Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1983

Implementing a scalar C compiler on the Cyber 205

Kuo-Cheng Li

Herb Schwetman

Report Number:
83-458

Li, Kuo-Cheng and Schwetman, Herb, "Implementing a scalar C compiler on the Cyber 205" (1983). *Department of Computer Science Technical Reports.* Paper 377.
https://docs.lib.purdue.edu/cstech/377

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

TR 458

Implementing a scalar C compiler on the Cyber 205†

Kuo-Cheng Li and Herb Schwetman

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

With the aid of the EM (Encoding Machine) Compiler Tool Kit, a preliminary version of a scalar C compiler was implemented on the Cyber 205 in a relatively short period of time. This C compiler emphasizes functionality more than efficiency. Several benchmark programs were used to measure the performance and to compare it with an equivalent C compiler for VAX/UNIX‡ system. In order to make it a production-quality C compiler, further enhancements will be necessary. This paper presents some motivating factors, implementation details, and proposes further work on developing the Cyber 205 C compiler.

KEY WORDS C Compiler tool kit Cyber 205 EM intermediate code

October 6, 1983

† This work was supported in part by the Purdue University Computing Center (PUCC).

‡ UNIX is a Trademark of Bell Laboratories.

Implementing a scalar C compiler on the Cyber 205

Kuo-Cheng Li and Herb Schwetman

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

1. Introduction

A supercomputer, the Cyber 200 Model 205 (hereafter the Cyber 205), was installed at Purdue University in the spring of 1983. It is the fastest computer in the world; only the CRAY-1 [Russ78] series [Buch83] is a competitor. However, as pointed out by Dijkstra, in his Turing award lecture [Dijk72], "*..as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem....*"; In other words, advances in hardware only create problems for software. Hardware is already fast, but how to take advantage of advanced hardware is the responsibility and challenge of software.

The "new" Cyber 205, although derived from the CDC STAR 100 [HiTa72] and therefore retaining some of its software, has only a limited amount of software which exploits its hardware capabilities. In the area of programming languages, there is only one high level language - (vector) FORTRAN. This FORTRAN does not satisfy the demands of all groups of users, e.g., it is awkward when dealing with character strings, complex program flow, recursion, etc., and also it provides no aggregate data structures. There are several languages available in today's computing world which have a structured syntax and hence can improve a user's productivity and increase a user's satisfaction; these languages are also reasonably efficient, allowing good utilization of the underlining powerful hardware.

There are at least three approaches to the design of high level languages for vector (or parallel) computers:

1. *Vectorization:* an automatic optimizer (e.g., [KKPL81]) which can detect inherent parallelism in a sequential program and generate code for the vector computer,
2. *Explicit, vector-oriented syntax added to an existing language:* syntactic and semantic enhancements to an existing (scalar) language which allow users to directly specify vector data types and vector operations, or
3. *New vector (or parallel) programming language:* design a new programming language tailored to vector (or parallel) processing; implement a compiler for this language.

We elected to pursue the second alternative for several reasons:

1. Various surveys [Weth80] [PeSt81a] [PeSt81b] have shown that vectorization alone is usually less satisfactory and less efficient for designing and constructing programs; many users of vector or parallel computers prefer languages with vector or parallel syntax,
2. The vector FORTRAN compiler provided by CDC already has an elaborate vectorizer, so programmers wishing to use this approach have a means of doing so,
3. We feel the need for a structured language with vector constructs will be an essential tool to support research into vector algorithms,
4. By extending an existing (familiar) language, users will not need to make an extensive investment in learning a new language; they can build on existing knowledge, and
5. By starting from an existing compiler, a reliable compiler for the extended language could be constructed more economically (than a brand new compiler).

The programming language C [KeRi78] has become popular as a system implementation and application language. For many of the reasons cited above we decided to implement C with *vector extensions* for the Cyber 205.

Before extending the language C, it was essential to have the existing (scalar) C implemented on the Cyber 205, so that *upward compatibility* could be achieved. With the aid of the Amsterdam EM (Encoding Machine) Compiler Kit [TSKS81], a scalar C compiler was implemented on the Cyber 205. The current version of this C compiler emphasizes *functionality* more than efficiency. Better performance can be achieved by later refinements, as is demonstrated in this paper.

This paper describes the implementation of this preliminary version of CC205 (C compiler for the Cyber 205). In the following sections, the EM Tool Kit, the Cyber 205 system, and the programming language C are briefly described. Then, the implementation, problems and performance issues are presented and discussed.

2. The EM Compiler Kit

2.1. Rationale

The Encoding Machine (EM) compiler kit is designed and implemented by Tanenbaum et al. at the Vrije University, Amsterdam, The Netherlands [TaSS80]. This Tool Kit is used mainly for assisting, in the sense of saving programming effort, the construction of (cross) compilers for different programming languages on different machines. The approach employed is based on the UNCOL (UNiveral Computer Oriented Language) approach [Stee60] with some restrictions.

The idea of UNCOL is that, for N languages and M machines, it should be necessary to build only N+M language processors, instead of N*M processors. In other words, only N Front-Ends and M Back-Ends are needed (see Figure 1). In Figure 1, the Front-Ends accept source languages and generate intermediate code which is input to the Back-End, and then target object programs are generated. However, this idea is probably too ambitious, so some restrictions are necessary. The restriction of the EM compiler kit is that only *algebraic languages* for *byte-addressable machines* are considered [TSKS81].

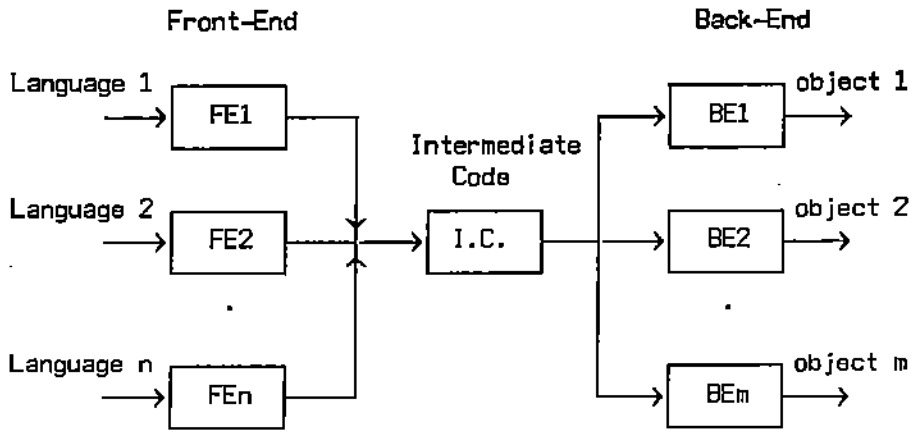

Figure 1. UNCOL diagram (n Languages and m machines)

2.2. Basic building blocks

Figure 2 shows the basic building blocks of the EM Tool Kit which is used in our implementation of CC205. As seen in Figure 2, the compilation proceeds as a series of translations, as follows:

- the source program is translated into compact EM assembly code by the Front-End,
- the EM assembly code is (optionally) translated by the peephole OPTimizer to optimized EM code,
- the EM code is translated by the Back-End into the assembly language program of the target machine (called META on the Cyber 205),
- the assembly language program is assembled into object modules by the Cyber 205 assembler, and
- the object code, plus entries from the C library, are made into executable modules by the Cyber 205 loader.

The Front-End is a modified version of the PCC (Portable C Compiler) [John78] [John80], in which the machine dependent part of PCC has been modified to accommodate the EM architecture and generate EM intermediate code. The EM intermediate code is similar to

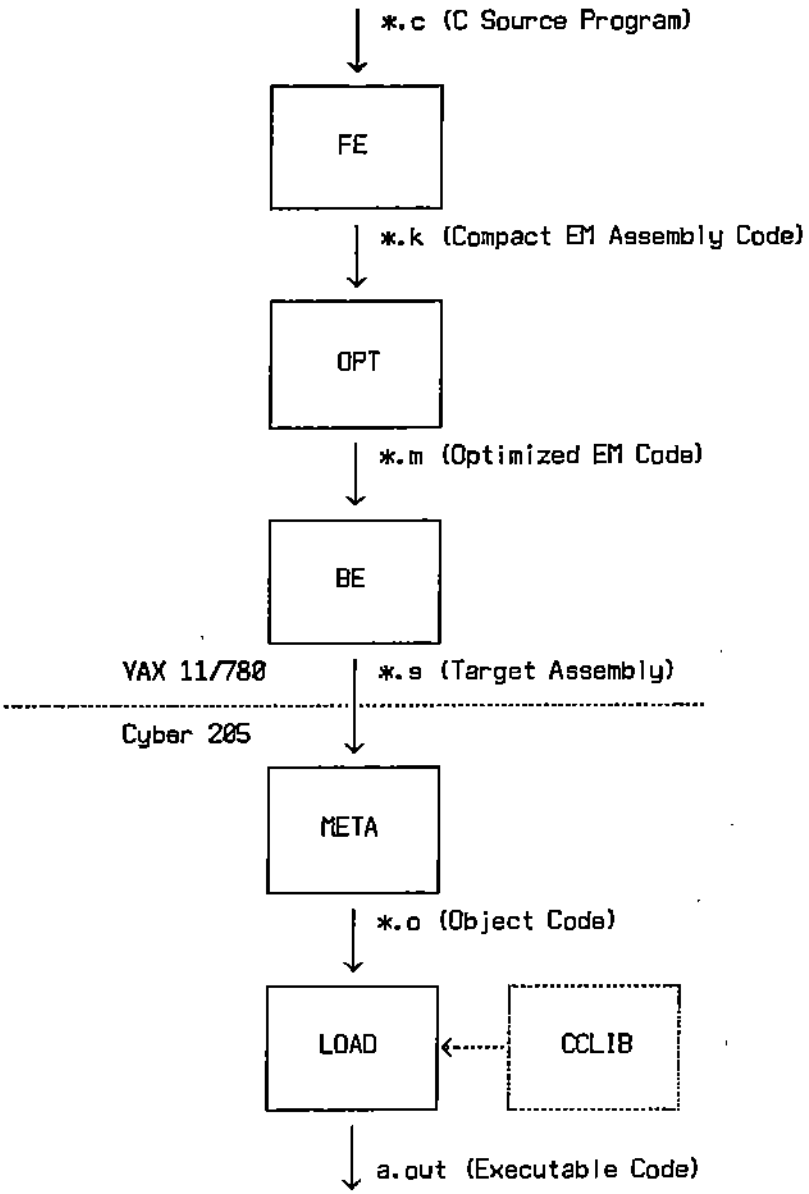

Figure 2. Basic building blocks of EM tool kit

P-code [NAJN75], but more general, since it is intended for a wider class of languages (not just Pascal).

Another advantage of the UNCOL approach is that only one intermediate code optimizer is necessary for all languages and machines. The optimizer in the Tool Kit is a peephole optimizer which scans the instructions in a window of a few instructions, replacing specified

code sequences by a semantically equivalent but (hopefully) more efficient ones [TaSS82]. The task of the Back-End is to convert the EM code into the object code (assembly language) of the target machine; an instruction mapping table from EM code to the target object code is the basis of this process.

2.3. Machine architecture

The Encoding Machine [Tane78] [TaSS80] [TaSS82] is a simple stack machine; its instruction set is stack-oriented (i.e. *reverse Polish* type). Fundamental operations include pushing a variable or constant onto the stack, poping the top of stack, and performing arithmetic operations on the top two stack items. Branching depends on the top one or two stack items, and the instructions for calling and returning from procedures, etc. also use the stack. These stack-oriented instructions were chosen to match closely the semantic primitives of common algebraic languages. Furthermore, the instruction set is designed to incorporate a highly compact encoding scheme, this encoding scheme is based on results of extensive empirical investigations [Tane78].

The Encoding Machine has no general purpose registers; it does have a few special purpose registers such as the Local Base register (LB, equivalent to the frame pointer *(fp)* in the VAX) and the Stack Pointer (SP). General purpose registers are normally used by compiled code to (1) hold intermediate results while computing complicated arithmetic or logical expressions and (2) hold frequently used local variables. However, several studies [Knut71][Tane78] have shown that a typical arithmetic expression is not complicated and, in fact, references, on the average, fewer than two operands. In addition, recent emphasis on structured programming (which advocates the use of many small procedures) means that the use of registers to hold local variables could require significant overhead in the prologues and epilogues of procedure calls. Thus, the need for general purpose registers is probably not as great as in earlier situations. Furthermore, reverse Polish code is much easier to generate than multi-register machine code, especially if highly efficient code is desired. If necessary,

part of the top of stack can be kept in a high speed memory (e.g., a cache) to achieve acceptable performance [Orga73].

2.4. The impact on our project

The UNCOL approach dictates that the Tool Kit be highly modular. Writing one Front-End for a new language is normally sufficient for implementing that language on several machines; similarly, writing a Back-End for one machine is normally sufficient for having it be used by several other languages on that machine. However, this did not benefit us, since our intention was to implement one language on one machine. Yet, using this Tool Kit did save us a lot of time and effort, when compared to developing a compiler "from scratch".

The possible problems we imagined at the beginning of this project were:

1. *Incompatibility between the EM and the target machine:* some EM codes may be mapped onto expensive target codes, and some target codes may not be mapped at all. As an example, the scalar instructions of the Cyber 205 are register-oriented, which are very different from the stack-oriented EM instructions, and
2. *Inefficiency due to the introduction of intermediate code:* the compilation process is slowed down because the generation of object codes is indirect, i.e. involves two distinct steps (EM code and assembly language).

3. The Cyber 205 system

3.1. General history

The Cyber 205 central computer, one of the few existing commercially available supercomputers, is a large-scale, high-speed computing system. It was first announced in 1980 and installed at Purdue University in the spring of 1983.

The Cyber 205 is a descendent of the CDC STAR 100 (1965 - 1975) which had several problem areas [HoJe81] including, slow main memory (1200 ns access time), slow scalar arithmetic (four times slower than CDC 7600 and IBM 360/195), long vector start-up times (3 - 7 μs), and unit vector increment (vector elements had to be in contiguous locations). The Cyber 203 in 1979 overcame the first two problems of the STAR 100 by utilizing LSI circuits in the CPU and bipolar memory, to obtain a 20 ns clock period (also called minor cycle) and 80 ns memory access time. In 1980, the Cyber 203 was re-engineered to overcome part of the third disadvantage, and its advanced features, such as stream processing, virtual addressing, and hardware macroinstructions were improved. This improved machine is designated the Cyber 205. However, because of the inherent properties of the machine architecture, the vector elements still need to be in consecutive memory locations. Nontheless, noncontiguous vector elements can be processed using the scatter/gather and compress/merge instructions.

The Cyber 205 is considered to be the fastest machine in the world, because it has a maximum processing rate of 400 MFLOPS *(Million FLoating-point Operations Per Second)* for 64-bit operations or 800 MFLOPS for 32-bit operations; these asymptotic rates are based on processing vector data using four arithmetic pipelines and linked triadic operations.

3.2. Machine architecture

Figure 3 is a simple block diagram of the Cyber 205. It is composed of the CPU, Central Memory (CM), and the Maintenance Control Unit (MCU), etc. The functional characteristics of the Cyber 205 are briefly described below; a more detailed description of each functional block can be found in [CDC81a]. The CPU comprises a scalar processor, a vector processor and several I/O ports. Since the current version of the C compiler uses only the scalar instructions of the Cyber 205, only the scalar processor will be described.

The Cyber 205 has a high speed *register file* with 256 full-word (64 bits) registers, which is used for holding operands and results for scalar instructions, and for instruction and operand addressing, indexing, and storing constants and field length counts, etc. There is a 64

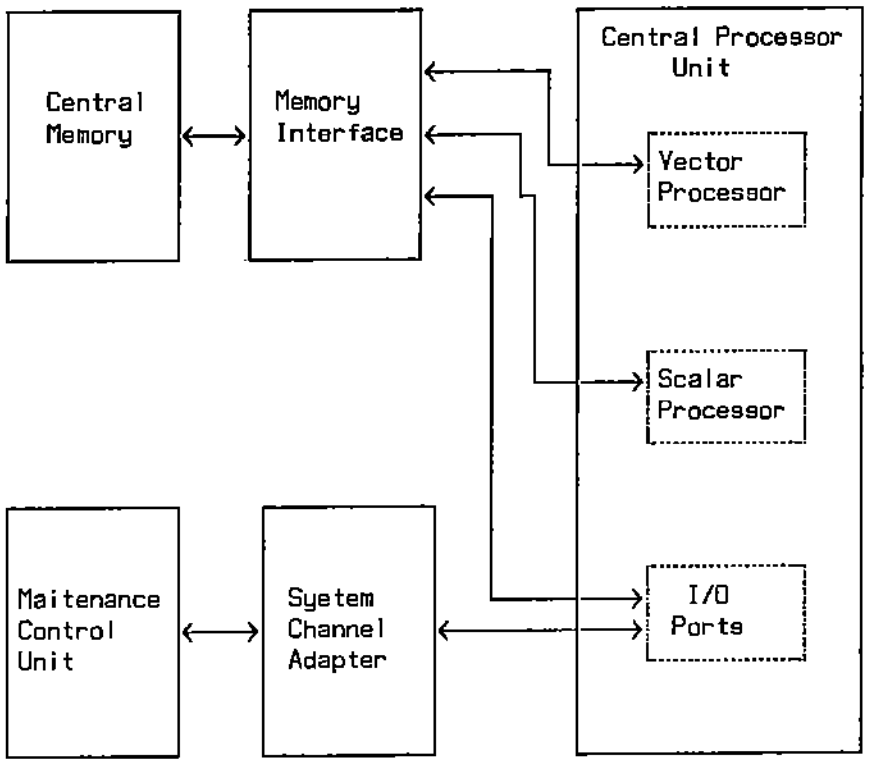

Figure 3. Cyber 205 Block Diagram

full-word semiconductor instruction stack for the optimization of programmed scalar loop iteration.

The *scalar processor* (see Figure 4) features five independent arithmetic functional units which are pipelined so as to accept new operands every clock cycle. An exception is the Divide/Sqrt/Convert unit for which a new operation can not be issued until the completion of the previous operation; this is the only portion of floating point units which is not pipelined. The Cyber 205 hardware supports floating point, integer, byte, and bit data types, where floating point and integer variables can be either full-word or half-word in length. A full-word integer is 48 bits, which is actually a floating point variable with zero exponent.

The *Load/Store unit* contains six address registers. A load instruction requires one address register, while a store requires two address registers. Hence, the Load/Store unit is capable of streaming load/store instructions at one load per minor cycle and one store per two

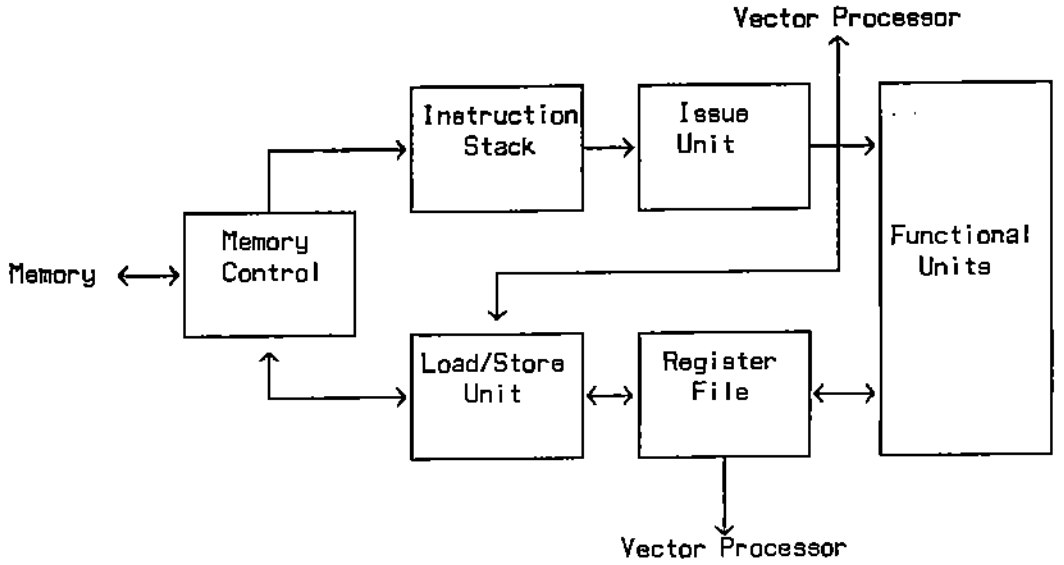

Figure 4. Scalar Processor Block Diagram

minor cycles, except for the STOC instruction - store character, which is one per 14 minor cycles. Furthermore, the Load/Store unit can load a word from central memory into the register file in 15 minor cycles (300 ns), and store a word into memory in 10 minor cycles. Therefore, a sequence of N loads can be executed in (N+14) cycles, whereas a stream of N stores can be executed in 2N+8 cycles. These times assume no memory conflicts or register-busy conditions. Five minor cycles may be required to release a register, and a minimum of four minor cycles to resolve a memory conflict.

The scalar process also has a *short-stop* feature which is the mechanism by which a result from any arithmetic unit may be used directly as the input to any arithmetic unit. This occurs in parallel with the storing of the result into the register file (see the timing analysis in Section 6.2.2). The time saved by short-stop (avoiding the store of the result into the register file and the retrieve of it back to the next arithmetic unit) is three minor cycles. Utilizing this benefit is not trivial as the timing constraints are critical. For example, instruction A, issuing at time T, generates a result which is to be used by instruction B; instruction B must be issued no later than $T+SS$, where SS is the short-stop time; if missed, instruction B can only access the

result from the register file.

The Cyber 205 has a *virtual memory mechanism* whose page table is the combination of 16 associative registers and a space table which resides in a restricted area of central memory. The virtual memory mechanism performs high speed address mappings from the logically contiguous addresses to the physically noncontiguous storage system. The virtual address space is 4 trillion words (addressed by 48 bits); half of the address space is available for each user; the other half is reserved for system use. Memory can be addressed in full-word, half-word (32 bits), byte (8 bits) and bit units. In the Purdue configuration, the size of physical memory is one million 64-bit words, which could be extended to two or four million words; the page sizes are 2048 words (small pages) and 65536 words (large pages).

From a user's point of view, many operations in the Cyber 205 are performed in a serial fashion, while other operations are performed in a semi-parallel (pipelined) mode. In fact, all operations are issued in strict sequence from the Instruction Issue Unit and form a single instruction stream. Also, since the scalar/vector processors operate on multiple data (due to the multiple functional units and the pipelined segments in each functional unit), this machine can be classified as a SIMD *(Single Instruction stream, Multiple Data stream)* machine [Flyn66].

The Cyber 205 is designed primarily for processing tasks which require intensive computations and/or large amounts of main memory. Some I/O operations and many other support functions are left to Front-End computers (e.g., CDC 6000's, or DEC VAX's) via the I/O ports and the Loosely Couple Network (LCN). Each I/O port is capable of 200 megabits per second maximum transfer rate; the LCN can sustain a transfer rate of up to 50 megabits per second (a peak rate which is rarely achieved). This functional hardware concept of distributive processing is the cornerstone of the Cyber 205 system architecture.

3.3. Software

The software of the Cyber 205 includes:

- a multiprogramming operating system - Cyber 200 VSOS (Virtual Storage Operating System) [CDC81b],
- programming languages such as FORTRAN, IMPL (a system implementation language similar to Fortran) and META (an assembler), and
- other utility programs such as LOAD (a link loader), OLE (Object Library Editor which is akin to *ar* in UNIX), etc.

META, the assembler for the Cyber 205 [CDC81c], generates relocatable binary output which is then linked and loaded by LOAD. META provides a conditional assembly capability, a procedure and function definition capability, a set capability to define, reference, and extend the list of expressions, and attribute assignment for symbols, etc. The mapping table in the EM Back-End is designed to map EM instructions into META assembly code.

4. The C programming language

The language C [KeRi78] was created by Dennis Ritchie and developed at Bell Laboratories at Murray Hill, New Jersey, in 1972. It was used in rewriting the assembly language version of the UNIX operating system on the DEC PDP-11 (except for a few very low level routines) so that transporting UNIX to another computer became mainly a matter of writing a C compiler for the target machine [Mill78] [MiTa82]. In spite of its intimate relationship with UNIX, C has earned a reputation as a good systems programming language and has even been called a *high level assembly language*. It is also a powerful application programming language [FeGe82]; e.g., it has been used in movie production (the computer graphics animation in the "Star Trek II" and "Return of the Jedi" were written in C [Robe83]).

Another important characteristic of C is its high degree of *portability*. This is due to fact that its data types and control structures are supported directly by most existing computers,

and the low level machine-dependent (e.g., I/O) issues can be resolved by use of run-time library functions. The Portable C Compiler (PCC) [John80] has made C movable to many other main frames with different idiosyncrasies, such as the IBM system/370, the Honeywell 6000, the Interdata 8/32, the VAX 11/780, and many microcomputers [BYTE83].

C has most features common to high level programming languages, features such as structured flow-control, recursion, fundamental data types (with structures, and unions), as well as several unique features, such as bitwise logical operations, increment and decrement operators, pointer arithmetic, static variables, register variables, fields, casts, etc. Its fundamental data objects are floating point numbers, integers of different sizes, and characters, and it has derived data types created by using pointers, arrays, structures, unions, and functions.

Efficient manipulation of bits is vital to systems programming. C has this capability which exists in only a few other high level programming languages. Furthermore, most hardware instructions deal with machine addresses directly, and C has pointers (which correspond to machine address) and the capability of doing pointer arithmetic. By virtue of these capabilities, C is capable of generating efficient code for critical segments and for constructing and manipulating efficient data structures.

Argument passing in C is *call-by-value; call-by-reference* can be achieved by passing the pointers (or the addresses) of data items. C is a typed, but not a strongly-typed, language; this will be helpful later when implementing the proposed vector extensions [PeCM83].

5. The design and implementation of CC205

5.1. Planning

The task of implementing a (scalar) C compiler (designated CC205) on the Cyber 205 using the EM Tool Kit was broken into five major stages:

1. *Install the EM Tool Kit on our VAX/UNIX system:* The EM Tool Kit, written in the V7 C language, was developed on the PDP-11/70 under the Version 7 UNIX operating system. At our site the C compiler executes on a VAX-11/780 running the Berkery 4.1bsd UNIX. This phase of the project required us to become acquainted with the EM Tool Kit (the EM package is, unfortunately, not well documented) and to tackle some of the machine dependent problems.

2. *Design and build the Back-End instruction mapping table:* This phase of work required knowledge of the Cyber 205 architecture and its META assembly language. As mentioned before, EM is a stack machine and may assume that the target machine has a *hardware stack*. In contrast, the Cyber 205 is register-oriented machine, and it does not have a hardware stack mechanism (i.e., hardware instructions for automatic manipulation of the stack pointer); therefore, a *software stack* is necessary.

3. *Design a C start-up routine and install the C run-time library:* A start-up routine was needed for interfacing the run-time C program and the Cyber 205 system. The C run-time library is a set of modules which is divided into three sub-libraries:

   gen - general functions, e.g., 'malloc', a memory allocation function,

   stdio - standard I/O functions, e.g., 'doprnt', a printing formator, and

   sys - system functions, e.g., 'read', 'write', 'creat', 'open', and 'close' etc., low level I/O functions.

   Since most of the library routines are already implemented in C and, also because of the high portability of C, the main part of this phase consisted mainly of developing the low level system functions.

4. *Bootstrap CC205 to the Cyber 205:* Steps 1, 2 and 3 led to a compiler which executed on the VAX/UNIX system and produced META code, which could then be uploaded to the Cyber 205 for assembly, loading and execution. The goal of the bootstrap phase of the project was to move the compiler itself onto the Cyber 205. Some machine dependent

and portability problems were expected. (Also, this was a good test of the compiler).

5. *Optimization:* A straightforward mapping of instructions resulted in a great deal of redundant code; further optimizations were needed.

5.2. The address space and C run-time stack

The Cyber 205 has a dynamic space, really a virtual space, lying between the code-data sections and the public library [CDC81b]; this could be used as the run-time stack required by procedure calls. A procedure call is expensive for a register-oriented machine. It may be even more expensive on the Cyber 205 because it may involve a vector instruction (*swap*) to swap the caller's environment registers and working registers (see Figure 5 for the register file) at its prologue; similar costs are incurred on procedure exit (in the epilogue), and the start-up time of the vector instruction 'swap' is nontrivial (either 28 or 56 minor cycles).

However, because the EM Tool Kit does not use any general registers, the procedure-call mechanism in our current C Compiler is different from the conventional one used in the Cyber 205. On procedure entry the prologue performs the following four actions (see Figure 6):

1. Save the return address,
2. Save the local base (LB),
3. Update LB, and
4. Allocate the space for local variables,

and the epilogue does the reverse actions, namely, restore LB and return to the caller. These actions mean that we must have a C run-time stack in the address space.

Figure 7 shows the block diagram of the address space. The Cyber 205 allows us to select the interleaved code-data format or the separated code-data format; the former format is the default. As shown in Figure 7, the C run-time stack is positioned in the area between the code-data segments and the VSOS dynamic stack. The stack pointer (SP) is initialized at

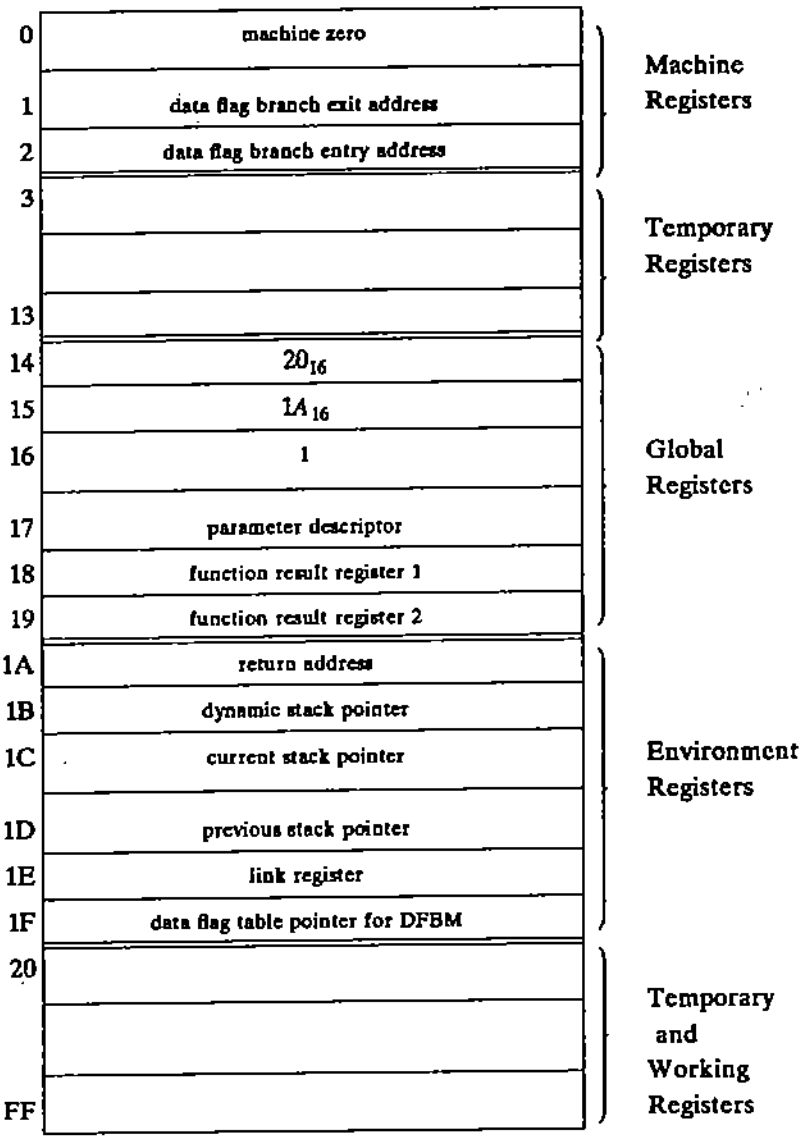

Figure 5. Register File

some proper location and grows upward, i.e., when pushing a data onto the stack, SP is decreased first, then the data is stored. Just below the C run-time stack is the VSOS dynamic stack which grows downward. It is used for the conventional procedure calls as mentioned above (the low level system functions in *sys* need to use conventional calls). A heap, growing upward against the VSOS dynamic stack, is used by the C run time storage allocation.

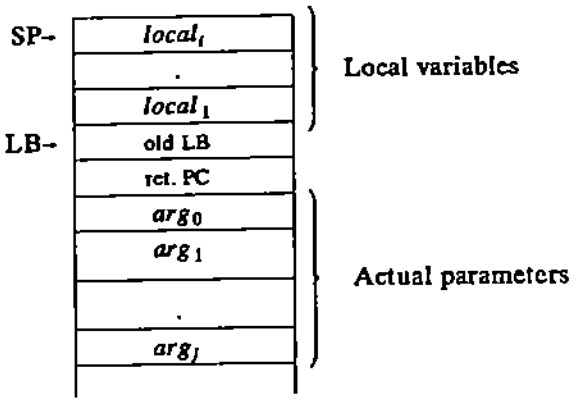

Figure 6. Activation Record for a Procedure Call

5.3. The instruction mapping from EM to META

5.3.1. General

The EM assembly language [TaSS80] has 12 pseudo instructions for storage declaration, and procedure indication, etc., and 132 machine instructions including loads, stores, arithmetic operations, comparisons, branches, and procedure calls, etc. Not all EM instructions are used in the C language conventions, hence the unused EM instructions are not generated by the C compiler (note that EM is designed for many languages, not simply for C).

The Cyber 205 has 214 instructions [CDC81a], including vector, vector macro, and monitor instructions, etc., and some directives [CDC81c] which are assembler mnemonics. For the scalar C compiler, only a few instructions were used.

5.3.2. The size specifications of data types and potential portability problems

The sizes of fundamental data types are parameterized in the EM Tool Kit. Table 1 shows the specifications employed in our first version of CC205. With these specifications, several potential portability problems (for existing C programs) may be expected, since in our current environment (VAX/UNIX C), a C program has the specifications shown in Table 2.

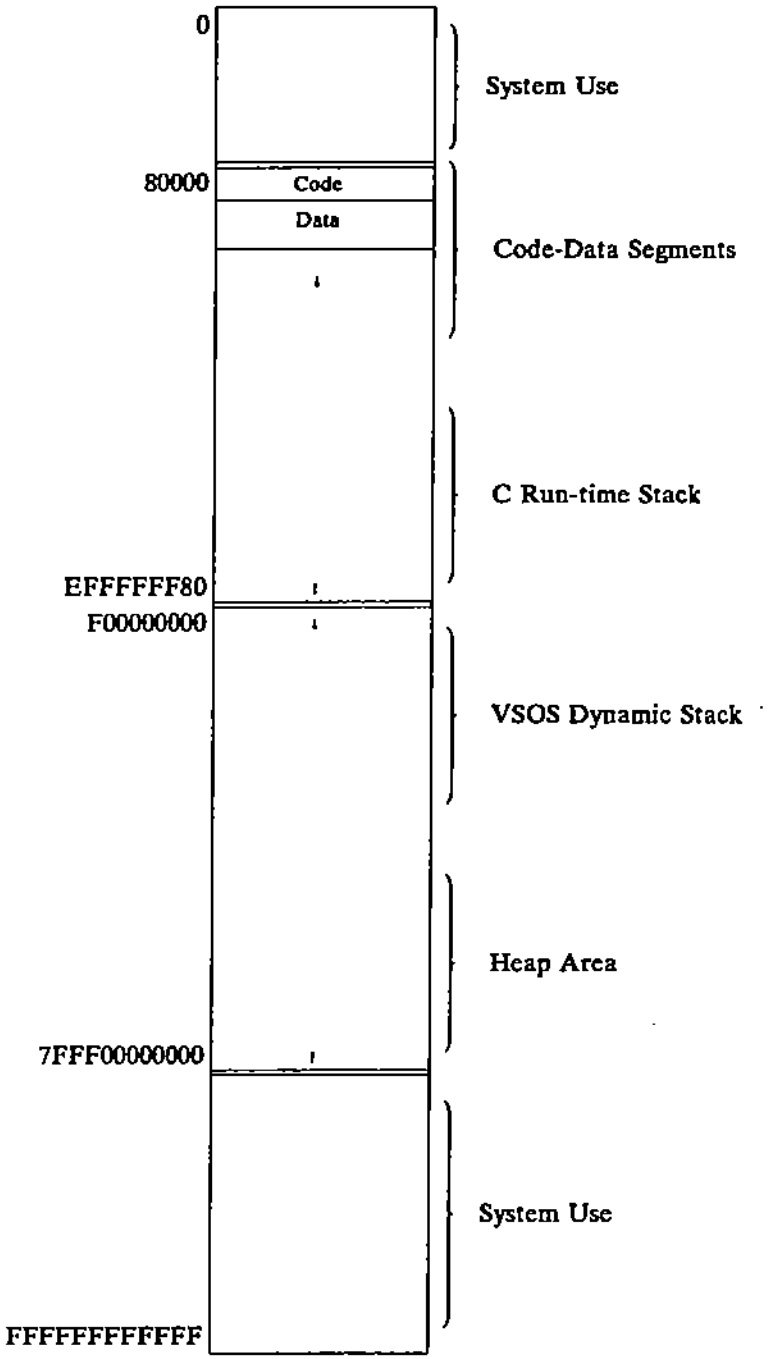

Figure 7. Address Space

Some of these possible portability problems include:

1. *size-dependent problems:* As shown in Table 1, 'short' is 64 bits, not 16 bits as defined in Table 2, this may cause data objects to be misinterpreted. For example, a memory word

Size	Number of bits
word	64
pointer	64
long	64
int	64
short	64
char	8
float	64
double	64

Table 1: Version 0 size specifications for CC205.

Size	Number of bits
word	32
pointer	32
long	32
int	32
short	16
char	8
float	32
double	64

Table 2: Size specifications for VAX C.

containing 0xFFFF in hex (65535 in decimal) is interpreted as -1 in the 16-bit short mode (i.e., the current VAX C environment) but as 65535 in 64-bit short mode.

2. *machine-dependent problems:* The full-word integer in the Cyber 205 is 48 bits, (not 64 bits as given in the specification); this forms a "hole" (the left most 16 bits of a word) in memory, which is not expected by the EM and the other C programs. The bit position is increased from lsb (least significant bit) to msb (most significant bit) in EM, while in the Cyber 205 it is in reverse order.

The first problem can be solved by modifying the programs to do *sign-extension* explicitly, and the second problem may be significant only when the existing C programs doing bitwise

operations, such as bitwise AND, OR, SHIFT, etc. This could be solved by modifying the bitwise operations to fit the new specification. For developing a new program, these should not be problems, since users should know the sizes of the data types.

5.3.3. Other characteristics

Characters in the Cyber 205 are, fortunately, represented in ASCII code, (which is significantly different from the traditional CDC series of computers in which a character is represented in 6-bit 'display code', with only the upper-case characters, plus digits, punctuation marks and operators). This makes C programs running on the Cyber 205 more conventional (i.e. not forced to use upper-case only). Also bootstrapping the compiler is more straightforward; however this incompatibility with the older CDC systems complicates the procedure of building CC205, because the current Front-End computer is a CDC 6600 and all files have to be stored in binary form on the 6600 before transferring them over to the Cyber 205.

Memory addressing on the Cyber 205 is in bit units, which is different from byte-oriented machines such as the EM and the VAX. Also the hardware of Cyber 205 does not provide for indirect addressing; this can be implemented using additional load (LOD) instructions.

5.3.4. Global definition, instruction mapping table and instruction macros

The *global definition table*, used as an included file for the CC205 generated META programs, symbolically defines the global registers (e.g., stack registers: SP and LB, constant registers, etc.) and the scratch registers. The *instruction mapping table* defines blocks of in-line code without labels, and other blocks are defined elsewhere as *instruction macros* which are sets of META procedures containing labels.

Because a *software* stack is employed, each EM instruction is, most of the time, mapped to at least three META instructions, e.g., an EM instruction *LOC c* (load constant *c* onto stack) is mapped as follows:

```
es      t1,c      * enter register t1 with constant c
is      sp,-64    * decrease stack pointer by one word (64 bits)
lod     sp,t1     * push (t1) onto the top-of-stack
```

5.4. The C start-up routine and low level system functions

The C start-up routine, used for entering the C run-time environment, performs several tasks, including:

1. Initializing the constant registers, the C run-time stack pointer, and the VSOS dynamic stack pointers,
2. Setting up the command-line arguments *(argc,argv)* and handling redirection of I/O files,
3. Opening the standard input (stdin), output (stdout), and error (stderr) files, and initializing the file descriptor table which is a table containing the Cyber 205's file logical unit numbers *(flun)* indexed by C's file descriptors *(fd)*, and
4. Jumping to the *main* routine.

At program termination, the routine *exit* is called to "flush out" the I/O buffers; control is then returned to VSOS.

The System Interface Language (SIL) [CDC81b] of the Cyber 205 is a set of subroutine calls which allow a task to exchange information with the operating system and to perform file I/O operations. The low level system functions of C are all implemented in META using SIL calls. The assembled low level functions with the rest of the (compiled and assembled) C run-time library are built into an object library (CCLIB) using OLE (Object Library Editor).

At the completion of this phase, we had a cross compiler available at the VAX site, i.e., C programs can be compiled on the VAX machine, and then the generated META programs can be uploaded to the Cyber 205 for assembly, loading, and execution.

5.5. Bootstrapping CC205

The process of bootstrapping [AhUl77] is shown in Figure 8, where the notation $T_S^{I\ O}$ means a translator T, written in language S, translates language I into language O. In the first stage, $CC205_C^{C\ M}$ (written in C, accepts C code and translates it into META code) goes through $CC_V^{C\ V}$ on VAX (CC in VAX machine code accepts C code and produces VAX machine code) produces a cross compiler ($CC205_V^{C\ M}$). Then, in the second stage, as indicated by the dotted line, $CC205_C^{C\ M}$ goes through the cross compiler $CC205_V^{C\ M}$ and produces $CC205_M^{C\ M}$. That is, the whole EM Tool Kit is processed as input to the cross compiler, a META version of CC205 is generated, and then it is assembled and link-loaded as a controllee file [CDC81b] (an executable file) on the Cyber 205. This completes the bootstrapping process, and a scalar C compiler is available on the Cyber 205 together with the C run-time library (CCLIB) and other libraries (e.g., LIBM - math library) and utilities (e.g., EXPAND - expands tabs into spaces).

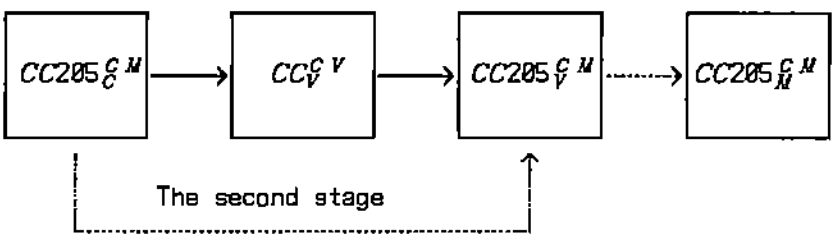

Figure 8. Bootstrapping Process

6. Performance Issues

6.1. Inherent inefficiency and possible cures

The preliminary version of CC205 did not take full advantage of the register file; in fact only a small portion of the register file was used. Furthermore, as mentioned in Section 2.4, the possible incompatibility and inefficiency (in compilation time) of using the EM Tool Kit, makes it difficult to compete with the hand crafted Cyber 205 Fortran compiler in terms of running time of compiled code. To improve the performance of CC205, several enhancements were considered; these possible enhancements included:

1. *Remove redundant code:*

   Because of the instruction mapping, the object program generated by CC205 contains many redundant instructions. For example, a value (in a register) is pushed (stored) onto the stack in one EM instruction, and the next EM instruction pops (loads) it out to the same register. This is a very common phenomenon in the current scheme of code-mapping, and, unfortunately, load/store takes much more time to execute than the other scalar instructions.

2. *Keep top portion of stack in the register file:*

   Keeping the top portion of the stack in the register file could improve performance significantly. However, META (really the 205 instruction set) does not allow dynamic addressing of registers (i.e., a register cannot point to another register); it only allows static addressing of registers (from assembled instructions). To "build" instructions to simulate dynamic addressing capability for registers might be possible, but it would probably be expensive. Also, this might incur considerable overhead in the Back-End, to implement a mechanism which manages the portion of the stack efficiently so that when the stack in the register file is full, part of the stack would be swapped into memory; also when referencing a variable, we need to know whether it is in the register file or in

the memory.

Another plausible approach is as follows: when a procedure call occurs, swap the current stack in the register file into memory, then do the 'prologue' operations and, at the end of the procedure, do the reverse operations. In this case, the top portion of the stack *(evaluation stack)* in a procedure is always kept in the register file, and the improvement over the current method should be significant. The problem here is that the growth of the stack in the register file is unpredictable. More precisely, keeping track of the size of the stack at assembly time or in the Back-End is sometimes impossible, because for some EM instructions (e.g., LOS, STS, BLS, ASS ,and DUS), the growth or shrinkage of the stack is runtime dependent. (According to the statistics gathered from 104 CCLIB function modules and the entire EM Tool Kit, these five EM instructions were never generated; also, the data in Table 6 on the maximum depth of the register stack shows that excessive register growth is probably unlikely).

3. *Maintain all scalar global variables in the register file:*

In order to further improve performance, a possible approach is to have all global variables (scalar variables and pointers to arrays or strings) reside in the register file. In other words, the preferred programming style would be changed so as to favor having many variables as global variables. This approach has some limitations: first, due to the limited capacity of the register file, only the first, say 200, global variables can be located in the register file; second, all source files must include a common set of global variable definitions. However, considering the potential improved performance, this may be worthwhile.

4. *Force local scalar variables to the register file:*

This approach, though it suffers from the argument that notable overhead may be incurred during procedure calls [Haik82][BaBK76], may be able to achieve good performance most of the time. This approach may require significant modifications of the EM

Tool Kit.

6.2. Optimization

6.2.1. A peephole optimizer

A META peephole optimizer, implemented as a post-processor to the Back-End, was added [Ande83]; this optimizer removed some of the redundant META instructions (see Approach 1 in Section 6.1). Basically, the optimizer performs pattern matching within a basic block, where a basic block is sequence of instructions delimited by a label, the end-of-procedure, or the end-of-file. A window of instructions is matched against target patterns, where the window size is defined as the length of the longest target pattern. Pattern matching is done by several finite automata which perform instruction-matching and then operand-matching. Matched instructions are replaced with a replacement pattern and the whole process is repeated until no more instructions are matched.

The current META optimizer removes small parts of redundant code; more redundant code could be removed by adding more target patterns. Pattern matching does incur heavy overhead for the whole compilation process, (see the compile time data in Table 7).

6.2.2. Back-End Optimizer

Based on Approach 2 in Section 6.1, the Back-End has been extended to have a register stack for the evaluation of expressions, i.e. the *evaluation stack* resides in the register file, rather than the normal run-time stack in memory. (We define this version as Version 0.1 of CC205, and the previous version with straightforward code mapping as Version 0.0). That is, instead of mapping instructions in the straightforward manner (i.e., all intermediate results stored on the memory stack), registers are used to hold the intermediate results within an expression. This results in a significant reduction in the number of *load* and *store* instructions (note that loads and stores are expensive instructions in scalar mode). As an example,

consider the statement $a = b*c + 3$, where c is an argument, and a and b are local variables. Table 3 shows the code generated by EM, Version 0.0 CC205 ($CC205_0$) and Version 0.1 CC205 ($CC205_1$). In Table 3, t0, t1, t2, and t3 are scratch registers and c_1n, c_2, c_2n, and c_3 are constant registers. As shown in the table, six EM instructions were mapped into 25 META instructions by Version 0.0 CC205, whereas only five META instructions were generated by Version 0.1 CC205.

EM	$CC205_0$	$CC205_1$	*Comments*
LOL -16	es t1,-2 lod [lb,t1],t2 is sp,-64 sto sp,t2	lod [lb,c_2n],t0	load b
LOL 0	es t1,2 lod [lb,t1],t2 is sp,-64 sto sp,t2	lod [lb,c_2],t1	load c
MLI 8	lod sp,t1 is sp,64 lod sp,t2 mpyx t2,t1,t3 sto sp,t3	mpyx t0,t1,t2	t2 = b*c
LOC 3	es t1,3 is sp,-64 sto sp,t1		load constant 3
ADI 8	lod sp,t1 is sp,64 lod sp,t2 addx t2,t1,t3 sto sp,t3	addx t2,c_3,t3	t3 = t2 + 3
STL -8	lod sp,t2 is sp,64 es t1,-1 sto [lb,t1],t2	sto [lb,c_1n],t3	store into a

Table 3. Instruction mappings for a = b*c + 3

The timing analyses [CDC82] of these two sets of META instructions are shown in the Tables 4 and 5, where the interpretation of each column is shown as follows:

Instructions	Issued	Stacked	Short-stop	Register	Memory
es t1,-2	0		1	4	
lod [lb,t1],t2	1			16	
is sp,-64	2		3	6	
sto sp,t2	3				
es t1,2	5	sto	6	9	
	16	sto			26
lod [lb,t1],t2	17			32	
is sp,-64	18		19	22	
sto sp,t2	19				
	26	sto			36
lod sp,t1	27				
is sp,64	28	lod	29	32	
	36	lod		51	
lod sp,t2	37			52	
mpyx t2,t1,t3	38				
	52	mpyx	57	60	
sto sp,t3	53				
es t1,3	55	sto	56	59	
is sp,-64	56	sto	57	60	
	57	sto			67
sto sp,t1	58			68	
lod sp,t1	60				
is sp,64	61	lod	62	69	
	68	lod		83	
lod sp,t2	69		84		
addx t2,t1,t3	70				
	84	addx	85	89	
sto sp,t3	85			95	
lod sp,t2	87				
is sp,64	88	lod	89	92	
es t1,-1	89	lod	90	93	
	95	lod		110	
sto [lb,t1],t2	96				
	110	sto			120

Table 4. Timing analysis (Version 0.0)

1. *Issued:* the time (in terms of minor cycles) when the instruction is issued,

2. *stacked:* the instruction stacked in front of the Floating-Point unit; at most one instruction at a time can be stacked,

3. *short-stop:* the time when the result is available at the short-stop register,

4. *register:* the time when the result is available at the register file, and

5. *memory:* the time when the result is available in memory.

According to these timing analyses, result *a* is available in memory in 120 minor cycles for the Version 0.0 generated instructions and in 32 minor cycles for the Version 0.1 generated instructions. Hence, the execution time of this statement for the Version 0.1 Compiler is roughly a factor of four shorter than that of Version 0.0 compiler.

The overhead associated with the procedure call is limited in this approach, because the scratch registers need to be saved *only* when there is a procedure-call argument, e.g., in a procedure call P(a, Q(b,c), d), the scratch register, storing variable d, needs to be pushed onto the memory stack when procedure Q is called. The activation record is extended as shown in Figure 9. Also, when *branching* occurs and the register stack is nonempty, a push/pop between the register stack and memory stack is needed.

Instructions	Issued	Stacked	Short-stop	Register	Memory
lod [lb,c_2n],t0	0			15	
lod [lb,c_2],t1	1			16	
mpyx t0,t1,t2	2				
	16	mpyx	21	24	
addx t2,c_3,t3	17				
	21	addx	22	25	
sto [lb,t1],t2	22				32

Table 5. Timing analysis (Version 0.1)

The static statistics gathered while compiling the 104 library routines and the entire CC205 compiler (which has total of 31 modules) are shown in Table 6. In Table 6, eight modules have depth zero; this is because they are data files. The average maximum depth of register stack is 3.4 (excluding the data files).

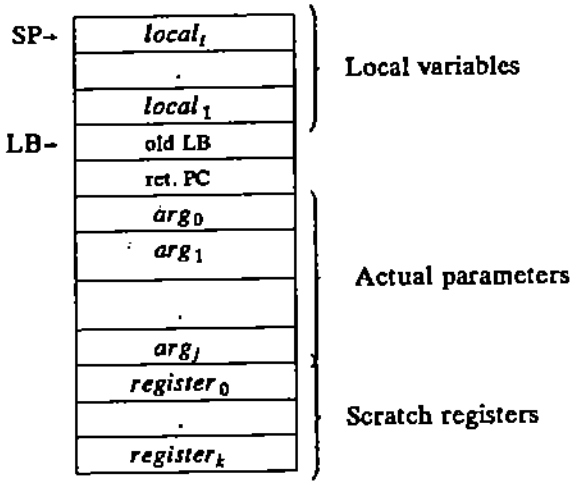

Figure 9. Activation record for a procedure call

Maximum Depth	Number of Modules
0	8
1	5
2	19
3	50
4	37
5	10
6	3
7	3

Table 6. Maximum depth of register stack for 135 modules

Obviously, another achievement of this optimization is that a significant amount of program space is saved (see for example Table 3).

6.3. Benchmark runs

Five benchmark programs were used to compare the preliminary versions of CC205 with the VAX C compiler. Four of them are from [HoBK83], namely, (1) the Sieve of Eratosthenes, (2) Floating-point, (3) Sorting, and (4) Fibonacci series benchmarks (see Appendix). Actually, in [HoBK83], there are five benchmarks; the fifth one is a disk file I/O benchmark.

Currently, we still have problems with CC205 handling random access files. The fifth benchmark is a Queueing Network Model solver - PMVA (Purdue Mean Value Analysis) [Schw80], which evaluates several thousand network population configurations.

The compile times (the CPU time required to generate assembly code) for these benchmark programs on the VAX and the Cyber 205 are shown in the Table 7, where the column labeled VAX C means the compile time using VAX *cc*, the columns labeled CEM_0, CEM_m and CEM_1 represent the compile times using the cross compilers (Version 0.0, Peephole Meta Optimizer and Version 0.1 respectively) running on the VAX 11/780, and the column labeled $CC205_1$ means the compile time using the Version 0.1 C compiler on the Cyber 205. From Table 7, we can see that the overhead (CEM_1 vs. CEM_0) incurred by optimization is about 13%.

Benchmarks	*VAX C*†	CEM_0‡	CEM_m‡	CEM_1‡	$CC205_1$‡
sieve.c	0.9	2.9	6.0	3.3	0.4
sort.c	2.0	7.1	19.9	8.2	1.6
fibo.c	0.7	2.7	4.6	3.2	0.3
float.c	1.3	2.9	8.3	3.3	0.4
pmva.c	31.9	48.1	245.8	51.9	14.1

† compiled on VAX

‡ compiled on Cyber 205

Table 7. Compile times (in seconds)

Table 8 shows the execution times of the compiled benchmark programs, where *%improvement* means the percentage improvement of the $CC205_1$ generated code over the $CC205_0$ generated code. The long execution time (24.6 seconds) of *fibo.c* on VAX C indicates that the VAX C is inefficient for a large number of recursions.

Benchmarks	*VAX C*†	$CC\,205_0^{\ddagger}$	$CC\,205_m^{\ddagger}$	$CC\,205_f^{\ddagger}$	% improvement*
sieve.c	2.8	1.61	1.00	0.67	58.39
sort.c	20.9	12.92	8.11	4.61	64.32
fibo.c	24.6	3.60	2.65	2.15	40.28
float.c	1.9	0.37	0.22	0.18	51.35
pmva.c	15.4	7.89	5.04	2.51	68.06

† executed on VAX

‡ executed on Cyber 205

* $(CC\,205_0 - CC\,205_1) / CC\,205_0$

Table 8. Execution times (in seconds)

6.4. Another size specification

In order to make CC205 more compatible with existing C programs, another set of size specifications of data types as given in Table 9 was tried. However, this version of CC205 turned out to be about twice as slow as the previous one, because it caused more instructions to be generated. To see this, when accessing a local 'int' variable using the previous size specifications, only one EM instruction is generated (e,g,. LOL -8, load local variable), but using the later specifications, two EM instructions are produced (e.g., LAL -4 and LOI 4, i.e., load address of the local variable then load indirect of four bytes).

Size	Number of bits
word	64
pointer	64
long	64
int	32
short	16
char	8
float	64
double	64

Table 9: Alternative size specifications.

The later version (with specifications from Table 9) has advantages for space-saving and

portability, but its disadvantage is poor performance. Since space is not a critical issue on the Cyber 205, and, based on the experience of bootstrapping CC205, portability is not a serious problem (see Section 5.3.2), we have retained the previous size specification.

7. Summary and Future Work

The EM Tool Kit has allowed us to construct a C compiler for a new machine (the Cyber 205) in a relatively short period of time (about four man months). The cost of using this approach lies in the (relatively) inefficient compiler and compiled code. Our initial experience indicates that application of successive optimization steps can lead to acceptable performance.

The project has several research goals, which will be pursued as the scalar C compiler becomes stable. One goal is the development of instrumentation in the compiler and in the generated code. This instrumentation will be used to assess the effects of the various attempts at performance improvements.

The other major goal is the introduction of extensions to the C language which can allow programmers exploit the vector features of the Cyber 205. The super-speeds of the Cyber 205 are realized only when problems can be formulated in terms of vectors and vector processing. We feel that C is especially well-suited to the introduction of vector (and sub-vector) data types and vector operations.

Our plan then can be summarized in three steps:

1. Implement stable scalar version of the CC205 compiler,
2. Modify the compiler to provide instrumentation for measuring performance of the compiler and the compiled code, and
3. Extend the CC205 compiler to add vector data types and vector operations.

Once the instrumented compiler with vector extension becomes available, a variety of research questions can be addressed. Among these, we include:

1. What are the benefits of vectorization?
2. How can scalar algorithms be converted to their vector analoges? can this be automated?
3. How can stack-oriented languages be efficiently implemented on machines without stacks?

We feel that our approach, vector-oriented extensions to an existing structured scalar language (together with an instrumented compiler), will allow us to engage in fruitful, scientific research on these and other topics. The fact that we can easily alter the compiler means that many approaches can be tried.

8. Acknowledgments

This project has received valuable assistance and support from many people. Saul Roson, Director of PUCC, has provided ideas and support from the beginning. Dale Talcott and Ken Adams, of the staff of PUCC, and John Jackson of CDC, have expended many hours helping with the Cyber 205 part of the project. The students in CS590, Jim Anderson, Steve Englestad and Jerry Gross, spent the summer of 1983 working on the peephole optimizer and moving CCLIB to the Cyber 205.

9. References

[AhUl77] A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[Ande83] J. Anderson "Design and Implementation of a META Peephole Optimizer," *work notes*, July 1983.

[BaBK76] P.A. Batson, E.R. Brundage, and P.J. Kearns, "Design Data for Algol-60 Machines," *ACM SIGARCH*, January 1976.

[Buch83] I.Y. Bucher, "The Computational Speed of Supercomputers," *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, August 1983.

[BYTE83] *BYTE (the small systems journal)*, Vol 8., No. 8, August 1983.

[CDC81a] "CDC CYBER 205 Hardware Reference Manual," *Control Data Corporation*, 1981.

[CDC81b] "CDC VSOS Version 2 Reference Manual, Volume 1 and 2," *Control Data Corporation*, 1981.

[CDC81c] "CDC CYBER 200 Assembler Version 2 Reference Manual," *Control Data Corporation*, 1981.

[CDC82] "Enginerring Specification," *Control Data Corporation*, No. 10358026, September 1982.

[Dijk72] E.W. Dijkstra, "The Humble Programmer," *Comm. ACM*, October 1972.

[FeGe82] A.R. Feuer and N.H. Gehani, "A Comparison of the Programming Language C and Pascal," *ACM Computing Surveys*, March 1982.

[Flyn66] M.J. Flynn, "Very High-Speed Computing Systems," *Proc. of IEEE, Vol. 54*, December 1966.

[Haik82] I.J. Haikala, "More Design Data for Stack Architectures," *Proceedings of the ACM '82 Conference*, October 1982.

[HiTa72] R.G. Hintz and D.P. Tate, "Control Data STAR-100 Processor Design," *COMPCON'72 Digest*, p1-4, 1972.

[HoBK83] J. Houston, J. Brodirck, and L. Kent, "Comparing C Compilers for CP/M-86," *BYTE*, August 1983.

[HoJe81] R.W. Hockney and C.R. Jesshope, *Parallel Computers*, Adam Higer Ltd. Bristol. 1981.

[John78] S.C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. of the 5th ACM Symposium on Principles of Programming Languages*, January, 1978.

[John80] S.C. Johnson, "A Tour through the Portable C Compiler," *UNIX Programming Manual*, U.C. Berkeley, 1980.

[KeRi78] B.W. Kernighan and D.M. Ritchie, "The C Programming Language," *Prentice-Hall, Inc.* 1978.

[KKPL81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. of the 8th ACM Symposium on Principles of Programming Languages*, January 1981.

[Knut71] D.E. Knuth "An Empirical Study of Fortran Programs," *Software-Practice and Experience*, January 1971.

[Mill78] R. Miller, "UNIX - A Portable Operating System?" *ACM SIGOPS*, July 1978.

[MiTa82] C.H. Minchew and K.C. Tai "Experience with Porting the Portable C Compiler," *Proceedings of the ACM '82 Conference*, October 1982.

[NAJN75] K.V. Nori, U. Ammann, K. Jensen, and H. Nageli, "The Pascal P Compiler Implementation Notes," *Eifgen. Tech.* Hochschule, Zurich, 1975.

[Orga73] E. Organick, "Computer Systems Organization, the B5700/B6700 Series," *Academic Press, New York*, 1973.

[PeCM83] R.H. Perrott, D. Crooles, and P. Millign, "The Programming Language ACTUS," *Software-Practice and Experience*, Vol. 13, p305-322, 1983.

[PeSt81a] R.H. Perrott and D.K. Stevenson, "Users' Experiences with the ILLIAC IV System and its Programming Languages," *SIGPLAN Notices*, July 1981.

[PeSt81b] R.H. Perrott and D.K. Stevenson, "Consideration for the Design of Array Processing Languages," *Software-Practice and Experience*, November 1981.

[Robe83] B. Roberts, "The C Language," *BYTE*, August 1983.

[Schw80] H. Schwetman, "Implementing the Mean Value Analysis Algorithm for the Solution of Queueing Network Models," *CSD-TR-555, C.S. Dept., Purdue Univ.*, October 1980.

[Stee60] T.B. Steel, "UNCOL: the Myth and the Fact," *Ann. Rev. Auto. Prog.* R. Goodman, (ed.) Vol 2, 1960.

[Tane78] A.S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM 21*, March 1978.

[TaSS80] A.S. Tanenbaum, H. Staveren, and J.W. Stevenson, "Description of an Experimental Machine Architecture for Use with Block Structureed Languages," *Informatica Rapport 54, Vrije University, Amsterdam*, 1980.

[TaSS82] A.S. Tanenbaum, H. Staveren, and J.W. Stevenson, "Using Peephole Optimization on Intermediate Code," *ACM Trans. on Programming Languages and Systems*, January, 1982.

[TSKS81] A.S. Tanenbaum, H. Staveren, E.G. Keizere, and J.W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," *Dept. of Math. and Computer Science, Vrije University, Amsterdam*, 1981.

[Weth80] C. Wetherell, "Design Consideration for Array Processing Languages," *Software-Practice and Experience*, April 1980.

10. Appendix - Listing of benchmark programs

```
/* sieve.c - Eratosthenes Sieve Prime Number Program */

#define true   1
#define false  0
#define size   8190

char   flags[size+1];
main()
{      int i,prime,k,count,iter;
       printf("10 iterations\n");
       for (iter = 1; iter <=10; iter++) {
```

```
            count = 0;
            for (i = 0; i < = size; i++) flags[i] = true;
            for (i = 0; i < = size; i++) {
                  if (flags[i]) {
                        prime = i+ i+ 3;
                        for (k = i+prime; k< =size; k+= prime)
                              flags[k] = false;
                        count++;
                        }
                  }
            }
            printf("\n%d primes\n",count);
}


/* fibo.c - The Fibonacci series benchmark */

#include      "STDIO"
#define NTIMES  10
#define NUMBER  24
main()
{     int     i;
      unsigned value, fib();
      printf("%d iterations: ",NTIMES);
      for (i=1; i < =NTIMES; i++) value = fib(NUMBER);
      printf("fibonacci(%d) = %u \n", NUMBER, value);
      exit(0);
}

unsigned fib(x)
int x;
{
      if (x > 2)
            return(fib(x-1) + fib(x-2));
      else
            return(1);
}



/* sort.c - Quicksort benchmark */

#include      "STDIO"

#define MAXNUM  1000
#define COUNT   10
#define MODULUS ((long) 0x20000)
#define C       13849L
#define A       173L

long    seed = 7L;
long    buffer [MAXNUM] = {0};
```

```
long   random();


main()
{      int i,j;
       long temp;

       printf("Filling array and sorting %d times\n",COUNT);
       for (i=0; i< COUNT; ++i) {
              for (j=0; j < MAXNUM; ++j) {
                     temp = random(MODULUS);
                     if (temp < 0L)  temp = (-temp);
                     buffer[j] = temp;
                     }
              printf("Buffer full, iteration %d\n",i);
              quick(0,MAXNUM-1,buffer);
              }
       printf("Done\n");
}

quick(lo,hi,base)
int lo,hi;
long base[];
{      int i,j;
       long pivot, temp;

       if (lo < hi) {
              for (i=lo, j=hi, pivot=base[hi]; i < j; ) {
                     while (i< j && base[i] < pivot) ++i;
                     while (j> i && base[j] > pivot) --j;
                     if ( i< j) {
                            temp = base[i];
                            base[i] = base[j];
                            base[j] = temp;
                     }
              }
              temp = base[i];
              base[i] = base[hi];
              base[hi] = temp;
              quick(lo, i-1, base);
              quick(i+1, hi, base);
       }
}


long random (size)
long size;
{
       seed = (seed*A + C) % size;
       return(seed);
}
```

```
/* float.c - Floating-point benchmark */

#define CONST1  3.141597E0
#define CONST2  1.7839032E4
#define COUNT   10000

main()
{       double a,b,c;
        int i;

        a = CONST1;
        b = CONST2;
        for (i=0; i < COUNT; ++i) {
                c = a * b;
                c = c / a;
                c = a * b;
                c = c / a;
                c = a * b;
                c = c / a;
                c = a * b;
                c = c / a;
                c = a * b;
                c = c / a;
                c = a * b;
                c = c / a;
                c = a * b;
                c = c / a;
                }
        printf("Done %f\n",c);
}
```