Link-Time Optimization of IA64 Binaries

Bertrand Anckaert, Frederik Vandeputte, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere

Ghent University, Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41 9000 Gent, Belgium
{banckaer, fgvdeput, bdebus, brdsutte, kdb}@elis.UGent.be

Abstract. The features of the IA64 architecture create new opportunities for link-time optimization. At the same time they complicate the design of a link-time optimizer. This paper examines how to exploit some of the opportunities for link-time optimization and how to deal with the complications. The prototype link-time optimizer that implements the discussed techniques is able to reduce the code size of statically linked programs with 19% and achieves a speedup of 5.4% on average.

1 Introduction

On the EPIC (Explicitly Parallel Instruction Computer) platform, the compiler determines which instructions should be executed in parallel. This responsibility corresponds to the belief that better performance can be achieved by shifting the parallelism extraction task from hardware (as in superscalar out-of-order processors) to the compiler: the hardware becomes less complex and the compiler can exploit its much wider view on the code [8].

Unfortunately compilers only have a fragmented program view: most compilers compile and optimize all source code files independently of each other. Even when all source code is compiled together, the libraries are still compiled separately, and hence not optimized for any specific program. The resulting lack of compile-time whole-program optimization is particularly bad for address computations: As the linker decides on the final program layout in memory, code and data addresses are not known at compile time. The compiler therefore has to generate *relocatable* code, which is most often far from optimal.

Optimizing linkers try to overcome these problems by adding a link-time optimization pass in the tool chain. Optimizing linkers take compiled object files and precompiled code libraries as input, and optimize them together to produce smaller or faster binaries. In this paper we present our link-time optimizer for the IA64 architecture. Our main contributions are:

- We extend the existing work on Global Offset Table optimizations by creating a second global pointer at link-time.
- We show how existing link-time liveness analysis can be adapted to deal with the rather peculiar register files of the IA64 architecture.
- We demonstrate how the set of branch registers can be exploited more effectively with whole-program optimization.

This paper is organized as follows. Section 2 presents a short overview of our link-time optimizer. IA64-specific whole-program analyses and optimizations are the topic of Section 3. Our results are summarized in Section 4, and Section 5 discusses related work. We conclude in Section 6.

2 Link-time Optimizer Overview

Our link-time optimizer for the IA64 architecture is developed on top of Diablo [3] (http://www.elis.ugent.be/diablo), a portable and retargetable link-time program editor framework. Any application developed with Diablo first links the compiled program object files and the needed library code. The linked program is disassembled, and an interprocedural control flow graph (ICFG) is constructed via call-backs to object file format and architecture back-ends. Given the rather clean nature of the IA64 application binary interface, the ICFG construction is trivial. Nodes in the ICFG model basic blocks, while edges model execution paths. Basic blocks consist of an instruction sequence, in which each instruction has both an architecture-independent and architecture-dependent part.

On the ICFG all whole-program analyses and optimizations are performed iteratively, since applying one optimization may trigger other optimization opportunities. The core of Diablo provides a number of architecture-independent analyses and optimizations, such as interprocedural liveness analysis and unreachable code elimination that operate on the architecture-independent part of the instruction representation. Additional architecture-dependent analyses and optimizations, such as peephole optimization, and semantics-based analyses, such as constant propagation, rely on call-backs.

Once all optimizations are applied, the code layout is determined (optionally using profile information), and the code is scheduled into the parallel instruction bundles of the EPIC, and assembled into binary code again.

3 IA64 Whole-Program Optimizations

This section discusses some IA64-specific whole-program optimizations.

3.1 Global Offset Table optimizations

Since the linker determines the final memory layout of the code and data in a program, the compiler does not know the final addresses of (statically allocated) global data. It must therefore assume that the data may be distributed throughout the 64-bit address space. Since 64-bit addresses cannot be encoded into a single instruction efficiently, the compiler generates code that indirectly accesses global data. Before each data access, the data's address is loaded from a Global Offset Table (GOT) using a special purpose global pointer (GP) register that always points to the GOT.

Unfortunately the compiler has to assume that one GOT will not suffice for the final program. First, the compiler does not know how much data will end up in the final program. Moreover, the size of a GOT is limited: all addresses in a GOT need to be accessed through the same base GP value, and the GP-relative offsets used to access the elements in the table is limited by 22-bit width of immediate instruction operands. As a result, each compiler module (a single source code file or a group of files compiled together) is given a separate GOT. Every time control flow enters a module, the GP's value is reset to point to the corresponding GOT. The major drawbacks of this solution are that (1) global data accesses require additional loads because of the indirection through the GOT, and (2) the GP value needs to be reset again and again.

The latter drawback can be overcome at link-time by combining the small GOTs of different modules into fewer larger GOTs, eliminating all GP resets when control flow crosses the corresponding module boundaries. The former drawback can be avoided for global data that is allocated nearby the GOT itself, by computing the data's address with an addition to the GP instead of loading it from the GP. Both solutions are well known and were implemented on the Alpha 64-bit platform [9]. On the IA64 however, the read-only data section is not located nearby the GOT, as on the Alpha, and therefore much fewer address loads can be converted into additions.

Our link-time solution to this problem is to create a second GP to point to the read-only data. This second GP is created by eliminating all existing uses of the general-purpose register GR3, after which we use GR3 as a second GP: loads through the original GP are then converted to additions to the second GP.

To eliminate the existing uses of GR3, we rename them. At each program point where renaming is required, either a free register already is available, which is detected through liveness analysis, or we create a free register by adding spill code that spills a register to the stack. Fortunately this spilling is rarely needed. In single-threaded applications on the IA64/Linux platform, the special-purpose Thread Pointer register (GR13) can always be used. In almost all other applications we examined, only the special `setjmp()` and `longjmp()` C-library procedures use all registers, forcing us to insert register spills.

By merging smaller GOTs into larger GOTs —mostly one GOT suffices— 4% of all instructions can be eliminated on average. By converting loads from the GOT into additions, the static number of load instructions is reduced on average with 16.1%, while the number of executed loads decreases with 11.1%. Roughly one third of these improvements results from using a second GP.

3.2 Liveness Analysis

As a result of the optimizations of the previous section, most of the address computations involving the GOT become superfluous. Other optimizations, such as copy propagation, also render certain instructions useless, meaning that the values they produce are never used, which are hence dead. In order to actually eliminate those useless instructions, interprocedural liveness analysis is needed. This backward data flow analysis solves the following flow equations [6]:

$$\forall\, n \in N : live_{in}(n) = Consumed(n) \cup (live_{out}(n) \setminus Defined(n)), \qquad (1)$$

$$\forall\, n \in N \setminus C : live_{out}(n) = \bigcup_{s \in succ(n)} live_{in}(s), \tag{2}$$

$$\forall\, c \in C : live_{out}(c) = (Saved(p) \cap live_{in}(r)) \cup Consumed(p). \tag{3}$$

N denotes the set of basic blocks and C denotes the set of basic blocks ending with a procedure call. Equation (1) states that all registers used in a block before being defined (= consumed) and all registers that are live at the end without being defined, are live upon entry to the block. Equation (2) implements the confluence of edges and equation (3) tells us which registers are live at procedure call-site c. Muth [6] describes in detail how to solve these equations.

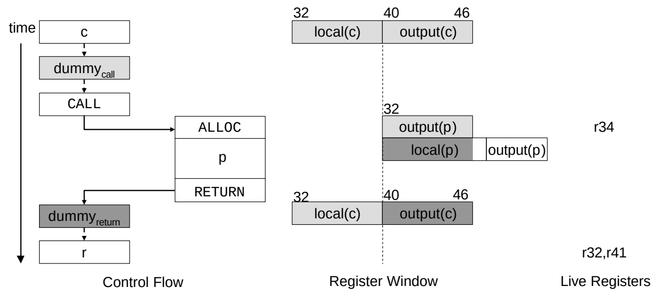

Fig. 1. Dummy blocks simulate the behaviour of the register stack.

Straightforward application of these equations is incorrect on the IA64 architecture, since register windows are used to ease parameter passing. With register windows, every procedure call involves automatic register renaming. This process is depicted in Figure 1. Each procedure has a limited view on the register stack file, which always begins at register $r32$, and is divided in local (including the input registers) and output registers. When a call is executed, the register window moves and the local registers of the caller are no longer visible to the callee. Supposing the first output register of the caller was $r40$, this register is named $r32$ after the call. The caller can resize its window with the `alloc` instruction. When a `return` is executed, the register window of the caller is restored.

This register renaming needs to be modeled correctly in our liveness analysis. Our solution consists of adding dummy blocks to the ICFG prior to calls and at continuation points, as depicted in Figure 1. Note that they are added at the call site, as the number of local registers at the call site determines the renaming. Their flow equations are:

$$\forall d \in Dummy_{return} : live_{in}(d) = Ren_{callee}(live_{in}(r) \cap output(c)) \tag{4}$$

$$\forall d \in Dummy_{call} : live_{in}(d) = (live_{in}(r) \cap local(c)) \cup (Ren_{caller}(live_{in}(p)) \cap output(c)) \tag{5}$$

The functions *local*() and *output*() return the set of the local and output registers of a given procedure respectively. $Ren_{caller}()$ maps the name of a register at the callee site to the name it has at the caller site. $Ren_{callee}()$ does the opposite. In our example $Ren_{caller}(r32) = r40$ and $Ren_{callee}(r40) = r32$.

As equation (1) operates only within a basic block, no additional measures need to be taken to assure its correctness. Equation (4) corrects the liveness information that is propagated to the end of a callee site as a result of equation (2). Without this equation the registers with names $r32$ and $r41$ would be considered to contain live values at the callee site, while the others might be considered to contain dead values (depending on the other successors). This is however incorrect due to the register stack mechanism: the value of the register named $r32$ in the register window of the caller is not visible to the callee, while the register named $r41$ is called $r33$ in the register window of the callee. The resulting liveness information should be that $r33$ contains a live value and that the other registers might contain dead values. As a result of equation (4) $live_{in}(dummy_{return}) = Ren_{callee}(\{r32, r41\} \cap \{r40, \ldots, r46\}) = Ren_{callee}(\{r41\}) = \{r33\}$, and the correct information is propagated to the callee site.

At first sight the correctness of equation (3) can be restored by adapting only the computation of the $Consumed(p)$ and $Saved(p)$ sets. The renaming cannot take place within these functions as these sets depend only on the callee p, while the renaming depends on the number of local registers of the caller, and this number is not necessarily constant for all callers of a procedure. Clearly a more complex solution is needed, resulting in equation (5). The first part states that local registers that are live at the return site are live at the call-site. Essentially these are live registers that are saved accross the call. The second part assures that values in output registers, live at the called procedure p are live at the $dummy_{call}$ block. The Ren_{caller} is needed because the output registers will be renamed after the call. In our running example this results in $live_{in}(dummy_{call}) = (\{r32, r41\} \cap \{r32, \ldots, r39\}) \cup (Ren_{caller}(\{r34\}) \cap \{r40, \ldots, r46\}) = \{r32\} \cup (\{r42\} \cap \{r40, \ldots, r46\}) = \{r32, r42\}$.

3.3 Branch Register Optimization

The IA64 architecture has eight branch registers, which can be used to store return addresses of procedure calls or to store target addresses of indirect jumps and procedure calls. However, our measurements on the code produced by the GCC compiler have learned us that only 2 of these 8 registers are used frequently: register B0 to store return addresses, and register B6 to store target addresses.

This inefficient use of the branch registers is due to calling conventions. When a procedure calls another procedure and stores the return address in a branch register, the second procedure has to know where the return address has been stored. Since a procedure from a separately compiled module or library cannot know all its callers, a fixed branch register has to be used, in this case B0. When the second procedure in turn calls another procedure, it also uses register B0 to store the return address. Therefore the first return address has to be saved before the call and restored afterwards.

To avoid this spilling of return addresses as much as possible, we have implemented the following link-time solution: all procedures get assigned a value ranging from 0 to 6, each indicating one of the branch registers B0-B5 and B7 in which the return addresses of the the procedure's callers will be stored. In the call graph of the whole program, leaf procedures (i.e. procedures that do not call any other procedure) are assigned the value 6. Other procedures are then iteratively given the minimum value of their callees minus 1, or, if this would result in a negative number, the (standard) value 0. Once all procedures are assigned a value, all instructions using the branch registers are adjusted accordingly. Whenever a procedure still has a callee with the same value, a register spill remains, but otherwise they are eliminated.

Please note that exceptions to this simple solution have to be made in the presence of indirect procedure calls, i.e. calls through function pointers. For such calls, the possible targets can only be estimated at link-time. The set of procedures that can be called indirectly is limited however, and a link-time optimizer can derive this set from the relocation information available in the object files of the program. All procedures in this set are assigned the value 0.

Still our branch register optimization is applied to 10% of all procedures, reducing the number of saves and restores of branch registers by 5% on average.

3.4 Code Layout and Scheduling

When the analyses and optimizations are finished, the control flow graph is serialized and the instructions are scheduled. Code placement may have an important impact on the performance of an application as it may improve caching and reduce the number of page faults. We implemented a profile-based closest-is-best technique, based on Pettis and Hansen [7]. As a result the number of cycles lost waiting for instructions is reduced by 38% on average. The average performance impact is moderate however because, except for the vortex benchmark, instruction latency is not a major bottleneck. In Vortex instruction latency accounts for 7% of the execution time, and there we achieve a speedup of almost 3%.

More important than code layout is code scheduling. This is particularly the case on the IA64 architecture, where the compiler needs to convey explicit information on the parallelism between instructions to the processor. Instructions that can be executed in parallel need to be clustered into instruction groups. These instruction groups are then mapped onto bundles. Each bundle has three 41-bit instruction slots and a 5-bit template that indicates the types of the instructions in the slots and the borders between instruction groups. The fact that the number of allowed combinations of instruction types and borders is limited to 32 complicates the scheduling process and as a result, it is often necessary to insert no-op instructions when no useful instruction can be found.

We implemented two local scheduling algorithms in our link-time optimizer: a list scheduler [1] and the so called *noptimizer*. The noptimizer is based on [4]. It can operate with different cost functions, to either minimize the number of no-op instructions, or to minimize the number of instruction groups and as a result the number of execution cycles. The noptimizer is a branch and bound version of the optimal scheduling algorithm, but its search depth is severely limited, to limit the execution time of the scheduler. Still, compared to the original binaries produced by the GCC compiler, the noptimizer is able to reduce 18% of all no-ops on average, with an overall compaction of 5%.

In order to increase the amount of parallelism, we also developed a global scheduling algorithm, the so-called *globtimizer*, which is roughly based on [10]. It is a branch and bound algorithm as well and it uses a cost function (number of no-op instructions, number of instruction groups, etc.) to optimize the instruction sequence. The idea here is to move instructions up and down between basic blocks using predication, schedule those basic blocks with a local algorithm and compare the quality of the solution with those of other configurations. The structure of the flow graph remains untouched however, so no basic blocks are merged or split like many other global scheduling algorithms do.

Currently some simple combinations and structures of basic blocks are considered to move instructions between blocks. Nevertheless, combined with the noptimizer, it is able to further reduce the number of no-op instructions by 23%.

4 Experimental Evaluation

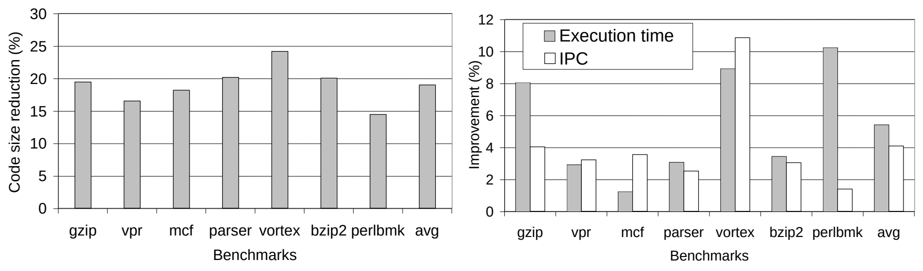

Fig. 2. Experimental results: code compaction, speedup and IPC improvement

To evaluate our link-time optimizer, we used 7 programs from the SPECint2000 benchmark suite. We compiled them with the GCC compiler (v3.2) and linked them with the glibc library (v2.3.1). The experiments were performed on a 4-way Intel Itanium multiprocessor system, running Linux 2.4.18.

Our results are summarized in Figure 2. The code size is reduced on average with 19%. The major contributions come from unreachable code elimination (9.64%), instruction scheduling and bundling algorithms (5.22%) and reduction of load instructions (3.27%). We also achieve an average speedup of 5.4%. The IPC (instructions per cycle) improves up to 11%, and 4.1% on average. The speedup is mainly caused by the reduction of load instructions and, in the case of vortex, by profile-guided code layout.

5 Related Work

Srivastava and Wall describe the optimization of GOT accesses on the Alpha architecture [9], for which both Alto [6] and Spike [2] are link-time optimizers. By contrast, Diablo [3], the framework we used for link-time optimization, is portable and retargetable. We also extended the existing work on GOT optimization on the Alpha by introducing a second GP to fully exploit this optimization on the IA64 architecture.

Numerous EPIC-specific scheduling algorithms have been developed. In [5] and [10], Integer Linear Programming is used to obtain an optimal local and global schedule respectively. We adopted the filosophy in [10] and designed a branch and bound version, in which the structure of the flow graph is also preserved. A technique for minimizing the number of no-op instructions is presented in [4]. We extended this work by integrating branch instructions tighter into the bundling process and by filling up partially filled bundles more carefully. More details on our extensions, including source code, can be found at http://www.elis.ugent.be/diablo.

6 Conclusions

We have shown how link-time optimization is able to improve the code quality of IA64 code, which is crucial given the EPIC paradigm. By optimizing the Global Offset Table address computations, the use of the branch registers and by improving the code schedules, code size reductions of 19% on average were achieved, together with an average speedup of 5.4%.

Acknowledgements

This research was funded by Ghent University, by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) and by the Fund for Scientific Research-Flanders (FWO-Flanders).

References

1. E. Coffman: Computer and Job-Shop Scheduling Theory. Jon Wiley & Sons. (1976).
2. R. Cohn, D. Goodwin and G. Lowney: Optimizing Alpha Executables on Windows NT with Spike. Digital Technical Journal. **9** (1998) 3–20.
3. B. De Bus, D. Kästner, D. Chanet, L. Van Put and B. De Sutter: Post-pass compaction techniques. Communications of the ACM. **46** (2003) 41–46.
4. S. Haga and R. Barua: EPIC instruction scheduling based on optimal approaches. Annual workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques. **1** (2001) 22–31.
5. D. Kästner and S. Winkel: ILP-based instruction scheduling for IA64. Proc. of Languages, Compilers and Tools for Embedded Systems. (2001) 145–154.
6. R. Muth, S. Debray, S. Watterson and K. De Bosschere: alto: A Link-Time Optimizer for the Compaq Alpha. Software Practice and Experience. **31** (2001) 67–101.
7. K. Pettis and R. Hansen: Profile guided code positioning. Proc. of the ACM SIGPLAN Conf. on Programming Language Design & Implementation. (1990) 16–27.
8. M. Schlansker and B. Ramakrishna Rau: EPIC: Explicitly Parallel Instruction Computing. IEEE Computer. **33** (2000) 37–45.
9. A. Srivastava and D. Wall: Link-time optimization of address calculation on a 64-bit architecture. Programming Languages Design and Implementation. (1994) 49–60.
10. S. Winkel: Optimal global scheduling for itanium processor family. Explicitly Parallel Instruction Computing Architectures and Compiler Techniques. (2002) 59–70.