GOLIAT: an Optimizing Linker for the IA-64 ArchiTecture

Bertrand Anckaert and Frederik Vandeputte

Ghent University
Department of Electronics and Information Systems (ELIS)
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

Email: {banckaer, fgvdeput}@elis.UGent.be

Abstract— **Modern compilers are extremely sophisticated and complex. This enables them to produce highly optimized code. However, due to the limited scope of compilers, the optimization opportunities that arise from analyzing and optimizing across module boundaries remain largely unexplored. We examined the viability of optimizing statically linked programs for the IA-64 architecture. As a result, the code size was reduced on average with 20% and the execution time with 4%.**

I. Introduction

SPEED has always been a key factor in the computing industry. This need for speed has its price, in this case complexity. Modern processors and compilers are extremely sophisticated. The IA-64 architecture for example incorporates many features to improve overall execution speed, like software pipelining, speculation, etc. On the other hand, more responsibility is given to compilers in order to exploit these features.

We investigate how we can optimize statically linked programs for the IA-64 architecture at link-time. With statically linked programs, all library code is included in the binary, which allows to perform some optimizations on the entire program. Previous research for the Alpha [1] showed that this approach is promising.

Goliat is built on top of Diablo [2], a framework for optimizing linkers. It allowed us to concentrate on the architectural details from the start, and leave the generic operations up to Diablo. As Diablo is mainly targeted towards compaction, we also investigated how we can make IA-64 binaries as small as possible.

II. IA-64 Architectural Issues

Compared to e.g. the simple Alpha architecture, the IA-64 architecture has a few properties that need special attention. One of these is the instruction format. A program consists of bundles of instructions, each bundle consisting of three instructions and a template, indicating the type of the instructions in the bundle.

As the template is only 5 bit wide, the number of combinations is limited. This means that a stream of instructions cannot simply be bundled one after another. Instead, NOP instructions have to be inserted to produce valid bundles. A careful mapping of instructions into bundles is necessary to minimize the number of NOP instructions.

Another important aspect of the IA-64 is the register stack. The purpose of the register stack is to minimize memory accesses by providing registers as parameter passing mechanism and temporary space, instead of using the usual stack. This is accomplished by performing register renaming at procedure calls and returns, so that the output registers of the calling procedure become the input registers of the called procedure. As will be discussed in section III, it affects analyses and optimizations and a careful modeling is therefore necessary.

III. Analyses and Optimizations

During our research, we developed and explored various optimizations and analyses, using the datastructures and functions already available in Diablo. Some standard optimizations we investigated are unreachable code and data elimination, liveness analysis and the corresponding useless code elimination, constant propagation, copy propagation and inlining.

Most of these optimizations require special attention because of the existence of the register stack. In fact, one has to simulate to some extent the register renaming. To do this, we used dummy instructions, thereby separating the standard algorithms from the architectural details.

To be more specific, a dummy instruction is inserted before and after every `call` instruction. Then, for liveness analysis [3] for example, the used en defined sets of those instructions are constructed in such a way that the modeling remains accurate. This can be done by looking at the size of the register stack frame, the number of local and output registers, and by marking certain of these registers as used and/or defined.

IV. Reducing Load Instructions

The IA-64 architecture has a special register known as the global data pointer (gp). This pointer is used to access the GOT section (i.e. Global Offset Table). Normally, there is one such table for each object file, containing the constant addresses relevant for that object file.

Each procedure of a module assumes that register gp points to the correct GOT table. This means that intermodular or indirect procedure calls first have to load the address of the correct GOT table into gp before making the actual call. When the procedure returns, register gp has to

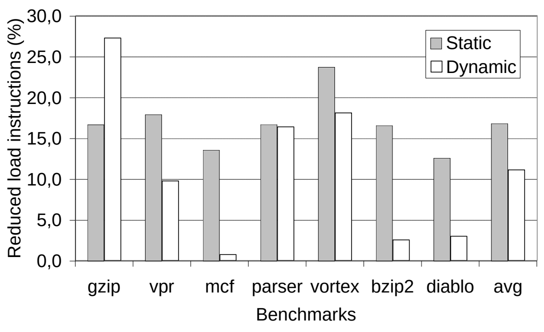

Fig. 1
Static and dynamic reduction of load instructions

be restored to its original value. This means an extra save and restore before and after procedure calls is needed.

With most statically linked programs however, there is only one such table. This means that register gp is constant during the entire execution of the program. So in fact, the save and restore instructions of register gp are redundant and can be eliminated. As these instructions appear quite often, removing these instructions reduces the code size with 4%.

Apart from that, another important optimization is possible. The GOT table is used to store constant addresses, so that they can be loaded into a register and used somewhere in the program. This means however that every time that address is needed, the correct address in the GOT table has to be calculated and a load has to be executed:

```
add rx = offset22, r1
ld ry = [rx]
```

A compiler looking at a single module does not know what the address will be in the final program nor whether it will be part of a shared library and must generate code that will work regardless. Whenever possible, it is far more efficient to encode those addresses directly into an instruction. As code and data addresses on the IA-64 architecture are 64 bit long and respectively start with `0x4..` and `0x6..`, we need to find an efficient way to encode these huge constants.

The IA-64 architecture provides an instruction to move a constant of 64 bit directly into a register. This instruction is expensive however, as it fills two slots of a bundle and the number of bundles containing this kind of instruction is very limited.

A better way is by trying to encode the address with the following instruction (with `constant22 + base = [rx]`):

```
add ry = constant22, base
```

For this to work, the base register has to contain a value of the form `0x4..` or `0x6..`. For data addresses, we are very lucky, because we can use register gp itself, as it contains the starting address of the GOT table, a data section.

As there normally is no register containing a constant code address, there is some extra effort needed there. Moreover, with this type of `add` instruction (having a 22 bit wide constant), only four global registers can be used as base register, two of them being a special purpose register.

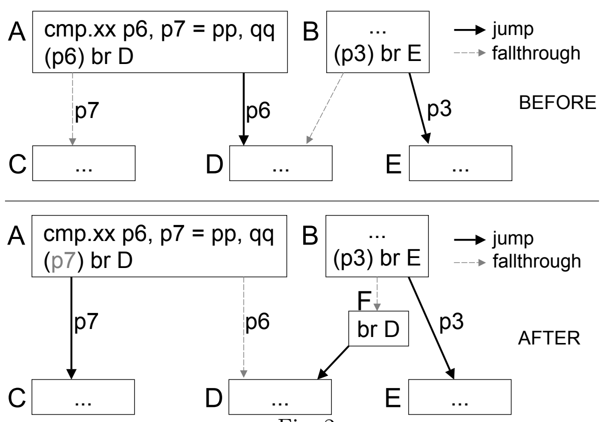

Fig. 2
Switching the fallthrough path of block A

Whatever register is chosen, it first needs to be freed in the entire program. Evidently, this step is quite intrusive and introduces some new problems, on which we will not elaborate.

As a result, many unnecessary load instructions are converted into simple and fast add instructions. As a byproduct, a preceding instruction, producing the address of the entry in the GOT table, can be eliminated by useless code elimination in most cases.

As you can see in figure 1, this optimization is very effective; many load instructions are eliminated, both statically as well as dynamically. As a result, this optimization has an important influence on the execution time of some benchmarks.

V. Code Placement

A good placement of hot code blocks can reduce the overall execution time, as it may reduce page faults, cache misses and the number of taken branches on hot paths [4]. In order to determine what the hot blocks and paths are, profile information is used.

For the code-layout, we used the closest-is-best principle [4]. By putting code blocks together that are frequently executed after one another, we reduce the probability that those blocks are mapped onto the same cache line. Moreover, those blocks will probably use up less memory pages.

Another profile-guided optimization is to minimize the number of jumps within hot code. For direct jumps, this can be done by putting the target block directly after the other, thus making the jump instruction redundant.

For conditional jumps, we can flip the predicate used with the jump instruction. In fact, the preceding compare instruction on the IA-64 architecture always produces two predicates with complementary values. The algorithm is illustrated in figure 2.

As a result of these optimizations, page faults are reduced by 35%, taken branches are reduced by 21% and instruction stalls by 38%.

VI. Instruction Scheduling

In section II, we mentioned that the mapping of instructions into bundles is very important. A good strategy is

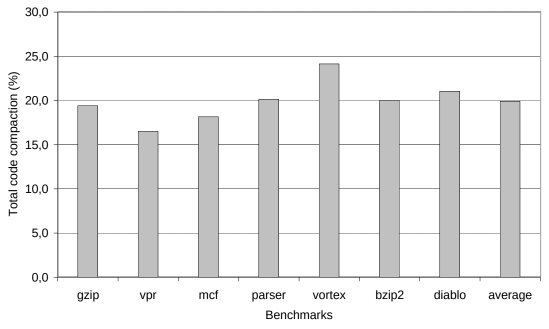

Fig. 3
Code compaction results

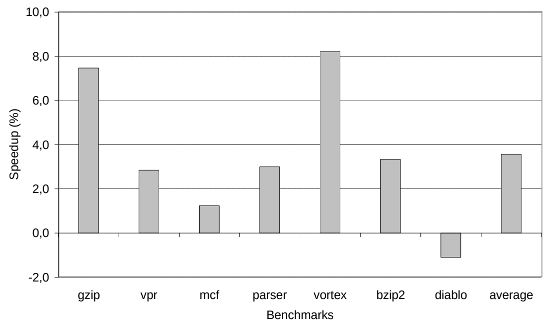

Fig. 4
Speedup results

to place the instructions of the hardest instruction types first [5]. Instruction types are called hard if there are few bundle types containing that instruction type.

Apart from bundling the instructions, they need to be scheduled first, meaning that an appropriate and efficient order must be determined. To do this, a dependency graph must be built, modeling the various dependencies that exist between instructions.

The goal is of course to have an optimal instruction scheduling as a result, but as the instruction scheduling problem is NP-hard [3], heuristics are inevitable.

There exist two types of scheduling algorithms, namely local scheduling algorithms and global scheduling algorithms [6]. Local algorithms operate within individual basic blocks, where global algorithms operate across basic blocks.

We implemented two local algorithms: list scheduling and the noptimizer. List scheduling is a very popular greedy algorithm, as it is simple, fast and effective [6].

We also developed another local algorithm, the noptimizer, which is based on the algorithm presented in [5]. The algorithm is a branch and bound version of the optimal scheduling algorithm, in order to keep the execution time within reasonable bounds. This algorithm is mainly targeted at reducing the number of NOP instructions that have to be inserted.

With this algorithm, the total number of NOP instructions can be reduced with 18% compared to the number of NOP-instructions in the original binary, reducing the total number of instructions with 5%.

We also developed a global scheduling algorithm, the globtimizer, which is roughly based on [7]. It is also a branch and bound algorithm and uses a cost function to determine the optimal result. This cost function can be the number of NOP instructions, the number of cycles, etc.

The idea is to move instructions between basic blocks upward and downward. Then those basic blocks are scheduled using a local scheduling algorithm.

Currently, only some very simple regions are considered. Despite that, the globtimizer is able to reduce the number of NOP instructions with 23%, if used in combination with the noptimizer for scheduling the basic blocks.

VII. Global Results

In figure 3 and figure 4, the total results we achieved with our research are illustrated: the code size is reduced with 20%, and execution time is reduced with about 4%. The original binaries were produced with gcc 3.2, using glibc 2.3.1.

The compaction is mainly achieved with dead code elimination (which eliminates redundant code blocks), our instruction scheduling and bundling algorithms and the reduction of load instructions. As you can see, the compaction ratio is uniform across all benchmarks.

The speedup is mainly caused by an improved code placement and by the reduction of load instructions. As you can see, the results show large variation. In fact, the results are very similar to the reduction of dynamic load instructions in figure 1, indicating that these load instructions are a real bottleneck for some benchmarks.

VIII. Conclusions

We have demonstrated that optimizing statically linked programs at link-time is indeed worthwhile for the IA-64 architecture, both towards speed as well as towards compaction. We showed that some properties of the IA-64 architecture need special attention in order to preserve correctness.

References

[1] R. Muth, S. Debray, S. Watterson, and K. De Bosschere, “alto : A link-time optimizer for the compaq alpha,” *Software – Practice and Experience*, vol. 31, pp. 67–101, Januari 2001.

[2] Bruno De Bus, Daniel Kästner, Dominique Chanet, Ludo Van Put, and Bjorn De Sutter, *Software Techniques for Program Compaction*, vol. 46, August 2003.

[3] S. Muchnick, *Advanced Compiler Design And Implementation*, Morgan Kaufmann Publishers, 1997.

[4] K. Pettis and R. Hansen, “Profile guided code positioning,” in *Proceedings of the ACM SIGPLAN '90 conference on programming language design and implementation*, July 1990, pp. 16–27.

[5] S. Haga and R. Barua, “EPIC Instruction Scheduling Based on Optimal Approaches,” in *1st Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, Austin, TX, 2001.

[6] B. De Sutter, “General-purpose architecture instruction scheduling techniques,” Tech. Rep. DG 98-09, Universiteit Gent, November 1998.

[7] S. Winkel, “Optimal global scheduling for itanium(tm) processor family,” in *Proceedings of the Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques*, November 2002.