SCISM IA-32 Binary Translator

Evgueni Koukourechkov, Nikolay Grozdanov, Georgi Gaydadjiev and Stamatis Vassiliadis
Computer Engineering Laboratory,
Electrical Engineering Department,
Delft University of Technology,
Delft, The Netherlands
{gusa, grozadanov, Georgi}@Dutepp0.ET.TUDelft.NL
http://ce.et.tudelft.nl

Abstract— **With today's IC technology approaching the edge of the Moore's law, it is emerging to obtain execution speed-ups by applying different methods rather than future clock speed increases. The execution time can be improved by exploiting the parallelism inherent in the binary code, i.e. Instruction Level Parallelism (ILP) [3]. In such a way, designing a new architecture from scratch and recompilation of the existing code (written and tested for years) can be avoided. The** *Scalable Compound Instruction Set Machine* **(SCISM) [6] organization addresses this problem by analyzing the instruction dependencies at execution time and by compounding them together for parallel execution according to a pre-defined categorization based on hardware utilization rather than opcode description. The main SCISM advantage is that it provides a design that remains binary compatible with the original instruction set architecture. This paper introduces SCISM software simulator able to read, translate and simulate parallel execution of IA-32 legacy code. The main goal is to provide a tool for easy exploration of the parallelism present in the native IA-32 binary code. The SCISM simulator is (open source) software project written in C++ under Linux. The compounding rules and all additional information, e.g. ISA description, hardware implementation [1] details etc., are provided through a set of plain text configuration files that can be easily modified. Preliminary results suggest that performance gains of about 30 % are feasible.**

Keywords—**ILP, binary translation, SCISM.**

I. Introduction

The need for high performance computer systems is still emerging. Researchers and designers do apply many different techniques on different design levels in order to improve computer systems performance. One of the most promising performance boosters is the future exploiting of the instruction level parallelism since the Moore's law is expected to decelerate when the component sizes decrease additionally.

Instruction-level parallelism (ILP) [3] is currently exploited in many modern computer architectures, e.g. IA-64. The acceptance process of such systems in the market is very difficult due to the huge amount of legacy code (written and tested for years) and currently in use. Redesign of all applications may require major effort and a significant investment. Since it has been shown that for a variety of reasons the performance gain of ILP aware systems is lower than the theoretical maximum, a better way to employ the parallelism inherent in the binary code is suggested. In such a way, designing a new architecture from scratch and recompilation/redesign of the existing code can be avoided. The Scalable Compound Instruction Set Machine (SCISM) [6] organization addresses this problem by analyzing the instruction dependencies at execution time and by compounding them together for parallel execution and provides designs that are binary compatible with the original (targeted) architecture.

This paper presents a tool (simulator) for exploring the SCISM advantages based on existing binary executables. The first targeted is IA-32 binary code. The simulator input consists not only of the binary (legacy) code to be analyzed but also of configuration information about the targeted instruction set and SCISM related information, e.g instruction groups description, compounding rules, type and count of functional units used etc., making the tool easy extendable to any other ISA.

This paper is organized as follows: Section II describes briefly the SCISM organization. Section presents the overall architecture of the binary translator and describes some implementation issues. Section IV shows some preliminary results obtained from the translator and finally, Section V concludes the paper.

II. The SCISM organization

SCISM is a machine organization suitable for RISC and CISC architectures [6]. It improves significantly the instruction-level parallelism by solving or diminishing the problems every superscalar machine suffers from, such as data interlocks (dependencies), branch instructions, and interrupt handling. The main idea behind SCISM is that

[1] in this document architecture and implementation are used as defined in [1]

all instructions are partitioned into several functional categories. This is done in order to minimize (optimize) the number of dependency rules among concurrent instructions for the sake of efficient (hardware) implementation. With other words, the grouping of instructions is based on the hardware utilization rather than opcode description, as used on other machines. For example, the total number of dependency rules in an architecture with *N* instructions and machine organization able to execute two instructions in parallel is *N x N*. Theoretically such architecture can exploit the instruction level parallelism maximally but hardware implementation of such amount of dependency rules may turn prohibitive complex. This is especially the case for CISC architectures.

All instructions inside one particular category in SCISM are considered as ”unique”, and hence cannot be executed concurrently since all members of a category use the same hardware resources. Differences among category members are considered as ”trivial” and are to be resolved by the hardware. The number of instructions that cannot be placed in any other category are lumped together in a single group, even are assigned an individual category. The former method simplifies the implementation by minimizing the total number of instruction categories, so it is often preferred. Inside a group of *n* instructions, where *n* is large (3 or more), it is very likely that dependencies will exist between instructions. One good example of such group are all arithmetic instructions utilizing the arithmetic logic unit (ALU), e.g. *add, subtract, logical AND, logical OR, compare*, and so forth.

High level abstraction of SCISM program execution flow is presented in Figure 1. The program instruction stream (*Program*) is processed by the compounding facility. This facility operates with an implementation-dependent (predetermined) scope and generates the *compound instruction program*. Such a transformation is based on a set of rules reflecting the system architecture, hardware organization and the parallelism between different instruction categories. These rules are referred as *compounding rules*.

The *compound instruction program* produced can be executed by the execution engine, which handles every compound instruction as a single instruction. The compounding process may be positioned mainly at two places - on compilation time and on execution time. Both approaches know different pros and cons. In the context of this paper the execution time compounding will be assumed.

The compound instruction should contain additional information for the purpose of parallel issuing and execution. In general such information can be incorporated in the compound instruction in the form of decoding or tagging. Tagging, however is the preferred way since it has been proven as mandatory technique for architectures that allow variable-length instructions, or that allow data to be intermingled with instructions [6], [5]. Please note, that instructions composing a compound instruction are not necessary consecutive instructions in the original program, allowing for out-of-order issue.

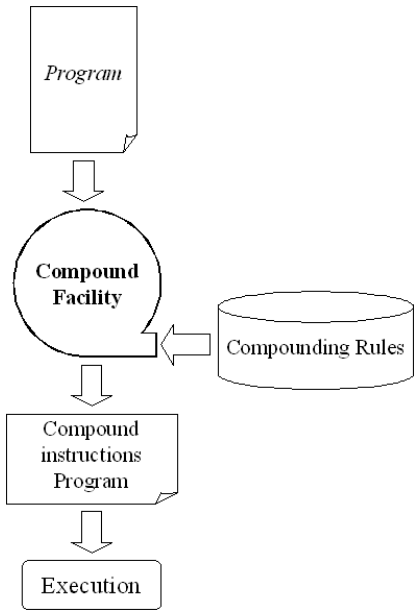

Fig. 1. SCISM program execution flow

Figure 2 shows the compound instruction format for a machine organization where up to four instructions can be executed in parallel. To mark the compounding boundaries of those instructions, four additional tag bits are required. In this definition a tag equal to 1 notifies that the next instruction can be executed in parallel with the previous one (or more), while 0 indicates the end of the current compound instruction. In Figure 2, the instructions I_j are in their original form and T fields represent the tags.

I_1	T	I_2	T	I_3	T	I_4	T

Fig. 2. SCISM compound instruction

One fundamental property of SCISM machine organization is that it enables preprocessing to be detached from instruction issue/decode. In order to achieve this, the compounding must be ”permanent”, with permanency depending on the physical location of the compounding facility. Such static compounding is also the major difference between SCISM and a superscalar machine that ”compounds” its instructions dynamically. For example, the compounding facility may be a software facility - in the form of a post compiler [5] or it may be in the form of hardware facility (preprocessor) located, for example, be-

tween the cache and the memory subsystems [6]. In such hardware preprocessor, the instruction stream to be compounded is the instruction text fetched during servicing of a cache miss. The output, i.e. the stream of instructions and their tags will be written in cache and will remain intact as long as the line resides in the cache, and is thus relatively ”permanent”. In case the line should be removed from the cache for any of a number of reasons, the associated tags become invalid and the line must be preprocessed again should it be required at some later moment.

III. The IA-32 binary translator organization

The purpose of the SCISM simulator is to perform binary translation and simulate parallelized execution of existing binary programs. Such binary programs (also referred as legacy code) are compiled and widely used applications for particular ISA, e.g. PowerPC or MIPS. Such ISA will be referred as *targeted architecture* in the scope of this document. The first targeted architecture is Intel IA-32 due to its wide share in the personal computers. Several clear steps are to be performed as a part of the simulation process. They are as follows: input data preparation, binary translation, simulation and profiling, and report generation.

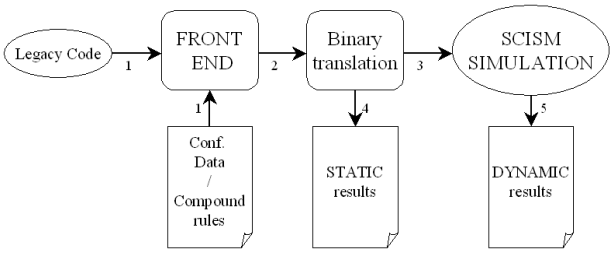

Fig. 3. The SCISM Simulator

The preparation stage performs several actions essential for the next stages of the simulation process. In more details they are: loading of the ISA description, loading of SCISM instruction groups definition, loading of the SCISM compound rules table, load of the SCISM implementation description and finally loading of the legacy code (program). All of the above are external text and binary files required for the simulation process. In this text we refer to this stage as *Front End* and its position in the overall organization is shown in Figure 3. The ISA description provides the information about each instruction of the targeted architecture. The instruction group definition and the compound rules table determine the SCISM organization to be used. The SCISM implementation description, e.g number and type of execution units, is need mainly for the simulation stage. The binary file that is the subject of simulation is disassembled in the first place. All of the above configuration and input data information is collected in dedicated internal representation containers. The organization with external configuration files provides high degree of flexibility and allows for rapid design space exploration, e.g the same binary file can be simulated on several different SCISM organizations.

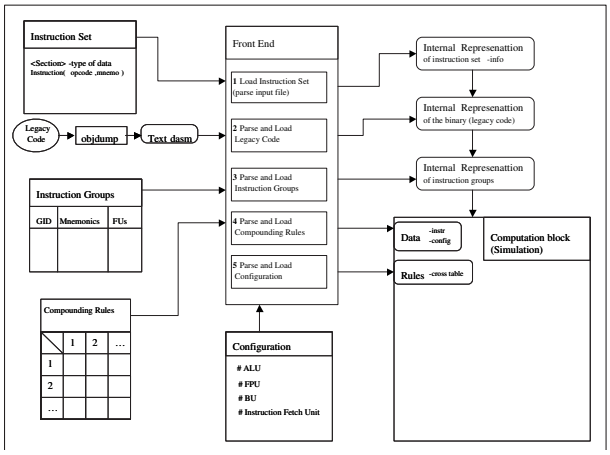

Fig. 4. Front End functionality

Figure 4 shows the main functions of the front end in more detail. The Front End Block is performing the following preparation tasks for the simulation:

1. Loading from the input files the data for instruction set, groups, compounding cross table rules and implementation to be simulated (types and count of storage and processing units). The data is loaded into their internal representation in the classes *InstrSet, CompConds, InstrGroups*.
2. Disassembly of the input binary file. In the current implementation the external program *objdump* (part of GNU development tools) is used.
3. Loading disassembled binary as a sequence of executable instructions in convenient for analysis internal representation class *InstrExec*.

The binary translation phase consists of two main subphases: Determination of inter-instruction dependencies (data, control and resource) and Instruction Compounding. The dependency determination is essential for the instruction compounding process and is envisioned to be a complex process. Three basic dependency types need be analyzed:

- Data dependencies, e.g. instruction $n + 1$ requires the data produced by n and hence can not be executed in parallel
- Control dependencies, e.g. all branch instructions that change the program flow heavily influence the compounded program
- Resource dependencies, e.g. two instructions require the same hardware resource simultaneously

In essence, three separate dependency graphs are implemented in a suitable data structure optimized for memory space.

Due to the fact that the internal dependencies of any binary program of the targeted architecture can be considered ”permanent” and can be detached from the compounding process, those two processes are assumed independent. With other words there are more compoundings possible for the same binary program (with the same internal dependencies). The actual compounding uses the dependencies and compounds the original instructions together, based on the SCISM compounding groups and rules definition. The result of this stage will be compounded instruction program. The compounding instructions are of variable length with a maximum being the maximum compound instruction width. Special internal representation is used to represent the SCISM compound instructions binary image. This stage will be referred as *Binary Translation* from now on. In addition to the compound instructions program, static code information is produced in the form of different instruction counts and other statistical information.

During the Simulation and profiling phase, the dynamic SCISM functional simulation is actually performed. The functional simulation verifies the proper program execution after the transformation stage has been applied. In addition dynamical statistic results and profiling information are collected. The report generation actually formats the collected static and dynamic data and produces a report file. This file is meant for the designer and allows him analyze and tune the SCISM compounding rules and/or implementation.

In Figure 3 an overview of the steps as described above above is presented. The simulator is designed using C++.

IV. Preliminary Results

The software project described in this paper is still ongoing. The current state of the project is as follows. The Front End is ready and the Binary Translation is almost finished. The data dependencies analyze and internal presentation turned to be more complex than previously estimated, hence the longer time needed to complete it.

At this moment only small programs can be loaded, analyzed and compounded. The maximum binary program length is about 400 lines. Approximately 30 % decrease of program lines was found, however, the compounding results should be verified on real programs instead of the currently used synthetic binaries. The functional simulation part is completely open and will be initiated only after all previous stages are completed.

A. Previous Work

The performance of IBM System/370 SCISM organization with compounding facility placed between the cache and the main memory was evaluated in [6]. A two-way compounding scheme was used under real-life commercial workloads. To avoid issues that clearly affect the performance of the superscalar processors but are entirely dependent on technology and implementation constraints, the number of instructions that can be executed in parallel where compared to the maximum performance of a theoretical superscalar machine which can issue and execute all instruction in pairs. The performance gain prediction was based on the number of instructions that can be executed in zero time. The rationale behind was as follows: If one instruction in a compound instruction executes in n cycles and another instruction executes in $m \leq n$, the later appears to execute in zero time. This factor was denoted as PZE (potential zero-cycle executions) and was used to represent the number of instructions that can be ”removed” from the instruction cycle time during the execution of a program. For example, considering a two-way compounding, PZE is 50%, since it is assumed that every instruction is half of a pair. In reality the improvement will always be lower due to factors such as cache size and branch prediction accuracy. The results of this evaluation showed that PZE in the range 35-43% can be achieved for the considered system. In addition a SCISM2 organization was evaluated, where the branches where removed from the instruction stream improving the PZE to 41-49% very close to the theoretical high-bound of 50%.

V. Conclusions

A tool that emulates the SCISM execution behavior on IA-32 binary programs has been presented. The tool internal structure has been discussed and the most important objects have been highlighted. In addition some preliminary results and the directions for future work where given. Previous work indicates the potential parallelism present in commercial workloads.

References

[1] Gerrit Blaauw and Frederick Brooks Jr., *Computer architecture*, Addison-Wesley, One Jacob Way, 1997.

[2] R.J. Eickemeyer, S. Vassiliadis, and B. Blaner, *An in-memory preprocessor for SCISM instruction-level parallel processors*, Technical Report TR-01 C407, IBM Glendale Laboratory, Endicott, NY, May 1992.

[3] John L. Hennessy and David A. Patterson, *Computer architectre a quantative approach*, third ed., Morgan Kaufmann, 2003.

[4] Intel, *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 2003.

[5] S. Vassiliadis and B. Blaner, *Concepts of the SCISM organization*, Technical Report TR-01 C209, IBM Glendale Laboratory, Endicott, NY, Jan 1992.

[6] S. Vassiliadis, B. Blaner, and R. J. Eickmeyer, *SCISM: A scalable compound instruction set machine*, IBM J. Res. Develop. **38** (1994), no. 1, 59–78.