ISSN 0361-7688, Programming and Computer Software, 2017, Vol. 43, No. 6, pp. 366–372. © Pleiades Publishing, Ltd., 2017.
Original Russian Text © K.A. Batuzov, 2017, published in Programmirovanie, 2017, Vol. 43, No. 6.

The Use of Vector Instructions of a Processor Architecture for Emulating the Vector Instructions of Another Processor Architecture

K. A. Batuzov

Institute for System Programming, Russian Academy of Sciences, Moscow, 109004 Russia

e-mail: batuzovk@ispras.ru

Received June 10, 2017

Abstract—The complexity of software is ever increasing, and it requires more and more computational resources for its execution. A way to satisfy these requirements is the use of vector instructions that can operate with fixed-length vectors of data of the same. A method for representing vector instructions of one processor architecture in terms of the vector instructions of another architecture during the dynamic binary translation is proposed. An implementation of this method that includes the translation of vector addition and memory access increased the performance of the QEMU emulator by a factor greater than three on an artificial example and 12% on a real-life application.

DOI: 10.1134/S0361768817060032

1. INTRODUCTION

The rush toward performance is the inevitable part of the history of computing. The ever increasing complexity of software requires more computational resources for its execution. A way to satisfy these requirements is the inclusion of vector instructions that can operate with fixed-length vectors of data of the same type in the processor architecture. Many widespread processor architectures support vector instructions: the x86 architecture includes the SSE extension [1], ARM includes NEON [2], and PowerPC has AltiVec [3].

Vector instructions can be included in the code in various ways—by writing assembler code by hand, by using intrinsic functions provided by a compiler, or by the automatic loop vectorization performed by a compiler.

In this paper, we consider the class of programs called emulators. Emulators are used for running programs written for one (guest) architecture or operating system on a processor of another (host) architecture or under another operating system. There are efficient solutions for the case when the guest and the host architectures are identical, but we consider the general case when these architectures are different.

In the general case, emulators use dynamic binary translation when the executable code of a program is translated during its execution by small acyclic fragments into the code of the host architecture. The overwhelming part of the time, the translated code rather than the code of the emulator is executing.

Ordinary methods for introducing vector instructions in the program cannot be used in dynamic binary translation. Since the translation is from a correct executable code of a certain processor architecture, no intrinsic functions can be encountered in this code. The automatic vectorization deals with loops, while the dynamic binary translator works with small acyclic regions. However, the original program probably already contains vector instructions, and they can be represented by the vector instructions of the host architecture.

In this paper, we investigate the possibilities of representing the vector instructions of a processor architecture in terms of the vector instructions of another architecture. The proposed method was implemented in the full system emulator QEMU. This implementation includes only the translation of vector addition and memory access operations. Experiments showed the speedup by a factor greater than three on an artificial example and by 12% on a real application.

2. PROBLEM DOMAIN AND RELATED WORK

2.1. Vector Instructions

Vector instructions can perform operations on a sequence of numbers of the same type rather than on a single number. These instructions operate with vec-

tors of a fixed bit length. The length of vectors can be different, depending on the specific architecture, but usually it is a power of two. In modern architectures, vectors of the length 64, 128, 256, and 512 bits are used. Each vector is interpreted as a sequence of elements. Depending on the size of the elements, their number can be different. For example, a 128-bit vector can be interpreted as a set of four 32-bit numbers or as 16 8-bit numbers. The interpretation depends on the instructions applied to the vector.

To organize computations, the processor architecture provides a set of *vector registers,* and a set of *vector instructions*. Often, these sets are implemented as extensions of the processor architecture. For example, the widespread `x86` architecture has `MMX`, `SSE`, and `AVX` vector extensions; and `ARM` has the `NEON` extension (also called the `Advanced SIMD`).

The most popular high-level programming languages, such as Java, C, C++, C#, and Python[1] do not support vector data types and operations. Vector instructions in the resulting code can be formed by the automatic loop vectorization by the compiler, or they can be included by a programmer by hand using Assembler or extended compiler capabilities.

2.2. Dynamic Binary Translation

The dynamic binary translation can translate the executable code from one binary representation into another; in particular, it allows one to execute programs written for one architecture on a different processor architecture. In this case, the architecture for which the program was originally written is called the *guest architecture,* and the architecture on which the program will be executed after the dynamic binary translation is called the *host architecture*.

The binary translation is performed dynamically (in the course of the program execution) because it cannot generally be performed statically (i.e., before the program execution). The translation is done by small fragments called *translation blocks*. The size and the structure of translation blocks can be different depending on a specific implementation. However, a typical requirement is that the block must be acyclic because the dynamic binary translation environment must be able to regularly get control to handle events, such as interrupts from various devices.

Typically, the translation goes in two phases: first a translation block is disassembled into an internal machine-independent representation, and then an equivalent code for the host architecture is generated based on this representation. The part responsible for the generation of the code from the guest architecture into the internal representation is called the frontend, and the part responsible for the generation of the code for the host architecture is called the backend.

2.3. Related Work

The dynamic binary translation is widely used for the implementation of emulators. The translation of scalar instructions is well studied and verified in practice. The set of scalar instructions includes a standard set of basic instructions that are implemented in all widespread architectures—the basic arithmetic operations, bitwise operations, shifts, comparison operations, conditional and unconditional jump instructions, and subroutine calls. Processor architectures can include a large number of specific instructions, but all of them either can be represented through the basic instructions or a helper function can be written for them that implements the required semantics and can be called from the resulting code for the host processor.

The translation of vector instructions is much less studied. The sets of vector instructions provide much more operations, and they significantly vary from architecture to architecture. The representation of operations in one set by operations of another set is generally fairly complicated because the same operations must be performed on all components of the vector; therefore, conditional control transfers can be used only in a very limited number of situations. In addition the implementation of the helper function performing the equivalent computations is also difficult. This is due to the fact that high-level languages lack the required data types, and the calling conventions for vector values vary much more from one compiler to another.

There are specific solutions for representing one set of vector instruction through other instructions. For example, the tool ExaGear Desktop produced by ElTech [4] designed for running programs written for x86 on ARM processors supposedly is able to translate the vector instructions of the SSE extension of x86 into the instructions of the NEON extension of ARM. This software is proprietary, and no information about its internal organization is freely available. However, the user documentation states that, in order to execute the source programs containing MMX/SSE instructions, the host architecture must support NEON [5]. Note that NEON has a larger set of instructions that MMX/SSE, and it easier to represent MMX/SSE instructions through NEON instructions than vice versa.

Another example of the dynamic binary translation of vector instructions into another set of vector instructions is provided by Valgrind [6, 7]. Valgrind is a dynamic binary translation framework designed for the analysis of binary code aimed at detecting bugs or performance analysis. The analyzers (called Valgrind tools) are written as machine-independent plugins

[1] These languages occupy the first five positions in the TIOBE index as of April 2017.

that work with the internal representation. The transformation from the binary code into the internal representation and back are done by the framework. Valgrind supports a number of different architectures with vector extensions, but it always assumes that the guest and host architectures are identical. Thus, Valgrind always translates vector instructions into vector instructions of the same architecture. Nevertheless, in order for the Valgrind tools to work correctly, the internal representation must be sufficiently architecture independent. In particular, in different architectures, the same operations of the internal representation must have identical semantics. In Valgrind, many operations of the internal representation are used to represent instructions of different architectures.

These observations allow us to assume that there are large intersections between different sets of vector instructions and that a part of instructions in one set can be represented through instructions of another set. This paper is devoted to the experimental verification of this assumption.

2.4. The QEMU Emulator

All the experiments described in the present paper were performed using the full system open code emulator QEMU [8]. For translation, QEMU uses the internal representation TCG [9]. TCG is a low-level representation of the program resembling assembler. It includes arithmetic and logic operations, conditional and unconditional jumps, loading from memory, saving to memory, bitwise operations, and function call instructions. All computations are performed on variables that can be global, local, or temporary. The global variables exist during the entire process of dynamic translation. They have meaningful names and preserve their values from one translation block to another. The local variables exist only within one translation block and temporary variables exist within one basic block. The local and temporary variables have no meaningful names; rather they have unique identifiers. The internal representation in QEMU supports only two data types—the 32- and 64-bit integers.

The processor state is described by the structure `CPUState`. Some fields of this structure can be accessed from the generated code using the load and store operations, while some other fields are declared as global variables in the internal representation. If a field is a global variable of the internal representation, then it cannot be operated on by accessing the corresponding memory by pointer. QEMU has no mechanism for tracking the events of overwriting variables when the memory is accessed by pointer, which can result in an incorrectly generated code at the stage of register allocation. As the code for the host system is generated, only variables can be allocated to registers. All memory accesses always remain memory accesses.

The guest system memory is emulated using a byte array. This array is accessed using the special instructions `qemu_ld` и `qemu_st`, which transform the loading from and saving to the virtual addresses of the guest system to accesses to the appropriate elements. These instructions may call the helper function that actually transforms the addresses. The resulting mapping is saved in the translation lookaside buffer (TLB). When the same memory page is accessed again, no addresses are actually computed, but the ready-to-use result from the TLB is taken.

The vector instructions are always emulated by accessing the memory by pointer. The elements of the vector are read one-by-one, then the required operation is performed on them, and finally the elements of the resulting vector are written to the memory.

3. DESCRIPTION OF THE IMPLEMENTATION

To implement the translation of the guest architecture vector instructions into the vector instructions of the host architecture, the following is needed:

- add the corresponding data type to the internal representation,
- add operations on the variables of this type,
- implement the work of `qemu_ld` and `qemu_st` with the values of this type,
- add the use of the new operations to the frontend,
- add the generation of code for the new operations to the backend.

The minimal set of operations must include the load, store, and move operations. All other operations can be emulated using the supported operations or using scalar operations on the vector elements.

To make the usage of vector instructions of the host architecture possible, the vector variables must be allocated to vector registers of the host architecture, i.e. they must be global variables of the internal representation. On the other hand, not all vector instructions of the guest architecture can be represented through vector instructions of the host architecture; i.e., a code must sometime be generated that accesses individual elements of the vector in memory. Therefore, two earlier incompatible methods of working with the processor state must be able to work with the same fields of the structure `CPUState` simultaneously.

A large number of combinations of the guest and host architectures makes the choice of the method of emulation for each specific instruction a nontrivial task. Therefore, we should design a convenient functions that will choose and generate the optimal emulation method for each operation, depending on the host architecture. Without such wrapper functions, the practical application of the proposed method will be too labor consuming.

At least in one processor architecture, there is a considerable overlapping of vector registers of different lengths. In the NEON extension of ARM, the 32-bit registers `s0–s31` joined by pairs to form 64-bit registers `d0–d15`; in turn, these registers are joined by pairs to form 128-bitregisters `q0–q8`. Such a situation is also encountered for scalar registers. For example the register ax in `x86` consists of the registers `al` and `ah`, and it is a part of the register `eax`. Presently, all these registers are considered as a single global variable corresponding to the longest register; if needed, its fragments can be accessed. This creates additional data dependences between nonoverlapping fragments; however, since short registers are rarely used, this does not significantly deteriorate the performance. The use of such an approach in the case of vector registers is undesirable because the reduction of the independent 32-bit registers by a factor of four will significantly decrease the performance of the applications that use these registers.

3.1. Pointer Analysis

In order to make it possible to work with the same fields of the processor state as with global variables and as with memory locations, the emulator must provide a mechanism for tracking the situations in which a memory access is overlapping with a global variable. This is a well-known problem called pointer analysis [10], and no methods for its sufficiently accurate solution in the general case are known. Fortunately, in the case under consideration we can make additional assumptions that significantly simplify the problem. These assumptions are consequences of the principles of the emulator operation and are as follows.

1. All variables reside in the emulator's memory, and memory accesses from the guest system cannot overlap with them. For the cases when the registers of the guest system can be mapped to the address space, the emulator has a special mechanism called `iomem`, which ensures the validity of accesses through the address space at the cost of decreased performance. All operations with the corresponding addresses call the appropriate helper functions that ensure that all the variables are correctly saved.

2. All global variables are assigned a memory location that is a field in `CPUState`, and all variables are accessed using the address of the beginning of this structure with a constant offset.

3. The vast majority of the emulator memory accesses are made relative to the beginning of `CPUState`.

Thus, for each memory access, it must be found out if it is addressed relative to the beginning of the structure `CPUState`, and if this is the case, then determine the offset. Then, knowing the offsets and sizes of all variables, one can find out which of them overlap with the given memory access. For the address of the beginning of `CPUState`, a special variable in the internal representation called `env` is assigned; it is always stored in a specially assigned register and its value is never modified.

The control flow graph of each translation block is acyclic; therefore, the data flow can be analyzed using only one pass in forward direction. For each variable at each point of the program we can determine

• if its value is a compile-time constant at this point of the program, and, if this is the case, then what is its value;

• whether this variable's value can be represented as `env + C`, where `C` is a compile-time constant; if this is the case, then determine the value of this constant.

If none of these conditions holds true, then we assume that the variable contains a value about which nothing is known.

All memory accesses in the internal representation use the address in the form *base* + *offset*, where both the *base* and the *offset* are variables. If at least one of these variables can be represented as `env + C` and the other one is a constant, then one can unambiguously determine the variables in the internal representation intersecting with the given operation. Otherwise, a conservative assumption that this operation can intersect with any variable may be used. In practice, this case in never realized.

The data about the overlapping of memory accesses with global variables is used in the liveness analysis, and the results of this analysis are used for register allocation. Two types of overlapping are differentiated:

• *complete,* when the memory access completely covers all bytes of the variable,

• *partial*, when the memory access has only some common bytes with the variable.

When the memory is read, all the variables with which this access overlaps must be saved to memory. When the memory is written, only the variables with which this access partially overlaps must be saved to memory, and the variables that are completely overwritten by this write operation must be marked as dead.

Consider an example. Suppose that the piece of code for ARM shown in Listing 1 is executed. Suppose that the translation to the vector operation of the host architecture is supported for the vector addition `vadd` but not supported for the pairwise vector addition `vpadd`. Then, in the intermediate representation, the corresponding vector operations will be generated for

Listing 1

```
vadd.i32    q0, q0, q1
vpadd.i32   d0, d0, d1
vadd.i32    q0, q1, q1
```

Listing 2

```
---- vadd.i32 q0, q0, q1
add_i32x4 q0, q0, q1

---- vpadd.i32 d0, d0, d1
ld_i32 tmp5, env, $0x858
ld_i32 tmp6, env, $0x85c
add_i32 tmp5, tmp5, tmp6
st_i32 tmp5, env, $0x858
ld_i32 tmp5, env, $0x860
ld_i32 tmp6, env, $0x864
add_i32 tmp5, tmp5, tmp6
st_i32 tmp5, env, $0x85c

---- vadd.i32 q0, q0, q1
add_i32x4 q0, q0, q1
```

Listing 3

```
;---- vadd.i32 q0, q0, q1
    movdqu    0x858(%r14), %xmm0
    movdqu    0x868(%r14), %xmm1
    paddd     %xmm1, %xmm0
    movdqu    %xmm0, 0x858(%r14)

;---- vpadd.i32 d0, d0, d1
    mov       0x858(%r14), %ebp
    mov       0x85c(%r14), %ebx
    add       %ebx, %ebp
    mov       %ebp, 0x858(%r14)
    mov       0x860(%r14), %ebp
    mov       0x864(%r14), %ebx
    add       %ebx, %ebp
    mov       %ebp, 0x85c(%r14)

;---- vadd.i32 q0, q0, q1
    movdqu    0x858(%r14), %xmm0
    paddd     %xmm1, %xmm0
    movdqu    %xmm0, 0x858(%r14)
    movdqu    %xmm1, 0x868(%r14)
```

vector addition, while for the pairwise vector addition the emulating code that reads elements one-by-one will be generated. The resulting intermediate representation is shown in Listing 2. The offsets of all memory accesses relative to `env` are constants, and these accesses overlap with the variable `q0` with the offset `0x858`, but they do not affect the variable `q1` whose offset is `0x868`. Therefore, the value of `q0` must be saved to memory after the first vector addition and loaded back before the last one. The variable `q1` can be loaded to the register once in the beginning of the translation block and saved at its end. The corresponding assembler code for x86_64 is shown in Listing 3. The register `%r14` is used to store the value of `env`.

3.2. Overlapping of Variables

The case of the complete or partial overlapping of variables is processed in the same way as the overlapping of memory accesses with variables. For each variable, two lists must be composed: the list of variables that are completely within this variable and the list of variables that partially overlap with the given one. These lists may be made up by hand or generated automatically.

For the automatic computation of overlapping of variables, note that here we are dealing only with global variables. All the global variables are also fields in the structure `CPUState`, and they can be described by their size and offset of the corresponding field from the beginning of the structure. Thus, to check if two variables overlap, it is sufficient to check if two given address ranges intersect. The total number of global variables is not large, and they are created only once when the emulator is initialized. Therefore, the iteration over all pairs of variables takes $O(N^2)$ operations, which is quite acceptable from the performance point of view.

3.3. Wrapper Functions

Different host architectures can support different sets of vector instructions. Therefore, as the source code is translated, the support of each potential vector instruction must be checked. To implement this feature, wrapper functions that check whether the corresponding vector instruction can be used. If this instruction is supported, then a vector operation is generated in the internal representation; otherwise, the emulation code is generated that handles each element of the vector individually.

The emulation code can be organized in different ways:

- as a series of calls to a helper function of which each performs the operation on one vector element,
- as a single call to a helper function that transforms the entire vector,
- as a sequence of scalar operations on the internal representation.

The last method is preferable from the performance point of view. However, for the operations that are difficult to represent in internal representation, the other methods are also acceptable.

4. EXPERIMENTAL RESULTS

The translation method of the guest vector instructions into the host vector instructions described above was implemented in QEMU version 2.9.50 (commit

Listing 4

```
        mov         r0, #0xb0000000
loop:
        vadd.i32    q0, q0, q1
        vadd.i32    q0, q0, q1
        vadd.i32    q0, q0, q1
        vadd.i32    q0, q0, q1
        subs        r0, r0, #1
        bne         loop
```

Listing 5

```
int a[256], b[256], c[256];
void foo (void) {
  int i;

  for (i = 0; i < 256; i++){
    a[i] = b[i] + c[i];
  }
}
```

9964e96dc9999cf7f7c936ee854a795415d19b60). More precisely, memory access vector operations and vector addition operations of 64- and 128-bit vectors were implemented. The guest architecture was `ARM`, and `x86_64` was used as the host architecture. No other operations besides addition have been implemented, but even the partial implementation of the proposed approach demonstrates a considerable improvement of performance. The implementation was tested on different types of programs:

- an artificial example in which the proposed method was expected to demonstrate a significant effect,
- the result of automatic vectorization by the `gcc` compiler,
- artificial programs aimed at the verification of validity of pointer analysis and overlapping of variables,
- the real-life video compression program `x264`, which intensively uses vector instructions written by hand.

The artificial program for checking the potential improvement in performance is a loop with a fixed number of iterations in which vector registers are repeatedly added. The assembler code of this program is shown in Listing 4.

The second type of tests was obtained by running the automatic vectorization algorithm [11] on a loop that adds the elements of two arrays and writes the result to the third array. The C source code is shown in Listing 5. By varying the types of array elements, four different test cases that used different types of addition were obtained. For adding one-byte numbers, the number of elements in the array was doubled because otherwise the compiler unrolls the loop. These tests were compiled by GCC [12] version 4.7.3 20130102 (prerelease) with the options `-O3 -ftree-vectorize -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a8 -fno-unroll-loops`.

Table 1. Performance measurements on various tests

Test	QEMU	QEMU-vect	Speedup
artificial	25.304	7.748	3.27x
autovect.i8	1.604	0.616	2.60x
autovect.i16	3.648	1.304	2.80x
autovect.i32	5.392	2.248	2.40x
autovect.i64	6.296	4.280	1.47x
x264	204.356	183.172	1.12x

The artificial programs for verifying the validity of pointer analysis and overlapping of variables were obtained from the tests of the two preceding types. The assembler code of these tests was modified by adding instructions operating on overlapping variables or unsupported instructions that should be translated into memory accesses in the emulator.

The validity of the results produced by these test programs was checked by outputting their resulting values.

As the real-life application, the program `x264` [13] (commit 3f5ed56d4105f68c01b86f94f41bb9bbefa3433b) was used to compress a short video clip. In this case, the validity of the result was verified by comparing the `md5` hash of the resulting file with the hash of the expected file. Since this implementation of the video codec is deterministic, such a comparison makes sense.

The results of performance measurements are shown in Table 1. The column "QEMU" corresponds to running the unmodified QEMU, and the column "QEMU-vect" corresponds to running QEMU with the modifications described in the paper. The row "artificial" corresponds to the artificial test. The rows "autovect.i8", "autovect.i16", "autovect.i32", and "autovect.i64" correspond to the tests obtained using the automatic vectorization. The number at the end shows the element size in bits. The row "x264" shows the results of running the video encoding program.

On the artificial program, the speedup is by a factor greater than three. On the automatically vectorized loops, the speedup is slightly lower because the contribution of vector instructions to the total execution time is lower. In the case of 64-bit numbers, the speedup is significantly lower because the current implementation uses less operations compared with the implementation for the 32-bit numbers. There is no considerable difference between the 8-bit, 16-bit and 32-bit numbers due to the features of the current implementation. In this implementation, the reads are

always performed by 32-bit fragments, and the vector addition of 32-bit vectors is made using scalar operations and masking the most significant bits. Thus, the difference in the number of operations in the basic and modified implementations remains unchanged.

In distinction from the other tests, the real-life application uses a greater number of various instructions without preferring the implemented addition operations. However, on this program we also observe a noticeable speedup by 12%, which is explained by the influence of operations of copying data arrays. When vector registers are used, a less memory loading and memory saving operations are needed.

5. CONCLUSIONS

A method for representing vector instructions of one processor architecture in terms of the vector instructions of another architecture during the dynamic binary translation was proposed. The difficulties occurring in the implementation of this method were analyzed and ways for resolving them were proposed. An implementation for the special case of the pointer analysis problem and processing of overlapping of global variables of the internal representation was developed. The method was tested in the open-source emulator QEMU. As a result, the speedup by a factor greater than three on an artificial program and 12% on a real-life application was achieved.

In the further work, it is intended to implement a larger set of vector operation and pass this implementation to QEMU developers. This is a practical task, all the studies needed for its accomplishment have been made, verified on a certain set of operations, and described in the present paper.

REFERENCES

1. *AMD64 Architecture Programmer's Manual,* Vol. 4: *128-Bit and 256-Bit Media Instructions.* https://support.amd.com/TechDocs/26568.pdf
2. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R Edition.* http://infocenter.arm.com/help/index.jsp?topic=//com.arm.doc.ddi0406c/index.html
3. *AltiVec™ Technology Programming Environments Manual.* http://www.nxp.com/docs/en/reference-manual/ALTIVECPEM.pdf
4. *Eltechs ExaGear Desktop. Run x86 Applications on ARM-based Devices.* https://eltechs.com/
5. *ExaGear Desktop System Requirements* (updated for v2.1). http://forum.eltechs.com/viewtopic.php?f= 4&t=4&sid?61125b0cdd4fdc640dee682449c870
6. Nethercote, N. and Seward, J., Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, in *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007),* San Diego, Calif., 2007, vol. 42, no. 6, pp. 89–100.
7. Valgrind: Code Repository. http://valgrind.org/downloads/repository.html
8. Bellard, F., QEMU, A fast and portable dynamic translator, in *Proc. of the Annual Conference on USENIX,* 2005, pp. 41–46.
9. QEMU Documentation/TCG. http://wiki.qemu.org/Documentation/TCG
10. Aho, A., Lam, M., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques, & Tools,* Boston: Pearson/Addison Wesley, 2007, 2nd ed.
11. Auto-vectorization in GCC — GNU Project — Free Software Foundation (FSF). https://gcc.gnu.org/projects/tree-ssa/vectorization.html
12. GCC, the GNU Compiler Collection. https://gcc.gnu.org/
13. x264, the best H.264/AVC encoder — VideoLAN. http://www.videolan.org/developers/x264.html

Translated by A. Klimontovich