SALTO : System for Assembly-Language Transformation and Optimization

Erven Rohou, François Bodin, André Seznec, Gwendal Le Fol, François Charot, Frédéric Raimbault

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SALTO: *System for Assembly-Language Transformation and Optimization*

Erven Rohou, François Bodin, André Seznec,

Gwendal Le Fol, François Charot, Frédéric Raimbault

N° 2980

Septembre 1996

THÈME 1

ISSN 0249-6399

SALTO: System for Assembly-Language Transformation and Optimization

Erven Rohou, François Bodin, André Seznec,
Gwendal Le Fol, François Charot*, Frédéric Raimbault†

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n° 2980 — Septembre 1996 — 27 pages

Abstract: On critical applications, particularly embedded systems, the performance tuning requires multiple passes. SALTO (System for Assembly Language Transformation and Optimization) is a retargetable framework for developing all the spectrum of tools that are needed for performance tuning on low-level codes (assembly-languages) on uniprocessors. SALTO enables the building of profiling, tracing and optimization tools. The user is responsible for giving a machine description of the target architecture, which includes instruction-set of the processor, precise hardware configuration and reservation-tables for all instructions, but high-level functions are provided to him for writing any tool corresponding to his needs.

Moreover SALTO will be a part of a global solution for manipulating assembly-code to implement low-level code restructuration as well as to provide a high-level code restructurer with useful information collected from the assembler code and from instruction profiling.

SALTO has been tested on Intel platforms running Linux (i486) and Solaris (PentiumPro) and on a Sparcstation running SunOs 4.1. A machine description for the Sparc v7 architecture is currently available. Two examples, a basic block instrumentation and a local reordering optimization, are given in the paper as illustration.

Key-words: assembly language, optimization, embedded systems, reservation tables, user interface, object oriented, compilation process

(Résumé : tsvp)

* {erohou,bodin,seznec,lefol,charot}@irisa.fr
† raimbaul@univ-ubs.fr

Unit´e de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
T´el´ephone : (33) 99 84 71 00 – T´el´ecopie : (33) 99 84 71 71

SALTO : un système pour la transformation et l'optimisation des codes assembleurs

Résumé : Pour la plupart des applications critiques, notamment les systèmes embarqués, l'optimisation des performances requiert plusieurs passes. SALTO est un environnement de travail reciblable qui permet le développement de l'ensemble des outils nécessaires pour l'analyse et l'optimisation de performances sur des codes assembleurs pour monoprocesseurs. SALTO facilite la construction d'outils pour le *profiling*, la génération de traces et l'optimisation. L'utilisateur doit fournir un fichier de description de la machine cible. Cette description, du jeu d'instructions du processeur, contient la configuration matérielle et les tables de réservations décrivant l'usage des ressources. Une interface de haut-niveau, orientée objet, est fournie pour l'écriture des outils dont il a besoin.

En outre, SALTO est destiné à être un élément d'une solution globale de manipulation de code assembleur tant pour la restructuration de codes de bas-niveau que pour la production d'information à destination d'un optimiseur de haut-niveau.

SALTO a été testé sur des plateformes Intel fonctionnant sous Linux (i486) et Solaris (PentiumPro) et sur Sparcstation sous SunOs 4.1. Une description de l'architecture Sparc v7 est disponible. Deux exemples, une instrumentation des blocs de base et un ordonnancement local, sont décrits dans cet article en guise d'illustration.

Mots-clé : assembleur, optimisation de performances, systèmes embarqués, tables de réservation, interface utilisateur, orienté objet, chaîne de compilation

1 Introduction

It is our belief that for many critical applications, particularly in embedded systems, the performance tuning of low-level code cannot be handled by a single-pass compiler. We believe that such performance tuning requires the cooperation of many tools including assembly-code schedulers targeted for the precise hardware configuration, tracing and profiling tools and instruction-layout optimizers. These tools should feed information back to the high-level compiler.

The increasing usage of high-performance embedded systems based on RISC/VLIW architectures has highlighted the need for tools that allow the easy implementation of fine-grain parallelism optimizations and assembly-code profiling and instrumentation, along with an accurate description of the target architecture and high retargetability. SALTO, presented in this paper, is a first step toward the availability of such a system.

SALTO is a retargetable framework for developing the whole spectrum of tools that manipulates assembly-language. The objective of such a system is to provide the user with a single environment that will allow him to implement the tools that are needed for performance tuning on low-level codes; this set of tools includes assembly-code schedulers, as well as profiling and tracing tools that provide the user with information on where to focus optimizations and how efficient they can be, therefore allowing tradeoff choices. Such a system is intended to address general computing as well as embedded systems for which optimizations are more critical and aggressive, but time-consuming techniques are more tolerable.

A large number of tools have been written to experiment with new optimizations or to try to point out particular mechanisms. This development phase is generally time consuming and requires much investment. Utilities able to trace or profile programs exist, but they are often provided "as is": they are not at all flexible, and only work for specific architectures. Studying a particular problem is likely to require rewriting from scratch large pieces of code. As SALTO is retargetable with the instruction-set architecture as well as with the precise target hardware, it is likely to be a major help for such studies.

SALTO overcomes many limitations of previous solutions: it does not implement any algorithm by itself, but its scope is broad enough to allow implementation of either profiling or optimization techniques. A large amount of work needed while working at the assembly-code level (code parsing, construction of the dependence graph, etc.) is performed automatically by the system. To the user, SALTO provides an object-oriented interface to deal with assembly-code. The objects contain a complete description of the control-flow graph of the program (when available) and a model of the target architecture. They are easily accessible through the user interface and provide a comfortable way to implement algorithms without having to worry about infrastructure.

With SALTO we plan to address the field of software analysis and optimization for superscalar and VLIW architectures.

Section 2 reviews related works and points out how our tool provides a more general and integrated solution for building tracers, profilers and optimizers. The general approach

leading to SALTO is presented in section 3. Section 4 gives an overview of the system. Section 5 details the features of our tool. Examples are given in section 6.

2 Related Work

To our knowledge no available single environment deals with the whole spectrum of code restructuration and execution profiling and tracing. Moreover most tools are not fully retargetable.

Much work has been done in the field of low-level code optimization and analysis of dynamic program execution. Optimization can be done at different levels of granularity. They range from code transformations for improving the parallelism in each basic-block to procedure motion for improving the instruction-cache behavior. Dynamic program execution can be analyzed by profiling or tracing. Profiling deals with the number of times a piece of code (instruction, block or procedure) is executed. This is particularly interesting for determining critical sections in codes. As object-code optimization is CPU consuming, the optimizations should focus only on those sections. Tracing furnishes a more precise analysis: the order of instructions is known and precise addresses are obtained.

2.1 Analysis

Many techniques for analyzing program behavior depend on instrumentation of either the source code or the executable file. PIXIE [26] is an instrumentation utility running on MIPS machines for executables. The instrumented program, which contains additional code, counts the execution of each basic-block. A counter is added in each block. Ball and Larus [3] studied how to optimally use the same technique by minimizing the number of counters required. They build a control flow graph of the procedure being instrumented and compute the maximal spanning tree before deciding where to add code. They proved this technique to be optimal. They implemented their strategy in a tool called QPT [3].

A more flexible solution called ATOM [12, 27] was proposed for the DEC Alpha. ATOM also depends on code instrumentation and provides common analysis and performance tools. A partial list given in [12] contains instruction profiling, system-call summary, memory checker, and many others. The major advantage is the ability for the user to implement instrumentations in the C language enabling a high degree of flexibility. ATOM can be seen as a library of predefined functions that ease the instrumentation of the code.

Gordon Irlam's SPY [22] runs on Sparc architectures running SunOs 4.x. SPY exploits special features of the Sparc microprocessor [11] to fetch the instruction to be executed. SPY provides as output a trace composed of the instruction addresses and the data addresses, if any. Other tools achieve analysis via instruction-set emulation. Cmelik and Keppel chose another strategy with SHADE [9]: they *dynamically compile* each instruction of the program, i.e., they build a block semantically equivalent to each assembly instruction of the original code as if it were a complex instruction and then execute the block. This approach enables

them to simulate an instruction-set on a different architecture (they can currently simulate Sparc and MIPS code on Sparc systems).

A survey of trace-driven memory simulation including trace collection, trace reduction and trace processing can be found in [29].

2.2 Code Generation and Optimization

Extensive work has been done on fine-grain optimization [23, 14, 1, 8, 24, 17, 13, 19, 25, 15, 18, 2, 30, 10, 5, 4] but very few on frameworks for enabling a fast implementation of such optimizations. Furthermore, we are not aware of a retargetable system that enables the implemention of assembly-code scheduling (with accurate resource usage models), partial-register allocation and instrumentation. Being able to "redo" partial-register allocation is a major capability that is necessary for achieving good performance when using software-pipeline techniques [2].

The main drawback of having SALTO as a separate tool is that it separates the code-generation and the optimization. Ideally, we want these phases to happen simultaneously [8] to be able to generate optimal code. In practice, mixing code-generation and scheduling (especially software-pipeline and global-scheduling techniques) is a very difficult task. Having code generation and scheduling performed at the same time, generally, results in having a non-optimized code-generator (compared to the one provided by the native compiler) and a weak code scheduler that does not implement anything beyond basic-block scheduling. Section 3 covers how we plan to couple the phases. In the remainder of this section we overview works that we believe are close to SALTO.

Code layout optimization is performed by CORD on MIPS based computers. CORD rearranges procedures in an executable to reduce paging and achieve better instruction-cache mapping. It uses a feedback file generated at run-time by an executable instrumented by PIXIE.

OCO [6] is an assembly-language optimizer which was developped at IRISA. It takes an assembly source code as input and produces an optimized version of the code. Optimization is based on local reordering and software pipelining. OCO has been used as a basis for SALTO.

The university of Karlsruhe has developed the BEG system (Back-End Generator) [21]. It produces a code-generator from rule based declarative descriptions. Basic blocks are reordered before and after global register allocation. The system is based on a simulation of the pipeline during instruction reordering. Pipelines that schedule more than one instruction per cycle are also supported. BEG can also produce local register allocators suitable for a great variety of target machines.

MARION [7] has been developed at the University of Washington. It is a code-generator generator. The MIPS R2000, Motorola 88000 and i860 architectures have been described with the specific language Maril. The hardware description includes registers, functional units, multiple issue and pipeline stages. Instructions are given properties that influence reordering. The description is based on reservation tables and on delays required to obtain the result of an instruction. MARION has been mainly used to study the interaction between

reordering and register allocation [8]. Compared to SALTO, no user interface is provided to modify assembly-code and we believe that it cannot be used with compilers other than `lcc`.

GCC is a C compiler maintained by the Free Software Foundation, Inc. [28]. It uses the internal representation RTL. The good performance achieved by GCC comes from its ability to apply the right optimization at the right moment within the compilation process. It is improved by a last optimization specific to the target architecture at the end of the process. Instruction reordering within the basic-blocks is performed twice, once before and once after the register allocation stage. The machine description includes the size of delay slots and the annulation conditions, the time needed to compute the result of an instruction and resource conflicts between instructions.

A compiler system named SUIF [20] (which stands for *Stanford University Intermediate Format*) is being developed at the Stanford University. It is built on top of a kernel that preprocesses source code and produces a representation of the program called an *intermediate format*. This format contains low-level description of the structures without losing useful high-level information like arrays, if-then-else structures and loops. Many passes are performed on this representation, implementing a single optimization or transformation at a time. The compiler performs several passes on this representation, but insertion of new passes can easily be made. Features are provided to allow implementation of algorithms ranging from data-dependence analysis or register allocation to symbolic analysis or detection of parallelism.

3 Motivation for SALTO

The complexity of today's architectures require that the compiler masters many parameters to produce efficient code. The impact of the modification of some parameters might not be straightforward. We believe that an efficient optimization can be achieved only with an iterative process where different tools exchange information. This section explains how we conceive such a multiple pass code-generation process and why retargetability is an important feature to implement in optimizers.

3.1 Multiple Pass Code Generation Process

The effective performance of an application on a superscalar or VLIW processor (embedded or otherwise) depends on many parameters which cannot be easily managed by a single-pass code-generation process. For instance, the effective performance depends on the instruction-cache behavior and on the interaction of basic-blocks. Profiling and tracing tools may provide the user (or a high level code restructurer or a compiler) with information on critical-code sections that should be highly optimized or, for instance, information for an accurate memory layout of basic-blocks, etc.

It is our belief that the code-generation process for performance critical applications must be iterative. It is also our belief, that such an iterative process can be afforded for

many embedded applications where time-consuming techniques may be used because of the performance requirement and the long lifetime of the code.

For such a multiple-pass code-generation process, a few tools directly working on the assembly-language are required: a code scheduler, a basic-block profiler, an instruction tracer, an instruction-layout optimizer, etc. The SALTO system we present in this paper is the main component of a unified framework for developing this spectrum of tools for manipulating assembly-languages, which we believe are needed for performance tuning. SALTO is highly retargetable with the assembly-language description as well as with the precise description of the targeted hardware.

3.2 The Optimization Loop

In this section, we illustrate the optimization loop of the code-generation of an application. Figure 1 illustrates the information flow between the different components in the optimization loop:

- Control flow and data dependencies are passed from the high-level code restructurer to the low-level code instrumenter. For feeding back information, links to the source code are also propagated during the compilation and code-generation phases.
- The low-level code optimizer feedbacks information to the high-level code restructurer: size, ideal execution time of basic-blocks, register pressure, etc.
- When instrumented, the execution forwards profiling information to the high-level code restructurer.

Figure 2 illustrates the optimization process:

1. The program is instrumented and profiled to determine statements execution time. The optimization will be focused on the most time consuming statements.
2. Information from the low-level code is forwarded to high-level code restructurer.
3. The high-level code restructurer uses this information as a guide for applying (or not applying) specific transformations. For instance, loop distribution or fusion is guided by information on register pressure as well as information on the execution time of the loop(s) body. Unrolling will be guided by the execution time of the loop body, but also by the size of the code.

   A code size criterion is applied to maintain correct behavior of the instruction-cache.
4. A new low-level code is generated. Then, low-level code optimizations are applied (basic-block scheduling, software-pipeline, etc.) and new information are fed back to the high-level code restructurer. Instruction-cache behavior is estimated. Phases 3 and 4 are reiterated to reach satisfactory performance.

Note that most constructors do not provide the user with the ability to use such an optimization loop.