Address Reference Generation in a Memory Hierarchy Simulator Environment*

Arnold J. Niessen Harry A. G. Wijshoff

High Performance Computing Division,
Department of Computer Science,
Leiden University
P.O. Box 9512, 2300 RA Leiden, The Netherlands
niessen@cs.leidenuniv.nl

*This work was supported in part by the Esprit Agency DG XIII under Grant No. APPARC 6634 BRA.

Contents

List of Figures

Abstract

Application driven address reference generation is a popular and frequently used technique for the simulation of architectures. These references can be produced by means of the insertion of instrumentation statements in the source code. However, this requires a major rewriting of the application source code under study. To alleviate this disadvantage, this report describes the use of C++ classes and operator overloading minimizing the amount of application code that is affected.

The following frequently used sparse matrix application codes are currently used in conjunction with this simulator: sparse matrix (SpM) LU-decomposition, (SpM x SpM) multiplication, with (SpM x V) sparse matrix vector multiply as a special case, and triangular solve. This report demonstrates the use of this simulator for one of these applications.

1 Introduction

A hierarchical memory system consists of several storage levels [7, 8, 13, 16, 17]. Each of these levels is faster and smaller than the level below. This way, a large virtual memory can be created with a low average access time. In most systems, the lower part of the hierarchy is made up of main memory (DRAM), while the middle part consists of one or more (hardware managed) caches and the higher part consists of the (compiler managed) register banks. Caches are fast small memories which make advantage of two important properties of applications: spatial locality and temporal locality. Temporal locality addresses the fact that a reference to a certain address is likely to occur very soon again, while spatial locality is the property that it is likely that shortly after a reference to a specific data item occurs, data items in the neighborhood of this item have a high probability being used.

Sparse matrix computations [4, 6, 14, 23] form a significant part of large scale applications. Although caches show good behavior, i.e., a high hit rate, on applications with regular memory reference patterns, their performance is disappointing for sparse matrix applications, because these computations exhibit problems which are specific to the nature of the data structure: indirect addressing, irregular memory patterns, fill-in, data movement, and the need for many integer instructions per floating point operation. These problems result in poor temporal and spatial locality.

Due to the growing gap between memory latency and processor cycle time, memory system performance is of vital importance for these and other applications. To gain insight in the performance of memory systems, we have developed a general-purpose memory hierarchy simulator environment. This simulator environment is implemented in C++ and consists of three parts. The first part is supplied by the user and consists of the application code to be analyzed. The second part acts as an interface between this application code and the actual simulator. It captures variable references at source code level from the first part using object-oriented programming techniques. Furthermore, it generates on-the-fly address references to feed the third part. The third part consists of the actual memory hierarchy simulator.

The interface part can be used in conjunction with any code written in C++, C, or even in Fortran using `f2c`. A vast amount of target hardware architectures can be simulated by changing the parameters of the memory hierarchy simulator or adding new simulation modules. The major advantage of our approach is that both the generated address references as well as the simulated architectures are completely independent of the compiler environment, compiler efficiency, and the platform on which the simulation runs.

The outline of this report is as follows: section 2 describes several methods described in related work on performance evaluation. Section 3 discusses the interface between the application code and the memory system simulator. It explains how the use of object-oriented technology enables the generation of address references at source code level. Subsequently, in section 4, we present simulator output for an application code fragment, originating from sparse matrix Cholesky decomposition. Finally, section 5 presents general conclusions about the simulator environment.

2 Related work

Because the performance of hierarchical memory systems becomes more critical as CPU speeds increases, a number of approaches has been taken to estimate the performance of such a memory hierarchy. These approaches include the use of program analysis by means of mathematical tools, execution of profiled applications, and simulation.

Performance prediction based on program analysis is done by Kelly and Pugh ([11]). However, their approach is limited to regular code fragments.

One of the disadvantages of executing a profiled application is that a working system, or at least a prototype of this system, must be available. Obviously, this may be prohibitively expensive or even impossible.

The use of address traces as input of a simulator has become rather popular. Because mathematical models have not been able to generate address traces that are representable for all characteristics of the application under study, applications remain required in order to generate these traces. Such trace driven simulation is very flexible. On the other hand, its limitations are: operating system overhead is not taken into account, long simulations represent only small simulated execution times, and the target architecture may differ from the trace generating architecture.

Address references may be collected in traces for later (post mortem) analysis, or for immediate (on-the-fly) use. Essentially, there are three methods to generate traces for post mortem analysis. First, trace-generating hardware may record all interactions between memory system and CPU, without impact on the application. Second, if this hardware is absent, a compiler may modify the object code at link-time by inserting code fragments recording address references [20]. Third, annotation statements can be added (automatically) to the source code, so that variable reference traces will be generated at run-time [5].

Because the storage requirements of trace data may be substantial, on-the-fly methods are preferred in case storage use or I/O time is at a premium. Source code modification can be used for on-the-fly simulation. This report describes *implicit* source code modification using operator overloading techniques of C++ [21]. It therefore combines the advantages of invisible link-time code modification with the flexibility and generality of source code modification. However, in contrast with traces generated at link-time, it is important to realize that compiler optimizations (see e.g. [1]) are not properly accounted for in the resulting traces in case traces are generated by instrumentation statements added to the source code. Therefore, in an attempt to recover this deficiency, some compiler optimizations (such as register allocation) have been incorporated into the simulator. More advanced compiler optimizations like constant propagation and common subexpression elimination will be lost.

3 Simulator implementation

This section elaborates on the grounds on which C++ has been chosen for the implementation of the interface part of the simulator environment. First, we discuss how references to scalar variables can be captured. Thereafter, we extend this technique to arrays.

3.1 Scalar variables

During the simulation of the memory system, every reference to a variable in the program is represented by a triple of the following form, where $tp \in \{\texttt{RD},\texttt{WR}\}$, ad denotes the address of a variable with a size of sz bytes:

$$(tp, ad, sz)$$

A simple method to produce such a trace by explicit source code modification is shown below. Consider, for example, the following statement:

```
int a,b,c,d;
a = b + c + d;
```

We use tracing functions `RD_int()` and `WR_int()` to generate address references, as described below:

```
int RD_int(int* address)
{ TRACE <RD, address, sizeof(int)>;
  return (*address);
}

int* WR_int(int* address, int value)
{ TRACE <WR, address, sizeof(int)>;
  (*address) = value;
}
```

We may rewrite the original statement into the following construct:

```
int a,b,c,d;
WR_int(&a,((RD_int(b)+RD_int(c))+RD_int(d)));
```

It records read actions on variables `b`, `c`, and `d`, followed by a write action on variable `a`. However, it does not record a write action on a temporary variable (or a register) which could be used to store `b+c`. In that case, a further modification of this statement yields references to this temporary variable as well, as can be seen below:

```
int a,b,c,d,tmp;
WR_int(&a,(RD_int(WR_int(&tmp,(RD_int(b)+RD_int(c))))+RD_int(d)));
```

Clearly, this approach is cumbersome and hence not suitable for quick hand-modifying prototyping. Furthermore, the generated trace depends on the architecture which executes the instrumented application code.

Two aspects of a scalar variable `i` are visible to the programmer during program execution: (1) the right-hand-side value (denoted by `i`), and (2) the left-hand-side value (or address, denoted by `&i`). Less visible is the fact *where* and *how* the data corresponding to such a variable (or corresponding to such an address) is stored. For example, if an integer variable `i` is stored in a register bank or in a cache, a reference to `i` is performed in a few nanoseconds, while this may cost many milliseconds if it is stored in a memory page which has been swapped to disk. Usage of the program visible address `&i` makes generated address references compiler-dependent. Hence, a method is wanted to associate a possibly different address with each variable which is used in the generated references. We may now store this extra information together with program variables, by implementing a class which simulates variables. Combining this approach with overloading all operators minimizes rewriting of source code.

We describe part of the implementation of the class `simint`, which simulates integer variables. This class is a derived class from a class `memobj`, which is used to handle objects stored in memory. A `simint` object `s` (simulating an integer `i`) of class `memobj` has an address `s.address` which is assigned to it by its constructor. The computation of this address is completely independent from the address `&i` as it has been assigned by the compiler, creating platform-independent simulation. It also stores its size and remembers in which memory level it is stored (for example, cache, integer register bank, or floating point register bank).

Figures 1(a..c) show part of this implementation. Every constructed object of `memobj` is automatically allocated an address. Member functions `memobj::RD()` and `memobj::WR()` generate on-the-fly references to the appropriate memory level.

Rather than using class `simint` to define variables in the application source code directly, this class is only used to derive classes `c_int` and `r_int`. The former class is used for variables which are not stored in the register bank (array elements and global variables), while the latter class is used for for register variables (for dynamic variables). We did not implement a derived class `t_int` for temporaries. Instead, elements of class `simint` are treated as temporaries by default. This way, all compiler generated temporaries are treated correctly. This allows replacing integer formal parameters in procedure headings by `simint`. Calls involving actual arguments of either type `c_int`, `r_int`, and `int` are allowed. Calls with actual arguments of type `int` are handled by a temporary variable of type `simint` which is now known (to the interface) to be a temporary. Hence, calls to procedures remain unmodified.

Figure 1(e) shows the calls to the tracing functions `RD()` and `WR()` when the statement in figure 1(d) is executed. Variables `tmp0` and `tmp1` are automatically generated by the constructor call in the return statement of the `operator+` function. The references to variable `tmp1` are redundant, and the use of advanced compiler techniques should be able to remove this constructor call. However, this call is not removed by the currently used compiler (g++, version 2.6.3). To overcome this problem, a detection method has been incorporated in the

```
class memobj { // Class for data objects stored in memory
  int memloc, size;
  int in_ml; // stored in memory level:
             // INTREG_LEVEL integer register bank
             // FPREG_LEVEL  f.p. register bank
             // CACHE_LEVEL  cache
public:
  memobj(const int sz, const int memory_level=INTREG_LEVEL)
  { size = sz;
    in_ml = memory_level;
    memloc = memory_allocate(size, in_ml);
  }
  void RD() const {memsys::mlp[in_ml]->mlRD(memloc,size,memsys::time);}
  void WR() const {memsys::mlp[in_ml]->mlWR(memloc,size,memsys::time);}
};
```

(a) Class `memobj`

```
class simint : public memobj {
  int value;
public:
  simint(const int dummy, const int memory_level): memobj(sizeof(int),memory_level){}
               // ^dummy to enable C++-like initializations simint i=9; simint j(10);
  simint(const int s) : memobj(sizeof(int)) {         WR(); value=s;       }
  simint(simint& s)   : memobj(sizeof(int)) { s.RD(); WR(); value=s.value; }
  simint& operator=(const simint& s)        { s.RD(); WR(); value=s.value; return *this; }
  friend simint operator+ (const simint& a, const simint& b);
}

simint  operator+ (const simint& a, const simint& b)
{ a.RD();b.RD();return simint(a.value + b.value);}
```

(b) Class `simint`

```
class c_int : public simint {
public:
  c_int()                : simint(0,CACHE_LEVEL){};
  c_int(const int s)     : simint(0,CACHE_LEVEL){         WR(); value=s;       }
  c_int(const simint& s) : simint(0,CACHE_LEVEL){ s.RD(); WR(); value=s.value; }
  c_int& operator=(const simint& s)             { s.RD(); WR(); value=s.value; return *this; }
};
```

(c) Class `c_int`

```
c_int a,b,c,d;
a=b+c+d;
```

(d) Code example

```
b.RD() c.RD() tmp0.WR() tmp0.RD() d.RD() tmp1.WR() tmp1.RD() a.WR()
```

(e) Calls to `RD()` and `WR()`

Figure 1: Implementation of integer simulating classes

interface to remove these redundant references. This method delays the generation of a write reference to the simulator until the next reference is made. In case this reference is a read of the same register within an assignment operator, the write/read combination is redundant and hence can be skipped. Likewise, if this reference is to another variable, or if it is a read outside an assignment operator, the delayed write reference is not redundant.

Similar classes were written for `simdouble` and `simfloat`. Multiple inclusion of the source code using appropriate macro definitions avoids code duplication. The total interface as described above consists of approximately 1000 lines.

The definition of the overloaded logical operators `&&` and `||` does not agree with the behavior of the corresponding built-in operators. The built-in operator `&&` does not evaluate its second argument, if the first argument evaluates to true. There is no way to mimic this approach with the following overloaded operator

```
operator&&(const simint& a, const simint& b)
```

Would overloading this operator be used, it would invalidate program semantics in case the application code contains constructs like the one below:

```
int valid[N];
int n=0;
while ((n < N) && valid[n]) n++;
```

However, we circumvented this problem by explicit casting the operands of type `simint` to `int` and the integer result back to a temporary `simint`. Hence, every use of this operator (viz. `f1 && f2`) is modified into a form where explicit casting is performed (viz. `(simint(((int) f1) && ((int) f2)))`). A similar procedure is applied to the logical operator `||`.

3.2 Pointers and arrays

So far, only scalar variables have been discussed. Other types of variables, including pointers, arrays, and structures, are more difficult to deal with. There is no simple solution to treat pointers in a similar manner as scalar variables. However, since our current research is heavily focused on sparse matrix computations, and most sparse matrix application codes are written in Fortran, we did not consider the use of more advanced features like pointers and multi-dimensional arrays.

Although it is possible to declare arrays of `c_int`, this results in many independent calls to `memory_allocate()`. This may scatter the elements over the memory, while arrays are usually stored in one contiguous piece of memory. Therefore, we enforce that before an array initialization occurs, the memory allocator is notified about the fact that the subsequent `size` element initializations should be treated as elements of the same array. In this manner, the memory allocator can assure that these elements will be stored in contiguous memory.

Figure 2 shows class `array` which 'captures' one-dimensional array initializations. References to array elements are captured by the overloaded `operator[]`, which applies both to left-hand-side and right-hand-side usage of array elements.

```
class int_array {
  c_int *pnt;
public:
  int_array(int s)
  { if (s != 0) {
      inform_memory_allocator(s, sizeof(int));
      pnt=new c_int[s];
    } else {
      pnt=NULL;
    };
  }
  ~int_array() { delete[] pnt;}
  c_int& operator[](int el) const { return pnt[el]; }
}

int_array A(10); // simulates int A[10];
A[3] = 4;
```

Figure 2: Implementation and use of array simulation

3.3 Identifying code fragments

Obviously, we have to identify every code fragment for which separate results have to be gathered. This identification is performed using the constructor/destructor facility offered to us by C++, as illustrated in figure 3. The first line of each code fragment consists of an initialization of an automatic variable, called `dummy`. Different identifiers, (for example, `FP` in figure 3(b)) are assigned to separate instances of this variable. The constructor belonging to the class `codename` of this object, notifies the simulator about the fact that a code fragment called `FP` is entered. Likewise, the destructor flags the exit of a code fragment. Figure 3(c) shows how simulation results can be obtained for each variable reference within a statement individually, by using temporary `float` variables `dv` and `rs`, whose use is *not* captured.

```
VAL[i] = VAL[i] - DENSEVAL[ind];
```

(a) Original code fragment

```
{ codename dummy(FP);
    VAL[i] = VAL[i] - DENSEVAL[ind];
}
```

(b) Code, to identify a statement.

```
{ float dv, rs;
  { codename dummy(FP1);
      dv = DENSEVAL[ind];
  }
  { codename dummy(FP2);
      rs = VAL[i] - dv;
  }
  { codename dummy(FP3);
      VAL[i] = rs;
  }
}
```

(c) Code, to identify each array reference

Figure 3: Instrumentation of code fragment

4 An example simulation

This section describes an example simulation run and shows a part of the output which is produced by the simulator.

4.1 Simulated architecture

The simulator models the memory hierarchy according to current state of the art and announced technology. Memory is either a banked memory or a Rambus[9] memory system. We use the cache-time analysis tool 'cacti' [22] to predict the access time of a cache, since access time changes with technology, size, line size, and associativity.

The architecture in this simulation example is a 3-level memory, consisting of standard main memory, and two caches, called L2-cache, and (on-chip) L1-cache. We present some of the reported data from the lowest memory level, a 16 kB direct-mapped cache, with write-allocate copy-back strategy, and 16 byte line size.

4.2 Application

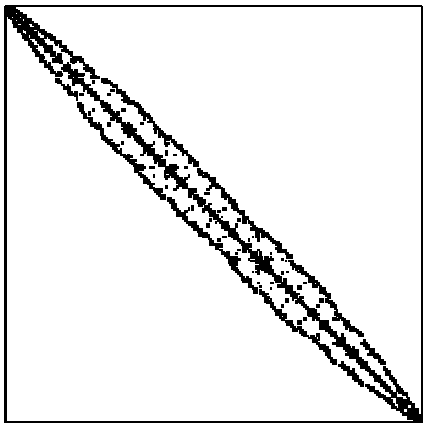
Various data structures have been developed to store sparse matrices efficiently [10, 12, 15, 18, 19]. The choice of the data structure for a sparse matrix is one of the major factors which determines performance. In many cases *sparse vector* storage is chosen for sparse matrices, because it is easy to handle, performs decently, and has low storage overhead [4, 6]. Row-wise storage has a high degree of spatial locality, resulting in a high cache hit rate. When the data structure is not static, the costs of element deletion and insertion is reduced if a linked list data structure is used. Linked list handling shows, however, very poor spatial locality. The simulation in this section uses linked lists to illustrate this effect.

A frequently used sparse matrix application is Cholesky decomposition. The application code we use is a column-wise Cholesky decomposition of a symmetric matrix from the Harwell-Boeing test set of matrices. This matrix, `bcsstk14.rsa`, has order 1806, and 32630 non-zero elements in the lower triangular part. To reduce the amount of fill-in the matrix is reordered with a minimum-degree algorithm. We used the minimum-degree implementation which is available in the software package SMMS93, written by F. Alvarado[2]. Figure 4 shows the matrix before and after reordering. The matrix is stored as a set of sparse vectors. Each sparse vector is stored as a linked list. For more details, we refer to [3, 4, 6, 14, 23].

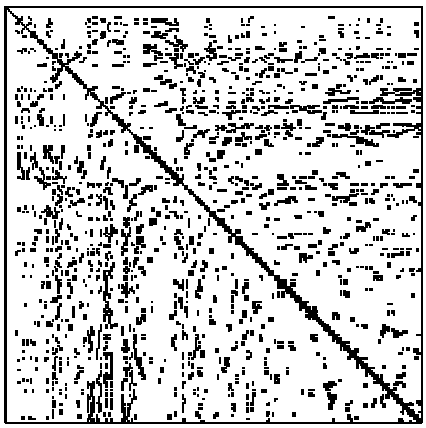
The application consists of subsequently initializing data structures, reading a sparse matrix, factorizing, and finally writing the output matrix. In between stages the memory system is flushed. Scalar variables are ignored in this example. Only references to array elements are traced.

4.3 Simulation results

Simulator results show that 60.2 % of the total simulated execution time is spent in a function called `SpMinDense`. For this function, we present part of the



(a) Matrix



(b) Minimum degree reordered matrix

Figure 4: Matrix `bcsstk14`

statistics, output by the simulator, for the L1-cache in figure 5. The first column presents the simplified code, with the instrumentation code removed. The second column shows the number of array accesses made to the corresponding line line in the source code. The third column shows the time spent for these array references. The fourth column shows the hit rate in L1-cache, while the measured average miss penalty is shown in the last column.

During memory accesses to consecutive locations, spatial locality is responsible for a hit rate of $\frac{100(N-1)}{N}$ %, where N is the number of data items per cache line. For the fourth source code line, `int ind = IND[i]`, this hit rate would be 75 %, since an integer is 4 bytes and the cache line size is 16 bytes. For the read reference to `VAL[i]`, which is an 8 byte data item, this hit rate would be 50 %. The real hit rate is, of course, higher due to temporal locality, or lower due to cache interferences. Measurements for these hit rates are 68.2

Code	Accesses (x1000)	Time spent (%)	Hit rate (%)	Miss penalty (ns)
`SpMinDense(int first);`				
`int i;`				
`for (i=first; i; i=LNK[i]) {`	6424	13.7	68.4	25.7
`  int ind = IND[i];`	6424	13.7	68.2	26.8
`  if (DENSEBIT[ind]) {`	6424	4.9	96.0	32.1
`  VAL[i] = VAL[i] - DENSEVAL[ind];`				
`  \  \  \_____`	4727	4.8	90.1	28.6
`  \  \_____`	4727	14.8	44.9	26.4
`  \_____`	4727	5.2	100.0	
`  DENSEBIT[ind] = false;`	4727	6.0	99.9	41.2
`  }`				
`}`				
Total (rd)	1123	100.0	74.2	26.6
(wr)			99.9	41.2

Figure 5: Simulation results L1-cache for `SpMinDense`

% and 44.9 %. Since part of this hit rate is due to temporal locality, we see that spatial locality is only partially useful for this application. This confirms that linked lists show poor spatial behavior.

5 Conclusions

In this paper we have presented a simulator which performs address reference trace capturing at source code level, using C++ classes and operator overloading. This technique aims at minimizing code rewriting. Rewriting is limited to the following two rewriting methods. Formal parameters, scalar variables and arrays are replaced by classes which simulate their behavior. Further, a peculiarity of the logical operators `||` and `&&` has to be imitated by rewriting all expressions involving these operators. In addition, if the source code fragment which generates an address reference has to identified, an instrumentation statement is required.

We have shown simulator output for an example sparse matrix application. The simulator presents a detailed analysis of the performance of all simulated memory levels, and relates this performance to specific code fragments or individual variable references. This makes it a useful tool for the evaluation of strategies to increase spatial or temporal locality, as well as for the evaluation of complete hierarchical memory systems.

Acknowledgements The authors would like to express their thanks to Aart Bik for his helpful comments.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley publishing company, 1986.

[2] Fernando L. Alvarado. *The Sparse Matrix Manipulation System: User and Reference Manual*. The University of Wisconsin, Madison, Wisconsin 53706, USA, May 18, 1993. SMMS93 software and documentation available from `ftp://eceserv0.ece.wisc.edu/pub/smms93`.

[3] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[4] Iain S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1990.

[5] Dennis Gannon et al. SIGMA II: A tool kit for building parallelizing compilers and performance analysis systems. Department of Computer Science, Indiana University, 1992.

[6] Alan George and Joseph W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., 1981.

[7] Jim Handy. *The Cache Memory Book*. Academic Press, 1993.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[9] Rambus Inc. Architectural overview, 1993.

[10] A. Jennings. A compact storage scheme for the solution of symmetric linear simultaneous equations. *The Computer Journal*, Volume 9:281–285, 1966.

[11] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. Technical Report CS-TR-3108, also UMIACS-TR-93-67, Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742, December 1993.

[12] S.J. Donal T. MacVeigh. Effect of data representation on cost of sparse matrix operations. *Acta Informatica*, 7:361–394, 1977.

[13] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1994.

[14] Sergio Pissanetsky. *Sparse Matrix Technology*. Academic Press, London, 1984.

[15] Udo W. Pooch and Al Nieder. A survey of indexing techniques for sparse matrices. *Computing Surveys*, 5(2):109–133, June 1973.

[16] Betty Prince. *Semiconductor Memories.* John Wiley & Sons, 2nd edition edition, 1991.

[17] Steven A. Przybylski. *Cache and memory hierarchy design : a performance-directed approach.* Morgan Kaufman Publishers, 1990.

[18] Youcef Saad and Harry A.G. Wijshoff. A benchmark package for sparse computations. In *Proceedings on International Supercomputing*, pages 500–509, 1988.

[19] Robert Schreiber. A new implementation of sparse gaussian elimination. *ACM Transactions on Mathematical Software*, 8(3):256–276, September 1982.

[20] Michael D. Smith. *Tracing with* pixie. Center for Integrated Systems, Stanford University, Stanford CA 94305-4070, April 1991. Version 1.1.

[21] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition edition, 1993.

[22] Steven J.E. Wilton and Norman P. Jouppi. An enhanced access and cycle time model for on-chip caches. WRL Research Report 93/5, Digital Western Research Laboratory, 250 University Avenue, Palo Alto, California 94301 USA, July 1993.

[23] Zahari Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publishers, 1991.