Reducing Program Image Size by Extracting Frozen Code and Data

Daniel Citron, Gadi Haber, Roy Levin
IBM Haifa Labs
University Campus
Haifa 31905, Israel
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 branches to code stubs 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 than 1%) that was identified as frozen during a training run, is also accessed with production datasets.

The technique was applied on three benchmark suites (SPEC CINT2000, SPEC CFP2000, and MediaBench) and results in image size reductions of up to 73%, 92%, and 85% per suite, The average reductions are 59%, 79%, and 78% per suite.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Performance

Keywords

image size, frozen code, frozen data, feedback directed

1. INTRODUCTION

Embedded systems are everywhere – in a multitude of consumer products, communication devices, and low-end computing devices, and they recently have made inroads into the desktop domain [16]. When embedded systems are required to process desktop workloads, however, 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 [23]. 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 are 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. Non-accessed code or data are marked as frozen and replaced by an interrupt to a code stub. This stub reads the frozen code/data from storage, replaces the interrupt, 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. [14] and [12] 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, both the compressed image program and the decompression subroutine, can consume substantial memory during run-time, making such techniques inefficient in many cases. IN reduced size instruction sets, the size of all instructions are diminished, usually from 32 to 16 bits. However, this greatly limits the quality of the compiled code. The 16-bit Thumb [25] 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, or software development tools. In this work, we implemented the reduction technique in FDPR (Feedback Directed Program Restructuring) [8, 5], 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 are presented in section 2.
- The methods for code and data reduction are explained in sections 3 and 4.
- The percentage of frozen code and data in the SPEC CPU 2000 and MediaBench benchmark suites are shown in section 5.
- The impact of the technique on existing systems and applications are discussed in section 6.

2. RELATED WORK

The idea of reducing 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 [11] and LZ (Lempel-Ziv) [15] algorithms. Others (such as *gzip* [4]) were designed in order to circumvent the aforementioned algorithms and derivative patents. Several others are tailored to compress code: [17] is a representative of such techniques.

However, decompressing before execution requires even more memory than loading the uncompressed executable. The gains achieved are in the area of 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 [25] and MIPS16 [20] 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 [14] uses dedicated lookup tables to decompress code that is fetched to the L1 ICache. Other hardware-based techniques are presented by Wolfe and Chanin [28] and Larin and Conte [12]. 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 [14].

Prior research that is based on profiling include Hoogerbrugge et al. [10], 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 $T = 0$ is there no impact on performance. This is exactly the case of frozen code.

It can be said that Virtual Memory (VM) [9] performs the same functionality as our proposed technique, except that VM promotes code and data that are accessed 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 [23].

The same argument is true for caching [9], 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.

Thawed Code A code area within the program that was marked as frozen but during runtime was accessed.

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

In general, 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. In the event of a reference to an unloaded code instruction, the interrupt handler is invoked and loads the relevant code regions from secondary storage. The interrupt is replaced with a direct or indirect branch (dependent on its placement in memory). As will be shown in the experimental results, the time consuming interrupt mechanism will be rarely (if at all) invoked during run-time, and will therefore, have no or little effect on performance.

The first step in the proposed reduction method is to globally reorder the program code, based on profiling data gathered with an appropriate representative workload. For example, consider the following pseudo assembly instructions before code reordering, in which hot code is shown in bold face:

```
        compare r1, r2
        jump-false L1
        (Frozen Then Part)
    L1: (Hot Continue Part)
```

The above code after reordering will have the following form:

```
        compare r1, r2
        jump-true L2
    L1: (Hot Continue Part)
    ...
    L2: (Frozen Then Part)
        jump L1
```

In the reordered code above, we reversed the condition of the conditional jump instruction, and managed to group the hot code together and place the frozen code further away in the program. Code reordering has important properties of reducing instruction cache misses and the number of branches in the code. Note that in order to maintain correctness, an additional unconditional jump instruction to $L1$ was added at the end of the relocated frozen code part.

The next step in the reduction algorithm is to replace the control transfers between the frozen and non-frozen code areas with interrupts. The interrupt is generated by placing invalid instructions in the code. In the unlikely event that the invalid instruction is reached during runtime, an interrupt exception invokes a loading subroutine which loads and then branches to the relevant code. Each invalid instruction contains the offset of the targeted code that needs to be loaded into memory at run-time. After replacing jumps with illegal instructions the example above will look like this:

```
      compare r1, r2
      jump-true L2I
L1:   (Hot Continue Part)
...
L2I:  invalid-opcode (containing the offset of L2)
...
L2:   (Frozen Then Part)
      invalid-opcode (containing the offset of L1)
```

Once a targeted frozen code is loaded into memory, the loading subroutine replaces the control transfers between the non-frozen code and the thawed code with appropriate branch instructions.

The complete layout of the program code at run-time can be shown in Figure 1. The program code layout consists

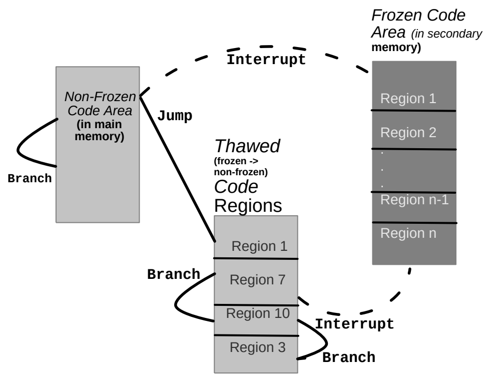

Frozen Code Area (in secondary memory)
Interrupt
Non-Frozen Code Area (in main memory)
Jump
Branch
Thawed (frozen -> non-frozen) Code Regions
Region 1
Region 2
Region n-1
Region n
Region 1
Region 7
Region 10
Region 3
Branch
Interrupt
Branch

Figure 1: Program code layout at run-time

of three main areas: non-frozen, frozen and thawed. The non-frozen area is laid out sequentially in main memory. The frozen code area is laid out sequentially on disk (or any other secondary memory device) This area is divided into "regions". In the event of a reference to a frozen instruction, its entire containing region is loaded into memory. Therefore, all control transfers between regions (i.e., inter-regional transfers) are replaced by corresponding illegal instructions, to enable the loading subroutine to handle them at runtime. Control transfers within a scope of a region do not need to be changed when loaded to memory by the loading subroutine. Finally, the thawed code area consists of various thawed (not necessarily successive) code regions which are allocated in memory at run-time. Control transfers between thawed and non-frozen code areas are updated to use direct or indirect branches, whereas transfers between thawed or non-frozen to frozen code areas continue to use the interrupt mechanism triggered by the illegal instructions.

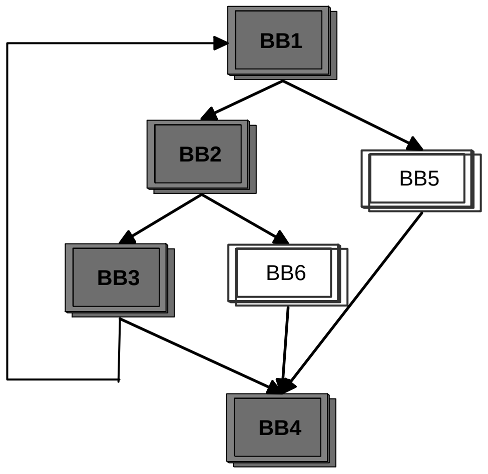


Figure 2: Basic Blocks layout before frozen code relocation

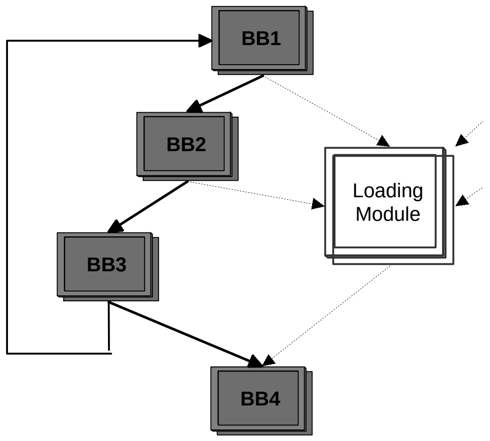


Figure 3: Basic Blocks layout after frozen code relocation

Figures 2 - 4 illustrate the code layout of a short example. Each rectangle in the figure represents a single basic block. The arrows represent the control flow between the basic blocks. Figure 2 illustrates the original code of a procedure prior to applying the relocation step. The procedure consists of four hot basic bocks labeled BB1 - BB4, and two frozen basic blocks labeled BB5 and BB6. Figure 3 illustrates the same procedure after relocating the frozen code to a separate area. The frozen basic blocks BB5 and BB6 are placed together in a separate (non-loadable) section and each control transfer to them from the other basic blocks is replaced with an illegal instruction, containing the offset target of the callee basic block within the area it was relocated to. The loading subroutine, which includes the code for intercepting the trap created when trying to execute the illegal opcodes, is placed in a different location of the non-frozen code area. The dashed lines represent the control transfer between the loaded frozen and the non-frozen code via the interrupt mechanism. Figure 4 illustrates a scenario in which the frozen basic block BB6 is thawed after being referenced by a non-frozen branch instruction during run-

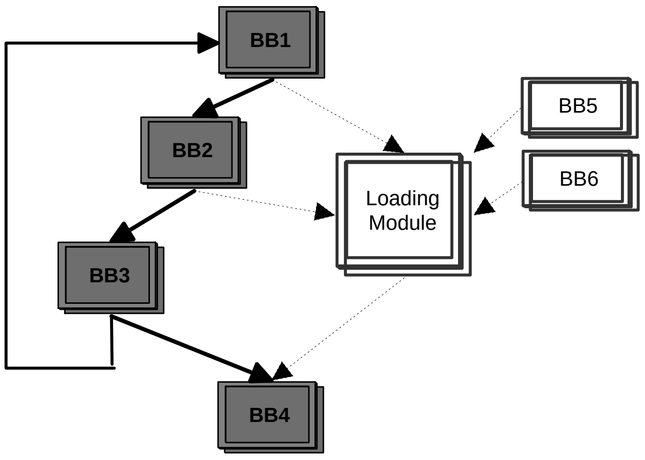


Figure 4: Basic Blocks layout after basic block thawing at run-time

time. As a result, the loading subroutine is invoked to load BB6 into a dynamically allocated memory area and to update the control transfer to and from the allocated BB6 code area, directly to its neighboring basic blocks BB2 and BB4.

The method for reducing the program image 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, 18, 21, 22, 24, 27].

2. Relocate all the frozen code segments into a single group and then place it in a non-loadable 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. Divide the frozen code segment into regions. A region is defined as a sequence of instructions in which every control transfer that branches within the scope of a region remains unchanged when loaded to memory. Inter-region branches, or branches back to the non-frozen code area, are replaced with appropriate invalid instructions. In the event that some frozen instruction is referenced at run-time, its entire containing region is loaded into memory. Each region is defined by its size and starting offset in the frozen segment. The frozen code area also includes a "region map" (located on disk or in secondary memory). This map supports efficient mapping from targeted offsets within the frozen code area to their corresponding containing regions.

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

5. Update each of the control transfer or fall through instructions that were identified in step 4, to invoke an interrupt. In this work the interrupts are 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 loading subroutine is automatically invoked. In order to occupy as little space as possible, minimal illegal code instructions are used. A single invalid instruction is comprised of a 5-bit zero opcode, followed by 1 *characterization bit*, and then followed by 26 bits of the target offset (calculated in step 4) of the instruction in its corresponding section. The invalid instruction does not exceed a regular fixed instruction opcode of 32-bits. The *characterization bit* is used for specifying whether the targeted code falls in the non-frozen text section (when called from the frozen section) or in the non-loadable frozen code section.

   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 instruction 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 described above. If the condition is non-reversible, an invalid instruction followed by the appropriate target offset is inserted immediately after the conditional branch.

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

   (e) For an indirect branch instruction via a register or memory, we differentiate between three types of indirect branch instructions:

      - **Branch via branch table:**[1] For such branches into or out of a frozen code area, each relocated target is replaced by an invalid instruction as described above.
      - **Function return:** For a branch instruction returning into or out of a frozen area, we replace the instructions following each call to that function, by invalid instructions as described above.
      - **Indirect function call:** For each function that can be called indirectly and is located in the frozen code area, an invalid instruction is added to its prologue.

[1]Generated by compilers mainly for efficient implementation of switch statements.

6. 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 loading subroutine is capable of loading the appropriate region from the relocated region and load it into the "thawed area". The loading subroutine also loads the code for intercepting the trap generated by the invalid instructions that were inserted in step5. When invoked at run-time:

   (a) The loading subroutine uses the characterization bit b and the target offset S of the referenced basic block in order to locate the region within the non-loadable frozen code section (reference to frozen code) or within the loaded text section (reference to non-frozen code).

   (b) If the target offset S is in the non-frozen code area, then retrieve its run-time address $S + B$ in memory, where B is the base loading address of the entire non-frozen code segment in memory.

   (c) If the target offset S is located in the non-loadable frozen code section, then:

      - The loading subroutine checks whether the referenced frozen instruction was already loaded into memory. Targeted instructions are marked *loaded* by maintaining a dynamic map (in our work implemented by an open hash table) containing all loaded target addresses in memory. The target addresses table is constantly maintained and updated by the loading subroutine.
      - If the referenced code is not in memory, the subroutine then:
         i. identifies the region containing the target offset S using the region map of the frozen code created in step 3.
         ii. dynamically allocates additional memory space for the region. [2]
         iii. loads the said frozen region into the allocated memory at address T.
         iv. marks the region as *loaded* by adding a new entry in the target addresses table.
      - If the referenced offset S is already in memory, then retrieve its address from the target addresses table.

   (d) Replace the triggering invalid instruction by a branch instruction to the target instruction in the thawed region [3]. If a direct jump cannot reach the targeted instruction, the triggering illegal instruction is replaced by a call instruction to an indirect jump wrapper code (added to the end of the non-frozen code area). This is similar to the mechanism that the linker uses in order to perform a far jump from one module to another. The wrapper code loads the targeted instruction from the target addresses table into a register and jumps via that register. [4]

   (e) Finally, the loading subroutine branches to the target address of the referenced instruction in order to continue execution.

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

In this work we replaced illegal instructions by appropriate branch instructions only when reached at run-time to trigger an interrupt. The loading subroutine may be further extended to replace all illegal instructions within a promoted region, containing target offsets already in memory. Such an enhancement to the loading subroutine can improve performance but will complicate its implementation.

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 loading mechanism for handling static variables in the IBM PowerPC uses a table, referred to as the Table Of Contents (TOC), consisting of the addresses of all static variables in the program. As a result, the instruction for loading a static variable has the following form:

load rt, off(r2)

where the $r2$ register, referred to as the "TOC anchor" register, holds the address of the TOC in memory, and serves as an anchor from which the offset off is used in order to reach the TOC entry containing the required variable's address[5].

In our proposed reduction mechanism this instruction is replaced by an illegal instruction comprised of an invalid opcode (different from the frozen code illegal instruction) followed by the same original encoding data $rt, off(r2)$. The corresponding entry in the TOC is then modified to hold the offset of the frozen variable within the non-loadable frozen data section. In the event that a frozen static variable is referenced at run-time, the loading subroutine triggered by the illegal instruction, allocates memory for the variable, loads it into the allocated area, replaces the content of its TOC entry by its loaded address in memory and modifies the triggering illegal opcode back to its original load instruction.

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

[2]If no available memory is left for allocating the region, least recently used regions may be freed from memory

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

[4]Note that two copies of the jump wrapper code may be needed in order to be reached from both the thawed and the non-frozen code areas

[5]On a platform that doesn't use a TOC, an alternative approach that mimics a TOC will be used.

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 [6].

2. Use the profiling information to identify all frozen data variables within the static data area. In this work, we use the profiling information to check whether the aforementioned 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).

4. Repopulate the contents of each frozen variable entry in the TOC, to hold the offsets of each frozen variable in the non-loadable section and attach a *characterization bit* with the value of 1 to the end of each offset. A value of 1 for the *characterization bit* indicates that the variable's offset falls in the non-loadable frozen data section.

5. Update each of the load instructions which refer to a frozen data variable, identified in step 1, by an invalid instruction containing an invalid opcode of 5 zero bits followed by the same encoding data of the original instruction. 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.

6. 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. When invoked at run-time:

   (a) The loading subroutine verifies whether the referenced frozen data was already loaded into memory using the characterization bit *b* attached to each offset in its corresponding TOC entry in step 4.

   (b) If the frozen data is not yet in memory, the loading subroutine then:

      i. loads the given frozen data variable into a dynamically allocated memory area.

      ii. replaces the content of its TOC entry by its loaded address in memory. This will automatically override the characterization bit *b* attached to the end of the previous offset, back to zero.

   (c) The loading subroutine then modifies the opcode of the triggering illegal instruction prefixed by the zero bits, back to its original load instruction opcode.

   (d) Finally, the loading subroutine returns the control back to the beginning of the modified load instruction, in order to continue with the normal execution of the program. In this work, a thawed frozen data variable is no longer considered frozen and therefore, remains in memory throughout the entire application execution cycle.

7. 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.

Note that when applying code reduction together with static data reduction, we can avoid duplicating the proposed reduction mechanisms, by automatically loading the static data referenced from thawed code regions.

5. EXPERIMENTAL RESULTS AND ANALYSIS

The technique proposed here was analyzed using a post-link optimization tool called FDPR (Feedback Directed Program Restructuring) reported in [8, 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 statistics presented here. In this section, we will analyze two benchmark suites: SPEC CPU2000 [26] and MediaBench [13] and show the percentage of frozen code and data they posses.

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 [26]: "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 imagine that applications such as *gcc* (C compiler), *vpr* (circuit placement), or *twolf* (circuit simulation) running on handheld devices, others such as *gzip* (compression), *parser* (word processing), and *eon* (visualization) are sure to be. And while many embedded processors do not support floating point operations, many others do so even better than desktop processors [16], leading us to include the SPEC CFP2000 suite in our analysis.

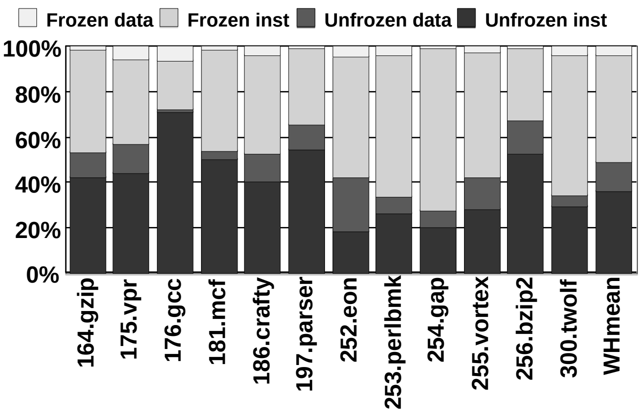
We believe that the types of applications presented in the suite will migrate to embedded processors while its successor, CPU2004, will be used in the workstation domain. As a consequence, we chose to analyze 32-bit, rather than 64-bit, executables. The C/C++ benchmarks were compiled on a Power4 running AIX [6] 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 [7]. This dataset was tailored to provide shorter running, yet representative executions. It is used extensively by profile based tools. Figure 5 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 MediaBench

The MediaBench [13] suite was compiled in 1997 by Lee, Potkonjak, and Mangione-Smith in order to provide a suite of applications for the embedded domain. The benchmarks are supplied with two datasets, which enables picking one as a *train* set and the other as a *reference set* (see [19] for specific input descriptions). Table 1 lists the inputs used for

[6]See section 6 for details regarding other platforms.
[7]Benchmark and dataset details can be found in [26].



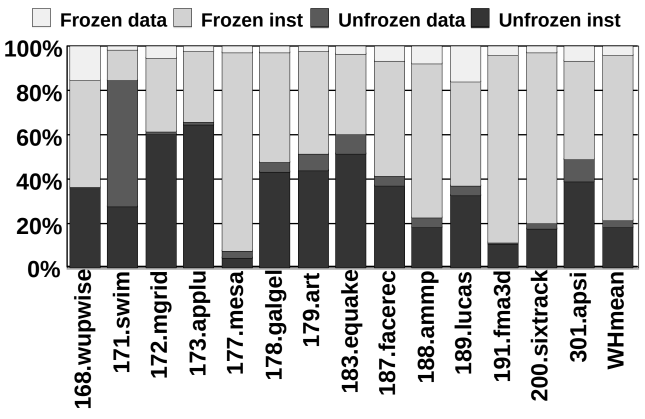


Figure 5: Percentage of frozen code and data in the SPEC CINT2000 and CFP2000 suites.

Table 1: MediaBench benchmarks and datasets.

Benchmark	*mode*	*Train input*	*Ref. input*
adpcm	dec	clinton.adpcm	S_16_44.adpcm
adpcm	enc	clinton.pcm	S_16_44.pcm
epic	dec	test_image.pgm.E	titanic3.pgm.E
epic	enc	test_image.pgm	titanic3.pgm
g.721	dec	clinton.g721	S_16_44.g721
g.721	enc	clinton.pcm	S_16_44.pcm
ghostscript	dec	tiger.ps	titanic2.ps
gsm	dec	clinton.pcm.gsm	S_16_44.pcm.gsm
gsm	enc	clinton.pcm	S_16_44.pcm
jpeg	dec	testimg.jpg	monalisa.jpg
jpeg	enc	testimg.ppm	monalisa.jpg
mpeg2	dec	mei16v2.m2v	tek6.m2v
mpeg2	enc	options.par	-
pegwit	dec	pegwit.dec	-
pegwit	enc	pegwit.enc	-

each benchmark[8] Most of the benchmarks are composed of two executables, an encoder and decoder, we shall present them as different applications [9].

Figure 6 shows the percentages of frozen code/data in the MediaBench suite. In these applications the ratio is even better than for CPU2000 and is 76/82% (code/data), which results in an average reduction of 78% in the image size. All the results in this section show that the potential for image size reduction is promising.

5.3 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 infrequent 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.

[8]We have omitted *pgp*, and *rasta* due to difficulties in compiling, profiling, and executing them.

[9]*mesa* is part of CFP2000 and omitted to avoid duplication.

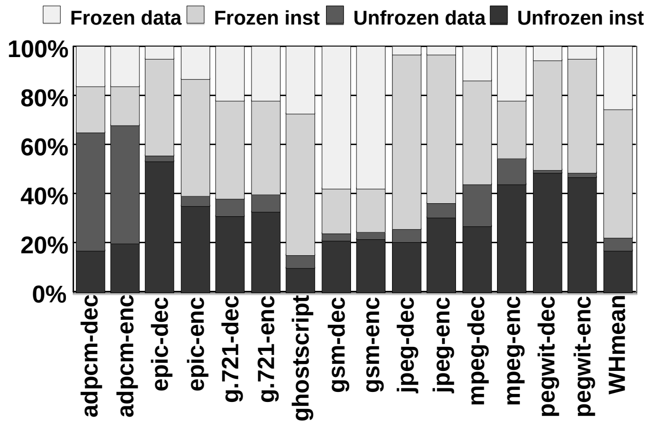


Figure 6: Percentage of frozen code and data in the MediaBench suite when the alternative input set is used.

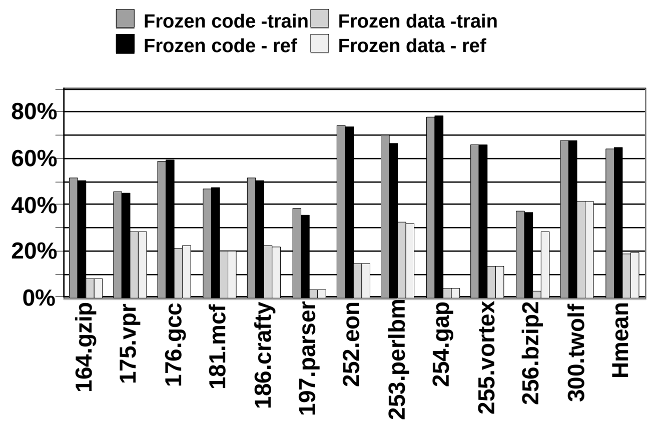
Thus, in order to quantify the quality of the training runs, we compared the amount of frozen code/data in both the train and reference datasets. Figures 7 and 8 show these comparisons for the CPU2000 and MediaBench[10] suites. The results show that the differences are indiscernible (except for *g.721*, which displays a slight variation). However, they are not identical. Table 2 summarizes the average differences in size and dynamic instruction count (both in absolute numbers and ratios).

6. DISCUSSION AND SUMMARY

The results in the preceding section indicate that there are code segments that may become unfrozen for different workloads. This indicates 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 secondary memory for each singular misprediction, which

[10]Couldn't obtain reference datasets for *mpeg2-encode* and *pegwit*.



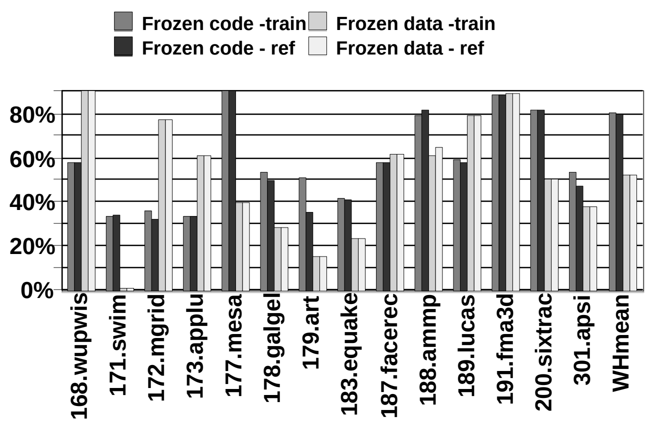


Figure 7: Comparison of frozen code/data between the train and reference data sets of CPU2000.

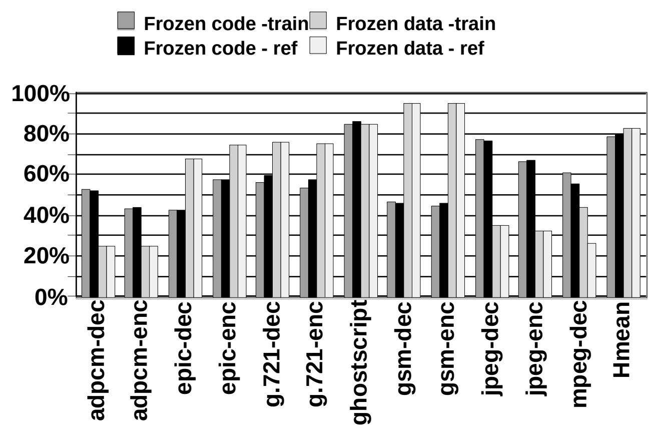


Figure 8: Comparison of frozen code/data between the train and reference data sets of MediaBench.

Table 2: Average differences (weighed harmonic mean) of frozen code/data between train and reference datasets.

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
MediaBench	code	KB	0.3
		%	0.09
	data	KB	0.05
		%	0.08

may take from micro to milliseconds, depending on the distance, state, and speed of the storage device and I/O channel. However, for every singular misprediction, we pay this penalty only on the 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 behavior between the train and the ref workloads. Therefore, the numbers presented here represent the worst-case scenario for our proposed mechanism on SPEC CPU2000.

The actual size of the gcc code that was considered frozen (based upon the train workload) yet turned out to be non-frozen with the reference workload, is around 4000 bytes, which is around 200 basic blocks. The entire gcc code contains 95,000 basic blocks, making the number of singular mispredictions 0.2% of all basic blocks. In addition, it turns out that all singular mispredictions are considered cold, i.e., rarely executed even on the reference workload. Thus, the the number of singular mispredictions is relatively small, and will most likely not result in a significant overhead.

Our prototype system was developed on a non-embedded system (AIX on a Power4) which might not need and exploit the full potential of the system. In order to partially test its usefulness we ran our experiments on a Linux system (2.6.5-7-pseries64) compiled with gcc version 3.3.3, the frozen code/data ratios are virtually the same. As many embedded systems use versions of Linux as their operating system [7] this is a first step in verifying our technique for embedded systems.

Nevertheless, for hard real-time applications and/or systems this technique might not be suitable. Our assumption that error handling code can suffer degraded performance is not true in every situation: error handling or rare situation code might be time critical (handling a nuclear reactor overload, for instance). Developers must take care to categorize and analyze their applications before applying this (or any other compression) scheme. For systems that have stricter timing guidelines, profiling on several distinct input sets, marking code as non-freezable, and analysis of thawed code can reduce both the size of the executable and the chance of accessing frozen code or data.

6.1 Summary

This paper presents a technique that reduces the runtime image of executables by stripping them of frozen code and data and storing them in secondary storage devices. The frozen code/data areas are detected using profiling techniques on representative runs. The executable is then restructured in order to bunch all frozen code/data together and accesses to these areas are replaced with interrupts. In the unlikely case that these code/data areas are accessed, a mechanism that promotes them to main memory is engaged. Our results have shown that this happens with less than 1% of the code/data. This techniques yields image sizes on the SPEC CINT2000, CFP2000, and MediaBench that are reduced by an average 59%, 79%, and 78% per suite.

7. REFERENCES

[1] G. A. B. Schwarz, S. Debray and M. Legendre. PLTO: A Link-Time Optimizer for th eIntel IA-32 Architecture. 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] http://www.gzip.org.

[5] G. Haber, E. A. Henis, and 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] G. Haber, M. Klausner, V. Eisenberg, 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.

[7] R. Harr. A Conversation with Jim Ready. *ACM Queue*, 1(2):6–15, April 2003.

[8] E. A. Henis, G. H. M. Klausner, 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 330–335, December 1997.

[14] 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.

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

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

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

[18] 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.

[19] MediaBench, http://cares.icsl.ucla.edu/MediaBench.

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

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] 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.