A Hybrid Code Compression Technique using Bitmask and Prefix Encoding with Enhanced Dictionary Selection

Syed Imtiaz Haider and Leyla Nazhandali
Virginia Polytechnic Institute and State University
302 Whittemore , Blacksburg, VA 24061
{syedh, leyla}@vt.edu

ABSTRACT

Memory is one of the most significant detrimental factors in increasing the cost and area of embedded systems, especially as semiconductor technology scales down. Code compression techniques have been employed to reduce the memory requirement of the system without sacrificing its functionality. Bitmask-based code compression has been demonstrated to be a successful technique that produces low compression ratios while having a fast and simple decompression engine. However, the current approach requires dictionary sizes of +16K bytes to produce acceptable results, adding significant overhead to the system. In this paper, we develop a new hybrid encoding method that combines the traditional bitmask-based encoding and prefix-based Huffman encoding as well as a new dictionary selection technique based on a non-greedy algorithm. The combination of these two new methods reduces the compression ratio by 9-20% and performs well with small dictionary sizes.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: Data compaction and compression

General Terms

Algorithms, Design

Keywords

Code Compression, Bitmask, Embedded Systems

1. INTRODUCTION

The demand for ultra-portable, high-performance, and low-power embedded systems is at an all time high with more than 98% of all programmable processors running in embedded mode. Today's cell phones, for example, handle entertainment/media functions, communication duties, and office tasks on relatively meager hardware resources. As more sophisticated functionalities are expected from these systems, their memory requirement will grow. Memory structures, however, require too much area, are too costly, and consume a significant amount of the energy budget to be generously used in embedded systems[1, 2]. As semiconductor technology scales down, the static energy consumption of memory structures becomes dominant. Therefore, the size of the memory has a much greater impact on the battery life of these systems [3].

To address this issue, code compression techniques have been employed to reduce the memory requirement of the system without sacrificing its functionality. In addition to evaluating a technique on compression ratio, one needs also consider the following for embedded systems: 1) placement of decompression engine, 2) the area overhead, and 3) performance hit caused by decompression latency.

There are two types of decompression engine designs: pre-cache and post-cache designs. In pre-cache designs, the engine sits between the processor and the main memory, whereas in post-cache designs it is placed between the cache and the processor core [4]. In a post-cache design, the system cache stores compressed instructions. A post-cache design can reduce the energy consumption of the system by employing a smaller cache and reduce traffic between main memory and the processor. Post-cache designs, however, need to be fast to keep pace with a processor core. Unlike pre-cache designs, there are no latencies for the decompression engine to exploit. The complexity and speed of a decompression engine is the major factor that can prevent a compression method from being post-cache compatible.

Dictionary-based compression methods are one of the best candidates for post-cache designs as they have simple and fast decompression mechanisms. Among these, bitmask-based schemes provide low compression ratios compared to simpler frequency-based techniques [5, 6, 7]. In addition to compressing highly frequent instructions, these methods compress many instructions that are not stored in the dictionary by saving a reference to a close-by dictionary entry (in terms of Hamming distance) and some bit toggling information.

Even though the current bitmask-based compression algorithms achieve compression ratios below 60%, they require large dictionary sizes of at least 4k-8k entries. With 32-bit instruction lengths, this means 16K to 32K bytes of static memory is needed just to store the dictionary. This is a substantial memory requirement and significantly increases

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

the cost of the processor, given the fact that most embedded processors have cache sizes in the same order. Adding a 32K-byte dictionary would considerably increases the power consumption of the core because leakage power is the dominant component of power loss in semiconductor technology today. Consequently, memory structures such as caches and dictionaries add a sizable power consumption overhead to the core regardless of their low transistor activity [3].

The purpose of this work is to design a compression scheme that 1) has a small and fast decompression engine, 2) requires a small dictionary, and 3) produces low compression ratios.

This paper explores the landscape of bitmask-based code compression. We carefully study the performance of the state-of-the-art implementation and present a detailed analysis of its strengths and limitations. Drawing upon our leanings, we develop a hybrid encoding method that combines traditional bitmask-based dictionary encoding with Huffman encoding and a new dictionary selection method based on a non-greedy algorithm.

Our results show that the combination of these two new methods reduces the compression ratio by 9-20% with larger improvements seen for smaller dictionary sizes.

2. BITMASK-BASED DICTIONARY COMPRESSION

A general dictionary compression scheme starts by creating a dictionary of a small number of instructions. The dictionary instructions are then replaced in the original code with indices to the dictionary. The number of bits used to represent the index is significantly fewer than the bits in the original instruction, resulting in the compression of the code. A frequency-based dictionary compression scheme simply picks the most frequent instructions as the dictionary entries.

Bitmask-based compression takes the frequency-based compression one step further by leveraging the short hamming distance between some of the instructions. There are two types of compressed instructions: dictionary entry (DE) instructions and dictionary children (DC) instructions. Figure 1 shows the physical construction of the different types of instructions as well as an example of bitmask compression. A DE instruction has its original instruction code stored in the dictionary. When compressed, this instruction only needs to reference the index of corresponding dictionary entry. A DC instruction depends on a different instruction, namely a DE, to be coded. When compressed, it contains a reference to the index of the DE instruction along with some bits that specify the bitmask required to recover the instruction from its associated DE. The C-bit specifies whether an instruction is compressed or not. The C-bit is followed by bits representing the dictionary index in compressed instructions. The number of these bits depends on the dictionary size. The mask code decides the number and type of masks that should be used to process the dictionary entry in order to retrieve the original instruction. In DC instructions they contain the information about the number and type of the masks required. The mask info bits contain more detailed information about the masks including their location and the actual bit toggles needed. The bit toggling information is kept as a string of 1's and 0's, where each 1 (0) means that specific bit in the DE should (not) be toggled.

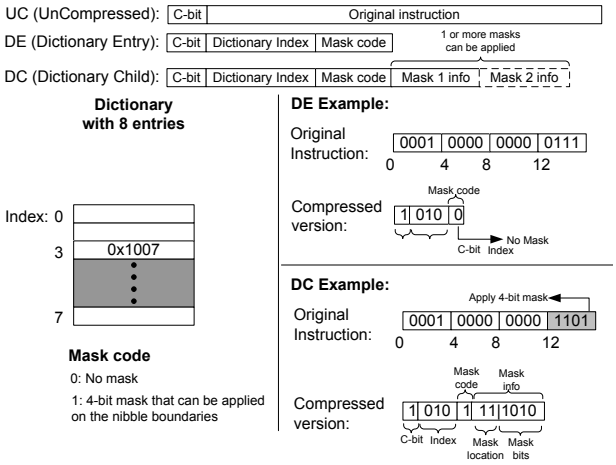

Figure 1: Instruction types and example of bitmask compression.

When a decompression system encounters a DE, it accesses the dictionary and retrieves the desired instruction specified by the dictionary index. A DC also has an index value; after the decompression system retrieves the contents of a dictionary entry it then proceeds to toggle bits as specified by the mask data in the DC (performed by an xor operation). For an uncompressed instruction the decompression agent just strips the first bit off the UC.

3. DICTIONARY SELECTION ALGORITHM

In a frequency-based dictionary compression, the selection of the dictionary is a straightforward task, as it picks the most frequent instructions. The same task, however, becomes an NP-hard problem in a bitmask-based version. That is because one has to balance and incorporate instruction frequency information and instruction coverage information to select good entries.

In the presence of several mask options, a certain instruction can be matched with numerous instructions at possibly different mask costs. The algorithm tries to select the dictionary entries so that many other instructions are mapped efficiently to the same dictionary entry resulting in shorter length for many instructions. In order to solve this problem, a graph is populated with nodes each representing a unique instruction in the binary. Given a preselected set of masks, an edge is created between two nodes if it is possible to match them using at least one of the masks. If more than one mask type can be used to match two instructions, the one with the minimum cost, i.e. shortest mask info bits is selected. Figure 2 presents the flowchart of dictionary selection algorithm for the latest work on bitmask-based compression method [7].

Although the algorithm shown in Figure 2 provides a fast and easy way to pick the dictionary entries, it fails in two aspects: First, the selection of entries round by round can be detrimental to future selections. If, for example two nodes with very high frequencies are connected to each other, selection of one as a dictionary entry will remove the other one as a dictionary child, even though that could have been an excellent dictionary entry itself. To alleviate this problem, the Seong algorithm employs a threshold for keeping some children with high frequencies in the graph for future

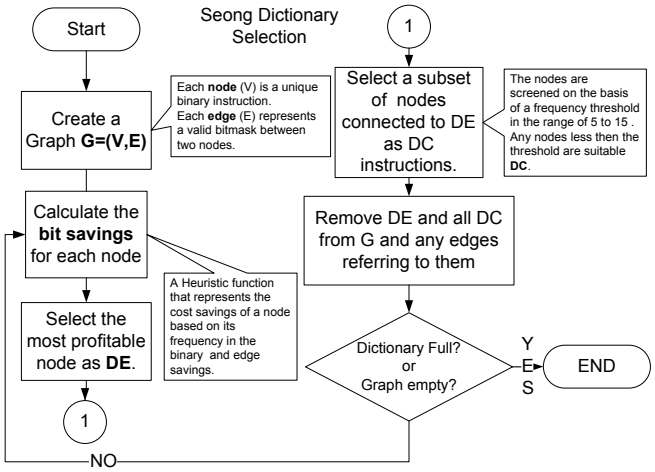

Figure 2: The original bitmask-based dictionary selection.

selection rounds. However, this may have an adverse effect because the threshold used to filter out possible dictionary entry candidates (refer to Figure 2), at times, may disallow too many instructions from becoming dictionary children. This is best shown in Figure 3. It presents the distribution of different types of instructions over instruction frequency for the gsm-toast application with the dictionary size of 512 entries. We see that the Seong selection scheme leaves a lot of instructions uncompressed compared to our new selection technique, which will be described shortly.

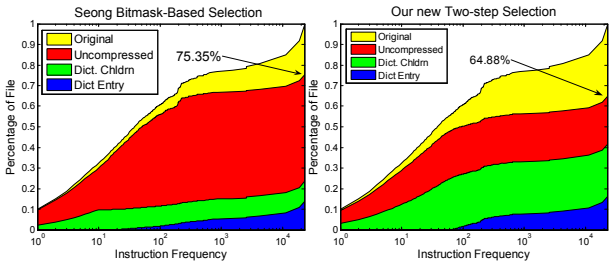

Figure 3: CDF of instructions in the original uncompressed file over frequency.

The second problem observed regarding the aforementioned algorithm is that it is not possible to associate a certain DC node with a specific DE node that results in maximum savings. This issue is shown with an example in Figure 4. The framework of regular selection does not pair Node C with the best DE instruction.

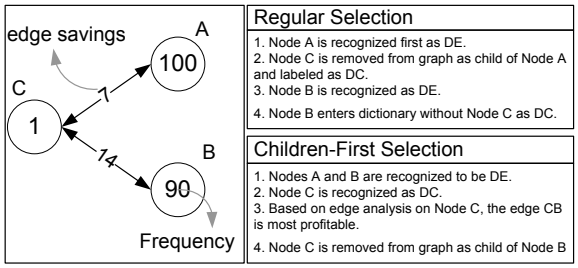

Figure 4: Example of children-first optimization.

In order to alleviate the problems described above, we propose a two-step dictionary selection with a specialized optimization. From our analysis of binary codes from mibench [8], we noticed that there are always instructions of high frequency that are profitable dictionary entries based on just frequency.

Given this observation, we can direct the algorithm to pick all these highly frequent instructions together, making sure none of them will be omitted in a greedy round-by-round approach. The selection of these nodes constitutes the backbone of the first part of our dictionary selection algorithm as shown in Figure 5.

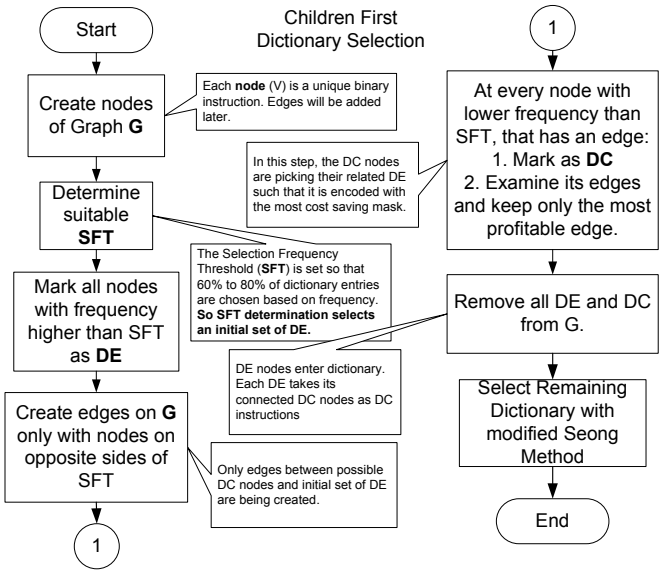

Figure 5: Our proposed bitmask-based dictionary selection.

In this algorithm, we need to first determine the minimum frequency that allows a node to be picked as DE in the first step. We call this value *Selection Frequency Threshold* or SFT. We have found that the SFT selection can be automated based on dictionary size. In order to achieve minimum compression ratio, our experiments show that the SFT should be selected such that a certain percentage of dictionary is selected in the first step. This percentage is very close across different benchmarks and depends on dictionary size only. In this algorithm, we pick the SFT so that 60%, 63%, 70% or 77% of the dictionary is filled in the first step for dictionary sizes of 512, 1024, 2048 or 4096 respectively. The next step is to create edges between nodes on opposite sides of the SFT. Note that all nodes above the SFT (i.e. with frequency higher than the SFT) are certain DE nodes and the nodes below the SFT are possible candidates to be their children. Since, a possible DC may be connected to more than one DE above the SFT, it gets to choose the best DE with maximum saving (i.e. child picks parent). We call this approach *children-first* optimization. In the final step of part 1 selection, the guaranteed DE nodes and their associated DC nodes are removed from the graph.

The remaining entries in the dictionary are selected in the second part. This step is very similar to the regular bitmask-based dictionary selection. However, there is no thresholding done when picking the DC nodes of a newly selected DE, as

it does not prove to be profitable anymore after completion of part 1.

This two-step children-first selection scheme eliminates the adverse affects of the greedy algorithm and applying thresholds when selecting children. Furthermore, it maps DC instructions to the best possible DE, resulting in an improved compression ratio. In the following section, we present our new hybrid encoding method in addition to a flexible mask selection approach.

4. FLEXIBLE MASK SELECTION AND A HYBRID ENCODING METHOD

In this section, we present flexible mask selection and a different encoding for masks. Previous work [7] studied various types of mask sizes and configurations. Fixed placement masks are proved to be the most profitable type of masks. A 4-bit fixed placement mask (4f), for example, can only be applied at nibble boundaries and a (2f) mask can be applied at half-nibble boundaries. Sliding masks, which can be applied anywhere on an instruction word, are not very profitable because they require more bits to store location information. The previous mask selection method consists of first picking a set of two masks and then the resulting subsets of this selection as possible mask configurations. The mask codes are then encoded as a standard 2-bit code.

We observe two limitations with this method: First, it is not flexible in terms of mask selection and confines itself to different combinations made out of two initially selected masks. We call this type of mask selection, *subset* mask selection. Second, it encodes all mask codes in equal length regardless of their distribution. We believe by providing flexibility both in the mask selection and mask encoding, the compression ratio can be improved significantly.

We allow the use of flexible non-subset mask selections such as: no-mask, 4f, 8f, (2f, 4f). Note that three different masks, namely 2f, 4f and 8f are used to create this selection. In previous work the 8f mask was regarded as a costly mask pattern and discarded from consideration. The reason behind this was *subset* selection forced combination of 8f with another mask, increasing the total mask cost beyond what can be acceptable. However, we found that many two mask patterns could be replaced with the 8f pattern, reducing the compressed instruction's length. Table 1 shows the costs of various masks we tried. Our analysis of different mask selections is presented in Section 5.

Table 1: Mask Costs

Mask	Location # of bits	Mask # of bits	Total # of bits
1s	5	1	6
2f	4	2	6
4f	3	4	7
8f	2	8	10
2f,4f	7	6	13
4f,4f	6	8	14

The bitmask instruction formats presented in Section 2 have three fixed-length sections for DE and DC instructions: c-bit, dictionary index, and mask code. DC instructions have additional bits for mask information. In all bitmask compressed files DE encoded instructions are the majority and contribute the most to the compression ratio because they have the shortest length. Figure 6 shows the distribution of compressed instructions in a sample program, rawcaudio, in terms of the mask configuration they use. As seen, DE instructions have the largest majority.

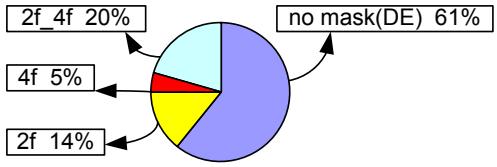

Figure 6: The breakdown of compressed instructions based on their mask configuration.

Huffman prefix encoding of the mask code is a simple and elegant way to improve compression ratio. The mask code for a DE instruction gets reduced to 1 bit. Some DC instructions do incur additional cost because their mask codes become longer, but the DE savings outweigh those costs. This hybrid method of combining prefix encoding of the mask code with regular encoding of the dictionary indices, is able to leverage the advantages of Huffman encoding without inheriting any of the weaknesses it would incur if entire instructions were Huffman encoded. Huffman encoding is most efficient when there are only a few symbols to encode, because codes are short and decoding is very simple.

The decompression engine required for this scheme is not any more complex than the one needed for the regular bitmask approach. The one-cycle engine modified by Seong in [6] from Lekatsas's design in [9] needs very little modification. Instead of decoding a fixed 2-bit mask code the new decompression engine will decode a prefix code of up to 3-bits, which is easily accomplished with little to no overhead.

5. RESULTS

Our experiments were conducted with 32-bit ARM binaries taken from Mibench [8]. In particular we used rawcaudio, rawdaudio, cjpeg, djpeg, gsm-toast, gsm-untoast, rijndael, basic-math, and susan. These applications are a good representation of software used across all types of embedded computing in the automotive, mobile phone, security, and media sectors.

Figure 7 shows the compression ratio of three different dictionary selection algorithms for four different sizes of dictionary. It can be seen that the two-step selection algorithm produces significantly better results compared to regular bitmask selection even without the children-first optimization. Also highlighted in Figure 7, is the inefficiency of the previous bitmask method at small dictionary sizes(512, 1024) where a simple frequency based dictionary children produces lower compression ratios.

Prefix encoding of the mask codes also improves compression ratio by sizable amounts. Small dictionaries benefit by 2-3% points and larger dictionaries benefit by 3-4% points; this is because dictionary entry instructions comprise a much larger majority of the compressed file when compressed with large dictionaries.

We experimented with different mask combinations that are not necessarily subsets of a two mask scheme done in previous work; and also studied the effects of increasing the number of mask configurations used. Note that in standard binary encoding of the masks, the number of combinations need to be powers of 2, otherwise, some codes will be wasted.

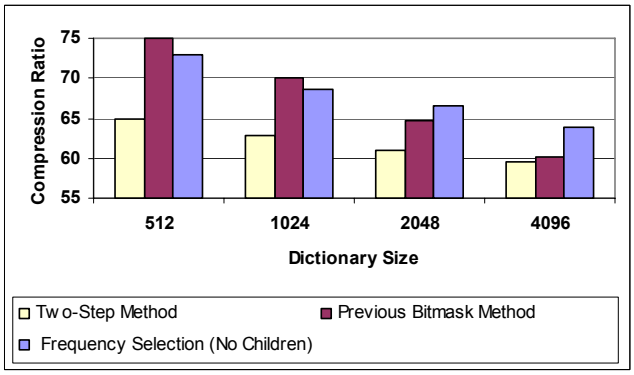

Figure 7: Compression ratio for different dictionary selection methods.

In Huffman encoding this is not the case. In any mask combination, one code is always reserved to represent "no mask" option. Therefore, in a 2-bit standard encoding, one is restricted to three mask configurations. Using prefix encoding allows the use of a few additional mask configurations. The limiting factor here is the complexity and speed of a prefix decoder, which increases as the number of symbols grows. The results from this exploration are shown in Figure 8. An interesting thing to notice is that every unconventional mask configuration outperforms the standard one shown in the first column. The bars for the standard mask code are missing for the last two combinations as they have more than 3 mask types, which along with the no-mask options exceeds 4 combinations, making it impossible to use a 2-bit mask code. The best results ultimately comes from the mask configuration shown in the second column: 4s, 8f, and 2f_4f. This configuration replaces the stand alone 2f mask with the 8f in a standard subset combination. The reason it outperforms the standard selection is that many dictionary children instructions using 2f_4f masks, use the the single 8f mask, which saves three bits per instruction.

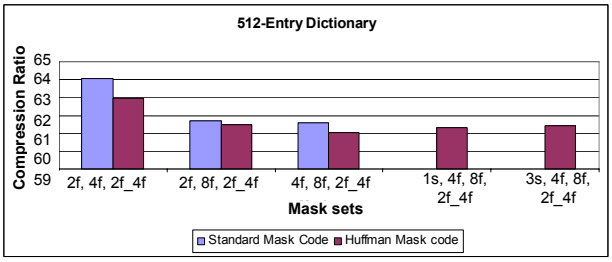

Figure 8: Compression ratios for different mask combinations.

Now with all the modifications made to the original bitmask-based compression framework, Figure 9 presents the final average compression ratios achieved over all the benchmarks. Our new method has better compression ratios and is very efficient at small dictionary sizes. The difference between using a 512 entry dictionary versus a 4096 entry dictionary with our method is less than 5% points compared to the 15% points difference with the original method.

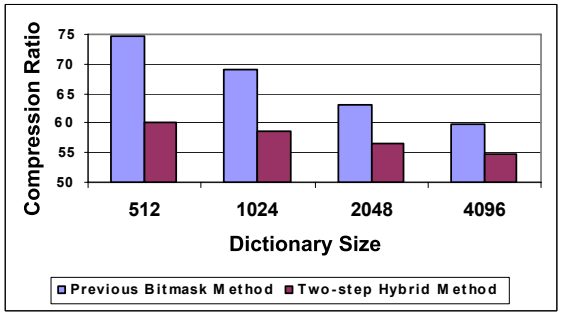

Figure 9: Compression ratios

6. CONCLUSIONS

We developed a new dictionary selection technique for bitmask-based dictionaries and a new flexible encoding scheme to store compressed instructions. We are able to achieve compression ratios previously seen in 4096-entry and 8092-entry dictionaries with dictionaries of 512 and 1024 entries. Overall we have reduced the compression ratios of small dictionaries by 20% and large dictionaries by 9%. Our compression technique is compatible with a 2-cycle post-cache decompression engine that extends the benefits of code compression to the internal instruction cache of a processor.

7. REFERENCES

[1] P. R. Panda. et al. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2), 2001.

[2] H. Shim. et al. Low-energy off-chip sdram memory systems for embedded applications. *Trans. on Embedded Computing Sys.*, 2(1):98–130, 2003.

[3] N. S. Kim. et al. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *MICRO 35: Proceedings of the 35th annual international symposium on Microarchitecture*, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[4] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. In *DAC '00: Proceedings of the 37th conference on Design automation*, New York, NY, USA, 2000. ACM Press.

[5] M. Ros and P. Sutton. A hamming distance based vliw/epic code compression technique. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, New York, NY, USA, 2004. ACM Press.

[6] S.-W. Seong and P. Mishra. A bitmask-based code compression technique for embedded systems. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, New York, NY, USA, 2006. ACM Press.

[7] S.-W. Seong and P. Mishra. An efficient code compression technique using application-aware bitmask and dictionary selection methods. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, Washington, DC, USA, 2007. IEEE Computer Society.

[8] M. R. Guthaus. et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[9] H. Lekatsas, J. Henkel, and V. Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *DAC '02: Proceedings of the 39th conference on Design automation*, New York, NY, USA, 2002. ACM Press.