Adaptive Filesystem Compression for Embedded Systems

Lan S. Bai† Haris Lekatsas‡ Robert P. Dick†
†l-bai,dickrp@northwestern.edu ‡lekatsas@vorras.com
Northwestern University Vorras Corporation
Evanston, IL 60208 Princeton, NJ 08540

***Abstract*—Embedded system secondary storage size is often constrained, yet storage demands are growing as a result of increasing application complexity and storage of personal data and multimedia files. Filesystem compression offers a solution. This paper formalizes the problem of automatic filesystem compression using multiple compression algorithms. The average latency of on-line file accesses is optimized under a constraint on filesystem capacity. Our solution is based on predictive control. Predicted latency implications are used to solve the file compression state selection problem using a multiple choice knapsack problem formulation. This approach is evaluated on filesystem traces and compared with other efficient heuristics. Our approach results in 34.1% reduction in file access latency compared to a straight-forward heuristic that decompresses frequently-accessed files and compresses least recently used files with more aggressive compression algorithms. It reduces file access latency by 67.7% compared to uniformly compressing files to the shallowest level required to meet storage capacity constraints.**

I. Introduction and Contributions

Embedded system size, cost, and power consumption are often constrained. This limits the amount of secondary storage such as flash devices or disk drives. As requirements for embedded systems become more complex, so does the software running on them. This increases file storage demands for both software and user data. File compression offers a method of increasing usable storage without increasing disk or flash device size. Most existing uses of filesystem compression are based on filesystem such as JFFS2 [12] or CRAMFS [2]. Such solutions use general-purpose compression algorithms and compress all filesystem contents regardless of usage patterns, degrading performance.

We propose to selectively compress files using a set of algorithms that trade off speed for depth of compression. This allows the benefits of compression to be maximized and performance degradation to be minimized, while honoring a storage constraint. We investigate the merits of automatically selecting compression algorithms for files based on prediction of their future usage. We propose a predictive control policy that estimates the latency imposed by each possible compression level for each file and minimizes the expected total latency subject to a filesystem size constraint. The cost minimization problem is formulated as a multiple-choice knapsack problem.

This paper makes the following contributions. First, it is the first to formulate the adaptive multiple-algorithm filesystem compression problem. Second, we propose an approach to solve this problem based on a predictive control policy.

II. Related Work

This section summarizes previous work on filesystem compression. Raita [9] proposed an automatic system for file compression that compresses inactive files with a single compression algorithm. Researchers at the Open Source Laboratory at the University of Szeged designed a block-based filesystem compression framework for JFFS2 that supports multiple compression algorithms [4]. They provide several compression modes to control compression algorithm selection. For example, if the mode is set to "size", the algorithm tentatively compresses the block with each available compression algorithm, picking the one permitting minimal size. As a result, selecting an algorithm for a block can consume a lot of time and energy. Moreover, the configuration cannot be automatically adjusted. E2compr [3] is an extension of the ext2 filesystem that incorporates user-selectable compression algorithms and requires manual reconfiguration. Cate [1] proposed using a daemon process to compress inactive files once per day. Their technique cannot deal with storage demands that exceed filesystem size between inactive file compression sweeps. Though support for multiple compression algorithms and automatic compression/decompression have been considered in previous work, nobody has implemented an automatic technique to adjust compression configurations on-the-fly according to storage demands while minimizing performance penalty.

This work was supported in part by NEC Laboratories America under the direction of Dr. Srimat Chakradhar and in part by the NSF under award CNS-0347941 and CSR-0720691. During much of the project, Dr. Lekatsas was affiliated with NEC Laboratories America.

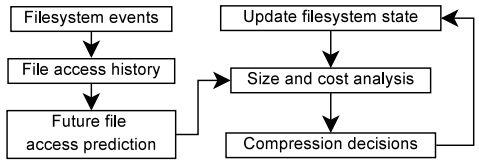

Fig. 1. Overview of adaptive filesystem compression.

III. Adaptive, Predictive Filesystem Compression

Adaptive filesystem compression changes the compression algorithms used for particular files depending on time-varying storage requirements and file access patterns. Generally, a filesystem initially contains few files. During use, the size can grow due to file creation and expansion of old files, and can decrease due to file deletion and truncation. The state of a filesystem is the mapping from available compression algorithms to the set of files it contains. Adaptive filesystem compression adjusts the effective total filesystem size to meet demands by compressing and decompressing files. When storage size requirements exceed filesystem size, it trades off performance for increased available space. Compression decisions are based on prediction of future access patterns based on previously-observed behavior in order to minimize average file access latency.

Figure 1 gives an overview of our adaptive, predictive filesystem compression framework. A history of file accesses is recorded and used to predict future file access patterns. Compression decisions are made based on these predictions and current filesystem state to minimize average access latency while meeting storage requirements. Filesystem state is updated according to compression decisions. In Figure 1, the loop including filesystem state and compression decisions indicates the interdependence of these steps and implies that compression decisions impact future compression decisions.

III.A. Problem Formulation

In this section, we formulate the offline version of the adaptive filesystem compression problem as an optimization problem. We then generalize to the on-line version of the problem.

Compression algorithms are classified by compression levels. *Compression ratio* is the compressed block size divided by the original block size. Deep compression levels correspond to low compression ratios, while shallow compression levels correspond to high compression ratios. We define the uncompressed state as compression at level zero, i.e., a compression ratio of one. The problem is to determine a sequence of filesystem states, with a change potentially occurring after each file access event. The objective is to minimize the average file access delay without exceeding the filesystem size constraint. We

TABLE I
DEFINITION OF $\tau(\epsilon)_a$

$\chi(\epsilon)$	θ^ϵ_γ	$\delta(\gamma, \epsilon-1)$	$\tau(\epsilon)_a$
r	0	0	$T_r \times r_\epsilon$
r	other than 0, 0		$(T \times S^\epsilon_\gamma \times R^\gamma_{\delta(\gamma,\epsilon-1)} + S^\epsilon_\gamma \times D_{\delta(\gamma,\epsilon-1)}) + (\theta^\epsilon_\gamma \times (S^\epsilon_\gamma \times C_{\delta(\gamma,\epsilon)} + S^\epsilon_\gamma \times R^\gamma_{\delta(\gamma,\epsilon)} \times T_w))$
w	0	0	$T_w \times r_\epsilon$
w	other than 0, 0		$(T \times S^\epsilon_\gamma \times R^\gamma_{\delta(\gamma,\epsilon-1)} + S^\epsilon_\gamma \times D_{\delta(\gamma,\epsilon-1)}) + (S^\epsilon_\gamma \times C_{\delta(\gamma,\epsilon)} + S^\epsilon_\gamma \times R^\gamma_{\delta(\gamma,\epsilon)} \times T_w)$
c	any	any	0
d	any	any	0

assume that compression/decompression speed (in bytes per second) of a compression algorithm is the same for all files.

Known:

1) F: set of all the files that ever exist in the filesystem.
2) B: absolute filesystem size.
3) E: set of all event times.
4) A: set of compression algorithms.
5) X: set of file operations. (read, write, create, delete) $X = \{r, w, c, d\}$.
6) S^ϵ_γ: original size of file γ after event ϵ. $\epsilon \in E$.
7) T_w: period of data writes to storage device. (s/byte)
8) T_r: period of data reads from storage device. (s/byte)
9) R^γ_α: compression ratio of compression algorithm α on file γ. $\alpha \in A$.
10) C_α: compression period (s/byte) of compressor α.
11) D_α: decompression period (s/byte) of compressor α.
12) ρ_ϵ: file accessed at event ϵ. $\rho_\epsilon \in F$.
13) χ_ϵ: file operation at event ϵ. $\chi(\epsilon) \in X$.
14) r_ϵ: requested data size of event ϵ.

Variables:

1) $\delta(\gamma, \epsilon)$: state of file γ after event ϵ. $\delta(\gamma, \epsilon) \in A$
2) θ^ϵ_γ: whether file γ changes state after event ϵ. θ^ϵ_γ is 1 if $\delta(\gamma, \epsilon) \neq \delta(\gamma, \epsilon - 1)$
3) $\tau(\epsilon)_o$: execution time of changing the states of files other than the accessed file.
4) $\tau(\epsilon)_a$: access time of the accessed file, including time of changing the state of the accessed file.

The method for computing $\tau(\epsilon)_a$ is shown in Table I. It is divided into six cases based on the previous compression state and whether the next state differ. We classify the cases to permit members of each case to share a cost function. Although we treat the uncompressed state as compression level 0, we make exceptions for estimating the cost at this level. If an uncompressed file is read without modifying its state, the access delay is merely the time required to transfer the requested data to memory. The second line shows the remaining cases for a file read operation. Either the file is compressed or the file's state is updated, resulting in the whole file being read into memory. If the file was compressed, it must also be decompressed. This corresponds to the first parenthesized term of the equation in the right column. The second part of the equation corresponds to the time required to update the file's state. It is zero when θ^ϵ_γ is zero, meaning that no state change occurs. The third row corresponds to writing to an uncompressed file without modifying its state, i.e., it is the data transfer time to the filesystem. In any other write request case, the file must be decompressed first and written back to storage even if its compression state does not change. When the contents of the file are modified, the whole file must be recompressed. Create (with zero size) and delete do not result in substantial data transfer so their access costs are zero. Delete changes the filesystem size requirement.

$\tau(\epsilon)_o$ is the sum of the costs of adjusting the states of other files immediately after event ϵ. The computation is based on the assumption that other files are not currently in memory. $\tau(\epsilon)_o$ includes the time of transferring the files to memory, decompressing, recompressing, and sending them back to secondary storage. The equation for $\tau(\epsilon)_o$ follows:

$$\tau(\epsilon)_o = \sum_{\gamma \neq \rho(\epsilon), \gamma \in F} \theta^\epsilon_\gamma \times (S^\epsilon_\gamma \times R^\gamma_{\delta(\gamma,\epsilon-1)} \times T_r + S^\epsilon_\gamma \times D_{\delta(\gamma,\epsilon-1)} + S^\epsilon_\gamma \times C_{\delta(\gamma,\epsilon)} + S^\epsilon_\gamma \times R^\gamma_{\delta(\gamma,\epsilon)} \times T_w)$$

Objective Function: The average access time of a given access sequence:

$$T = \frac{1}{|E|} \sum_{\epsilon \in E} (\tau(\epsilon)_a + \tau(\epsilon)_o)$$

Constraint: Secondary storage usage should be no more than the filesystem size B at all times.

$$\forall \epsilon \in E : \sum_{\gamma \in F} S^\epsilon_\gamma \times R^\gamma_{\delta(\gamma,\epsilon)} \leq B$$

The problem can be classified as a mixed integer non-linear programming problem with linear constraints. For a problem instance containing n files, m compression levels, and an event sequence of length k, there are $n \times m \times k$ variables and the solution space has a size of m^{nk}. Practical instances generally contain hundreds of files. Therefore, this problem is intractable for existing mixed integer programming solvers.

In the proceeding discussion, one complication was neglected. In reality, the filesystem compression problem must be solved on-line, without knowledge of future events.

IV. A PREDICTIVE CONTROL POLICY BASED ON COST ESTIMATION AND COST MINIMIZATION

Optimally solving the problem defined in Section III-A is hard because the objective function depends on events that occur in the future and also filesystem state changes made in the future. We propose to break this loop by decomposing the problem to two subproblems: cost estimation and compression level selection. Cost estimation determines an expected latency for each file at each possible compression level based on current file state and future file access pattern. Compression level selection chooses a compression level for each file to minimize the total expected latency over all files in the filesystem under the filesystem size constraint.

IV.A. Cost Estimation

In this section, we define a cost function to estimate the expected latency of setting a file to a particular compression level. The expected latency of a particular file state selection has two components. One is the performance penalty paid for updating the state of the file; it is zero if the state remains unchanged. The other is the latency of future accesses to the file starting from the updated compression state. Note that decompressing a file may require compressing other files to honor the size constraint. However, this cost function considers each file independently. The interference among different files will be handled in the solution to the second subproblem, in which size constraints are imposed so decompressing one file to a larger size leads to selecting a more aggressive compression algorithm for another file. The cost estimation function is defined as

$$C(\gamma, \delta) = modify\_cost(\gamma, \delta) + N \times (F_w \times write\_cost(\gamma, \delta) + F_r \times read\_cost(\gamma, \delta)) \quad (1)$$

F_w and F_r are write and read frequencies for file γ in the next N events. N is a constant that indicates how many events into the future are considered during optimization. It is important to distinguish read and write operations because they have different impacts on performance. Writing to a compressed file requires both decompression and compression; compression is usually slower than decompression. Therefore, it can be more beneficial to decompress

a file if it will soon be written many times and decompression may not be necessary if the file will instead be read many times. N must be carefully selected to accurately estimate the impact of future file accesses. Equation 1 implies that the filesystem state stays the same during the next N events. A very large value of N decreases the accuracy of this assumption because the filesystem state will have more opportunity to change. On the other hand, very small values of N decrease accuracy as a result of underestimating the impact of current decisions on future accesses. 500 was experimentally determined to be a good value for N.

IV.B. Prediction Model

We propose to use the read and write frequencies in a history window to predict F_r and F_w. Other prediction methods were considered and evaluated, e.g., prediction with a weighted history window or universal prediction model. Despite the ability of more sophisticated prediction models to capture higher order process patterns, they do not bring improvements in prediction accuracy for this application compared to the proposed method. Though this prediction model assumes the file access request is a 0th order process, it works well with very low prediction error on real file access traces as demonstrated in Section V. Selecting the history window size involves a trade-off between considering sufficient history and capturing time-varying behavior. 200 was experimentally determined to be a good window size.

IV.C. Cost Minimization

This section describes our method of selecting the compression states of all files in the filesystem to minimize the sum of expected latencies over all files under a constraint on used space. We formulate the problem as a multiple-choice knapsack problem (MCKP).

Let binary variable $x_{i,j}$ indicate whether to compress file i to compression level j. Since each file can only be saved in one state, the sum of $x_{i,j}$ for each file i is 1. Let $C_{i,j}$ be the expected latency of compressing file i to compression level j, and let $S_{i,j}$ be the effective size of file i at compression level j. The compression state selection problem follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in F} \sum_{j \in A} C_{i,j} \times x_{i,j} \\ \text{subject to} \quad & \sum_{i \in F} \sum_{j \in A} S_{i,j} \times x_{i,j} \leq B, \\ & x_{i,j} \in \{0,1\}, \ \forall i \in F \ \sum_{j} x_{i,j} = 1 \end{aligned} \tag{2}$$

The formulation in Equation 2 is very close to the MCKP problem. Given a number of sets, each containing multiple items, where each item is associated with a profit and a weight, the MCKP problem requires the selection of one item from each set. The selection is optimal when the total profit is maximized and the total weight of the selected items is below a constraint. A compression level selection problem instance can be converted to an MCKP problem instance by considering each potential compression level to be an item. The associated weight is the effective size of the file at this compression level. The associated profit is the reduction in expected latency compared to longest latency. For example, if C_{ij} is the expected latency for compressing file i to level j, then the corresponding profit P_{ij} is $\max\{C_{ij}, j \in A\} - C_{ij}$. MCKP is $\mathcal{NP}$-hard. Fortunately, this problem has received substantial attention and a number of algorithms were developed to solve it in pseudo-polynomial time [8, 10]. We adopted Pisinger's algorithm for the MCKP problem.

V. Experimental Results

We evaluate the proposed filesystem compression technique on a number of filesystem traces. First we describe our trace-driven simulation environment based on the processor and filesystem of a cellphone platform. We then describe several heuristics and present experimental results.

TABLE II
File Traces

Trace	Length	Number of files	Reads	Writes
1	3970	132	3540	430
2	5791	156	5329	462
3	4465	139	3872	593
4	5604	109	5263	341
5	6569	225	5760	809

TABLE III
Compressed Filesystem Access Times (s)

Trace	Size bound	Random Promote	Uniform	LRU Promote	F_MCKP	Oracle F_MCKP
1	0.6	12.51	7.83	5.09	4.28	4.26
1	0.5	22.69	7.83	5.09	4.62	4.57
1	0.4	34.90	17.16	11.75	6.47	6.17
2	0.6	23.21	28.62	17.09	16.23	16.73
2	0.5	39.17	28.62	17.12	16.30	16.81
2	0.4	120.57	60.90	37.74	22.83	21.55
3	0.6	20.12	16.66	5.84	4.23	4.22
3	0.5	32.61	16.66	6.67	5.14	4.66
3	0.4	50.99	34.41	14.24	10.79	6.01
4	0.6	31.02	24.19	12.02	8.32	7.98
4	0.5	41.35	49.15	26.00	12.20	9.90
4	0.4	100.55	49.15	44.28	21.35	16.53
5	0.6	26.62	42.61	13.00	7.62	7.64
5	0.5	51.89	81.67	27.83	11.46	10.37
5	0.4	110.43	81.67	54.82	49.43	40.58

V.A. Evaluation Environment / Experimental Setup

We used file access traces as our benchmarks. A trace is a sequence of filesystem operations. Each record contains information such as file path, file size, file operation, etc. Unfortunately, there are no publicly-available file traces collected from embedded systems. We therefore used trace data from DFSTrace [6]. We chose one set of their traces, mozart, which provide system-level information. The properties of the five parsed traces are shown in Table II. The five dominant file types appearing in real cellular phone filesystems were considered: executable, image, bitmap, configuration file, and English text. We assigned each file in the trace a type according to the file type distribution from the filesystem on a cellular phone prototype. To determine the initial compression levels of all files, we compressed all files to the shallowest level permitting the capacity restriction to be met. We considered three compression algorithms: LZO, GZIP, and Burrows-Wheeler transform (BWT). We tested their performances on an embedded system with a 400 MHz Intel XScale processor. The average compression ratio for each type of file was experimentally measured. Flash read and write rates were determined to be 9 MB/s and 6 MB/s respectively based on data given in flash memory datasheets [5, 11]. We wrote a program to parse the trace, implement the evaluated heuristics, and compute the average access latency. The original MCKP code [7] solved self-generated random test cases. We modified it to accept problem instances from our filesystem prediction infrastructure.

V.B. Accuracy of Frequency Prediction Technique

To evaluate the effectiveness of our file access frequency prediction method, we compute the absolute prediction error defined as

$$e = \frac{\sum_{\epsilon \in E, \gamma \in F} \left(|f_r(\gamma,\epsilon) - f_r\prime(\gamma,\epsilon)| + |f_w(\gamma,\epsilon) - f_w\prime(\gamma,\epsilon)| \right)}{N \times M}$$

$f_r(\gamma,\epsilon)$ and $f_w(\gamma,\epsilon)$ are the actual frequencies of reads and writes to file γ. $f_r\prime(\gamma,\epsilon)$ and $f_w\prime(\gamma,\epsilon)$ are the predicted values. N is number of events in each file trace. M is the number of files. As can be seen from the results in Table IV, the average error is 0.0045. This indicates that our prediction method is very accurate.

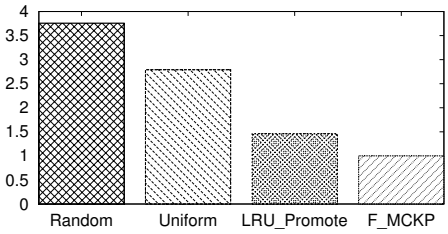

Fig. 2. Access latencies of heuristics relative to proposed technique.

TABLE IV
ABSOLUTE FREQUENCY PREDICTION ERROR

Trace number	1	2	3	4	5
Error	0.0053	0.0051	0.0048	0.0039	0.0032

V.C. Evaluation of Heuristics on Filesystem Access Traces

We compare the proposed heuristics with a base case in which a single compression algorithm is used for all files. We introduce three straight-forward heuristics, and the proposed efficient but low-latency technique, for filesystem compression with multiple compression levels. We also test our approach with an oracle cost estimator that knows future accesses to investigate the impact of prediction error on decision quality. The compared cases are listed below.

1) **Uniform:** Use the shallowest compression level necessary to honor the size constraint. All files are compressed.
2) **Random_Promote:** Always decompress the accessed file. If this would cause a storage space violation, randomly select other files and incrementally deepen their compression levels until the filesystem size constraint is met.
3) **LRU_Promote:** Decompress the accessed file if it has been used more frequently than a pre-defined threshold. Iteratively deepen the compression levels of LRU files until the filesystem size constraint is honored.
4) **F_MCKP:** Our approach. Choose file compression levels that minimize the expected access latency.
5) **Oracle_F_MCKP:** F_MCKP with oracle knowledge of the future file access sequence.

Table III shows the total access times for five traces with different compression approaches. The total access time is composed of time to (1) access files in the trace and (2) update file compression states. The performance penalty of the compression selection algorithm itself is not included. The second column indicates the ratio between storage size and total file size, it is a measure of the filesystem size constraint. The following conclusions can be derived:

1) **Oracle_F_MCKP** generally produces the best results. In three of the 15 cases, **F_MCKP** outperforms **Oracle_F_MCKP**. This may seem counterintuitive because **Oracle_F_MCKP** has no file access trace prediction error. However, **Oracle_F_MCKP** assumes that the filesystem state is constant during future accesses. This assumption does not hold in practice because filesystem state changes in the future depending on the decisions made by **Oracle_F_MCKP**. This error in filesystem state prediction can either be exaggerated or offset by inaccuracies in file access prediction introduced by **F_MCKP**. As a result, on rare occasions **F_MCKP** produces slightly better results than **Oracle_F_MCKP**. Despite its reliance on prediction, the total delay resulting from **F_MCKP** is always within $1.8\times$ that of **Oracle_F_MCKP**.
2) The worst results are generated by **Random_Promote** and **Uniform**, which ignore access patterns.
3) All methods supporting multiple compression levels but **Random_Promote** always beat **Uniform**. This demonstrates the potential to improve performance by using more than one compression algorithm.

Figure 2 summarizes the improvement of **F_MCKP** over the other three heuristics. **Oracle_F_MCKP** is not present in the figure since it requires knowledge of future events. **F_MCKP** is normalized to 1. Each entry gives the normalized geometric mean over all traces and filesystem constraint values shown in Table III. In addition to considering flash, we have also evaluated these cases with a disk-equipped embedded system. The results are similar, and are omitted due to the space constraint. Given that the energy consumption proportional to the total file access delay, the impact on energy consumption would be similar.

The proposed solution may impose some computational burden and may increase file access latency when implemented on moderate-performance embedded processors. We have tested the performance of the MCKP solver with problem instances generated from the file traces. The average duration is 1.19 ms on an 400 MHz Intel XScale PXA250 processor. The average frequency of calling the MCKP solver is 6.0% given a filesystem size constraint of 0.5. The average file access latency with **F_MCKP** is 1.81 ms. For the same trace and filesystem size constraint, the average file access latency with **Uniform** is 6.37 ms. Therefore, on a ARM-class embedded processor, the reduction in average access latency of our approach compared with uniformly compressing every file is $[6.37 - (1.81 + 1.19 \times 0.06)]/6.37 = 70.5\%$.

VI. CONCLUSIONS

Filesystem compression with adaptively controlled compression levels can permit good performance and increased usable filesystem size. We have formulated the problem of adaptive filesystem compression with multiple compression algorithms and proposed a solution to the on-line version of the problem based on access latency estimation and optimization. Compared with the alternative of compressing the entire filesystem with a single compression algorithm, our technique reduces average file access latency by 67.7%. Compared with a heuristic based on promoting LRU files to deeper compression levels, our technique reduces file access latency by 34.1%.

REFERENCES

[1] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 200–211, 1991.

[2] Cramfs: Cram a filesystem onto a small ROM. http://sourceforge.net/projects/cramfs.

[3] Transparent compression for the ext2 filesystem. http://e2compr.sourceforge.net.

[4] JFFS2 improvement project. http://www.inf.u-szeged.hu/jffs2/compression.php.

[5] Kingston flash memory. http://www.kingston.com/flash/.

[6] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software–Practice and Experience*, pages 705–736, 1996.

[7] David Pisinger. MCKP code. http://www.diku.dk/~pisinger/mcknap.c.

[8] David Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European J. of Operational Research*, pages 394–410, 1995.

[9] T. Raita. An automatic system for file compression. *The Computer J.*, 30(1):80–86, 1987.

[10] Prabhakant Sinha. The multiple-choice knapsack problem. *Operations Research*, 27(3), 1979.

[11] Toshiba flash memory. http://www.toshiba.com/taec/Catalog.

[12] David Woodhouse. JFFS: The journalling flash file system. In *Ottawa Linux Symp.* RedHat Inc., 2001.