graph
```
static void update_dfg(Node p);
```

VII. Global variables and definitions

```
#define pms_500

#include<stdio.h>
#include<string.h>
#include"c.h"
#include"tools.h"

#define INT_BIT (sizeof(unsigned int)*CHAR_BIT)

/* IN(x) means pointer to In-set of dfg-node x. IN(x)[y] gives value of
   y-th unsigend integer in this set                                    */

#define IN(x)  (sets+(x)*setsize)
#define OUT(x) (sets+(ndfg+(x))*setsize)
#define GEN(x) (sets+(2*ndfg+(x))*setsize)
#define KILL(x) (sets+(3*ndfg+(x))*setsize)

#define L_IN(x) (live+(x)*setsize)
#define L_OUT(x) (live+(ndfg+(x))*setsize)
#define L_USE(x) (live+(2*ndfg+(x))*setsize)
#define L_DEF(x) (live+(3*ndfg+(x))*setsize)


int   offset, maxoffset;    /* Current stackoffset for defining locals, and
                               maxoffset                                  */
char  buf[BUFSIZ];          /* line buffer                                */
Node  *tail;                /* Aux. pointer to linearize DAG */
DFG   dfg, current,lastdfg; /* Data flow graph of current function and
                               current node and last created node (for
                               linkage) */
int   ndfg;                 /* Number of dfg nodes                        */
int   node_number=0;        /* Counter to number DAG nodes                */
Daglist  labls=NULL;        /* List of labels for current DFG node        */
List  defl=NULL;            /* List of all definitions in function        */
int   ndefl=0;              /* Total number of definitions                */
Node  *Def;                 /* Array of all definitions (List is converted
                               to array at some point in the program      */
Bitfield sets;              /* In, Out, Gen and Kill sets                 */
Bitfield live;              /* Same for live variable analysis            */
int   setsize;              /* Number of integers necessary to store In,
                               Out.. etc. sets */
int dfn;                    /* Auxiliary for depth-first numbering of the
                               dfg */
DFG   *dfa;                 /* Array of pointers to dfg-nodes in
                               depth-first order */
int slsize=0;               /* Size (bytes) of list of symbols */
Aliaslist free_aliasnodes=NULL; /* List of free aliasnodes */
List pointerlist=NULL;          /* List of existing pointer tuples */
Daglist free_dagnodes=NULL;     /* List of free dag nodes */
struct pntr unknown={(Symbol)-1,1,True}; /* The instantiation of the famous
                                            UNKNOWN symbol */
```

VIII. Index