H-0224 (H0405-002) May 2, 2004
Computer Science

IBM Research Report

Reducing Program Image Size by Extracting Frozen Code and Data

Daniel Citron, Gadi Haber, Roy Levin
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Reducing Program Image Size by Extracting Frozen Code and Data

Daniel Citron, Gadi Haber, Roy Levin
IBM Research Labs in Haifa
citron,haber,royl@il.ibm.com

Abstract

Constraints on the memory size of embedded systems require reducing the image size of executing programs. Common techniques include code compression and reduced instruction sets. We propose a novel technique that eliminates large portions of the executable image without compromising execution time (due to decompression) or code generation (due to reduced instruction sets). Frozen *code and data portions are identified using profiling techniques and removed from the loadable image. They are replaced with interrupts that load them in the unlikely case that they are accessed. The executable is sustained in a runnable mode.*

Analysis of the frozen portions reveals that most are error and uncommon input handlers. Only a minority of the code (less that 1%) which was identified as frozen during a training run, is also accessed with production datasets.

Applying this technique results in a maximal code reduction of 90% (average 62%) in the image size of the SPECINT CPU 2000 benchmarks and 40% (average 20%) in the data image size. For the SPECFP CPU 2000 maximial code reduction reached up to 90% reduction (average 80%) and static data reductuion reached up to 90% (average 50%).

1 Introduction

Embedded systems are everywhere: in a multitude of consumer products, communication devices, low end computing devices, and are recently making inroads into the desktop domain [13]. However, while required to process desktop workloads they are constrained by power dissipation, energy consumption, and size. Particularly, they are constrained by memory size.

The constraints are, not only on the physical memory hierarchy, but on the virtual memory size as well. In fact, many embedded systems have poor or no support for virtual memory [20]. Thus, reducing the size of loaded executable images is of paramount importance.

This paper suggests a novel technique for reducing image size: elimination of code and data that is not accessed within representative trace executions. These portions are called *frozen code* and *frozen data*. A profile of a program is gathered based on representative workloads and non-accessed code or data that are not accessed are marked as frozen and replaced by an interrupt. The interrupt handler reads the frozen code from storage and resumes execution.

This overcomes the liabilities of two common techniques for image size reduction: (i) compression/decompression; (ii) reduced size instruction sets; The former requires that the executable is stored in a compressed state and is decompressed before execution. Thus, it is possible that critical code or data must be decompressed before it can be used. [11] and [10] are two examples of this technique. Run-time decompression [3] can help solve the issue of non-runnable executable files. However, in run-time decompression techniques, both, the compressed image program and the decompression subroutine, can consume a substantial memory size during run-time, thus making it inefficient in many cases. The latter technique reduces the size of all instructions, usually from 32 to 16 bits. However, this greatly limits the quality of the compiled code. The 16-bit Thumb [22] instruction set is a famous representative of this technique.

Our proposed technique manages to reduce the image size significantly and efficiently using known code and data relocation techniques, making it a convenient method that can be embedded into hardware, operating system level or software development tools. In this work we implemented the reduction technique in FDPR (Feedback Directed Program Restructuring) a post-link tool that is part of the IBM AIX operating system.

The rest of the paper explores the implementation, potential and experimental results of the proposed scheme:

- Previous and related research will be presented in section 2.
- The methods for code and data reduction will be explained in sections 3 and 4.
- The percentage of frozen code and data in the SPEC CPU 2000 benchmark suite will be shown in section 5.

2 Related Work

The idea to reduce the size of code and data has been approached in many different manners. Compressing the executable on non-volatile storage and decompressing before execution is probably the most popular. Many compression techniques are based on the Huffman [9] and LZ (Lempel-Ziv) [12] algorithms. Others (such as *gzip* [6]) were designed in order to circumvent the aforementioned algorithms and derivative patents. Several others are tailored to compress code, [14] is a representative of such techniques.

However, decompressing before execution requires even more memory than loading the uncompressed executable. The gains achieved are for storage and network transfer, not memory image size. At the other end of the spectrum are schemes that reduce the size of the individual instruction's representation. The Thumb [22] and MIPS16 [17] instruction sets are composed of 16-bit instructions that implement 32-bit architectures. These implementations trade code size for number of registers and operation variety. Ultimately performance is degraded.

Hardware based decompression is another popular technique. IBM's *codepack* technique [11] uses dedicated lookup tables to decompress code that is fetched to the L1 ICache. Other hardware based techniques are Wolfe and Chanin [25] and Larin and Conte [10]. The disadvantage of these techniques is that they incur a potential penalty for every line brought into the cache, and increase hardware costs, although under some circumstances performance improvement is possible [11].

Prior research that is based on profiling include Hoogerbrugge et al . [8] who interpret non-critical code, but have to include a possibly large interpreter in their code. Debray and Evans [3] compress *cold* code, code that is executed less than a threshold of T (during a train run). This technique incurs a performance degradation when these areas are encountered. Only when $T = 0$ there is no impact on performance. This is exactly the case of frozen code.

It can be said that Virtual Memory (VM) [7] performs the same functionality as our proposed technique, only code and data that is accessed is promoted from disk to memory. However our technique can overcome the relatively large granularity of VM (default of 4K per page) and the lack of memory management on many embedded systems [20].

The same argument is true for caching [7] where only code/data that are used are fetched into the higher levels of the memory hierarchy. Nevertheless, in embedded systems where the main memory can fit into the caches of high-end servers our technique is valid and useful.

3 The Code Reduction Method

The following list defines the key terms that will be used in this paper:

Dead Code A portion of the program that never executes for any program trace. Compiler optimizations usually remove these sections.

Frozen Code A code area within the program file that is *not* executed when run on a representative workload.

Cold Code A code area within the program file that is *rarely* executed, relatively to other parts of the program, when run on a representative workload.

Hot Code A code area within the program file that is *frequently* executed, relatively to other parts of the program, when run on a representative workload.

Frozen/Cold/Hot Data Variable A data variable within a given executable file, that is not/rarely/frequently accessed when run on a representative workload.

The code reduction method relocates all frozen basic blocks in the given code, groups them together in a separate non-loadable module and replaces each control transfer to and from them by an appropriate interrupt. The interrupt mechanism is in charge of invoking the appropriate handler for loading the relevant code segments from the non-loadble module containing the targeted basic blocks. As will be shown in the experimental results, since the relocated basic blocks are frozen, the time

consuming interrupt mechanism will be rarely (if at all) invoked during run-time, and will therefore, have no siginifcant effect on performance.

The method for reducing the program code size is described as follows:

1. Use the profiling information to identify all the frozen code segments within the code. In this work we preferred to use existing post-link tools for gathering the required profile information and for analyzing given executable files. For details on post-link tools or link-time tools for global code analysis and restructuring see [1, 2, 15, 18, 21, 24, 19].

2. Relocate all the frozen code segments into a single group and then place it in a non-loadble section area of the executable file, or optionally, in a separate file. In our work the frozen code was placed in a separate non-loadable section of the executable file in a non-compressed form.

3. Identify all control flow instructions, and fall through instructions, into and out of the frozen code segments, and among the frozen code segments. For each of these instructions compute the offset of their target. This is named the *target offset*. In this work, the target offsets of both frozen and non-frozen basic blocks were calculated from the beginning of the section in which they were placed.

4. Update each of the control transfer or fall through instructions, to invoke an interrupt. In this work the interrupts were triggered by inserting invalid instructions in the code. During run-time, in the unlikely case that a frozen basic block is referenced, the invalid instruction interrupt is then thrown by the system and a frozen code loading subroutine is automatically invoked. Note that in order to occupy as little space as possible, minimal illegal code instructions are used. To each of the invalid opcode instructions we attach a target offset of the relocated frozen segment as calculated in step 3, to be used by the interrupt handler. As a result, a single invalid instruction comprised of a zero opcode, which is set by 5 zero bits in the IBM PowerPC RISC architecture, followed by 1 *caracterization bit* for specifying whether the targeted code falls in the non-frozen text section or in the non-loadable frozen code section, and then followed by 26 bits of the target offset of the targeted code segment in its corresponding section. The invalid instruction did not exceed a regular fixed instruction opcode of 32 bit size. These invalid instructions are inserted into the code as follows:

   (a) A direct unconditional branch to or from a frozen code segment is replaced entirely by an invalid operation code followed by the corresponding target offset.

   (b) A conditional branch instruction which branches into or out of a frozen code segment is modified to branch to an intermediate location consisting of the invalid instruction followed by the appropriate target offset.

   (c) A conditional branch instruction which falls through or out of a frozen code segment, will have its condition reversed and then be modified as above. If the condition is non-reversible, an invalid instruction followed by the appropriate target offset is inserted immediately after the conditional branch.

   (d) An indirect branch instruction via a register or memory, is modified to branch to an intermediate location consisting of the invalid instruction.

   (e) A non-branch instruction that falls out of a frozen code, will have an invalid instruction inserted immediately after.

5. Add the code of the loading subroutine handler to the given executable file or alternatively, place it in an appropriate linkable module and link it (either statically or dynamically) to the executable file. During run-time, the subroutine loads the referenced frozen basic block into memory. In our work, at each time, only a single referenced basic block is loaded to memory. However, the notion of basic block was in many cases extended to a group of continous basic blocks which have an obvious control flow among them. When invoked at run-time:

   (a) The loading subroutine uses the caracterization bit b and the target offset S of the referenced basic block that were attached to the invalid opcode in step 4, in order to locate the basic block within the non-loadable frozen code section in the case of a refernce to a frozen code, or within the loaded text section, in the case of a refernce to a non-frozen code.

   (b) If the referenced basic block is located in the non-loadable frozen code section, then:

      - the loading subroutine verifies whether the referenced frozen basic block was already loaded into memory. Referenced basic blocks are marked *loaded* by replacing the code of each referenced basic block in the non-loadable frozen code section (or file) by zeroes followed by its loaded memory address.

- If the frozen code is not in memory, the subroutine then:
    i. dynamically allocates additional memory space for the basic block.
    ii. updates the lowest or highest boundaries of the frozen code area to include the additional allocated space.
    iii. loads the said frozen basic block into the allocated memory at address T.
    iv. marks the basic block as loaded by replacing its instructions in the non-loadable frozen code section (or file) by zeroes followed by the loaded address T.
- If the referenced frozen basic block is already in memory, then retrieve its address, prefixed by zero bits, from the non-loadable frozen code section.

(c) Update the trigerring invalid instruction which invoked the loading subroutine, by a corresponsing branch instruction to the loaded basic block, using the same mechanism that the linker uses in order to perform a far jump from one module to another [1].

(d) Finally, the loading subroutine branches to the target address of the refernced basic block in order to continue execution. Note that once a frozen basic block was referenced by the application it is no longer considered frozen and therefore, continues to remain in memory throughout the entire application execution cycle.

6. Insert an action listener at the entry point of the program file, for invoking the loading subroutine handler in the event of invalid instructions.

Figure 1 illustrates the relocation phase of frozen code areas. Each rectangle in the figure represents a single basic block. The arrows represent the control flow between the basic blocks. The first figure illustrates an example procedure before the frozen code in it was relocated. The procedure consists of four hot basic bocks labeled BB1 - BB4, and two frozen basic blocks labeled BB5 and BB6. The second figure illustrates the same procedure after relocating the frozen code. The frozen basic blocks BB5 and BB6 are placed together in a separate (non-loadble) section and each control transfer to them from the other basic blocks in the procedure is replaced by a corresponding illegal instruction containing the offset target of the callee basic block within the area it was relocated to. The loading module, which includes the code for intercepting the trap created when trying to execute the illegal opcodes, is placed in a different place in the program. When invoked at run-time, the loading subroutine decompresses BB5 and BB6 if needed, loads them into dynamically allocated memory area and transfers the control to and from them using their target offsets, after adding the run-time address of the loaded section in which they reside. The dashed lines represent the control transfer between the loaded frozen and the non-frozen code via the trapping mechanism, which is done via the interrupt mechanism.

4 The Data Reduction Method

The method for reducing the static data in the given program file is similar to the code reduction method. All frozen data variables that are not referenced in a representative trace, are relocated, grouped together and then placed in a separate section. Each load instruction of the relocated data variables is then similarly replaced by invalid instructions which trigger a trap mechanism in charge of loading the variables into memory.

The static data reduction method is described as follows:

1. Identify all the load instructions in the code which reference the static data variables and which need to be updated during data positioning. These instructions are updated by the linker once the global data variables are placed in the executable file. As a result, these instructions have appropriate linker relocation information attached to them. The proposed method can use this relocation information in order to identify them. For more details on global data placement at post-link time please see [16].

2. Use the profiling information to identify all frozen data variables within the static data area. In this work we use the given profiling information to check whether the above load instructions, which reference a certain data variable, are all frozen.

3. Relocate all the frozen data variables, group them together and then place them in a non-loadable section area of the executable file (or in a separate file).

[1] it is assumed that the loading subroutine has the necessary permissions to write to the text area of the application

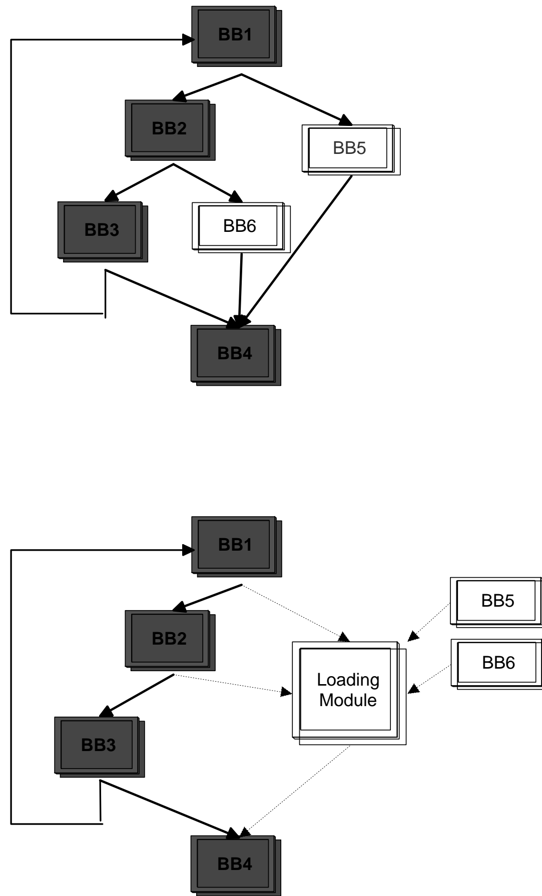

Figure 1. Basic Blocks layout before and after frozen code relocation

4. Update each of the load instructions which refer to a frozen data variable, by an invalid opcode instruction containing the offset of the frozen data variable in the non-loadable section to which it was relocated in step 3. This is similar to the way it is done with frozen code segments. During run-time, in the unlikely case that the frozen data is referenced, an invalid instruction interrupt will be thrown by the system and a loading subroutine will be automatically invoked by catching the thrown trap generated by these illegal instructions.

5. Add the loading subroutine to the given executable file or, alternatively, place it in an appropriate linkable module and link it to the executable file. During run-time, the subroutine loads the refernced frozen data variable. When invoked at run-time:

   (a) The loading subroutine verifies whether the referenced frozen data was already loaded into memory.

   (b) If the frozen data is not yet in memory, the subroutine loads the given frozen data area (or a relevant part of it) into a dynamically allocated memory.

   (c) Finally, the loading subroutine returns the address of the variable in memory or alternatively, performs the actual original load instruction. The ability to retrieve the correct address of the frozen data variable is obtained by adding the address of the loaded frozen data area to the target offsets planted in the code in step 4. Similar to the code reduction method, referenced data variable is no longer considered frozen and therefore, remains in memory

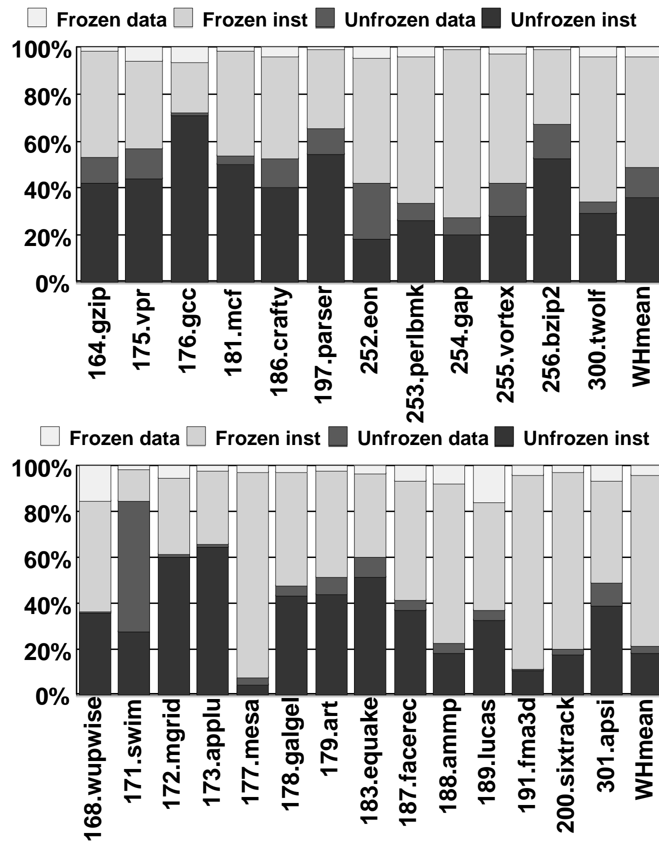

Figure 2. Percentage of frozen code and data in the SPEC CINT2000 and CFP2000 suites.

throughout the entire application execution cycle.

6. Insert an action listener which will invoke the loading subroutine handler in the event of invalid instructions, at the entry point of the program file.

5 Experimental Results and Analysis

Although the proposed technique can be applied in the Operating System (OS) or use dedicated hardware, we chose to implement it using a post-link optimization tool called FDPR (Feedback Directed Program Restructuring) reported in [4, 5]. FDPR is part of the IBM AIX operating system for the IBM pSeries servers. FDPR was also used to collect the profile information for the optimizations presented here. In this section we will analyze two benchmark suites (SPEC CPU2000 [23] and show the percentage of frozen code and data they posess.

5.1 SPEC CPU2000

The CPU2000 suite is primarily used to measure workstation performance but was designed to run on a broad range of processors as stated in [23]: *SPEC designed CPU2000 to provide a comparative measure of compute intensive performance across the widest practical range of hardware*. Although it may be hard to believe that applications such as *gcc* (C compiler), *vpr* (circuit placement), or *twolf* (circuit simulation) are run on handheld devices, others such as *gzip* (compression), *parser* (word processing), and *eon* (visualization) are sure to be. And while many embedded processors don't support floating point operations many others do so even better than desktop processors [13], leading us to include the SPEC CFP2000 suite in our analysis.

We believe that the types of applications presented in the suite will migrate to embedded processors while its successor, CPU2004, will be used in the wokstation domain. As a consequence we chose to analyze 32-bit, rather than 64-bit, executables. The C/C++ benchmarks were compiled on a Power4 running AIX version 5.1 using the IBM compiler `xlc` v6.0 with the flags: `-O3`. The Fortran benchmarks were compiled using the `xlf` v8.1 compiler with the flags: `-O3`.

The profiles were taken using the suite's *train* input set [2]. Figure 2 shows the percentage of frozen code and data in the SPEC CPU2000 suite. The results (CINT first) show that an average (weighted harmonic mean) of 64/80% of the code and 19/52% of the data is frozen. This results in executables which are 58/79% smaller than the originals.

5.2 Train vs. Reference

In order for our scheme to work without *any* performance degradation we must ensure that the frozen code and data areas are either related to error handling or unfrequent case handling. In both cases it is assumed that the code has been written in order to preserve correctness and generality of the program, even though performance will be degraded. Obviously, this will not be the case for every application. *176.gcc* of CINT2000, the gcc compiler, contains hundreds of command line flags. It is virtually impossible to devise a "representative" trace that can cover all valid executions.

Thus, in order to quantify the quality of the training runs we compared the amount of frozen code/data in both the train and reference datasets. Figures 3 show these comparisons for the CPU2000. The results show that the differences are indiscernible. However they are not identical. Table 1 summarizes the average differences in size and dynamic instruction count (both in absolute numbers and ratios).

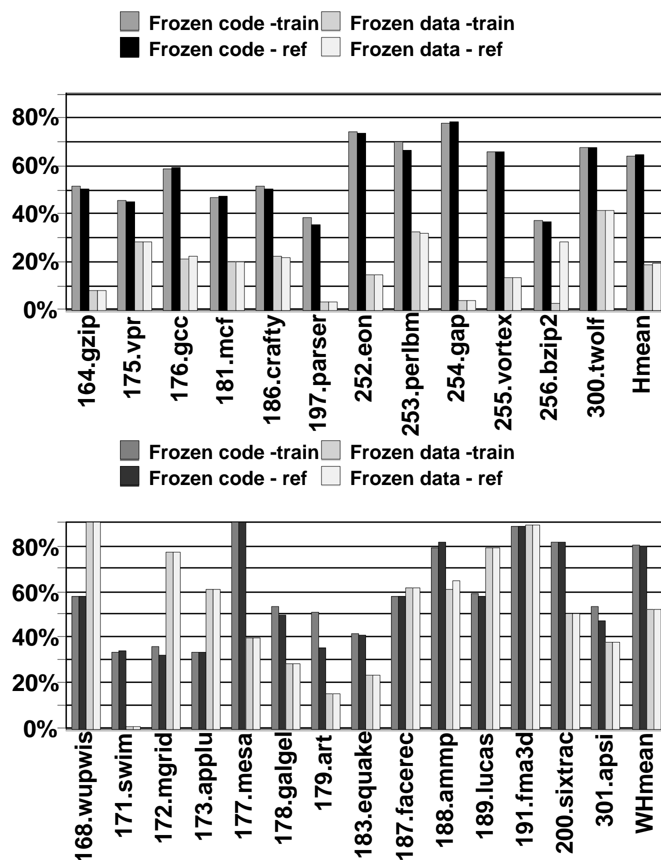

Figure 3. Comparison of frozen code/data between the train and reference data sets of CPU2000.

[2] Benchmark and dataset details can be found in [23].

Suite	*Type*	*Metric*	*Diff.*
CINT2000	code	KB	12
		%	0.32
	data	KB	1
		%	0.05
CFP2000	code	KB	5
		%	0.53
	data	KB	0.1
		%	0.34

Table 1. *Average differences (weighed harmonic mean) of frozen code/data between train and reference datasets.*

The above results indicate that there are code segments that for different workloads may become unfrozen. This means that they are not error correction code and should not have been taken out of the loadable text section. We will refer to these basic blocks as *singular mispredictions*.

The main performance penalty of the proposed reduction method derives from the fact that we require access to the disk for each singular misprediction, which may take up to 50 or more ms, depending on the speed of the disk and I/O bus. However, for every singular misprediction we will pay this penality only on first encounter. Future references are replaced by corresponding branch instructions by the loading subroutine handler.

In order to learn about the estimated penalty of the singular mispredictions, we selected the *gcc* benchmark as a proper candidate for investigation as it contains the highest number of differences in the behaviour between the train and the ref workloads. Therefore, the numbers brought here represent the worst case scenario for our proposed mechanism on the SPEC CPU 2000.

As it turns out, the actual size of the gcc code that is considered frozen with the train workload, yet turns out to be non-frozen with the ref workload, is around 4000 bytes, which is around 200 basic blocks. The entire gcc code includes a total of 95,000 basic blocks, making the number of singular mispredictions around 2% of all the basic blocks. In addition, it turns out that all singular mispredictions are considered cold, i.e., rarley executed even on the ref workload. This means that the number of singular mispredictions is relatively very small, and will most likely not result in a significant overhead.

References

[1] G. A. B. Schwarz, S. Debray and M. Legendre. PLTO: A Link-Time Optimizer for th eIntel IA-32 Architecture. In *In Proceedings of Workshop on Binary Rewriting*, September 2001.

[2] R. Cohn, D. Goodwin, and P. G. Lowney. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical of Digital Equipment Corporation*, 9(4):3–20, 1997.

[3] S. Debray and W. Evans. Cold Code Decompression at Runtime. *Communications of the ACM*, 46(8):55–60, August 2003.

[4] M. K. E. A. Henis, G. Haber and A. Warshavsky. Feedback Based Post-link Optimization for Large Subsystems. In *Proc. Of Second Workshop on Feedback Directed Optimization (FDO), Haifa, Israel*, November 1999.

[5] G. H. E. A. H. V. Eisenberg. Reliable Post-link Optimizations Based on Partial Information. In *Proceedings of the 3rd Workshop on Feedback Directed and Dynamic Optimizations (FDDO)*, December 2000.

[6] http://www.gzip.org.

[7] J. Hennessy and D. Patterson. *Computer Architecture : A Quantitative Approach*, chapter 5. Morgan Kaufman Publisher, 3rd edition, 2000.

[8] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. V. D. Wiel. A code compression system based on pipelined interpreters. *Software Practice & Experience*, 29(11):1005–2023, September 1999.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[10] S. Larin and T.Conte. Compiler Driven Cached Code Compression Schemes for Embedded ILP Processors. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 82–92, December 1999.

[11] C. Lefurgy, E. Piccininni, and T. Mudge. Analysis of a High Performance Code Compression Method. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 93–102, November 1999.

[12] A. Lempel and J. Ziv. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–349, May 1977.

[13] M. Levy. Embedded CPUs Do More, Run Faster. Microprocessor Report, February 2004.

[14] S. Lucco. Split-stream dictionary program compression. In *Proceedings of the Conference on Programming Languages Design and Implementation*, pages 27–34, June 2000.

[15] C. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A post-link Optimizer for the Intel Itanium Architecture. In *Proceedings of the Second International Symposium on Code generation and Optimization*, pages 15–26, March 2004.

[16] G. H. M. K. V. E. B. Mendelson and M. Gurevich. Optimization Opportunities Created by Global Data Reordering. In *First International Symposium on Code Generation and Optimization (CGO'2003)*, March 2003.

[17] http://www.mips.com/content/Products/Architecture.

[18] R. Muth, S. Debray, and S. Watterson. alto: A Link-Time Optimizer for the Compaq Alpha. Technical Report 14, Department of Computer Science, The University of Arizona, December 1998.

[19] I. Nahshon and D. Bernstein. FDPR - A Post-Pass Object Code Optimization Tool. In *Proc. Poster Session of the International Conference on Compiler Construction*, April 1996.

[20] G. V. Neville-Neil. Programming without a net. *ACM Queue*, 1(2):16–22, April 2003.

[21] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad., and B. Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, August 1997.

[22] D. Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, 2000.

[23] SPEC CPU2000: http://www.spec.org/cpu2000/.

[24] A. Srivastava and D. W. Wall. A practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[25] A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 81–91, December 1992.