Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

Techniques for Improving the Cache Performance in Parallel Applications

Inbum Jung Joonwon Lee

Department of Computer Science,
Korea Advanced Institute of Science and Technology
373-1, Kusong-Dong, Yusung-Gu, Taejon 305-701, Korea
e-mail : {jib,joon}@camars.kaist.ac.kr

Abstract

The performance of parallel programs has suffered from memory access latencies induced by cache misses. In this paper, to investigate the causes of these cache misses, data parallel applications were executed on shared memory multiprocessors. The experiment showed that cache conflict misses occupied most of the cache misses. This was due to the cross interference among the grains composed of the part of data arrays. To address this problem, a tailored grain size was devised from the underlying cache architecture. Besides the interference among grains, cache performance was sensitive to the way data were constructed. To make data structure for exhibiting good cache behavior, a stride merging-arrays method was presented. This method entailed the reduction of cache conflict misses and reduced the useless prefetches in cache lines with multiple words. Simulation results show that these techniques may enhance the performance of parallel applications due to the improved cache performance.
Keywords: Parallel Application, Cache Misses, Grain Size, Prefetch, Merging

1 Introduction

Cache memory is very effective at increasing program performance since the locality is exploited enough by holding regions of recently referenced memory. However, cache misses result in memory access latencies in retrieving corresponding data from the main memory. During this penalty period, processors must stall until the data arrive. Though fast processors are used, since high cache miss-rates aggravate processors' utilization, applications that ignore caching effects may severely degrade performance.

The workload of parallel programs is partitioned into many grains based on granularity policies. The grains are distributed to processors working in parallel processing. However, the grain size that does not consider underlying cache memory may degrade cache performance since the interference among grains composed of data arrays may increase cache conflict misses. In this paper, data parallel programs are executed on shared memory multiprocessors, and the weight of the cache conflict misses is measured. And to reduce cache conflict misses due to the interference among grains, a tailored grain size is suggested, based on the number of processors and the cache size of a processor.

Besides the interference among grains, the cache misses are sensitive to the way data are constructed. The merging-arrays technique has been exploited for reducing cache conflict misses. However, this existing technique results in the useless data prefetches when applications have simultaneously referenced multiple arrays in the same dimension using the different indices. To address this problem, a stride merging-arrays method is devised, based on the grain size in parallel applications.

Our simulation results show that the tailored grain size may reduce the cache conflict misses through the decrease of the interference among grains. Also the stride merging-arrays method increases cache performance due to not only the reduced cache conflict misses but also the useful data prefetches. These two techniques improve the performance of simulated parallel applications.

This paper is structured as follows: in Section 2 related work is presented. In Section 3 the simulation environment is presented. In Section 4 benchmark applications are described and their performances are measured. In Section 5 a tailored grain size is suggested and evaluated through the simulation. In Section 6 a stride merging-arrays is suggested and evaluated through the simulation. Finally, the conclusion is presented in Section 7.

2 Related Works

Much research has investigated cache effects and bus traffic patterns under various parallel programs [1, 2, 3]. These studies reported that the data sharing characteristics affected the performance of cache coherency protocol, and also that the performance of coherent caches relied on the quantity of sharing data and its locality in the program. Eggers [4] researched the effects of cache line size on cache and bus performance using traces of application programs. This study

described both processor locality and false sharing as important factors in the system performance when increasing the cache line size. Torrellas [5] reported that the cache miss rate decreased when the application was constructed based on processor's cache characteristics, allocating locking variables to separate cache lines, and grouping data according to shared patterns.

Cache performance depends on the locality of references. If the sequence of addresses referenced by applications cannot all be stored in the cache, cache misses occur. It is not possible to build a cache that is large enough to hold the working sets of all possible applications, nor is it possible to code all applications to avoid all cache misses. However, several optimization techniques based on small source-code changes were used for improving cache performance [6, 7, 8]. Lam [6] experimented with a matrix multiplication using the blocked algorithm under various cache structures. This study calculated the optimal block size based on given cache parameters that could avoid self-address interference. But this research did not deal with cross address interference among grains. Lebeck [7] suggested a visualized tool called CPROF (Cache Profiling) that classified cache misses as compulsory, capacity, and conflict, and then provided cache performance information at the source line and data structure level. This tool improved performance by helping a programmer determine appropriate program transformations like merging-arrays, padding, aligning structures, and so forth.

In the previous studies, the impact of the grain size on parallel applications was not examined. Even if the sum of cache memories increases as more processors are employed, the anticipated cache performance is not achieved, since the cache performance is hurt by the interference among the grains and the mis-constructed data arrays.

3 Simulation Methodology

3.1 Simulated Multiprocessor

The simulation environment consists of a functional simulator that executes parallel applications and an architectural simulator that models the shared memory multiprocessor. An efficient program-driven simulator, MINT(Mips INTerpreter)[9] is used as a functional simulator. We construct an architectural simulator based on a multiprocessor with a bus-based structure. Each processor is assumed to be a RISC processor with the same cache size and each instruction is executed in a single cycle except the memory reference.

3.2 Cache Parameters and Timing Considerations

We assume that cache structure is 128 Kbytes direct-mapped with 16 bytes cache line size. A cache line is composed of 4 words. The simulated cache coherency protocol is write invalidation scheme[10]. On current microprocessors, the main memory access-time is about 80 ns, the clock rate is 250 Mhz(e.g. MIPS R10000, UltraSparc-II) and the system bus width is 128 bits. Table 1 shows timing values used in the cache coherency protocol based on these parameters including 1 address cycle and 1 bus operation cycle.

Table 1: Timing Parameters for Cache Coherency Protocol

Events (operations)	Penalties (cycles)
A write on a shared line (invalidate signal)	3
A cache miss (The missed line is supplied by an another cache)	7
A cache miss (The missed line is supplied by the main memory)	22

4 Benchmark Applications and Their Performance

4.1 Benchmark Applications

The four data parallel applications that we have chosen are BMM(Blocked Matrix Multiplication), LU(LU Decomposition), FFT(Fast Fourier Transform) and BS(Bitonic Sorting) [11]. All of these applications are written in C and use the synchronization and sharing primitives provided by the SGI's parallel macros package. All programs are run with 8 processes on 8 processors. In our benchmark applications, a grain size is defined as the part of data arrays. We use the coarsest grain size that is achieved by dividing the workload of a program by the number of processors, since it incurs the least address interferences between grains occupying a processor. Table 2 shows the workloads of these benchmark programs. The partitioned grains are then executed by a static scheduling policy [12, 13]. The static scheduling policy designates grains for each process before a program is executed. Since the grains allocated to all processes are not changed until the program finishes, the data locality can be exploited.

Table 2: Benchmark Applications

Applications	Data items	Data size
LU	a 512×512 matrix	512 Kbytes
BMM	three 256×256 matrices	1.5 Mbytes
FFT	two arrays with 65,536 elements	1 Mbytes
BS	two arrays with 32,768 elements	512 Kbytes

4.2 Performance

Table 3 provides some statistics about the applications when each application is run individually with the simulation environments described above. As can be seen, all applications show that the cache miss times are given much weight in total execution time. The cache miss time indicates the

processor stall time due to cache misses. In particular, this experiment also illustrates that cache conflict misses occupy most of the cache misses in all applications.

Table 3: Statistics for Application Performance

Applications	Execution time ($\times 10^6 cycles$)	Cache miss time ($\times 10^6 cycles$)	Rate of cache conflict misses (%)
LU	48.7	20.1	58.1 %
BMM	83.1	20	94.9 %
FFT	171.4	130.2	98 %
BS	383.2	369.5	99 %

5 A Tailored Grain Size

5.1 Principle

To reduce the cache conflict misses induced by the cross interference both among grains, we suggest a tailored grain size on the basis of the number of processors and the cache size of a processor. This method is effective when the sum of cache memories exceeds the data size of a program, and it also assumes a cache indexed with virtual addresses and the same cache size for every processor. Figure 1 shows an example of the address alignments when a tailored grain size is applied at 4 processors. As shown in this figure, the grains occupying the same processor have distributed address spaces as much as the sum of a cache size and a tailored grain size. Thus, the cross interference among grains does not incur, so cache conflict misses are decreased.

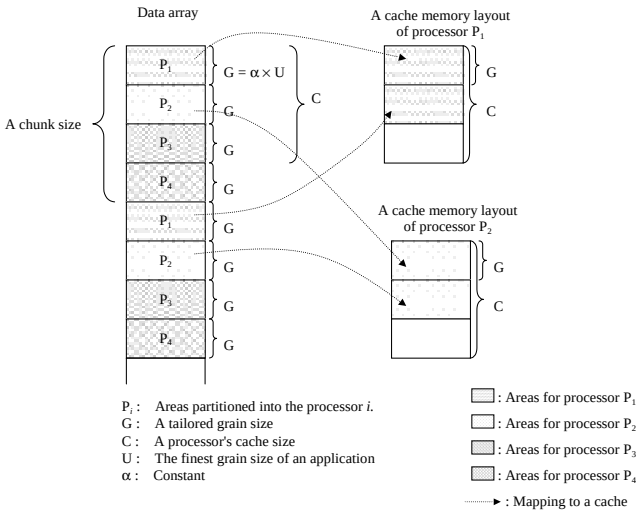


Figure 1: Address alignments using a tailored grain size.

A tailored grain size is composed of a multiple of the finest grain size of an application. A chunk size is defined by multiplying a tailored grain size by the number of processors. To apply this tailored grain size at several data arrays, the address space of each array must be a multiple of the chunk size. Thus, when doing the memory allocation, the array structure should be aligned. The following equations are driven from the Figure 1.

$$G = \alpha \times U, \quad G \times N = C + G$$

G : a tailored grain size
N : number of processors
C : a processor's cache size
U : a finest grain size
α : constant

From these equations, a tailored grain size is computed by

$$\alpha \times U = \lceil \frac{C}{(N-1)} \rceil$$

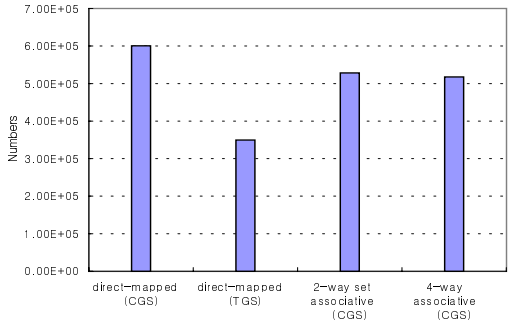
5.2 Performance

For our experiments, the tailored grain sizes were calculated for the LU and BMM programs since they had shown fixed memory reference patterns during their execution. To evaluate the performance of the tailored grain size, besides the performance of it, programs were run under a direct-mapped cache using the coarsest grain size and set associative caches using the coarsest grain size. Set associative caches can also decrease cache conflict misses, but they may suffer from the cost of increased hit times. Hill [14] found about a 10% difference in hit times for a direct-mapped cache versus a 2-way set associative cache and a 12% difference for a 4-way set associative cache. In these experiments, the cost of hit times for set associative caches was not reflected.

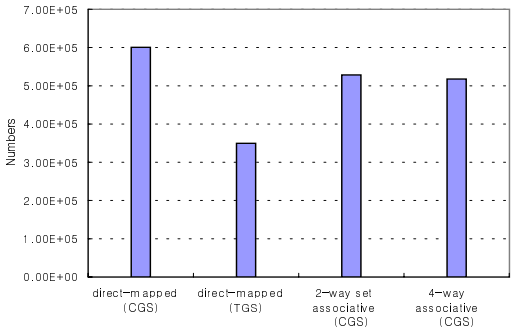
The LU program is run on 8 processors with 128 Kbytes direct-mapped cache and executes the decomposition for a 256×256 matrix. When the finest grain size is 1 row(2 Kbytes in size), the tailored grain size is then approximately 10 rows from above equation. Figure 2(a) shows that the tailored grain size presented the better performance than other cases. *TGS* denotes a tailored grain size and *CGS* denotes the coarsest grain size. The tailored grain size showed the better performance of about 41.8% than the coarsest grain size under the direct-mapped cache and about 24.1% than the 2-way set associative cache and about 23.5% than the 4-way set associative cache. This result was due to the reduced cache misses with the tailored grain size as shown in Figure 2(b). The tailored grain size induced the reduced cache misses of about 41.7% than the coarsest grain size under the direct-mapped cache and about 33.8% than the 2-way set associative cache and about 32.4% than the 4-way set associative cache.

The BMM program uses three matrices of 256×256 elements. It is executed on 12 processors with a 128 Kbytes direct-mapped cache, since a tailored grain size is more useful when the sum of cache memories exceeds the data size of

a program. When the finest grain size assumes a row composed of 1×1 blocks(2 Kbytes in size), the tailored grain size is approximately a 6×6 block. Figure 3(a) shows that the tailored grain size produced the better performance than other cases. The tailored grain size showed the better performance of about 33.4% than the coarsest grain size under the direct-mapped cache and about 7.3% than the 2-way set associative cache and about 6.7% than the 4-way set associative cache. As shown in Figure 3(b), the tailored grain size entailed the reduced cache misses of about 40.1% than the coarsest grain size under the direct-mapped cache and about 15.4% than the 2-way set associative cache and about 13.8% than the 4-way set associative cache.

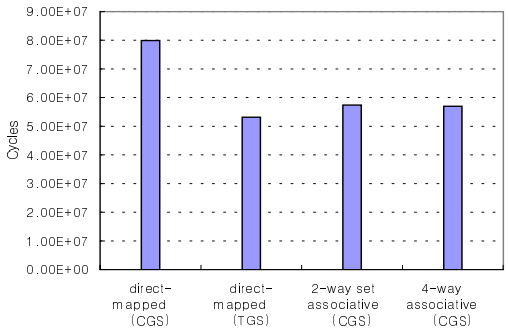


(b) Cache misses of LU program

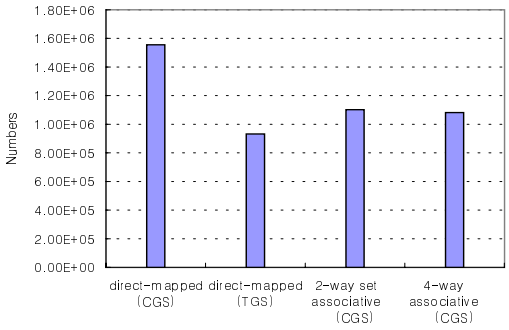


(b) Cache misses of LU program

Figure 2: Comparison of performance in LU program



(a) Execution time of BMM program



(b) Cache misses of BMM program

Figure 3: Comparison of performance in BMM program

6 A Stride Merging-Arrays Method

6.1 Principle

The merging-arrays technique has been exploited for reducing cache conflict misses. Figure 4 also shows examples for the non-merging arrays, the existing merging-arrays, and the stride merging-arrays methods in C programming language.

The existing merging-arrays technique has been discussed throughly in previous work[7, 8]. However, the existing technique is applicable only when programs have simultaneously referenced multiple arrays in the same dimension using the same indices. If the different indices are simultaneously used in applications, the useless prefetches can be induced in cache lines during the execution of grains. The reason is that the grain size of parallel applications is not considered when merging the arrays. In data parallel applications, the grains mean the part of data arrays and occupy the contiguous address space and request the elements of other arrays in the sequential address space as much as a grain size.

Figure 5 shows data placement on the cache memory with 4 words within a cache line, when the existing merging-arrays method is applied on the FFT. Since the FFT simultaneously accesses two arrays with different indices, if a processor executes a grain composed of memory references like $Y[0] = X[0] + X[4]$, $Y[1] = X[1] + X[5]$, $Y[2] = X[2] + X[6]$, and $Y[3] = X[3] + X[7]$, data from $Y[4]$ to $Y[7]$ loaded

```
/* non-merging arrays */
int   X[SIZE];
int   Y[SIZE];

/*   existing merging-arrays   */
struct   merge   {
            int   X:
            int   Y;
};
struct merge data[SIZE];
```

```
/*     stride merging-arrays     */
struct   merge   {
            int   X[GRAIN_SIZE];
            int   Y[GRAIN_SIZE];
};
 struct merge data[SIZE / GRAIN_SIZE];
```

Figure 4: Examples of merging arrays in C

into cache memory are unused data for computation. Thus, useless prefetches are induced.

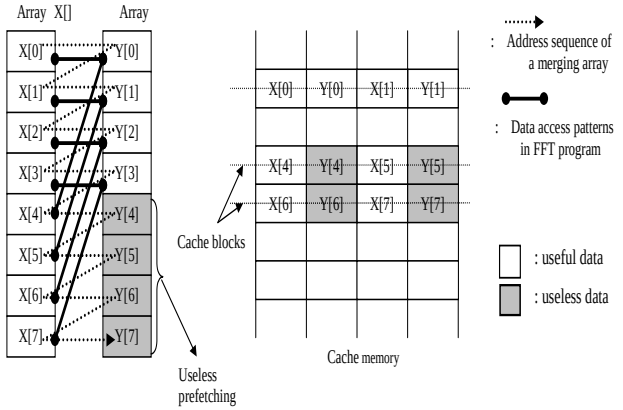


Figure 5: Address alignments using the existing merging-arrays

To address this problem, we devise a stride merging-arrays method based on the grain size in parallel applications. A grain size is regarded as a stride for merging the arrays. The stride merging-arrays method merges multiple arrays into a single array by exploiting a grain size as the merged unit. In this method, although the different indices are used, the useless prefetches do not occur in cache lines with multiple words since the accessed elements are exploited during the execution of grains.

Figure 6 shows data placement of the FFT program on the cache memory when the stride merging-arrays method is used. In this Figure, when a processor computes $Y[0]$, $Y[1]$, $Y[2]$, and $Y[3]$ as in the Figure 6, $X[4]$, $X[5]$, $X[6]$, and $X[7]$ are sequentially loaded into a cache line instead of $Y[4]$,$Y[5]$,$Y[6]$, and $Y[7]$ loaded in the existing merging-arrays method. Since these loaded data are used during the execution of a grain, they are useful prefetches. Thus, using the stride merging-arrays method, besides the reduction of cache conflict misses, the useful prefetches substantially improve cache performance.

6.2 Performance

To evaluate the stride merging-array method, we executed the FFT and BS programs on the environment described in Section 3. Table 4 shows the performances of applications achieved by the stride merging-arrays and the existing merging-arrays. In the FFT, the stride merging-arrays method represents an improved performance of about 12.2% with reduced cache misses of about 28.8% as compared with the existing merging-arrays method. In the BS, the stride merging-arrays method also shows an improved performance of about 12.3% with reduced cache misses of about 22.1%, as compared with the existing merging-arrays method.

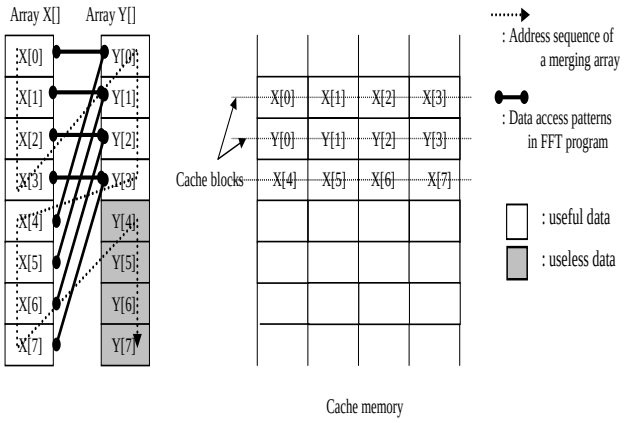


Figure 6: Address alignments using the stride merging-arrays

Table 4: Performance of two merging-arrays methods

Applications	Existing Merging		Stride Merging	
	Execution Time	Cache Misses	Execution Time	Cache Misses
FFT	65.3	6.3	57.2	4.5
BS	25.4	1.8	22.1	1.4

(Execution Time : $\times 10^6$ cycles , Cache Misses : $\times 10^5$ numbers)

7 Conclusion

In this paper, we studied the cache behavior that arises when parallel applications had been run on shared memory multiprocessors. We applied the coarsest grain size to our benchmark applications and then executed the grains on a static scheduling policy to exploit the cache locality. In our benchmark applications, the wasted time due to the cache misses occupied most of the elapse time, even if the sum of processor caches approached or exceeded the size of the program data. In particular, a great portion of the cache miss resulted from the cache conflict misses due to the cross interference among the grains occupying the same processor.

To address this problem, a tailored grain size was devised based on the underlying cache architecture. This technique decreased the cache conflict misses due to the reduction of the interference among the grains. When applying the tailored grain size to the LU and the BMM programs, it decreased about 41.7% of the cache misses in LU program and 40.1% of those in BMM Program as compared with those achieved by the coarsest grain size. These reduced cache misses resulted in a performance improvement of about 33.3% for the LU program and about 33.5% for the BMM program. In particular, comparing with set associative caches, the tailored grain size using the direct-mapped cache resulted in a better performance than that of the set associative caches, due to its lower cache miss rates. This was due

to the fact that the tailored grain size was more effective in reducing cache conflict misses than set associative caches.

To reduce the cache misses, a merging-arrays technique had been used. However, the existing technique produced the useless prefetches in cache lines since the grain size of parallel applications was not considered when merging the arrays. To address this problem, we suggested a stride merging-arrays technique that improved the prefetching effects by considering the stride of the indices on the merged arrays. The stride merging-arrays method increased the cache utilization, due to not only the reduction of cache conflict misses, but also to the useful data prefetches. The stride merging-arrays was applied to both the FFT program and the BS program. In the FFT program, the cache misses were decreased about 28.8% as compared with the existing merging-arrays, and therefore these caching effects resulted in the improved performance of about 12.2%. In the BS program, the stride merging-arrays method also produced the reduced cache miss of about 22.1%, and the improved performance of about 12.3%.

References

[1] A. Agarwal and A. Gupta. Memory-reference Characteristics of Multiprocessor Applications under Mach. In *In Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 215–225, May 1988.

[2] S.J. Eggers and R.H. Katz. A Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation. In *In proceedings of the 15th International Symposium on Computer Architecture*, pages 373–382, May 1988.

[3] A. Gupta and W.D. Weber. *Analysis of Cache Invalidation Patterns in Shared Memory Multiprocessors, In Cache and Interconnect Architectures in Multiprocessors*. Kulwer Academic Publishers, 1990.

[4] S.J. Eggers and R.H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *In Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 256–270, April 1989.

[5] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In *In Proceeding of 1990 International Conference on parallel Processing*, pages 266–270, August 1990.

[6] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[7] A. R. Lebeck and D. A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, October 1994.

[8] D. A. Patterson and J. L. Hennessy. *Computer A Quantitative Approach, 2nd Ed.* MORGAN KAUFMANN PUBLISHERS, INC., 1996.

[9] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *In Proceeding of 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems(MASCOTS)*, pages 201–207, January 1994.

[10] J. Archibald and J-L. Baer. Cache coherence protocols : Evaluation Using a Multiprocessors Simulation Model. *ACM Transaction on Computer Systems*, 4(4):273–298, November 1986.

[11] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing(Design and Analysis of Algorithms)*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[12] Y. C. Chung and S. Ranka. Applications and Performance Analysis of a Compile-time Optimization Approach for List Scheduling Algorithms on Shared Memory Multiprocessors. In *In Supercomputer 1992*, November 1992.

[13] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors. In *The 6th ACM International Conference on Supercomputing*, July 1992.

[14] M. D. Hill. A case for direct mapped caches. *Computer*, 21(12):25–40, December 1988.