Profile-Driven Selective Program Loading

Tugrul Ince and Jeffrey K. Hollingsworth

Computer Science Department University of Maryland College Park, MD 20742
{tugrul,hollings}@cs.umd.edu

Abstract. Complex software systems use many shared libraries frequently composed of large off-the-shelf components. Only a limited number of functions are used from these shared libraries. Historically demand paging prevented this from wasting large amounts of memory. Many high end systems lack virtual memory and thus must load the entire shared library into each node's memory. In this paper we propose a system which decreases the memory footprint of applications by selectively loading only the used portions of the shared libraries. After profiling executables and shared libraries, our system rewrites all target shared libraries with a new function ordering and updated ELF program headers so that the loader only loads those functions that are likely to be used by a given application and includes a fallback user-level paging system to recover in the case of failures in our analysis. We present a case study that shows our system achieves more than 80% reduction in the number of pages that are loaded for several HPC applications while causing no performance overhead for reasonably long running programs.

1 Introduction

Software systems have been constantly getting more functional and complex. Most systems are not composed of a single executable file any more; they are a combination of many executables and shared libraries. These executables often use components developed by others. Frequently, users of these general purpose shared libraries are interested in only a fraction of a library's functionality.

Ideally, only the necessary parts of shared libraries should reside in main memory during execution. Traditionally systems have relied on demand paging to only load those parts of libraries that are actually used into memory. However, systems such as IBM's BlueGene and Cray XT series lack local disks on each compute node and therefore avoid virtual memory and demand paging for performance reasons [1,2]. Thus the available memory on such systems is limited to what is physically available. Moreover, this memory is shared between applications and their data. Reducing the memory footprint of application text space is therefore crucial for large and complex applications that deal with large datasets.

During launch of an application, the executable file and all of shared libraries are loaded into memory [3]. Typical applications today use several large shared libraries and relatively small executable files. A typical PETSc application takes

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 62–73, 2010.
© Springer-Verlag Berlin Heidelberg 2010

over 16 MB of memory to load the application and shared libraries although the actual executable only occupies 0.02 MB of that space. Many large US DOE applications have text segment sizes of around 100 megabytes.

In this work, we propose a system that reduces the memory footprint of shared libraries by eliminating unused parts of libraries from an executable. Our approach relies on an efficient profiling mechanism that lets us determine a list of functions that are not executed in the most common case. Our tool requires manipulating the way shared libraries are loaded into the memory. On systems that support Executable and Linkable Format (ELF), this is achieved by modifying program headers such that the unused functions are not loaded into memory when the program is launched. If, for some reason, any function that has not been loaded is accessed at runtime, our system includes an error recovery mechanism that loads that function into the memory and allows the application to continue execution.

2 Architecture Overview

Figure 1 shows the architecture of our system. It is composed of a profiler, and a rewriter.

Our design obeys the "make the common case fast" motto. A profiler is used to get a trace of executed functions for a given application. Then, a list of unused functions is generated for each shared library. Since code has to be loaded in page-sized units, removing a single function does not save any pages since there is usually other code around that function that still needs to be loaded. Therefore, we need to re-arrange code and group unused functions so that we can remove them from the loadable sections altogether. Our tool moves all unused functions to the end of the code section and modifies ELF program headers to make those parts unloadable. Finally, our tool writes the modified shared libraries to the disk to make the changes permanent. These rewritten libraries are used during subsequent production runs without further processing.

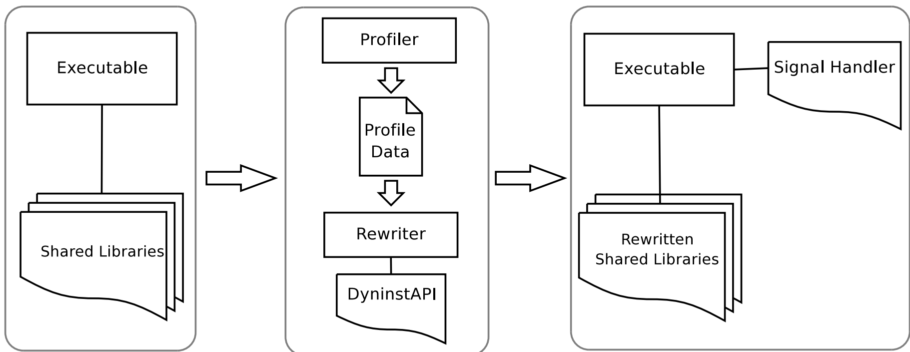

Fig. 1. Overview of the system: Executables and shared libraries are profiled and rewritten

3 Target Applications and Platforms

Our system can be applied to any executable file or any shared library on any ELF platform (e.g. most Unix-based systems). However, for efficiency and simplicity, we limit ourselves to certain kinds of applications and platforms. Our prime target is applications that use a limited number of functions from many shared libraries.

High end HPC systems benefit significantly from our system. These systems generally do not support demand paging since they do not have disks on compute nodes. Therefore, the available memory is limited. Since these systems heavily rely on running parallel applications on thousands of nodes, saving even very little memory space on each node adds up quickly. As a result, we assert that parallel applications and clusters are a good match for our system. Although we only support x86 systems for now, our solution is also suitable for systems like BlueGene too.

4 System Design

Our system is composed of three main components: A profiler, a rewriter, and a signal handler. Profiling is performed to identify a list of functions that are used during regular runs of the executable. Profiling data is fed to our binary rewriter which uses Dyninst [4] to access functions, their control flow graphs, basic blocks, and finally instructions. For each shared library, our binary rewriter performs the following tasks:

1. Calculation of updated start addresses for each function that is being moved. Functions that are used often will be placed before the functions that are rarely or never used.
2. Code generation for moved functions. Call instructions, address calculations for global offset table (GOT), contents of jump tables, and function pointer calculations are updated using the new locations.
3. Symbol updates so that cross library calls can be directed to the correct location.
4. Rewrite of the updated shared library to a new file.

Our signal handler is used during actual execution for error-recovery.

In the following sections, implementation details and challenges of each process are discussed.

4.1 Profiling

In order to extract a list of functions that are usually not used by a program, we first observe several executions of the program and obtain a list of functions that are used by this program. We use a profiler to obtain this list of functions used during a specific run of the program. We profile the applications using different input datasets and node counts. We combine all training runs and note all functions ever called.

There are various profilers that serve different needs. For our analysis we used *sprof* [5] and our own tool based on Dyninst [4]. *sprof* is a GNU profiling tool for shared libraries. It profiles one shared library at each run. Our profiling tool, on the other hand, rewrites the shared libraries with instrumentation code and can profile all shared libraries at a single run.

4.2 Rewriting

Once the shared libraries are profiled, this profiling data is used to identify functions that will always be loaded, and those that will only be loaded on demand upon first call. Our system modifies LOAD entries in ELF headers so that the loader selectively loads functions that are known to be used. To maximize memory savings, functions are grouped into two sets: Used and Unused. Grouping functions requires moving around their machine code in the binary. Since the correct execution of an instruction usually relies on where it is located in binary, we perform extensive analysis to make sure that the external behavior of an instruction does not change once it is moved. This analysis involves updating any relative addressing operation, updating GOT address calculations, and updating function address transfers. The shared library is then rewritten to the disk.

Our tool is not as aggressive on executable files as it is on shared libraries. It only links them with a signal handler to handle faults for an unloaded page.

Avoiding Loading Unused Functions: To reduce the memory footprint of applications, we first need a mechanism that will allow us to avoid loading parts of executables and shared libraries while enabling the process to access these regions when necessary.

One way of getting around this problem is to split each shared library into two shared libraries: one that contains functions that will likely be called, and one that contains the remaining functions. If a function that is not currently available is called during runtime, a signal handling mechanism could load the shared library that contains this function and transfer control to that function. However, this scheme requires loading this whole shared library even though there is a single function that is used. As a result, it is not very effective in recovering from an unexpected call. Moreover, splitting shared libraries is complex since it requires moving most functions and symbols while regenerating the symbol table and procedure linkage table so that cross-library calls can be satisfied. It also requires adding a mechanism to access global variables across shared libraries.

Our mechanism, on the other hand, is very simple and effective. During the rewriting phase we modify appropriate ELF headers to accommodate selectively loading chunks of the original program or shared library. Loaders rely on program headers in ELF binaries and only load parts of the binary that has a matching LOAD entry in the program headers. Our rewriting mechanism modifies these LOAD entries by changing the address ranges and adding new ones if necessary so that it only loads desired (used) part of the library. The loader then loads the appropriate regions in the binary, leaving out the parts that are unlikely to be used.

Update Relative Calls: In a shared library with position independent code, most call instructions employ relative addressing. The exact target address is calculated using the current program counter and the offset that is stored in the call instruction. These offsets need to be updated during the function shuffling process if either callee or caller is moved. When the relative position of a call instruction changes with respect to its target, the address computation generates an incorrect address for the target unless the offset in the call instruction is correctly updated. As a result, our rewriting mechanism goes over every such instruction and updates the offsets to match the modified layout.

Update Symbols: There is no guarantee that a given shared library will always be loaded at a specific address each time a process launches. As a result, addresses of functions in shared libraries cannot be known before launch time. Moreover, if a call instruction and its target are in different shared libraries, their relative position with respect to each other cannot be known before launch time. In such cases, any call instruction, rather than jumping directly to the callee, has to go through the procedure linkage table. The dynamic linker looks up the callee upon the first call to that function. The look up process consists of matching the mangled name of the callee with a list of symbols that appear in the shared libraries. When a matching symbol is found, the address it contains is written to the corresponding procedure linkage table entry.

If a function is moved within a shared library without updating corresponding symbols, the dynamic linker cannot correctly look up for the actual address of this function since the symbol information still points to the old location of this function. Our rewriting mechanism locates such symbols and updates them with the new addresses of associated functions.

Update Jump Tables: Most current compilers make use of jump tables for n-way branches (e.g. switch statements in C). During the compilation process, such control flow structures are converted into an indirect jump instruction that reads addresses of targets from a table called jump table. In a shared library that contains position independent code, jump tables contain offsets rather than absolute addresses. These offsets correspond to the difference between the address of each target and the address of the global offset table of that specific binary.

If the function referenced in a jump table is moved, the jump table becomes invalid because the relative offsets of the function within the library have changed. Our rewriting mechanism updates each jump table entry for moved functions. We use Dyninst [4] to locate the jump tables for us since they are not marked by the compilers. An offset is computed such that it equals the displacement of the moved function from its old position in the shared library to its new one. That offset is added to each entry in the jump table associated with an indirect jump in this function.

Update Function Address Transfers: Function pointers are simply variables that contain addresses of functions. A function pointer becomes invalid when the associated function is moved to another address. Our tool recognizes

writes to function pointers and updates them accordingly if the target functions are moved.

Since addresses of functions in shared libraries cannot be known before runtime, there has to be some runtime computation for writes to function pointers. There are two ways these addresses are computed:

1. Computed by the loader: Each function pointer that resides in the data section has an associated entry in the relocation table. The loader updates these function pointers during relocation. Moving a function invalidates associated function pointer. Our tool checks each relocation entry and updates the ones that point to moved functions so that they will point to the correct location after relocation.
2. Using global offset table address at run-time: In some cases, function addresses are computed using relative displacement of a function from the start of the global offset table. Moving the target function to another location requires updating this computation. This case requires thorough analysis since the address computation might take place at any valid code region. Therefore, an instruction-by-instruction analysis is performed to identify such computations. Once they are identified, offsets used in the address computation are updated.

4.3 On-Demand Mapping

Our system provides a mechanism to recover when a function we did not load is called. As part of the offline analysis, the executable file is linked with a new shared library that contains a signal handler. During the execution, if the control is transferred to some instruction that is not available in memory, the process generates a segmentation fault signal (SIGSEGV). Our signal handler, in turn, locates that function and maps it into memory (In reality, the whole page that contains this function is mapped since most systems only allow mapping an entire page). The execution then resumes, and the function that has just been loaded takes control.

5 Experimental Results

To demonstrate our system, we performed our analysis on two sample PETSc applications (ex2 from the ksp package and ex5 from the snes package) [6] and a physics application, GS2 [7,8].

PETSc (Portable, Extensible Toolkit for Scientific Computation) "is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations." It uses MPI for parallelization. It has linear and non-linear equation solvers and supports C, C++, Fortran and Python. The sample PETSc program we used has 79 lines of code and is linked with 25 shared libraries. PETSc suite is composed of 879,772 lines of code.

GS2 is a physics application developed to study low-frequency turbulence in magnetized plasma. It is typically used to assess the microstability of plasmas produced in the laboratory and to calculate key properties of the turbulence which results from instabilities. It is also used to simulate turbulence in plasmas which occur in nature, such as in astrophysical and magnetospheric systems. GS2 is composed of 53,105 lines of code and is linked with a total of 20 shared libraries.

All shared libraries we examined were compiled with the debug flag on and without optimization. We used Open MPI [9] for an implementation of message passing interface.

5.1 Environment

We tested our system on a 64 node cluster owned and operated by UMIACS at the University of Maryland. Nodes are connected using Myrinet. Each node has two 32-bit x86 processors and an off-the-shelf Linux distribution.

5.2 Results

Tables 1, 2 and 3 show how much saving one can achieve on a typical application. In our experiments the space savings of the text space ranged from 34.6% to 100% for all shared libraries over 7 KB of text space. The total weighted average of space savings is 82.0%. There are some libraries that are used fairly often such as libopen-rte.so, which is a library in the Open MPI suite. On the other hand, some libraries such as libMdsLib.so are not used at all although they are linked with the application.

Figure 2 shows a comparison of normalized running times between the original and modified applications. Error bars show the normalized standard deviation of running times for each type of execution. The difference in running times is not large enough to make any conclusive statements. Our observations have supported the assertion that our tool does not cause any performance overhead for applications that run more than a few seconds.

In our experiments, modified GS2 runs took 5 seconds less than the unmodified program (36 minutes 33 seconds vs. 36 minutes 38 seconds). Functions used by modified binaries occupy fewer pages; therefore, cache misses might be less frequent. Also, the paging system of the operating system might be spending less time loading and unloading pages.

On the other hand, applications that run for only few seconds might experience some slowdown due to initial signal handler registration. *ex5* from PETSc's *snes* package takes 1.05 seconds on average. In our experiments modified executable experienced about 19% slowdown since it did not run long enough to compensate for the initial signal handler registration. Conversely, *ex2* from PETSc's *ksp* package runs for about 2.7 seconds and the modified executable experiences more than a 6% speedup. This result shows that this application runs long enough to compensate for the initial cost of running modified binaries.

Table 1. PETSc results for *ex5* from snes package

Library Name	Original		Modified		Text (Pages)	Text (KB)
	Text (Pages)	Text (KB)	Text (Pages)	Text (KB)	Reduction %	Reduction %
petsc	260	1034	68	266	73.85	74.24
petscdm	161	640	19	72	88.2	88.79
petscksp	335	1337	39	153	88.36	88.59
petscmat	772	3085	40	157	94.82	94.92
petscvec	204	813	52	205	74.51	74.76
petscsnes	20	77	20	77	0	0
mpi_cxx	10	36	5	16	50	54.93
mpi	142	564	37	144	73.94	74.45
open-pal	62	241	34	129	45.16	46.48
open-rte	55	215	34	131	38.18	39
m	28	108	3	8	89.29	92.27
X11	146	578	7	22	95.21	96.13
lapack	866	3458	2	2	99.77	99.94
blas	80	315	3	7	96.25	97.9
stdc++	133	529	12	45	90.98	91.54
gcc_s	12	45	2	5	83.33	88.95
Xau	2	3	2	3	0	0
Xdcm	3	7	3	7	0	0
gfortran	123	485	4	9	96.75	98.13
dl	2	4	2	4	0	0
nsl	14	55	2	7	85.71	87.59
util	2	2	2	2	0	0
TOTAL	2021	13632	348	1472	82.78	89.2

Table 2. PETSc results for *ex2* from ksp package

Library Name	Original		Modified		Text (Pages)	Text (KB)
	Text (Pages)	Text (KB)	Text (Pages)	Text (KB)	Reduction %	Reduction %
petsc	260	1034	72	282	72.31	72.73
petscdm	161	640	3	8	98.14	98.75
petscksp	335	1337	49	193	85.37	85.56
petscmat	772	3085	49	193	93.65	93.74
petscvec	204	813	54	213	73.53	73.8
mpi_cxx	10	36	5	16	50	55.56
mpi	142	564	47	184	66.9	67.38
open-pal	62	241	37	141	40.32	41.49
open-rte	55	215	36	139	34.55	35.35
TOTAL	2001	7965	352	1369	82.41	82.81

Table 3. GS2 results

Library Name	Original		Modified		Text (Pages)	Text (KB)
	Text (Pages)	Text (KB)	Text (Pages)	Text (KB)	Reduction %	Reduction %
MdsLib	21	80	0	0	100	100
MdsShr	21	80	0	0	100	100
TdiShr	220	875	3	9	98.64	98.97
TreeShr	38	150	0	0	100	100
fftw	70	276	25	96	64.29	65.22
rfftw	58	228	8	28	86.21	87.72
mpi_f77	13	48	2	4	84.62	91.67
mpi	142	564	40	156	71.83	72.34
open-pal	62	241	36	137	41.94	43.15
open-rte	55	215	36	139	34.55	35.35
TOTAL	700	2757	150	569	78.57	79.36

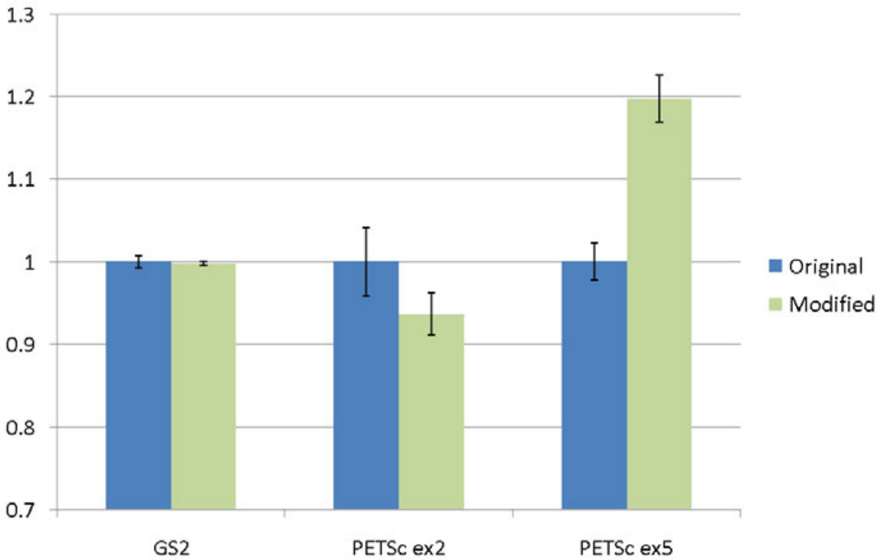

Fig. 2. Normalized running times. Error bars show the standard deviation for that category.

Since most HPC applications take several minutes to complete, our impression is that modified binaries will likely not cause any overhead and might cause some speedup.

Just like any other on-demand tool, the performance of our system might suffer due to mapping. However, this operation is rarely necessary. We ran experiments on GS2 to show how often such error recovery operations are triggered at

runtime. Our tool had to load only 7 pages out of 700 pages that contain code. Each such mapping takes 3.1 miliseconds on our system, averaged over 5000 forced mappings observed during a separate experiment, and is transparent to the user.

6 Related Work

Reducing the memory footprint of programs has been extensively researched. Previous works include a wide range of techniques from code compression [10,11,12,13] to procedure abstraction [14,15,16] and dead code elimination [16].

Code compression is the act of reducing the size of program code by its equivalent representation in another form [17]. It is usually applied to executables that run on embedded systems. Xie et al. developed a system where only the instructions that are least frequently used are compressed [12]. Just like we do, they first profile the executable and identify the regions that are least likely to be used. These regions are then compressed. They leave frequently accessed regions uncompressed to reduce the performance hit. A decompressor generates the original uncompressed code if a block of code that was compressed is accessed at runtime. Lefurgy et al. evaluate a hardware assisted code compression system from IBM PowerPC 405 [13]. In this system all program code is compressed. They note that they achieve performance increase in many situations thanks to the prefetching of instructions. Since their system relies on CodePack hardware support available on PowerPC 405, it is restricted to this platform.

Other approaches to code size reduction techniques include dead and redundant code elimination, procedure abstraction, and instruction level modifications. [16] explains various code elimination methods including unreachable, redundant and dead code elimination. These methods are demonstrated in their binary-rewriting tool, *squeeze*, along with interprocedural constant propagation and strength reduction. They also perform procedure extraction for single entry-single exit sections at the binary level. They make use of various optimizations such as instruction reordering and platform specific improvements such as reducing the cost of function prologues and epilogues. Van Put et al. propose optimizations including constant propagation and unreachable code elimination as well as procedure extraction in their binary rewriter tool, DIABLO [18]. They also demonstrate how their system can be used to rewrite Linux kernel for specific embedded systems.

Komondoor and Horvitz propose procedure extraction at the source code level [14,15]. Zmily and Kozyrakis propose BLISS which successfully targets reducing text space, energy use and execution time [19]. They selectively replace 32-bit instructions with 16-bit instructions. Since more instructions fit into the instruction cache, performance of the system increases. They also remove repeated sequences of instructions leaving a single copy, just like procedure extraction. Lau et al. show how *echo* instructions can be used to remove duplicates of identical or similar regions of code [20]. *echo* is a proposed instruction that directs processor to execute a sequence of instructions in the binary. Authors perform

procedural abstraction as well as replacing similar sections of code with a single echo instruction.

Zhang and Krintz propose a system that unloads code regions from a modified java virtual machine after their execution is over [21]. They note that 61% of code is only used at the start-up period and can be unloaded after their execution. Although our system currently does not unload code regions once they are loaded, this functionality is a straightforward extension to our current system.

7 Conclusion

In this paper we proposed a new system that reduced memory footprint of executables linked with many shared libraries. After an offline rewriting phase, we managed to reduce the number of loadable pages in target shared libraries by an average of 82.0%. We also demonstrated that our tool causes no performance overhead for reasonably long running programs. Upon a call to a function that is not loaded into memory by the loader, our error recovery mechanism maps the page which contains that function into memory and continue execution without a failure. These properties make our system a desirable optimization for frequently executed applications with multiple shared libraries.

References

1. Adiga, N.R., et al.: An overview of the bluegene/l supercomputer. In: Supercomputing 2002: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, pp. 1–22. IEEE Computer Society Press, Los Alamitos (2002)
2. Kelly, S.M., Brightwell, R.: Software architecture of the light weight kernel, catamount. In: Cray User Group, pp. 16–19 (2005)
3. Levine, J.R.: Linkers and Loaders. Morgan Kaufmann Publishers Inc., San Francisco (1999)
4. Buck, B., Hollingsworth, J.K.: An api for runtime code patching. Int. J. High Perform. Comput. Appl. 14(4), 317–329 (2000)
5. Drepper, U.: Using elf in glibc 2.1. Technical report, Cygnus Solutions, Sunnyvale, CA (1999)
6. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2009), `http://www.mcs.anl.gov/petsc`
7. Kotschenreuther, M., Rewoldt, G., Tang, W.M.: Comparison of initial value and eigenvalue codes for kinetic toroidal plasma instabilities. Computer Physics Communications 88(2-3), 128–140 (1995)
8. Dorland, W., Jenko, F., Kotschenreuther, M., Rogers, B.N.: Electron temperature gradient turbulence. Phys. Rev. Lett. 85(26), 5579–5582 (2000)
9. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (September 2004)

10. Cooper, K.D., McIntosh, N.: Enhanced code compression for embedded risc processors. SIGPLAN Not 34(5), 139–149 (1999)
11. Debray, S., Evans, W.: Profile-guided code compression. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 95–105. ACM, New York (2002)
12. Xie, Y., Wolf, W., Lekatsas, H.: Profile-driven selective code compression. In: DATE 2003: Proceedings of the Conference on Design, Automation and Test in Europe, p. 10462. IEEE Computer Society, Washington (2003)
13. Lefurgy, C., Piccininni, E., Mudge, T.: Evaluation of a high performance code compression method. In: MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pp. 93–102. IEEE Computer Society, Washington (1999)
14. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 40–56. Springer, Heidelberg (2001)
15. Komondoor, R., Horwitz, S.: Effective, automatic procedure extraction. In: IWPC 2003: Proceedings of the 11th IEEE International Workshop on Program Comprehension, p. 33. IEEE Computer Society, Washington (2003)
16. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Trans. Program. Lang. Syst. 22(2), 378–415 (2000)
17. Beszédes, A., Ferenc, R., Gyimóthy, T., Dolenc, A., Karsisto, K.: Survey of code-size reduction methods. ACM Comput. Surv. 35(3), 223–267 (2003)
18. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology, Athens, pp. 7–12. IEEE, Los Alamitos (December 2005)
19. Zmily, A., Kozyrakis, C.: Simultaneously improving code size, performance, and energy in embedded processors. In: DATE 2006: Proceedings of the Conference on Design, Automation and Test in Europe, 3001 Leuven, Belgium, pp. 224–229. European Design and Automation Association (2006)
20. Lau, J., Schoenmackers, S., Sherwood, T., Calder, B.: Reducing code size with echo instructions. In: CASES 2003: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 84–94. ACM, New York (2003)
21. Zhang, L., Krintz, C.: Profile-driven code unloading for resource-constrained jvms. In: PPPJ 2004: Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, Trinity College Dublin, pp. 83–90 (2004)