# Binary-to-Binary Translation

# Literature Survey

University of Texas at Austin

Department of Electrical and Computer Engineering

Juan Rubio

Wade Schwartzkopf

March 16, 1998

# Binary-to-Binary Translation

Juan Rubio and Wade Schwartzkopf

University of Texas at Austin

Department of Electrical and Computer Engineering

*Abstract – There is much code presently written for the Texas Instruments TMS320C5x line of processors. However, the Texas Instruments TMS320C54x processors have surpassed the C5x line in many ways. It would be very useful if one could take the programs written on a C5x and use them on a C54x processor. In this way, one could take advantage of the capabilities of the newer processor without having to spend time and money redeveloping programs already existing on the older platform. But this is not possible because C5x binaries are incompatible with the C54x. In order to use C5x programs on a C54x processor, those programs now must be rewritten. We propose to develop the framework for a binary-to-binary translator that can automatically convert binary code from the C5x to the C54x. Our goal is to create a translator that can translate a subset of C5x code into C54x code that accurately emulates the behavior of the original C5x code on a C5x processor. While doing this, we plan to develop an intermediate representation for the code of these processors. It is our hope that this representation will be flexible enough to allow others to extend our translator to work with other digital signal processors.*

## *I. Introduction*

Binary translation is the process of directly converting machine code from one processor into code that runs on another. This is often useful when one wants to transfer code from an old machine to a newer one. This is the case with the C5x and C54x processors. The C5x is commonly used, but the C54x family is newer, faster and has more capabilities than the C5x family. Therefore, it would be very useful if one could take the programs running on C5x processors and translate them to use them on the newer C54x processors. If one could do this, he could take advantages of the capabilities of a newer processor without having to rewrite his software.

In this paper, we will briefly review the history of binary translation in section II. In section III, we will discuss the alternatives to binary translation and why we choose to work on it rather than the other choices we have available. We will then consider problems that one can encounter with binary translation in section IV. Finally, we will explain what we expect to accomplish in this project.

## *II. History*

One of the first attempts at binary translation was Hewlett Packard's translator for the HP-3000 in 1987 [1, 2, 3]. The code for the HP3000 was fairly simple, and typically there was only one file to translate.

As computers became more complicated so did binary translation. In modern computers, it is common to have several programs that work together. Timing intricacies, parallel processing, exception handling, interactions with operating systems, and read-write ordering problems have also made the task of binary translation more difficult.  The first major successful binary translator to deal with these problems was Digital's VEST (VAX Environment Software Translator) that translated VAX code into code for their new Alpha AXP computers [1].  VEST was completed in 1993, and Digital has since proven themselves the leader in the area of binary translation.  Since VEST, Digital has built Freeport express, which translates from SPARC to Alpha, and FX!32, which translates x86 to Alpha.

Many other notable attempts at binary translation have also been made [3], but to date, there have been few if any successful attempts at binary translation for digital signal processors.

## *III. Choices in Translation*

Binary translation is not the only method in which one can take code from one machine and run it on another.  There are four main methods of code translation.  They are, in order from the slowest to the fastest: software interpretation, microcoded emulation, binary translation, and the use of native compilers [1].  Each of these is described briefly below.

### A.  Software Interpretation

A software interpreter retrieves each instruction, one at a time, and performs it on a software-maintained version of the old architecture.

Software interpreters have to their advantage that they can always faithfully reproduce the behavior of the old code.  In fact, most binary translators have a software interpreter to fall back on situations where the binary translation fails.  The disadvantage of software interpreters is their poor performance.  They are much slower than the other three methods of translation.

## B.  Microcoded Emulation

A microcoded emulator uses hardware to translate code from one machine into code for another.  Microcoded emulators are specific to one machine and cannot be generalized to other targets.  And, of course, microcoded emulation is limited to architectures that have a microcoded hardware layer.

## C.  Binary Translation

Binary translation generates a sequence of instructions that reproduces the behavior of the old program on the new platform.  It is this method of translation that this project will address.

We chose to work on binary translation because it runs much faster than software interpretation.  Code produced through binary translation often executes on the target machine at least as quickly as the code did on the original processor.  Microcoded emulation is faster than software interpretation, but it is still not as fast as binary translation, and binary translation has the additional advantage that, unlike microcoded emulation, it has the potential to be easily generalized to other platforms.  Native compilers have a speed advantage over binary translation, but, unlike native compilers, binary translation works for all executables, not just the ones for which the source code is available.

### D. Native Compilers

If the source code from which the machine code was developed is available, it is best just to recompile the code on the new architecture. This is the simplest, fastest, and most efficient solution to creating code that will run on a target machine. Compilers can often make optimizations that the above three alternatives cannot make. However, source code for programs we wish to translate is often not available, and it is possible that the code was not produced by a compiler, but rather was written by hand. In these cases, a compiler cannot be used.

## *IV. Difficulties*

While translating simple straight-line code may be a difficult task, binary translation often requires more than just the simple replacing of op-codes and adjusting of the order of the operands. It is essential to maintain exactly the behavior of the program. This is often a difficult task even when there are many similarities between the source and target processors. Some factors that must be considered when one is doing binary translation include branching, pipelining and self-modifying code.

### A. Branching

The number of instructions in the translated code will not necessarily be the same as the number of instructions in the original code. It is also possible that a translated instruction’s length could differ from its corresponding instruction for the source architecture (i.e. 8 bits to 16 bits). This means that the locations of routines in the translated code could be at different addresses than they were in the original code. Because of this it will likely be necessary to adjust the target address of some branch instructions. While it is relatively simple to adjust the length of statically determined branches, extra work must be taken to handle dynamically determined branches. With dynamically determined branches, every instruction can be the target for a branch instruction, and which instruction will eventually be the target of the branch cannot generally be determined at translation time.

### B. Pipelining

Pipelining creates data dependencies that determine when the processor can read from or write to data locations. The code produced for the target machine must not violate these constraints even though the number and order of instructions may be changed.

In addition, pipelining's effects on delayed branches make the matter of translation even more complicated [4]. This applies to all instructions and conditions that interrupt the sequential execution of a program.

### C. Self-Modifying Code

The third difficulty with binary translation is self-modifying code. Because self-modifying code is generally specific to the machine code on which the program was written, it is difficult to create a binary translator which can handle it. It is difficult to locate sections of self-modifying code and more difficult to translate this code if it can be found.

## *V. Future Work*

Our goal is to create a binary translator that converts code for the TI TMS320C5x DSP into code for the TMS320C54x DSP. We do not expect for the translator, at least initially, to be able to translate all possible instructions for the C5x, but we will define a subset of the C5x's instruction set which the translator must be able to accurately translate.

We plan to use an intermediate form in the translator to represent the desired behavior of the processor. It is hoped that by doing this the translator will be more easily generalized to work with other processors.

It is our hope that we can develop a translator that converts behavior on C5x processors to equivalent behavior on C54x processors, and that we can create a framework for this translator upon which others can build.

## *VI. References*

1. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, "Binary Translation", *Comm. of the ACM*, vol. 36, pp. 69-81, Feb. 1993.
2. A. Bergh, K. Keilman, D. Magenheimer, and J. Miller, "HP 3000 Emulation on HP Precision Architecture Computers," *Hewlett-Packard Journal*, pp. 87-89, Dec. 1987.
3. C. Cifuentes and V. Malhotra, "Binary Translation: Static, Dynamic, Retargetable?", *Proc. of the International Conference on Software Maintenance*, IEEE-CS Press, Monterey, CA, pp. 340-349, Nov. 1996.
4. V. Zivojnovic, S. Tjiang, and H. Meyr, "Compiled Simulation of Programmable DSP Architectures", *Proc. of IEEE 1995 Workshop on VLSI Signal Processing*, Osaka, Japan, pp. 187-196, Oct. 1995.
5. C. Mills, S. Ahalt, and J. Fowler, "Compiled Instruction Set Simulation", *Software Practice and Experience*, vol. 21(8), pp. 877-889, Aug. 1991.

6. V. Zivojnovic, S. Pees, and H. Meyr, “LISA – Machine Description Language and Generic Machine Model for HW/SW Co-Design”, *Proc. of IEEE 1996 Workshop on VLSI Signal Processing*, San Francisco, California, pp. 127-136, Oct. 1996.
7. V. Zivojnovic and H. Meyr, “Compiled HW/SW Co-Simulation”, *Proc. of the 33rd Design Automation*, Las Vegas, Nevada, pp. 690-695, June 1995.
8. C. Cifuentes and S. Sendall, “Specifying the Semantics of Machine Instructions”, technical report 422, Department of Computer Science, The University of Queensland, Dec. 1997.