15

Compiling at 1000 MHz and beyond

A commentary on future compiler technology

D.B. Wortman
Department of Computer Science
University of Toronto, 10 Kings College Road, Toronto, Ontario, Canada M5S 1A4
E-mail: wortman@cs.toronto.edu

Abstract

The rapid changes in computer hardware technology that have occurred in the past are expected to continue well past the year 2000. This paper investigates the impact that these trends will have on the programming language compiler. We examine the dual questions of

1. How will the internal organization and structure of compilers need to change in order to adapt to new technology?
2. How will the processing performed by compilers need to change in order to keep up with changes in the underlying hardware?

Keywords

Compilation, programming language processors, technology forecast

1 CLASSICAL COMPILER STRUCTURE

We begin with a very brief description of the structure of a traditionally organized compiler for a conventional programming language to provide context for the discussion that follows. This is the model of compilation that is discussed in most academic compiler text books (Fischer & LeBlanc 1991). No real compiler will follow this model exactly but the compiler processing described below occurs in some form in almost all compilers.

Compilation can be described as a number of processing steps that are (conceptually) performed in sequence.

- *Preprocessing* A manipulation of the source program text that occurs before the text is broken down into lexical tokens. For example the preprocessor used for C programs implements macro definition and use, conditional compilation and source file inclusion.

Systems Implementation 2000 R. Nigel Horspool (Ed.)

- *Scanning* The source program text is processed to divide it into language specific *lexical tokens.* Extraneous material like whitespace and comments is discarded.
- *Parsing* The sequence of lexical tokens produced by scanning is analyzed to determine if it constitutes a legal program in the language being compiled. Some intermediate data structure (e.g. an *abstract syntax tree*) is produced that captures the essential structure of the program for subsequent processing.
- *Semantic Analysis* The program is analyzed to determine if it is correct with respect to the non-syntactic constraints imposed by the language being compiled. (e.g. type correctness, declaration of symbols before their use)
- *Code Generation* The program is transformed from its intermediate representation into a sequence of instructions for some real or artificial computer. Generally this involves selecting a sequence of instructions to implement each construct (e.g. expression, statement) in the language. Instruction selection may depend on context, this is a form of machine-dependent optimization.
- *Optimization* The purpose of optimization is to transform the program (without changing its meaning) so that the program will execute more quickly. There are two main types of optimization.
  Machine independent optimization transforms the program in ways that do not depend on the target machine for the compilation. Usually this involves reducing the amount of work that is done during program execution. Examples of machine independent optimizations include constant expression evaluation, moving loop independent code out of loops, optimizing procedure call overhead through tail recursion elimination.
  Machine dependent optimizations transform the program so as to take advantage of the characteristics of the target machine. Machine dependent optimizations include allocating registers to variables, minimizing pipeline stalling branches, structuring data access for improved cache performance.

Any particular compiler will perform these steps in some number of *passes* or *phases.* Preprocessing and scanning are often called the compiler *front end,* parsing and semantic analysis are the *middle end* and code generation and optimization are the *back end.* Much of the processing in a compiler involves the management of several major data structures.

- *Source Program* First files and characters, then lexical tokens and later some intermediate representation. These data structures can be quite large. Many compilers use some form of *abstract syntax tree* as the primary intermediate representation. The tree is created during parsing and then modified by subsequent processing.
- *Object Program* During back end processing the compiler will maintain some data structure that represents the object program being produced.

This may be a simple linear buffer although singly and doubly linked lists are also widely used.

- *Symbol Table* Used to record information about all symbols (e.g. identifiers) occurring in the program. Depending on the programming language being compiled and compiler design decisions this may be implemented as one large table or a collection of smaller interlinked tables (e.g. type information might be stored in a separate table, separate tables might be used for each scope of declaration). When compiling large programs these tables will typically have thousands or even tens of thousands of entries. For example there may be table entries for all symbols from included interfaces, e.g. class libraries in C++. Most production compilers use some form of hash table (or a similar data structure) to optimize table lookup time.
- *Control Tables* Many compilers use table driven processing algorithms that operate from large constant tables. For example YACC based parsers or Glanville-Graham style code generators.
- *Optimization Data Structures* Some optimization algorithms use very large data structures (e.g. a control flow graph for the program) during their processing (Muchnick 1997). On contemporary machines the size of these data structures is one factor that limits the feasibility of some optimizations.

The algorithms that manipulate these data structures in most contemporary compilers were designed when main memory usage was a significant issue and the processor/memory bandwidth was not a significant issue.

2 THE FUTURE OF HARDWARE

Most projections of hardware technology for the year 2000 and beyond (IEEE Spectrum Staff 1997, Burger & Goodman 1997) suggest that we will have available a vastly more powerful and capable computing environments. Moore's Law (the number of transistors on a chip doubles every 18 month) is now expected to be a reasonable forecast through 2017.

- Clock speeds for affordable processors will reach the many hundred megahertz range. Current top end processors with clock speeds in excess of 600 MHz clocks are currently available, clock speeds of 1 GHz or more may be reached by the year 2000. Near term changes to processor instruction sets will be in the direction of multimedia support and in the introduction of short instructions to improve instruction cache density. Most machines will be highly *superscalar*, attempting to execute multiple instructions per machine cycle. The number of instructions issuable per cycle will increase in future processors. Some manufacturers are investigating VLIW (very long instruction word) architectures as a means to achieve future perfor-

mance gains. Instruction pipelines will likely deepen as technology changes increase the number of transistors on a chip.

- The typical contemporary processor has relatively small level one (L1) caches for instructions and data and a somewhat larger second level (L2) cache. On current high end processors the time penalty for a L1/L2 cache miss is equivalent to several hundred processor instructions. In the future we expect the L1 and L2 caches to grow in size and the penalty for cache misses to become more severe.
- Physical memories, even on modest workstations will be in the hundreds to thousands of megabyte range. One Gb memory chips should be available in the not too distant future.
- Local area networks will have capacities measured in the terabyte range while wide area networks will reach multiple gigabyte speeds. This trend may increase the popularity of the *network* computer, a workstation with little or no local disk storage.
- Disks with 10s of gigabyte capacity will be readily available at reasonable prices. RAID style disk organizations will increase in popularity.
- DVD, the successor to CD-ROMs will provide online read only storage capacities in the 500 MB to 1 Gb range.
- The cyclical popularity of parallel processing may reach another peak as the limits of single processor architectures are reached. The success of multiprocessing will depend on the level and quality of support provided by the underlying operating systems and or our ability to develop software that makes effective use of parallel hardware.

On the other hand there are trends which will make the task of compilation more difficult in the future.

- The ratio processor speed to memory speed will *increase*, leading to larger primary and secondary caches interposed between the processor and physical memory. Chip makers will attempt to counter this imbalance with larger on-chip primary and secondary caches, larger register files and denser instruction sets. Some manufacturers are prototyping processor chips with large on-chip memories (Anon 1996).
  Making most effective use of the available processor/memory bandwidth and achieving high cache utilization will become the dominant performance issues as processor speeds continue to outstrip memory access speeds.
- Memory access and cache interactions will continue to be a performance bottleneck for fast IO devices and networks.
- Processors will have more general registers that the compiler must try to use effectively.

The net effect of these changes will require significant changes in the way compilers are constructed which will improve compiler performance and maintainability. A trend in recent hardware designs (e.g. the DEC Alpha and the Intel/HP IA-64) has been to simplify the hardware to improve processor speed and to require the compiler to expend considerably more effort on code selection and optimization. The recently announced Intel/HP IA-64 (Merced) architecture(Crawford & Huck 1997) is indicative of the trends in this area

- A much larger number of general purpose and floating point registers to be managed by the compiler (128 each in the IA-64).
- Predictive and Speculative execution modes that transfer instruction scheduling effort from the hardware to the compiler.
- Six way multiple instruction issue.
- Designs that minimize the impact of branching on processor pipelines.
- Speculative load instructions to help mitigate memory access latency.
- Support for parallel execution of instructions in the processor.
- Processor support for multiple instruction sets (IA-32 and IA-64 in Merced) that will greatly complicate the compiler's code generation algorithms.

These changes will also require compilers that strive to produce highly optimized object programs to work even harder since most of these changes make optimization more difficult and/or more costly to perform. The next section discusses a number of issues related to compiler structure and the new forms of compiler processing that may be required.

3 THE FUTURE OF COMPILERS

These inevitable trends in hardware technology will lead to significant changes in the way that programming languages are processed and in the role of the software that we presently know as compilers . There are two issues to consider with respect to the future of compilation

1. What changes will be required to allow compilers to continue to operate with acceptable performance?
2. What changes will be required to allow compilers to generate object programs that operate with acceptable performance?

The first issue will affect the structure of compilers and the techniques that are used to process programming languages. The second issue affects the type of algorithms that will need to be used in compilers (e.g. superoptimization.)

3.1 Compiler Organization and Structure

Most contemporary compilers large monolithic programs. I expect that compilers will become much more *Object Oriented* in their internal architecture. This change should lead to improved maintainability and portability of compilers. An example of the evolution that I hope will occur is the novel structure of the DEC SRC Modula-3 compiler developed by Kalsow, Muller & Heydon (1995). In this compiler

- There are four major Object types: *values (bindings), statements, expressions* and *types.*
- Each of these major Objects provides an interface to a family of derived objects representing language specific entities. For example the value object includes any kind of value that can be named in Modula-3 including constants, variables, functions, procedures, modules, types, fields, etc. There is a derived statement object for each type of statement in the language.
- A major effort was made to encapsulate all knowledge about each language feature in a single module. For example there is a single module that contains all of the processing (syntax analysis, semantic analysis, code generation) for each type of statement. Information hiding is used extensively to limit access to information about each language construct to the module in which the construct is processed.

Kalsow (1996) reports that the object oriented structure greatly enhanced compiler maintainability. Most compiler bugs could be fixed by corrections within a single object. The Modula-3 compiler was an interesting experiment in a novel structure for a compiler. It was an incomplete experiment since the compiler did not include an optimization phase so the issue as to whether this design would work well with a heavy weight optimizer is still open to question. Kalsow (1996) points out that optimization is performed using optimization specific data structures that can be created separately so that there is some reason to believe that this object oriented design *is* compatible with heavy weight optimization.

For highest performance, compiler code and data should fit into the processors L1 cache. To achieve this goal, we may see a return to a compiler organization based on many small passes that each make some small contribution to the compilation process. Perhaps something reminiscent of Ershov's 30 pass Algol-60 compiler (Yershov 1965).

3.2 Compiler Internal Data Structures

Larger physical memories will make it feasible to maintain very large memory resident data structures, e.g. compiler tables and the abstract syntax tree for

a program. This will largely eliminate the algorithms used in many contemporary compilers for keeping part of these data structures on disk. Large in memory data structures should be used only if they do not seriously impact processor data access. Many data structure algorithms (e.g. searching a linked list) induce very poor cache utilization. We will probably see a much wider use of persistent memory structures (i.e. permanently memory resident parts of compiler tables) such as the symbol table entries for standard library routines and the control tables required to drive table driven algorithms. Koehler & Horspool (1996) have demonstrated a prototype mechanism for caching symbol compiler tables. Given the expected bandwidth improvements in local and wide area networks, their approach might be very attractive in network computer environments.

Most production compilers use some form of hash table for efficient symbol table lookup. Used naively hashing can cause very poor memory/cache performance. New algorithms will need to be developed that optimize use of processor memory bandwidth and maximize cache hit ratios. For example an old technique that may return to popularity is the *unique identifier number* technique. With this technique, the scanner assigns a unique integer key to each *distinct* identifier that it processes (e.g. every occurrence of the identifier `foobar` in the program will be mapped to the integer key `231`). This assignment can be done efficiently using a relatively small hash table or similar data structure. Thereafter, this integer key is used to access information about the identifier, e.g. the integer key is used to directly access the compilers symbol table whenever middle or backend processing need information about the identifier. This technique provides lookup that is as efficient as hashing but with better memory and cache characteristics. The processing required to handle reuse of the same identifier in different scopes of declaration is straightforward to implement.

3.3 Effects on Front End Processing

Although techniques for designing very efficient compiler front ends have been known for some time (Waite 1986) much more will be need to be done to make memory access within the front end as efficient as possible. Much more than in the past, compiler front end algorithms will have to be tuned for efficient memory access and high cache utilization. For example, very significant performance gains can be achieved if the scanner is designed so that its code and local data fit in the processors L1 cache(Morgan 1997). Waite recommends an approach that minimizes the number of times each character is accessed during front end processing. This recommendation is more important today with the increased processor/memory bandwidth problem.

Another likely trend in front end processing is to eliminate it entirely. Programming environments that store programs in a pre-tokenized internal repre-

sentation are becoming mature products that are widely available. Compilers will process this internal representation directly rather than working from traditional source files.

Context Inhalation is the process of efficiently dealing with external files that are incorporated into a compilation unit. For example, use of `#include` in C or the `IMPORT` declaration in Modula-3. It has long been recognized (Cashin, Joliat, Kamel & Lasker 1981) that inefficient context inhalation can significantly reduce compiler efficiency. Litman (1993) implemented a form of preprocessing that predetermined which symbols in an imported interface were actually used in compiling a given piece of source code. Inhalation could then by optimized to read in only these symbols. We expect techniques like this to be more widely used in the future as a way to reduce the context inhalation overhead.

3.4 Effects on Program Representation

Compression of object code, source code and compiler tables will become a common approach to dealing with processor/memory and memory/disk bandwidth bottlenecks. (Ernst, Evans, Fraser, Lucco & Proebsting 1997). Many implementations of Java use a compressed library of standard classes as a mechanism for reducing input bandwidth. Compilers (or link editors) will generate object modules and executables in a compressed format.

3.5 Effects on Code Generation

The near term changes in processor instruction sets that will affect code generation and code selection most are the addition of multi-media instructions, the ability to deal with small data types (e.g. 8 or 12 bit integers) and the development of RISC style architectures with several sizes of instructions. We are already seeing efforts by some manufacturers to develop RISC machines with short encodings for frequently used instructions as a way of reducing the required memory bandwidth for instruction fetch.

In the past, code selection and generation for RISC architectures was relatively straight forward because there was often only one possible instruction to implement each operation (e.g. integer addition). The projected changes to processors will make code selection for RISC machines much more like code selection for CISC machines. Extensive pattern matching on an intermediate representation may be required to determine when the multimedia instructions can be utilized. A variety of data sizes and tradeoffs between instructions of various lengths will require more decision making and complexity in the code selection process.

Compiler will have to pay much more attention to data and program layout

in order to achieve good cache utilization. For some recent work in this area see (Hashemi, Kaeli & Calder 1997).

3.6 Effects on Optimization

The expected increases in the ratio of processor speed to memory speed and the widespread use of multilevel caches will only increase the need for extremely sophisticated optimization techniques to achieve high processor utilization and acceptable performance. The significantly faster processors and the much large memories that I am assuming will increase the size of optimization problems that can be handled in practice, but the fundamental NP-completeness of optimization problems will endure. Optimization is one of the most important areas of for current research in compiler technology(Muchnick 1997). There are many issues that modern optimizing compilers attempt to deal with

- *Register Allocation* attempts to optimize program execution speed through intelligent use of a (relatively small) fixed number of hardware registers. Graph coloring techniques (Chaitin 1982) are used to select registers for instructions and to allow some program variables to reside in registers. This area will become less significant as the number of registers available in a processor increases. The techniques suggested by Sites (1979) may again become relevant.
- *Interprocedural Analysis* attempts to discover optimizations (e.g. access to shared variables) that occurs across function call boundaries. Whole-program optimization will become a significant research area(Blainey & Archambault 1997).
- *Predictive and Speculative Execution* New processors will use branch path prediction and speculative execution of instructions as a way of gaining speed. New optimization algorithms(Hwu 1997, August, Hwu & Mahlke 1997) will be required to deal with this change in processor functionality.
- *Parallelism optimizations* attempt to discover parts of a program that can be executed concurrently. This optimization is useful even on single processor systems because such systems are typically superscalar (executing more than one instruction per machine cycle) and pipelined (instruction execution occurs in a number of distinct stages).
- *Object Oriented Optimizations* deal with optimization opportunities that arise in object oriented languages such as C++ and Java. Typical optimizations are compile time determination of virtual function call targets and changes to reduce storage management (garbage collection) overhead.
- *Machine Dependent optimizations* for contemporary RISC processors include instruction scheduling (to avoid processor stalls), data mapping and reorganization to maximize cache performance.

We expect to see increased efforts in the area of *run-time* optimization of programs, either through increased run-time gathering of profile information as suggested by Ammons, Ball & Larus (1997) or by dynamic optimization done *during* program execution (Goodwin 1997). In this later case some part of the optimization process is embedded in the run time system and is invoked to optimize the program for a particular execution.

3.7 Effects on the Compilation Process

Although faster processors can mask a lot of inefficiency, programs are becoming larger and are using more complicated interfaces (e.g. class library hierarchies in C++ and Java) efficient compilation will continue to be of concern especially in organizations whose commercial success depends on efficient software development and/or maintenance.

Faster local area networks will make various forms of load sharing and distributed processing more attractive. For example the *Load Sharing Facility* developed by Zhou, Zheng, Wang & Delisle (1994) provides for transparent migration of processing across a heterogeneous network of workstations. The range and attractiveness of this type of load sharing will increase with faster local area networks.

If parallel processing comes back into vogue then we may see compiler designs that attempt to exploit parallel processing. Parallel processing could be exploited at the compilation level as is currently done by programs like parallel make (Baalbergen 1988) or within the compiler itself. Vandevoorde (1987), Wortman & Junkin (1992) and others have demonstrated prototype compilers that were able to take advantage of small scale parallel processing. The extension of compiler internal processing to highly parallel machines as suggested by Hillis & Steele (1986) will be an interesting and challenging problem.

3.8 Data Caching, Incrementalism and Dynamic Compilation

Some existing compilers cache data about source programs (e.g. context inhalation hints). Incremental compilers are often used in interactive environments where a program is (conceptually at least) recompiled as it is being edited (Adams, Tichy & Weinert 1994). In a related context Linton & Quong (1989)showed that most programs change very little from one compilation to the next so that incremental modification to program executable files could significantly reduce the time required to link edit programs. In cases where compilers interoperate with a programming environment that tracks source code changes it would be feasible to recycle parts of object programs and only

recompile the parts that have changed. Algorithms for effective incremental compilation are well known and could easily be extended if the savings in memory bandwidth and compiler processing were large enough.

An extreme form of incrementalism which may be attractive in the future is *just in time compilation.* The initial executable representation of a program is derived directly from the internal representation used in the middle end of compiler processing (e.g. something equivalent to the abstract syntax tree that is input to code generation). The code generation (and optimization) phases of the compiler are available as a part of the runtime system. The first time execution reaches some part of the program, code generation is dynamically invoked and that part of the program is transformed into machine instructions. The advantages of this technique arise from two observations

- A small portion of the source code (typically 10..20%) is responsible for almost all of the executed instructions in a typical execution.
- In typical program executions, large portions of the source code (e.g. 50..60%) are *never* executed.

Some current Java implementations feature just in time compilation. Poletto, Engler & Kaashoek (1997) recently demonstrated a prototype system for dynamically generating code for C.

4 SUMMARY

In this paper we have discussed the effect that anticipated changes in hardware technology will have on the structure and operation of compilers. The hardware changes open up many worthwhile avenues for improvement but also lead to new and more challenging problems especially in the area of program optimization.

ACKNOWLEDGMENTS

The material in this paper has benefited from discussions with my colleagues in IFIP Working Group 2.4 including Robert Morgan, Gerhard Goos, Martin Hopkins, Nigel Horspool and William Waite.

REFERENCES

Adams, R., Tichy, W. & Weinert, A. (1994), 'The cost of selective recompilation and environment processing', *ACM Transactions on Software Engineering and Methodology* **3**(1), 3–28.

Ammons, G., Ball, T. & Larus, J. (1997), Exploiting hardware performance counters with flow and context sensitive profiling, *in* 'Proceedings of the ACM Sigplan Symposium on Programming Language Design and Implementation', pp. 85–96.

Anon (1996), 'Mitshubishi java device'.
URL: *http://www.melco.co.jp/service/m32r/*

August, D., Hwu, W. & Mahlke, S. (1997), A framework for balancing flow control and prediction, *in* 'Proceedings of the 30th International Symposium on Microarchiitecture'.

Baalbergen, E. H. (1988), Design and implementation of parallel make, *in* U. Association, ed., 'Computing Systems, Spring, 1988.', Vol. 1, USENIX, pp. 135–158.

Blainey, B. & Archambault, R. (1997), The toronto portable optimizer: Engineering large scale optimization, *in* 'CASCON 97 Compiler Optimization Workshop', IBM Canada Centre for Advanced Studies.

Burger, D. & Goodman, J. (1997), 'Billion transister architectures', *IEEE Computer* **30**(9), 46–48.

Cashin, P., Joliat, M., Kamel, R. & Lasker, D. (1981), Experience with a modular typed language: Protel, *in* 'Proceedings of the 5th International Conference on Software Engineering', pp. 136–143.

Chaitin, G. (1982), Register allocation and spilling via graph coloring, *in* 'Proceedings of the ACM Sigplan Symposium on Compiler Construction', pp. 98–105.

Crawford, J. & Huck, J. (1997), Motivation and design approach for the ia-64 64-bit instruction set architecture, *in* 'Microprocessor Forum'.

Ernst, J., Evans, W., Fraser, C., Lucco, S. & Proebsting, T. (1997), Code compression, *in* 'Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation', pp. 358–365.

Fischer, C. & LeBlanc, Jr., R. (1991), *Advanced Compiler Design Implementation*, Benjamin/Cummings Publishing Co.

Goodwin, D. (1997), Interprocedural dataflow analysis in an executable optimizer, *in* 'Proceedings of the ACM Sigplan Symposium on Programming Language Design and Implementation', pp. 122–133.

Hashemi, A., Kaeli, D. & Calder, B. (1997), Efficient procedure mapping using cache line coloring, *in* 'Proceedings of the ACM Sigplan Symposium on Programming Language Design and Implementation', pp. 171–182.

Hillis, W. D. & Steele, Jr., G. L. (1986), 'Data parallel algorithms', *Communications of the ACM* **29**(12), 1170–1183.

Hwu, W. (1997), Predicated microprocessor architectures and their enabling compiler technology, *in* 'CASCON 97 Compiler Optimization Workshop', IBM Canada Centre for Advanced Studies.

IEEE Spectrum Staff (1997), *Technology 1997: Analysis and Forecast Issue*, Vol. 34, IEEE Spectrum.

Kalsow, B. (1996), 'Re: Object-oriented compiler construction', USENET

News comp.compilers.

Kalsow, B., Muller, E. & Heydon, A. (1995), 'Src modula-3: a tour of the compiler'.
URL: *http://www.research.digital.com/SRC/modula-3/html/compiler.html*

Koehler, B. & Horspool, R. (1996), 'Ccc: A caching compiler for c', *Software - Practice & Experience* .

Linton, M. & Quong, R. (1989), 'A macroscopic profile of program compilation and linking', *IEEE Transactions on Software Engineering* **15**(4), 427–436.

Litman, A. (1993), 'An implementation of precompiled headers', *Software - Practice & Experience* **23**(3), 341–350.

Morgan, R. (1997), private communication.

Muchnick, S. S. (1997), *Crafting a Compiler with C*, Morgan Kaufman.

Poletto, M., Engler, D. & Kaashoek, M. (1997), tcc: A system for fast flexible and high-level dynamic code generation, *in* 'Proceedings of the ACM Sigplan Symposium on Programming Language Design and Implementation', pp. 109–121.

Sites, R. L. (1979), How to use 1000 registers, *in* 'Proceedings of 1st Caltech Conference on VLSI', Caltech CS dept, pp. 527–532.

Vandevoorde, M. T. (1987), Parallel compilation on a tightly-coupled multiprocessor, Master's thesis, Massachusetts Institute of Technology.

Waite, W. (1986), 'The cost of lexical analysis', *Software - Practice & Experience* **16**(5), 473–488.

Wortman, D. & Junkin, M. (1992), A concurrent compiler for modula-2+, *in* 'Proceedings of the ACM Sigplan Symposium on Programming Language Design and Implementation', pp. 68–81.

Yershov, A. (1965), Alpha - an automatic programming system of high efficiency, *in* 'IFIP Congress '65'.

Zhou, S., Zheng, X., Wang, J. & Delisle, P. (1994), 'Utopia: A load sharing system for large, heterogeneous distributed computer systems', *Software - Practice & Experience* .

BIOGRAPHY

Professor David B. Wortman has over thirty years experience in the design and construction of compilers including compilers for XPL (1970), SP/k (1977), Euclid (1980), Modula-2+ (1988). He has published many papers in the area of programming language implementation. Professor Wortman is currently a member of the faculty in the Department of Computer Science at the University of Toronto. He became a member of IFIP Working Group 2.4 in 1977.