Optimization Methods on Embedded System

Summary of the Ph.D. Dissertation

Ferenc Havasi

Supervisor:

Prof. Tibor Gyimóthy
Head of the Software Engineering Department

Ph.D. School in Computer Science
University of Szeged
Software Engineering Department

Szeged
2013

Introduction

In today's world, embedded systems have more and more applications. Cell phones, DVD players, MP3 players, GPS receivers, car electronics and similar devices are now an integral part of our lives. One of the main feature of these devices is that their resources are generally much more limited than the resources of the personal computers. So optimizing their software components is always a priority.

A good indication of the penetration rate of embedded systems is that even in 2002 just 2% of the microprocessors were used in personal computers, and the rest (98%) in embedded systems [15]. The economic and social impact of this on societies will increase, and new results may have great practical significance.

In my thesis, three key results are presented, namely one XML-related result and two flash file system optimizations:

1. **XML Semantic Extension and Compaction:** we designed a new metalanguage called SRML that allows one to define semantic rules for computing XML attributes. This method helps one to store XML files in a more compact form, and it also improves the efficiency of the XML compressors by 9-26%. [6] [9] [10]

2. **Size Optimisation with ARM Code Compression:** we developed a new decision tree based ARM code compression algorithm. We also designed the structure of an efficient implementation (a compression framework) for JFFS2, which originally uses a general purpose compressor called zlib. Our method can save 12-19% in space in a general file system image, relative to the original one. This framework is now an official part of JFFS2 and Linux Kernel, and the compression method was patented with US patent number 6,917,315. [3] [14]

3. **Performance Optimization with Improved B+ Tree:** we improved the B+ tree algorithm for flash file systems. The new data structure and algorithm handle the data stored on flash and in the memory as well and make its performance much more optimal, and provide a power loss safe solution. The algorithm is now a part of the official UBIFS file system and Linux Kernel, and it is used in the Nokia N900 smart phone.[5]

	Thesis	Publications
1.	XML Semantic Extension and Compaction	[6], [9], [10]
2.	Size Optimization with ARM Code Compression	[3]
3.	Performance Optimization with an Improved B+ Tree	[5]

Table 1: Theses and publications

1 XML Semantic Extension and Compaction

These days XML is one of the most popular, general, and widely used document formats on desktop computers, on servers and in embedded systems. Applying one of the results we got, XML files can

be stored more efficiently, which can be very useful especially in the case of embedded systems and network applications.

The main idea is based on an analogy between XML documents and attribute grammars (AG), which shown in Table 2.

Attribute Grammars	XML
nonterminal	element
formal rules	element specification in DTD
attribute specification	attribute specification in DTD
semantic functions	—

Table 2: Analogies between AG and XML

As one can see, there is an important concept in Attribute Grammars that has no XML counterpart: the semantic function. To bridge this gap, we defined a new metalanguage called SRML to make it possible to define semantic computation rules for XML attributes. In the case of DTD, this definition is stored in a separated XML file; in the case of XSD, it can be stored inside the XSD in its appinfo part.

Attribute grammars can be classified according to their evaluation methods. By analogy, we can also define these classes in an XML environment. In the thesis we defined S-SRML and L-SRML.

One of the applications of SRML is XML compaction, as shown in Figure 1.

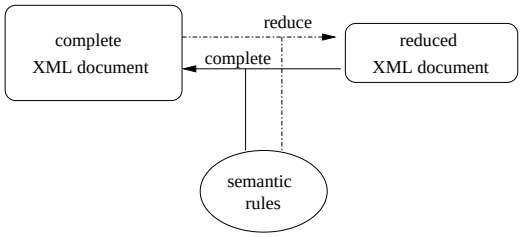


Figure 1: The Compacting/Decompacting method

The method *reduce* removes those attributes from the original *complete XML document* that are calculable using the rules described in SRML, while the method *complete* does just the opposite of this.

The computation rules described in SRML can be produced in one of the following ways: it may be defined by an expert who knows the corresponding XML format, or can be learned by machine learning algorithms using concrete XML files as its training set. These two methods can also be combined: rules produced by an expert can be extended by a machine learning algorithm. Figure 2 gives the structure of the implementation.

This compaction technique can also be used to improve XML file compression since a compressor algorithm will be more effective if more correlations can be recognized in the compressed files. Owing to this, an XML compressor (like XMill) can generally achieve better compression ratios on compressing XML files than a general purpose compressor like gzip. However, even XML-specific compressors cannot detect the semantic relationships among XML attributes. Hence if we compact an XML file before

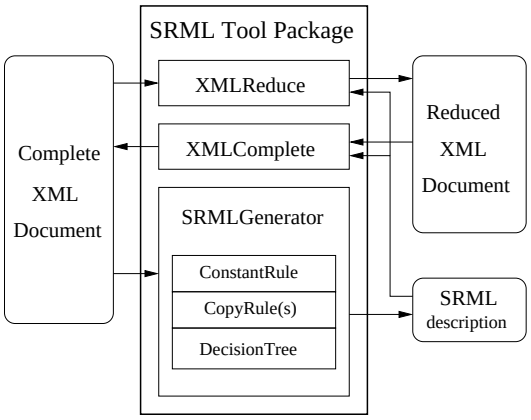


Figure 2: The structure of the implementation

the compression, it will produce better results compared to compressing it with an XML compressor alone.

Our method was tested with CPPML files, which were intended to store C++ source files in an XML format. It can be generated by the Columbus Reverse Engineering package [13]. Table 3 shows the size of the CPPML files, the size of the compacted size of the CPPML files (the SRML size is included) using handwritten SRML, machine generated SRML and also the combined method. The compaction rate we achieved lays between 57-79%.

File	Manual	Machine	Combined
SymbolTable (399 321)	296 193 [74.17 %]	313 873 [78.60 %]	281 088 [70.40 %]
Jikes (2 233 822)	1 736 285 [77.72 %]	1 737 872 [77.79 %]	1 367 244 [61.20 %]
AppWiz (3 547 297)	2 238 308 [63.09 %]	2 589 526 [73.00 %]	2 038 569 [57.46 %]

Table 3: Combining machine learning and manual rule generation

Table 4 shows the results of improving the efficiency of XMill. In the first column there is the original size of the CPPML file, while in the second there is the XMill compressed size. In the third there is the result of the combination of XML compaction with hand-made rules and the XMill compressor, while in the fourth the combination is XML, compacting with hand-made and machine-learned rules and XMill compressor. We improved the efficiency of the compression of the XMill XML compressor program by 9-26%.

File	Original	Manual	Combined
SymbolTable	19 786	18 008	17 876
399 321	4.95 %	4.50 %	4.47 %
		8.98 %	9.70 %
Jikes	114 275	108 458	92 102
2 233 822	5.11 %	4.85 %	4.12 %
		5.09 %	19.40 %
AppWiz	145 738	134 217	106 773
3 547 297	4.10 %	3.78 %	3.01 %
		7.90 %	26.73 %

Table 4: XMill compression for combined and manual compaction

The first suggestion of adding semantics to XML documents was mentioned in [12]. The authors

furnished a method for transforming the element description of DTD into an EBNF Syntactic rule description. It introduced its own SRD (Semantics Rule Definition) comprised of two parts: the first one describes the semantic attributes[1], while the second one gives a description of how to compute them. SRD is also XML-based. The main difference between the approach outlined in their article and ours is that we provide semantic rules not just for newly defined attributes but also for real XML-attributes. Our approach makes the SRML description an organic part of XML documents. This kind of semantic definition could offer a useful extension for XML techniques.

We are not aware on any study on generating rules for XML files. We came across an article that generates rules for Attribute Grammars, which was introduced in [4]. The idea is to provide a way of learning attribute grammars. The learning problem of semantic rules is transformed into a propositional form. The hypothesis induced by a propositional learner is then transformed back into semantic rules. AGLEARN was motivated by ILP learning. This method is similar to ours as it learns and uses semantic rules based on examples as training data, but it is only effective on attributes with very small domains. In contrast to our method, it searches for precise rules that can use approximated rules as well.

The basic idea behind the method and introducing the metalanguage and its main applications are my own results, which were published in [6]. The compression application, the learning framework published in [9] are joint results with Miklós Kálmán. The XSD adaptation published in [10] is mostly the work of my co-author, Miklós Kálmán.

2 Size Optimization with ARM Code Compression

Some years ago embedded systems had very limited storage resources in general. Nowadays, because of the low cost of flash chips, the relative importance of the size factor has decreased in the case of multimedia devices, but in the case of the functional devices it is still high.

One of the simplest solutions for saving space is that of compression. One of the nicest ways of compression is when the file system itself has a transparent compression feature; then neither the users nor the developers have to deal with it. JFFS2, which is one of the most popular Linux flash file systems, has this kind of compression capability. JFFS2 uses zlib as a compressor, which is a general purpose compressor.

As we said in the previous section, a compressor can be more efficient if it has more pre-knowledge or background knowledge about the data. Because of this, a special purpose compressor is generally more efficient than a general purpose one.

In the case of embedded systems one of the biggest large-scale data types is that of executable code. One of the most popular embedded architectures is ARM. On account of this, our study focused on developing an ARM code compressor that could also be combined with currently available solutions.

Our algorithm is a model-based compression, which uses a decision tree model to predict the probability distribution of the next token. This probability distribution is used by an arithmetic coder for encoding and decoding.

Our method is based on the work of C. W. Fraser [1] and M. Garofal [2]: we combined their results and improved them for efficient ARM code compression.

[1] These are newly defined attributes which differ from those in XML files.

Fraser worked on an intermediate representation (*IR*) not on binary code. He achieved a compression ratio of 0.19 on IR, but this did not take into account the size of the model, which can be very large.

Garofalakis et al. introduced some methods for the efficient building of decision trees: they used an MDL (Minimal Description Length) measure for pruning.

Our algorithm also built a decision tree, and used an MDL heuristic, but the building method was adapted and improved in the following ways:

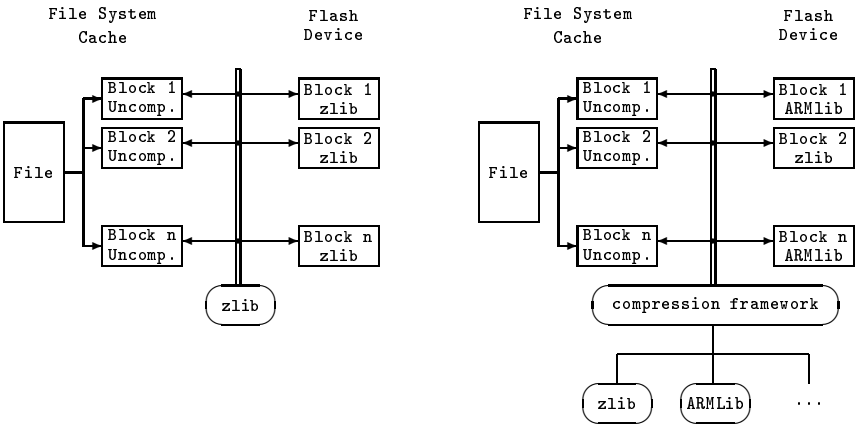
- The tokens were designed to be effective for the ARM code: a 32 bit-long ARM instruction divided into 4 bit-long pieces and we rearranged the instructions for ease the predictability.
- We used 17 predictors:
  - 16 reduced predictors: the previous 16 tokens (2 ARM instructions)
  - 1 computed predictor: it shows the order of the next token inside the corresponding ARM instructions (1-8).
- Two types of decision tree models were used with different performance indicators: binary and multi-value decision tree. In both cases it was possible to build a tree for the entire training set (in this case, the model is one big tree), or cutting up the same training set into random sets (in this case, the method will use several smaller trees as model).
- During the model construction we used an MDL-based stopping heuristics (unlike in the Garofall method, which only pruned the tree after building): we used entropy to estimate the compressed size of the code (because the compression ratio of the arithmetic coder is close to it), and with the subtree of the model, we used a fixed compression ratio to predict the compressed size, because it was compressed using a general purpose compressor.

Besides the development of the new code compression algorithm, we also elaborated the structure of an efficient implementation. This solution replaces the original zlib compressor in JFFS2 with a compression framework, which can be configured to choose the best compression ratio. It calls all the available compressors (zlib and our ARM code compressor called ARMLib) and it chooses the smallest result for each block. Its structure is shown in Figure 3b.

Figure 4 shows the results of this combined solution, which produces a 12.6-19.3% better compression ratio than the original zlib-only solution on iPAQ H3600. The drawback of the method is its slowness. Where it is the most noticeable to the user, is the boot time, as can see in Figure 5.

The implemented framework became an official part of the JFFS2 file system and the Linux Kernel. The compression algorithm was patented with US patent number 6,917,315 [14].

The compression algorithm is a common result [3] with my co-author, Tamás Gergely. The structure of the efficient method of integrating the algorithm into the Linux Kernel, and the implementation itself are my own results. The authors of the US patent are myself, Tamás Gergely and Árpád Beszédes, and its beneficiary is the Nokia Corporation.



a: Original **b**: Advanced

Figure 3: The improved compression of JFFS2

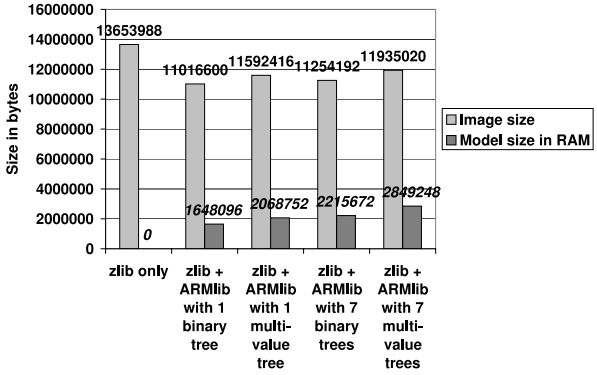


Figure 4: Sizes of compressed file system images

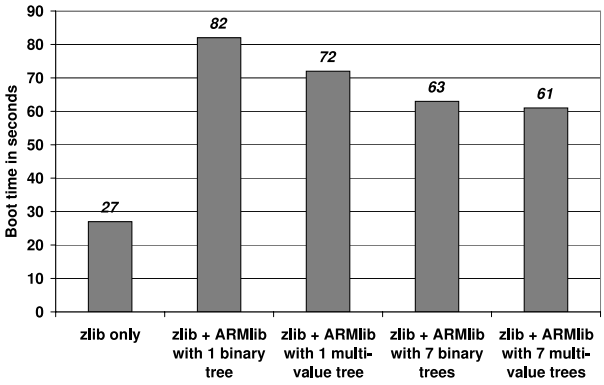


Figure 5: Boot times

3 Performance Optimization with an Improved B+ Tree

Most mobile devices handle data files and store them on their own storage device. In most cases, this storage device is flash memory. On smart devices an operating system runs programs and uses some kind of file system to store data.

Using traditional file systems on flash devices is not a straightforward process because most of the traditional file systems are designed for hard drives, and flash chips have different properties. Both of them handle the data in blocks (with hard drives, the blocks are called sectors; with flashes they are called "erase blocks"), but in the case of flash chips there is a hard limit: one erase block can be erased only about 100.000 times; after that the flash chip will be unreliable. This is why most of the ordinary file systems (FAT, ext2/3, NTFS, etc.) are unusable on flash directly, because all of them have areas which are rarely rewritten (FAT, super block, etc.), and this area would soon be corrupted.

One of the most common solutions to balance the burden of the erase blocks is FTL (Flash Translation Layer), which hides the physical erase blocks behind a layer, and under it exchanges physically the erase blocks to reach nearly equal erase counts. This solution is used by most of the pen drives and memory cards available.

For occasional usage, FTL is an appropriate compromise. For more frequent usage, or where the performance penalty is unacceptable, it is necessary to look for another solution. One such solution may be the use of flash file systems that have been specially designed for flash chips.

One of the most popular Linux flash file systems was the second version of JFFS, called JFFS2. The basic idea behind this is quite simple: the file system is just a concentric journal. In essence, all of the modifications on the file system are stored as a journal entry. When mounting, the system scans this journal and then replays the events in the memory, creating an index to register which file is where. If a new free block is needed, the system selects the one with the most, now obsolete entries in the log, moves the still active entries and erases the block. The cost of the mount process is its slowness of time, what's more, the overall index information has to be stored in memory. This causes problem especially in large flash chips (over 512MB) where the JFFS2 is practically unusable.

Because the root of this problem lies in the base data structures and operating method of the JFFS2, we really need to construct a new file system to eliminate the linear dependency. To achieve this, it is necessary to store index information on the flash so as to avoid always having to rebuild it when mounting. It was necessary to find a solution to store this index information in a flash-friendly and even effective way. Since a lot of file systems use B+ tree to store index information, we also decided to use this as a starting point.

A modified version of the B+ tree can be found in the LogFS file system [8], which is a flash file system for Linux. It is still in the development phase, and probably will be never finished because UBIFS offers a much better alternative. This B+ variant is called a wandering tree. The general workings of this tree can be seen in Figure 6.

Like the ordinary B+ tree algorithm, during a node insertion it is normally necessary to modify a pointer at just one index node. In the case of flash memory the modification is costly, so this wandering algorithm writes out a new node instead of modifying the old one. If there is a new node (such as E'), it is necessary to modify its parent as well, up to the root of the tree. It means that one node insertion (not counting the miscellaneous balancing) requires h new nodes, where h is the height of the tree. It also generates h dirty (obsolete) nodes, as well. Because h is $O(log_d(n))$, where n is the tree node

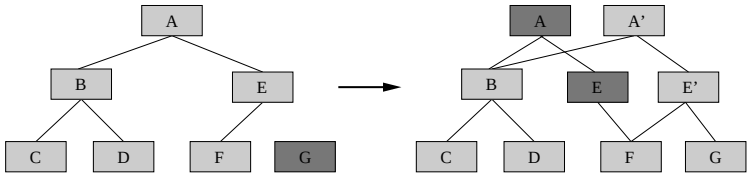


Figure 6: The wandering tree before and after insertion

number, the cost of this operand is still $O(log_d(n))$. If this algorithm is used in a trivial manner, the resulting garbage makes the storing data very inefficient, so it will be practically unusable. Making a wandering tree more efficient is the main result of this thesis.

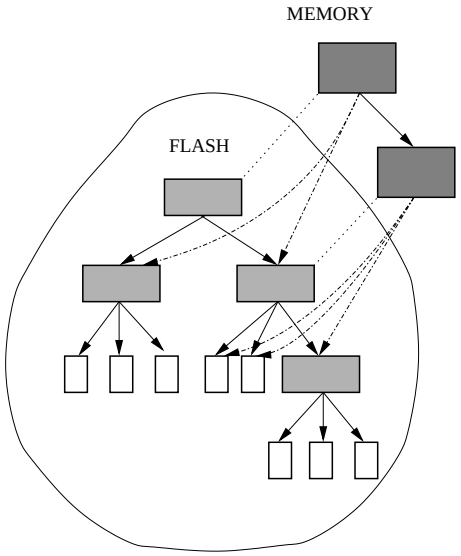


Figure 7: The data structure of TNC

Our improved data structure and the algorithm are both called the TNC (Tree Node Cache). It is a B+ tree, the structure of which can be seen in Figure 7.

Its building and modification is performed dynamically, and simultaneously performs caching, performance improving and fault management functions. The key properties of our improved B+ tree are the following.

- TNC is partly in the flash memory, and partly in the memory. The parent of all index nodes, that are in the memory, is also in the memory. The children of all index nodes which are in the flash memory, are also in the flash memory.
- Each index node read in the memory stores the address where it was read from, and also stores a flag if it was modified or not.
- Its operations are the following.

Search (read):

1. Read the root node of the tree into the memory, then point to it using the pointer p.

2. If p is the desired node, return with the value of p.

3. Find at node p the corresponding child (sub tree), where the desired node is.

4. If the child found is in the memory, set pointer p to it, and jump to point 2.

5. The child is in the flash memory, so read this into memory. Mark this child in p as a memory node.

6. Set pointer p to this child, and jump to point 2.

Clean-cache clean-up (e.g. in the case of low memory):

1. Look for an index-node in the memory which has not yet been modified, and for which all of its children are in the flash memory. If there is no such index-node, then exit.

2. Set the pointers in the identified node's parent to the original flash address of the node, and free it in the memory.

3. Jump to point 1, if more memory clean-up is needed.

Insert (write):

1. Write out the data as a leaf node immediately. UBIFS writes them out to the BUD area[2], which is specially reserved for leaf nodes, just to make it easier to recover when necessary.

2. Read (search) all of the nodes into memory that need to be modified using the B+ algorithm. (In most cases it is just one index node.)

3. Apply the B+ tree modifications in the memory.

4. Mark all modified nodes as dirty.

In the method described above node insertions can be collected, and we can apply them together with significantly lower flash overheads.

Commit (Dirty-cache clean-up):

1. Look for a dirty index node that has no dirty child. If found, call it node n.

2. Write out a new node n onto the flash, including its children's flash addresses.

3. Mark the place dirty where the node n was previously located, and update the flash pointer in the memory representation of the node to the new flash address.

4. Mark the parent of node n as dirty (if it is not the root node), and mark node n as clean.

[2]There are two kinds of data erase block in UBIFS, namely the BUD erase block and the non-BUD erase block. UBIFS stores only the leaf nodes in the BUD erase blocks, while all other types of data nodes are stored in non-BUD erase blocks.

5. Jump to point 1 until there is a dirty node.

Deletion:

1. Read (search) all of the nodes into memory that need to be modified using the B+ algorithm. (In most cases it is just one index node.)
2. Apply the B+ tree modifications in the memory.
3. Mark all modified nodes as dirty.

In the case of power loss, the information stored in the memory is lost. To prevent this from happening, UBIFS combines TNC with a journal, where the following information is stored:

- A journal entry with a pointer to new BUD erase blocks. BUD erase blocks in UBIFS are reserved areas for leaf nodes. If the BUD area is full, a new free erase block will be reserved for this purpose.
- Delete an entry after each node deletion.
- A journal entry after each commit with a list of still active BUD areas.

In the event of power loss, the correct TNC tree can be recovered by performing the following steps:

1. Start with the tree stored on flash.
2. Look for the last commit entry in the journal. All of the events that occurred from that point have to be scanned.
3. All of the node insertions stored in the BUD areas marked in the journal, and all of the deletion nodes stored in the journal have to be replayed in the memory.

The method we applied to test the efficiency of the TNC was the following: we unpacked the source of Linux kernel version 2.6.31.4 onto a clean 512MB file system, and deleted data using the commands below. During the test, the system counted how many flash operands (in terms of node size) were created with and without TNC.

We measured the performance using different TNC configurations. A TNC configuration has the following parameters:

TNC buffer size : The maximal size of the memory buffer that the TNC uses to cache. If it is full, it calls commit and shrink operands.

Shrink ratio : In the case of shrink, the shrink operand will be called until this percentage of the TNC nodes is freed.

Fanout : B+ tree fanout number: the maximum number of children of a tree node. ($2d$, where d is the order of the B+ tree.)

Max. TNC size	Without TNC	With TNC	Shrink Ratio	With TNC / without TNC
5000	2161091	38298	25 %	1.77 %
10000	2211627	31623	25 %	1.43 %
15000	2191395	24632	25 %	1.12 %
20000	2244013	20010	25 %	0.89 %
25000	2192044	12492	25 %	0.57 %
5000	2163769	36273	50 %	1.68 %
10000	2250872	31570	50 %	1.40 %
15000	2225334	22583	50 %	1.01 %
20000	2225334	20002	50 %	0.92 %
25000	2183596	12457	50 %	0.57 %
5000	2215993	36759	75 %	1.66 %
10000	2290769	32578	75 %	1.42 %
15000	2244385	29956	75 %	1.33 %
20000	2238633	20002	75 %	0.89 %
25000	2205709	12958	75 %	0.59 %

Table 5: The number of the flash operations (measured in terms of node size)

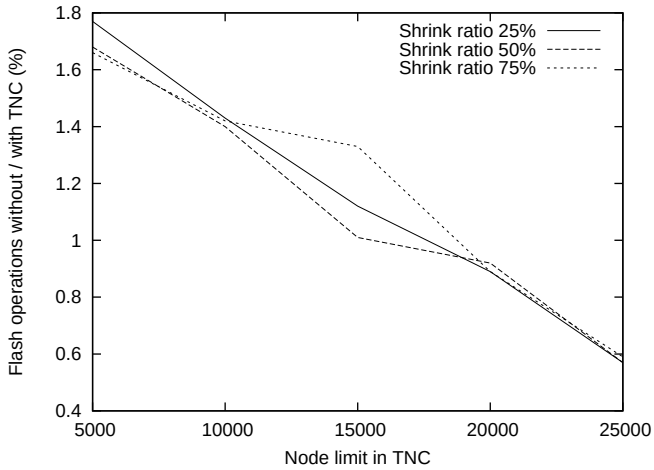


Figure 8: The performance of TNC flash operations compared to those got using the simple wandering algorithm

Table 5 and Figure 8 show the results of measuring the flash performance when the TNC *buffer size* and *shrink ratio* were varied. As can be seen, the TNC saves 98.2-99.4% of the flash operands. Increasing the TNC size, more of the flash operations are saved, but varying the shrink ratio has no noticeable effect here.

Table 6 shows what happens if we change the fanout value of the tree. The number of TNC nodes decreases, but the size of a TNC node increases, because a TNC node contains more pointers and keys. The size of the flash operations is the product of these two factors, and it has a minimum fanout value of 32.

In the remaining tests we took different samples from the source code of Linux kernel version

Fanout	Without TNC in nodes	With TNC in nodes	Max TNC in nodes	TNC node size	Max TNC in MB	Flash ops in MB
4	1134784	48392	64801	176	10.88	8.12
8	2168308	12405	23189	304	6.72	3.6
16	1304212	3577	9662	560	5.16	1.91
32	1024363	1317	4669	1072	4.77	1.35
64	1140118	3420	3671	2096	7.34	2.35
128	767005	1245	1586	4144	6.27	3.35
256	930236	1641	980	8240	7.7	4.35

Table 6: The effect of varying the TNC fanout

I/O Size (MB) \ Fanout	8	16	32	64
50	3302	1456	703	351
100	6364	2818	1355	671
200	12925	4620	2224	1106
400	23518	8978	4282	2861
600	43320	18426	8846	5840
800	44948	22070	12273	8527

Table 7: The maximal TNC size as a function of tree fanout

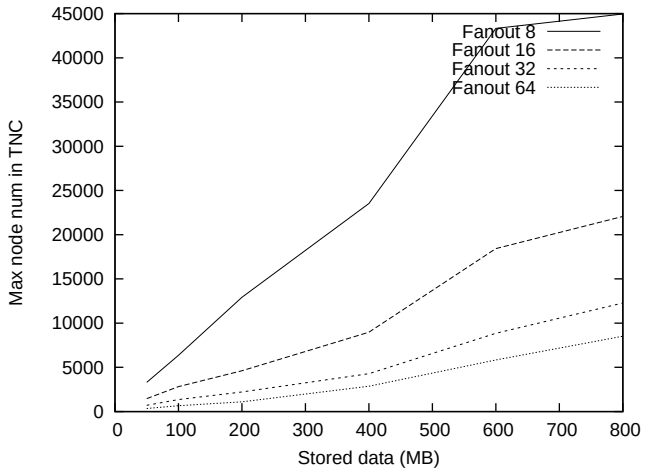


Figure 9: The maximal TNC size as a function of tree fanout

2.6.31.4. Table 7 and Figure 9 tell us the maximal TNC size (setting no limit) when the fanout is varied, and the size of the I/O operands (size of the "file-set" above) as well.

The authors of [11] outlined a method that had a similar goal to ours, namely to optimize the B+ tree update on a flash drive. The method collects all the changes in the memory (in LUP = lazy-update-pool), and after it has filled up, data nodes are written out in groups. It also saves flash operations, but unlike our method, using LUP means a lower read speed because, before searching in the tree, it always has to scan the LUP. In the case of the TNC, there is usually a higher read speed

because the nodes (at least the modified ones) are in the memory. Our own method is power-loss safe, but the authors of [1] do not say what happens when the information is stored in the LUP. The advantage of their method is the following: the node modifications can be grouped more freely (not just sequentially), so it may be easier (and require less memory) to close the tree operations intersecting the same tree area.

The goal outlined in [16] is also a B+ tree optimization on a flash memory. It collects any changes made in the memory and stores them in the "Reservation Buffer". It is filled up and these changes are written out and grouped by a "Commit Policy" into flash as an "Index Unit". It makes use of another data structure called the "Node Translation Table" to describe which node has to be transformed by which Index Unit. To search in the tree, it is necessary to scan both the Node Transaction Table and the Index Units.

The method described in [7] is essentially an improved version of that described in [16]. Instead of the simple "Reservation buffer", it utilizes the "Index Buffer", which monitors the tree modifications and if any intersect the same node, it closes them or, where possible, deletes them. In the case of commit, it collects data concerning the units belonging to the same nodes, and writes them out to one page.

These results are my own results, which were published in [5]. The TNC became an official part of UBIFS and the Linux Kernel, and was oncorporated in the Nokia N900 smart phone.

References

[1] Christopher W. Fraser. Automatic inference of models for statistical code compression. *SIGPLAN Not.*, 34(5):242–246, May 1999.

[2] Minos Garofalakis, Dongjoon Hyun, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for constructing decision trees with constraints. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 335–339, New York, NY, USA, 2000. ACM.

[3] T. Gergely, F. Havasi, and T. Gyimóthy. Binary code compression based on decision trees. *Proceedings of the Estonian Academy of Sciences Engineering*, 11(4):269–285, 2005.

[4] T. Gyimóthy and T. Horváth. Learning semantic functions of attribute grammars. *Nordic Journal of Computing*, 4(3):287–302, Fall 1997.

[5] F. Havasi. An improved B+ tree for flash file systems. In *SOFSEM 2011: Theory and Practice of Computer Science: 37th Conference on Current Trends in Theory and Practice of Computer Scienece, LNCS 6543*, pages 297–307, 2011.

[6] Ferenc Havasi. XML semantics extension. *Acta Cybernetica*, 15(2):509–528, 2002.

[7] Ha-Joo Song Hyun-Seob Lee, Sangwon Park and Dong-Ho Lee. An efficient buffer management scheme for implementing a B-Tree on NAND flash memory. *Embedded Software and Systems*, pages 181–192, 2007.

[8] Robert Mertens Jörn Engel. Logfs - finally a scalable flash file system. CSCI 390 Spring 2008 Senior Seminar.

[9] M. Kálmán, F. Havasi, and T. Gyimóthy. Compacting XML documents. *Information and Software Technology (Impact Factor 1.522)*, 48(2):90 – 106, 2006.

[10] Miklós Kálmán and Ferenc Havasi. Enhanced XML validation using SRML. *International Journal of Web & Semantic Technology (IJWesT)*, 4(4):1 – 18, 2013.

[11] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. Lazy-update B+-Tree for flash devices. *Mobile Data Management, IEEE International Conference on Mobile Data Management*, pages 323–328, 2009.

[12] G. Psaila and S. Crespi-Reghizzi. Adding Semantics to XML. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 113–132, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[13] Á. Beszédes-Á. Kiss M. Tarkiainen R. Ferenc, F. Magyar. Tool for reverse engineering large object oriented software. *SPLST*, pages 16–27, June 2001.

[14] Á. Beszedes T. Gergely, F. Havasi. Model based code compression, US patent number 6,917,315, 2003.

[15] J. Turley. The two percent solution. Embedded Systems Design, 2002.

[16] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient B-Tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3):19, 2007.