# Speeding-up SIMD instructions Dynamic Binary Translation in Embedded Processor Simulation

Luc Michel, Nicolas Fournel and Frédéric Pétrot
TIMA laboratory,
CNRS/Grenoble INP/UJF, Grenoble, France.

***Abstract*—This paper presents a strategy to speed-up the simulation of processors having SIMD extensions using dynamic binary translation. The idea is simple: benefit from the SIMD instructions of the host processor that is running the simulation. The realization is unfortunately not easy, as the nature of all but the simplest SIMD instructions is very different from a manufacturer to an other. To solve this issue, we propose an approach based on a simple 3-addresses intermediate SIMD instruction set on which and from which mapping most existing instructions at translation time is easy. To still support complex instructions, we use a form of threaded code. We detail our generic solution and demonstrate its applicability and effectiveness using a parametrized synthetic benchmark making use of the ARMv7 NEON extensions executed on a Pentium with MMX/SSE extensions.**

## I. Introduction

Audio, video and communication applications are the core activities of embedded devices. Thanks to the integration technology, it has been possible already in the late 90's to realize in software many hardwired device parts, leading to a "revival of the SIMD idea"[1]. Since then, a lot of work has been done regarding compilation tools to automatically exploit[2], extract[3] and simulate systems making use of such instructions[4]. As integration continues, software solutions are preferred to increase the reuse capabilities, and MPSoCs using processors with SIMD extension are now common in embedded platforms. The simulation speed of these platforms is a key point to allow concurrent hardware and software design. One good candidate for fast simulation of instruction sets is the dynamic binary translation approach, that has been recently advocated by several works [5], [6]. However, although binary translation is well suited for the scalar code, it generally ends up as a host function call to an instruction interpretation function for the SIMD instructions in retargetable solutions. This slows down drastically the simulation of programs making use of these extensions, even though most host processors are high performance processors that include some sort of SIMD unit indeed not used by binary translation.

The purpose of this paper is to propose a solution applicable to retargetable dynamic binary translation that can make use of the host computer SIMD capabilities. As the nature of the SIMD extensions of the different instruction set architectures (ISA) are quite different, what can seem to be a trivial problem ends up as a fairly complex one, at least when targeting the ability of doing a translation between many extensions.

978-3-9810801-7-9/DATE11/©2011 EDAA

The paper is organized as follows. Section II presents the principles of dynamic binary translations, as a prerequisite to the rest of the paper, and details the solutions that have been proposed for handling SIMD instructions in this context. Section III introduces our solution making use of a dedicated intermediate representation. Finally, Section IV gives and analyses experimental results, prior to discuss the future works in Section V.

## II. Related work

Binary translation is aiming at transforming instructions of one ISA to another. This process can be executed at two different moments: offline, we then talk of static binary translation (SBT) [7], [8], or at run-time, we then talk of dynamic binary translation (DBT) [9], [10].

The main and original goal of binary translation has been migration and compatibility. Indeed, numerous works propose solutions based on binary translation (static or dynamic) aiming at exploiting new features of more recent versions of an architecture (for example UQBT with the SPARC ISA [7]). Others propose solutions easing migration from some architecture to another [8], [11], [12].

More applications of this technique have emerged in the recent years, for example Just-in-time compilation for Java, native binary acceleration [13] or virtualization and system simulation [10].

In the DBT field, efforts have been made to build machine adaptable binary translator. In that way, the translator can be adapted to numerous architectures with a reduced effort [10], [14], [9]. These translators are all based on an intermediate representation so that the complete process of binary translation can be described in a two-phases manner, as proposed on figure 1. The intermediate representation is machine independent to guaranty the re-targetability of the translator. This representation can be viewed as a generic machine instruction set, it is even called micro-operations in QEMU [10] or I-RTL in Walkabout [9].

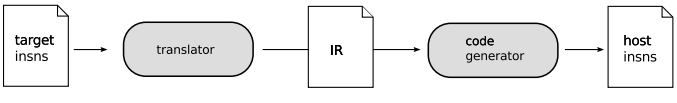


Fig. 1. Machine adaptable dynamic binary translation process

The first phase is in charge of translating the target instruction into the machine independent intermediate representation.

Most of the case, this phase is called translator [10], [9]. The second phase is then in charge of encoding the intermediate representation into host instructions. It is called code generation, by analogy with the compilation notion, [10] or encoder [9].

Among all the works proposed until now, many have addressed the issue of translating binaries from a scalar target architecture to another scalar architecture. This case is the most direct one, since by falling back in an elementary intermediate representation, we can always find a translation for instructions regardless of their complexity [10].

Other works have proposed translations from scalar architectures to parallel architectures, more precisely to VLIW architectures DAISY [11], IA64 [12]. These solutions are a little more complex than the previous ones since on top of finding a matching between instructions, these solutions have to extract the instruction level parallelism (ILP) to take advantage of the host architecture for better performances.

Our main concern is to simulate embedded processors including SIMD extensions efficiently. To the best of our knowledge, only few works address this question and none in a retargatable way. Among them we can cite an effort from FX!32 [8] to translate MMX SIMD instructions to the Alpha instruction set. Another effort aims specifically at executing SIMD instructions (MMX/SSE) on an IA64 architecture [15].

The usual strategy consists in calling a specific function, named helper in QEMU, that implements the SIMD instruction behavior. We believe that taking advantage of the host SIMD capabilities is a must, and therefore our proposition is to propagate instruction set parallelism from target to host through extensions of the intermediate representation.

## III. Dynamic Binary Translation of SIMD Instructions

### A. Specificity and complexity of SIMD instruction sets

SIMD instructions perform parallel operations on multiple data (Single Instruction Multiple Data – SIMD). We can find today multiple ISA extensions providing SIMD instructions to general purpose CPUs. Among them, we specifically looked at MAX instructions for the PA-RISC, MIPS' MDMX and DSPASE, PPC's AltiVec, SPARC's VIS, Intel's MMX/SSE and ARM NEON. All these instructions set extensions share the same characteristics, described below. For performing parallel operations on multiple data, SIMD instruction performs the operation (or sequence of instructions) on registers interpreted as array of values. This array of data can have a variable number of values of various size, for example a 128 bits wide register can be viewed as two 64 bits, four 32bits, eight 16 bits or sixteen 8 bits values. On top of that the variety of the operations applied to the data is huge. It ranges from the classical arithmetic operation (add, sub, shift, ...) to saturated or rounding arithmetic. Among this large range of instructions, each SIMD instruction set represents a unique subset choice made by the designers. On top of these classical instructions, SIMD extensions can even integrate some exotic instructions such as the `vmul.p8` Neon instruction performing polynomial multiplication which has no equivalent in other SIMD ISA.

We have then to make a careful choice of the micro-operations to add to the intermediate representation of the DBT. The main two constraints we respect for this extension are:

- limit the number of new IR micro-operations in order to limit the burden on the code generator and the overall performances of the binary translator.
- add enough micro-operations in the IR to allow a maximum coverage of the SIMD instruction sets.

Indeed, adding too much micro-operations will tend to add one micro-operation per SIMD instruction. This will not solve the problem since the code generator (the second phase of DBT) will have an heavy work to do to guaranty the emulation of each of the micro-operations. In the other hand if we do not add enough micro-operations all SIMD instructions cannot be expressed and the translator will have a huge task to guaranty this translation. We concentrate then on an extension of the IR with a set of instructions which is close to the intersections of the studied SIMD instruction sets. The IR micro-operations will be 3-addresses operations since it is the most general case and allows to represent the 2-addresses versions easily, whereas the reverse is not true.

As opposed to scalar DBT, finding instruction equivalence in SIMD DBT has to take care the SIMD specificities: parallelism and register interpretation.

### B. Different translation cases

In the process of translating from one SIMD instruction to an other, we face similar situations compared to scalar DBT. The main difference is the number of parameters to take into account in this process. We will illustrate this all along this section with concrete examples of translation from ARM NEON instruction set to Intel MMX/SSE.

*a) Direct mapping between instructions:* fortunately the easiest case of the scalar DBT is also present in the SIMD DBT. It is the presence of an exact equivalence between a target SIMD instruction and an host SIMD instruction. The behavior of the SIMD DBT in this situation is quite similar to the one of the scalar DBT. All we have to do is to guaranty to convey operands to correct registers and retrieve the result from the correct register. This case can be called a *direct mapping*. The main diverging aspect between scalar and SIMD *direct mappings* lies in the fact that we have to guaranty that there is the same level of parallelism between the two instructions, *i.e.* the same interpretation of registers (couple of number and size of the values).

This case is widely applicable on arithmetic operations of SIMD instruction sets. Figure 2 illustrates the DBT of an ARM Neon `vadd.i16` into an Intel MMX/SSE `paddw`. The IR micro-operation used to propagate the parallelism is named `simd_128_add_i16` and represents the SIMD instruction performing 8 parallel adds on 16 bits values in 128 bits registers.

Table I gives some examples of *direct mappings* between the ARM Neon add instructions and the Intel SSE ones. These examples are only 128 bits adds, but equivalent mapping can


Fig. 2. Direct mapping between `vadd.i16` Neon instruction and `paddw` MMX/SSE instruction

TABLE I
MAPPING BETWEEN ADDITION INSTRUCTIONS

| Operation | Neon instruction | MMX/SSE instruction |
|---|---|---|
| add 8 bits | `vadd.i8 Qd, Qn, Qm` | `paddb xmm1, xmm2` |
| add 16 bits | `vadd.i16 Qd, Qn, Qm` | `paddw xmm1, xmm2` |
| add 32 bits | `vadd.i32 Qd, Qn, Qm` | `paddd xmm1, xmm2` |
| add 64 bits | `vadd.i64 Qd, Qn, Qm` | `paddq xmm1, xmm2` |

also be found for 64 bits instructions and for other arithmetic instructions such as **`sub`**, **`and`**, **`or`** and **`xor`**.

*b) No direct mapping:* in a less favorable case, there exists no direct mapping between instructions of the instruction sets. Most of the cases, this lack of mapping is due to a lack of generality of the operations performed by the target SIMD instruction. In that case it is only of little interest to have a micro-operation in the IR for that instruction. The strategy in such cases is to split the target SIMD instruction in more elementary operations present in the IR. This technique is once more identical to the one used in scalar DBT, but we have still more parameters to take into account during the process, *i.e.* parallelism level and registers interpretation.

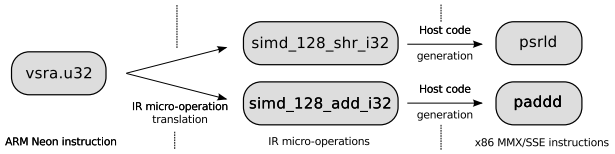


Fig. 3. The `vsra` Neon instruction is translated into two TCG micro-operations

Figure 3 gives an example of this situation with the translation of the ARM Neon **`vsra.u32`** instruction which is performing a right shift on operands and accumulate the shifted results in the output register. This SIMD instruction is translated into two elementary IR micro-operations **`simd_128_shr_i32`** and **`simd_128_add_i32`**. The code generator can then find an equivalent for each micro-operation, *i.e.* **`psrld`** and **`paddd`**.

*c) The exceptional case:* a third and least favorable case is finally possible. This situation is quite seldom, but due to the way instructions have been chosen for integration in the SIMD instruction sets, it can be encountered. This situation happens when an SIMD instruction of the target can be translated into a corresponding IR micro-operation but no equivalent is available in the host SIMD instruction set. As shown in table II, all versions of the shift are available in ARM Neon SIMD instruction set. This is even true for all SIMD instruction sets except the Intel SSE SIMD instruction set. As it can be realized from this table, there exists no instruction for shifting 8 bits values. As this operation is present in all other instruction sets, it is present in the IR.

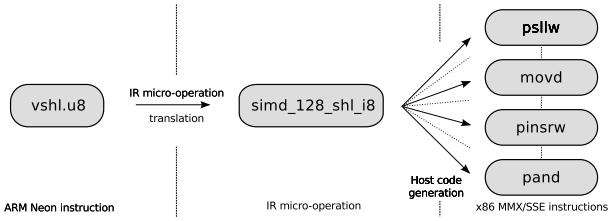


Fig. 4. The left shift micro-operation is translated into multiple MMX/SSE instructions

TABLE II
MAPPING BETWEEN LEFT SHIFT INSTRUCTIONS

| Operation | Neon instruction | MMX/SSE instruction |
|---|---|---|
| shl 8 bits | `vshl.i8 Qd, Qm, #imm` | N/A |
| shl 16 bits | `vshl.i16 Qd, Qm, #imm` | `psllw xmm1, xmm2` |
| shl 32 bits | `vshl.i32 Qd, Qm, #imm` | `pslld xmm1, xmm2` |
| shl 64 bits | `vshl.i64 Qd, Qm, #imm` | `psllq xmm1, xmm2` |

The code generator has then to solve this situation by generating multiple host instructions, as shown in figure 4. The example given in this figure is for the translation of a 8 bits logical left shift emulated by a 16 bits version. Figure 5 gives more details of the sequence of MMX instructions used for this emulation of the 8 bits logical left shift of 4 (we used 64 bits MMX version for the sake of readability). As shown on the figure, the MMX version contains the 16 bits left shift operation followed by other MMX instructions in charge of masking the uselessly propagated bits due to the mismatch of operation width (8 to 16 bits). We may hope that this least favorable case is still much better than doing it with a sequence of scalar instructions.

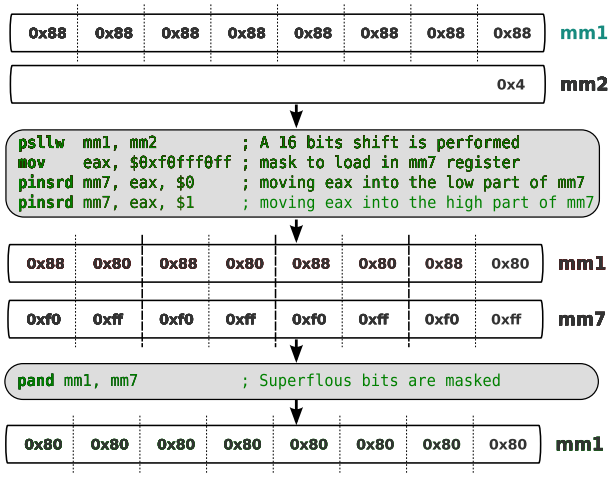


Fig. 5. Reproducing 8 bits left shift with 16 bits left shift and a mask

*d) Comparison instructions:* As far as comparisons are concerned, PowerPC Altivec, Sparc VIS, MMX/SSE and Neon instruction sets provide the result for each element in the output operand, whereas the MIPS DSPASE sets flags. Because

of this unbalanced distribution, we select the micro-operations that produce their results in the output operand.

## IV. Experiments

The proposed IR extension has been ported in the QEMU binary translator. The translator for the ARM Neon instruction set and the code generator for the Intel MMX/SSE instruction set have been implemented.

### A. Functional tests

In order to validate the correctness of our implementation, we ran different multimedia format conversions with `ffmpeg` to compare the SIMD DBT from ARM Neon to Intel MMX/SSE results to the ones obtained for the CortexA8 of a BeagleBoard. Thanks to the intrinsic parallelism of SIMD instructions the number of cases is limited.

### B. Benchmarks and performance measurement

In order to estimate the performances of the SIMD DBT, we have generated synthetical benchmarks containing between 0% and 100% of SIMD instructions. These benchmarks covered the three cases presented before (`vadd.i16`, `vsra.u16` and `vshl.u8`).

These benchmarks have been executed on the original version of QEMU using `helper` functions as well as on our improved version. Figure 6 shows the benchmark results for the three Neon instructions. The curves represents the execution time of the improved version $T_i$ relative to the original version's execution time $T_o$: $T_i/T_o$.

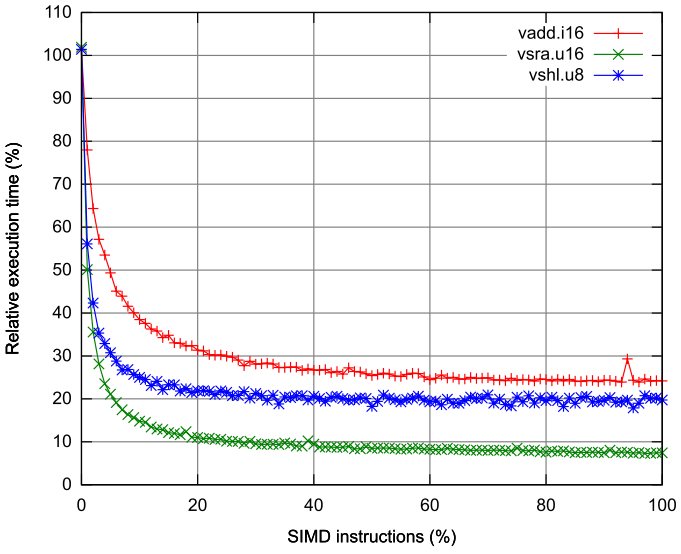


Fig. 6. Performance measurement

As it can be realized from the figure, with a 0% of SIMD instructions, our version reaches the same execution time than the original version. This means that the modification made to the DBT has no negative impact on the scalar DBT performances.

At the other end of the curve, (100% of SIMD instruction point), we can realize that the speed up for the three cases is good, since the worst result is a 4x improvement (for the `vadd.i16` case – *direct mapping*). Among the three cases, the easiest one does not give the best speedup as its helper version also has the least complexity.

Finally, we can observe that regardless of the ratio of SIMD instructions, the improved solution proposes a good enhancement of the performances. It even reaches a speedup close to its final speedup at about 20 %, which is due to the huge weight of SIMD simulation in the original version.

## V. Perspectives

As these preliminary results are encouraging, we believe that two main, although ambitious, follow-ups are possible. The first one concerns the definition of an abstract representation of the semantic of the SIMD instructions to (1) optimize the intermediate instruction set and (2) allow to automate the generation of the translation. The second one concerns the dynamic translation of VLIW codes.

As a final remark, even though we focused on embedded to desktop processor translation, the strategy we propose is also viable for desktop to desktop translation and could benefit to virtualization technologies.


## Acknowledgment

The authors would like to thank the Catrene office and the French Public Authorities for supporting this work through the Catrene COMCAS project.


## References


[1] M. Schlett. Trends in embedded-microprocessor design. *Computer*, 31(8):44 –49, aug. 1998.

[2] R. Leupers. Compiler design issues for embedded processors. *IEEE Design and Test of Computers*, pages 51–58, 2002.

[3] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. *International Journal of Parallel Programming*, 31(6):411–428, 2003.

[4] P. Mishra and N. Dutt, editors. *Processor Description Languages, Volume 1*. Morgan Kaufmann Publishers Inc., 2008.

[5] M. Gligor, N. Fournel, and F. Pétrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *CODES+ISSS*, pages 71–80. ACM, 2009.

[6] M. Becker, G. Di Guglielmo, F. Fummi, W. Mueller, G. Pravadelli, and Tao Xie. RTOS-aware refinement for TLM2.0-based HW/SW designs. In *Design, Automation Test in Europe Conference*, pages 1053 –1058, March 2010.

[7] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33(3):60–66, 2000.

[8] P. J. Drongowski, D. Hunter, M. Fayyazi, D. Kaeli, and J. Casmira. Studying the Performance of the FX!32 Binary Translation System. In *the First Workshop on Binary Translation*, 1999.

[9] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *DYNAMO '00*, pages 41–51, 2000.

[10] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *the USENIX Annual Technical Conference*, pages 41–46, 2005.

[11] K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *ISCA 24*, pages 26 – 37, 1997.

[12] C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer*, 33(3):47–52, 2000.

[13] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM conference on Programming language design and implementation*, pages 1–12, 2000.

[14] B. Yuncheng, L. Alei, and G. Haibing. Design and implementation of crossbit:dynamic binary translation infrastructure. *Computer Engineering*, 33(23):100 – 134, 2007.

[15] J. Li, Q. Zhang, S. Xu, and B. Huang. Optimizing dynamic binary translation for SIMD instructions. In *Code Generation and Optimization*, March 2006.