Free Rider: A Source-Level Transformation Tool for Retargeting Platform-Specific Intrinsic Functions

FREE RIDER: A Source-Level Transformation Tool for Retargeting Platform-Specific Intrinsic Functions

STANISLAV MANILOV, School of Informatics, University of Edinburgh
BJÖRN FRANKE, School of Informatics, University of Edinburgh
ANTHONY MAGRATH, Cirrus Logic International (UK) Ltd.
CEDRIC ANDRIEU, Cirrus Logic International (UK) Ltd.

Short-vector SIMD and DSP instructions are popular extensions to common ISAs. These extensions deliver excellent performance and compact code for some compute-intensive applications, but they require specialised compiler support. To enable the programmer to explicitly request the use of such an instruction, many C compilers provide platform-specific intrinsic functions, whose implementation is handled specially by the compiler. The use of such intrinsics, however, inevitably results in non-portable code. In this paper we develop a novel methodology for retargeting such non-portable code, which maps intrinsics from one platform to another, taking advantage of similar intrinsics on the target platform. We employ a description language to specify the signature and semantics of intrinsics and perform graph-based pattern matching and high-level code transformations to derive optimised implementations exploiting the target's intrinsics, wherever possible. We demonstrate the effectiveness of our new methodology, implemented in the FREE RIDER tool, by automatically retargeting benchmarks derived from OPENCV samples and a complex embedded application optimised to run on an ARM CORTEX-M4 to an INTEL EDISON module with SSE4.2 instructions (and vice-versa). We achieve a speedup of up to 3.73 over a plain C baseline, and on average 96.0% of the speedup of manually ported and optimised versions of the benchmarks.

CCS Concepts: •**Software and its engineering** → **Translator writing systems and compiler generators; Source code generation;** Preprocessors; •**Computer systems organization** → *Embedded software;*

Additional Key Words and Phrases: Intrinsic functions, graph matching, source-level transformations, retargeting

ACM Reference Format:
Stanislav Manilov, Björn Franke, Anthony Magrath and Cedric Andrieu, 2015. FREE RIDER: A Source-Level Transformation Tool for Retargeting Platform-Specific Intrinsic Functions. *ACM Trans. Embedd. Comput. Syst.* V, N, Article 1 (January 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Instruction set extensions are computer architects' favourite weapon of choice to add domain-specific acceleration to processor cores with their mature and proven software and hardware ecosystems. For example, INTEL has devised various streaming SIMD extensions (first MMX, then SSE to SSE4 and AVX) to speed up graphics and digital signal processing. Similar capabilities are offered by the ALTIVEC floating point and integer SIMD extensions designed by APPLE, IBM and FREESCALE SEMICONDUCTOR. In the embedded space, ARM offers DSP and multimedia support through their SIMD extensions for multimedia and NEON extensions. Whilst conceptually similar, these different instruction set extensions differ significantly in detail, e.g. in their word and subword size, supported data types, and use of processor registers.

Despite improvements in compiler technology, including automatic vectorisation [Nuzman and Zaks 2008; Nuzman et al. 2011], short-vector instructions offered by the architecture are typically accessed through platform-specific compiler *built-in functions*. This is due to the superior performance of hand-tuned vector code, which often outperforms auto-vectorised code [Mitra et al. 2013].

Author's addresses: S. Manilov and B. Franke, School of Informatics, University of Edinburgh; A. Magrath and C. Andrieu, Cirrus Logic International (UK) Ltd., Westfield House, Edinburgh EH11 2QB.
 1539-9087/2015/01-ART1 $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

Built-in functions, also called *intrinsics*, are functions available for use in C, but their implementation is handled specially in the compiler: the original intrinsic call is directly substituted by a machine instruction. For example, MICROSOFT and INTEL's C/C++ compilers as well as GCC and LLVM implement intrinsics that map directly to the X86 SIMD instructions. The use of intrinsics enables programmers to exploit the underlying instruction set extensions and to increase the efficiency of their programs, but their use inevitably results in non-portable code. Obviously, this seriously restricts the re-use and porting of software components such as libraries, which have been heavily optimised for one particular instruction set extension and where no plain C sources are available.

In this article, which extends an earlier conference paper [Manilov et al. 2015], we develop a novel technique for cross-platform retargeting of code comprising platform-specific intrinsics. The **key idea** is to accept the presence of intrinsics as an opportunity and a source of information, rather than an obstacle. We develop a graph based matching approach, which aims at substituting existing intrinsics with those available on the target machine and possibly additional code providing compatibility. We provide descriptions of intrinsics for a number of different instruction set extensions using a custom description language, covering the syntactic and semantic specification of intrinsics. These descriptions are translated to graph representations by our FREE RIDER tool, which then translates any C program written using one set of intrinsics (e.g. those for an ARM CORTEX-M4 core) to make use of intrinsics of any other platform (e.g. INTEL SSE). Any pair of the available architectures can be used, in either direction. This translation process might also include additional source code transformations such as loop unrolling to account for different SIMD word sizes of the source and target platforms, respectively.

1.1. Motivating Example

Consider the example in Figure 1, which illustrates the steps involved in translating a vector addition loop using intrinsics for an ARM CORTEX-M4 to an INTEL SSE-enabled processor.

In Figure 1a ARM-optimised code is shown, which exploits the `UADD8` intrinsic available on the CORTEX-M4 platform and which provides convenient C-level access to a quad 8-bit unsigned addition instruction implemented in the processor's ISA. Using the `UADD8` intrinsic four pairs of one-byte values are added using a single processor instruction (line 10). To account for this implicit loop unrolling the surrounding loop is incremented in steps of four (in line 9), whilst also enabling 32-bit data accesses (rather than four individual 8-bit accesses). This is achieved by the access macro `PV`, which performs the necessary 32-bit cast operation. The measurable benefit of using the `UADD8` intrinsic in Figure 1a is a speedup of about four over a plain C implementation such as shown in Figure 1b (on a FREESCALE KINETIS K70 implementation of the ARM CORTEX-M4 core). However, higher performance for the platform-specific code comes at a price – the code in Figure 1a is **not portable** and does not work on platforms other than the ARM CORTEX-M4.

Porting of the code in Figure 1a to another platform is hindered by the fact that a plain C version such as shown in Figure 1b is often **not available**. In this situation, the user could (a) **manually derive** the plain C implementation and then try to vectorise this code, either manually or using an auto-vectoriser, or (b) use our FREE RIDER tool and methodology for **automatic retargeting**.

Now consider the automatically retargeted code, optimised for an INTEL processor with SSE extensions, in Figure 1c. It exploits the `_mm_add_epi8` intrinsic, which provides access to an 8-bit addition instruction that operates on two groups of sixteen elements. Using the `_mm_add_epi8` intrinsic sixteen pairs of one-byte values are added in a single processor instruction (line 7). Accordingly, the loop increment has been adjusted to sixteen (line 6), and 128-bit data accesses are provided by the access macros `LV` and `SV`, shown in Figure 1d. Without user intervention platform-specific ARM code has been retargeted to an INTEL platform whilst **retaining the performance benefit** of the original ARM intrinsic. Compared to a plain C baseline (such as the one in Figure 1b) the code in Figure 1c is about ten times faster[1].

[1]Memory access overheads prevent the speedup to reach its ideal value of sixteen.

```c
char A[128], B[128], C[128];

// ... initialize A and B ...

// Packed vector access
#define PV(x) (*((uint32_t*)(&x)))

// Compute loop with UADD8 intrinsic
for (int i = 0; i < 128; i+=4) {
  PV(C[i])=__UADD8(PV(A[i]),PV(B[i]));
}
```

(a) Platform-specific code using the ARM `UADD8` intrinsic. This code cannot be compiled with a compiler, which does not support that intrinsic, and thus cannot be executed on non-ARM platforms.

```c
char A[128], B[128], C[128];

// ... initialize A and B ...

...
...

// Compute loop
for (int i = 0; i < 128; ++i) {
        C[i] = A[i] + B[i];
}
```

(b) Portable, but frequently **unavailable** plain-C implementation. Portable code versions are often not maintained or even dropped from code repositories as platform-specific optimizations are introduced.

```c
char A[128], B[128], C[128];

// ... initialize A and B ...

// Compute loop with Intel SSE
    intrinsic
for (int i = 0; i < 128; i+=16) {
        SV(C[i], _mm_add_epi8(LV(A[i]),
    LV(B[i])));
}
```

(c) Platform-specific code using INTEL `_mm_add_epi8` intrinsic. Conceptually, the code looks similar to (a), but features a larger loop unrolling factor due to the target architecture's wider SIMD word size.

```c

// Load vector
#define LV(x)     (_mm_loadu_si128 ((
    __m128i*)(&x)))

// Store vector
#define SV(x, y) (_mm_storeu_si128 ((
    __m128i*)(&x),y))
```

(d) Auxiliary load/store macros for INTEL SSE vectors complement the code in (c). On the INTEL platform vector accesses require special vector load and store intrinsics, whereas the ARM code in (a) only requires suitable casting.

Fig. 1: Motivating example illustrating the use of the intrinsics to speed up a vector addition loop. The code in Figure 1a is optimised for an ARM CORTEX-M4. This code makes use of the ARM-specific `UADD8` intrinsic and **will not compile** for e.g. an INTEL platform. Equivalent plain-C code as shown in Figure 1b is often **not available**. Figure 1c shows the vector addition loop from Figure 1a **translated** to an INTEL platform, now using the INTEL `_mm_add_epi8` SSE intrinsic. This translation requires not only substitution of the ARM intrinsic, but additional code transformations. These comprise the introduction of suitable **short vector accesses** (Figure 1d), further **loop unrolling** to match the wider SIMD word size of the INTEL architecture and **dead store elimination** of redundant flag setting operations implicitly contained in the original ARM `UADD8` intrinsic, which are not used in this example, but need to be emulated where required.

FREE RIDER does not require a plain C implementation such as the one in Figure 1b, but directly retargets platform-specific code *where no plain C implementation exists*. Translation of intrinsics involves a number of processing steps briefly outlined in Figure 2. We start with the code in Figure 1a, but we do not have access to a plain C implementation such as the one shown in Figure 1b. As a first step of the transformation process the `UADD8` intrinsic is expanded in the internal representation of FREE RIDER – it is essentially expressed as a vector of four additions followed by a vector of four compare-and-set operations. This is shown in Figure 2b and follows closely the specification

The __uadd8 intrinsic returns:

— the addition of the first bytes in each operand, in the first byte of the return value
— the addition of the second bytes in each operand, in the second byte of the return value
— the addition of the third bytes in each operand, in the third byte of the return value
— the addition of the fourth bytes in each operand, in the fourth byte of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If res is the return value, then:

— if res[7:0] ≥ 0x100 then APSR.GE[0] = 1 else 0
— if res[15:8] ≥ 0x100 then APSR.GE[1] = 1 else 0
— if res[23:16] ≥ 0x100 then APSR.GE[2] = 1 else 0
— if res[31:24] ≥ 0x100 then APSR.GE[3] = 1 else 0.

(a) Specification of the ARM CORTEX-M4 __uadd8 intrinsic [ARM Ltd. 2010].

```
// Compute loop –
// Abstract vector form


for (int i = 0; i < 128; i+=4)
{
    vector_4 [j = 0 .. 3]{
        C[i+j] = A[i+j] + B[i+j];
        APSR.GE[j] = (C[i+j] > 0x100) ? 1:0
    }
}
```

(b) Code in abstract vector representation.

```
// Reduced abstract representation
for (int i = 0; i < 128; i += 16) {
    (C[i] = A[i] + B[i]) x 16
}
```

(c) Unnecessary writes to APSR removed. Unroll factor increased.

```
// Compute loop – Intel SSE intrinsic
for (int i = 0; i < 128; i += 16) {
  SV(C[i],_mm_add_epi8(LV(A[i]),LV(B[i])));
}
```

(d) Equivalent INTEL SSE code, which makes use of the _mm_add_epi8 intrinsic.

Fig. 2: Motivating example illustrating the transformation from the use of the UADD8 intrinsic on the ARM CORTEX-M4 core (in Figure 2a) to the use of the INTEL _mm_add_epi8 SSE intrinsic (in Figure 2d). The transformation requires not only substitution of the ARM intrinsic, but additional code transformations, which comprise suitable short vector accesses and further loop unrolling to match the wider SIMD word size of the INTEL architecture. In addition, the overflow checking logic is removed, as the APSR register is not read later in the program, making the writes to it unnecessary.

of the UADD8 intrinsic from Figure 2a. The next step is to analyse which output of the intrinsic is actually used by the program. In the case of the motivating example the result of the addition is later used (for outputting the result, further computations, etc.), but the APSR register is never read. This register exists in the ARM CORTEX-M4 core to set flags indicating different program status - zero result, negative result, overflow (as is the case of UADD8), and others. Since the register is not read, writing to it is a waste of processing resources, so the compare-and-set operations are removed altogether. Another operation performed at this step is to find appropriate target SIMD intrinsic that consists of the remaining core operation – addition. In the case of INTEL SSE this is the _mm_add_epi8 intrinsic, which has internal representation (c = a + b) x 16 - it is a vector of sixteen additions. Since the widths of the vector operations do not match, the loop is unrolled to fit an _mm_add_epi8 operation. This step is shown in Figure 2c. Finally, when the resulting abstract representation matches exactly a target instruction it is replaced by that instruction together with appropriate access macros. The resulting code after retargeting, shown in Figure 2d, matches the INTEL SSE implementation from Figure 1d.

We have implemented this methodology in the FREE RIDER tool and demonstrate its effectiveness using a set of compute-intensive OPENCV computer vision benchmarks [Bradski 2000]. Automatically retargeting these benchmarks from an ARM NEON platform to an INTEL EDISON module

with short-vector SSE4.2 instructions, we achieve on average 96.0% of the performance of manually retargeted and optimised ports. Furthermore, an evaluation against a full-scale robotic application [Meier et al. 2012], which implements the computer vision component of a high-end autopilot for unmanned aerial vehicles (UAV) delivers a speedup of 3.73 over a plain C baseline, when ported from an ARM CORTEX-M4 platform to INTEL EDISON using our methodology.

1.2. Contributions

This paper makes the following contributions:

(1) We develop a novel, automated methodology for retargeting C code containing platform-specific intrinsics, whilst making efficient use of those intrinsics offered by the target platform,
(2) we combine in our approach high-level descriptions of intrinsics, graph based matching and source-level code transformations to account for differences in the SIMD word sizes between machines, and
(3) we evaluate our methodology using compute-intensive OPENCV benchmarks as well as full applications and demonstrate performance levels competitive with manual retargeting efforts.

This article is based on an earlier conference paper [Manilov et al. 2015], which has been extended in following ways: We provide a more complete specification of the FREE RIDER Description Language and the structure of the files generated by FREE RIDER tool (in Section 3.3.1), we provide an additional evaluation of our methodology and retarget the OPENCV benchmarks from INTEL SSE to ARM NEON to demonstrate both directions of translation, and we discuss the impact of our methodology on code size (in Section 4).

1.3. Overview

The remainder of this paper is structured as follows. In Section 2 we briefly introduce the background on compiler intrinsics, target platforms and applications. In Section 3 we present our methodology for retargeting of platform-specific intrinsics involving a high-level description of intrinsics, a graph-based matching algorithm and source-level code transformations. The results of our evaluation on benchmarks and full applications are presented in Section 4, before we establish the context of related work in Section 5. Finally, in Section 6 we summarise and conclude.

2. BACKGROUND

2.1. Target Platforms

The specific target platforms used in our research are the ARM CORTEX-M4 core and INTEL X86 processors with short-vector SSE4.2 instructions (in particular, INTEL EDISON, but our work applies without modification to any X86 architecture that supports SSE4.2) and ARM NEON enabled processors. By providing further target descriptions other platforms such as POWERPC/ALTIVEC could be supported, but this is beyond the scope of this paper.

The CORTEX-M4 processor is specifically developed to address digital signal control markets. It is designed so that it has low power consumption while offering high-efficiency signal processing functionality, provided instruction set extensions accessible through ARM specific intrinsics.

ARM NEON is ARM's SIMD extension that targets more computationally demanding tasks, e.g. video processing, voice recognition, or computer graphics rendering. Architectures that support NEON have higher computational performance compared to the CORTEX-M4 processor and are typically found as application processors within mobile devices such as smartphones or tablets.

INTEL X86 on the other hand includes a huge family of processors, from embedded low-power chips to high-end server CPU's offering a one-size-fits-all instruction set. SSE4.2 is an instruction set extension that allows INTEL X86 processors to execute SIMD instructions on vectors up to 128-bit wide. This allows such processors to be used efficiently for multimedia and graphics processing.

SIMD operations, both for ARM and INTEL, are accessible to the C programmer by means of intrinsic functions. An intrinsic function is not explicitly defined by the programmer, but is provided

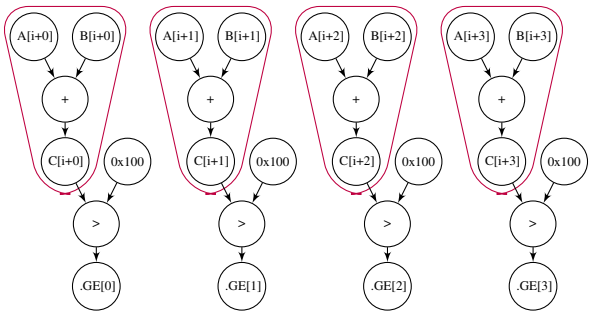

Fig. 3: Graph representation of the __UADD8 intrinsic. Arithmetic operations – circled in red – represent the transferable core of the vector add instruction, whereas the remaining parts implement the necessary comparisons for overflow checking and flag setting. Frequently, the flag setting operations can be eliminated if the flags are not used later on, i.e. they are "dead variables".

(as a built-in function) by the compiler, which replaces an intrinsic function call with a hard-coded sequence of low-level instructions [Batten et al. 2000]. Examples for intrinsic functions are the `UADD8` intrinsic for the ARM CORTEX-M4 processor and the `_mm_add_epi8` intrinsic for the SSE instruction set extension, both of which are part of the motivating example (Figure 1).

In general, the operands of CORTEX-M4 SIMD instructions are 32-bit wide fields. Depending on the instruction each operand is treated as a single 32-bit number, two 16-bit numbers, or four 8-bit numbers. The available operations that can be performed range from simple operations like addition, to very specialised operations like the `SMLALDX` instruction which performs dual 16-bit exchange and multiply with addition of products and 64-bit accumulation. A list of the groups of available operations are: addition (13 instructions), subtraction (13 instructions), sum of absolute differences with or without accumulation (2 instructions), halfword multiply with addition or subtraction, with or without exchange, and with or without accumulation (12 instructions), parallel add and subtract halfwords with exchange (12 instructions), sign-extend byte, with or without addition (4 instructions), half-word saturation (2 instructions), status register based selection (1 instruction).

At the same time, the operands of INTEL SSE4.2 instructions are 128-bit wide fields when they signify a vector, or any other type from the C programming language when they signify vector elements, bitmasks, or shift values. The 128-bit wide fields can be treated as vectors of two, four, eight, or sixteen elements of 64, 32, 16, or 8 bit values, respectively, depending on the instruction. The available operations are not as specialised as those of the CORTEX-M4 SIMD processor, but rather resemble standard processor instructions that operate on vectors instead of single elements.

While there are also miscellaneous utility (e.g. cache control) instructions for the INTEL SSE instruction set we primarily target arithmetic instructions. The integer instructions can be grouped in the following categories: addition (8 instructions), subtraction (8 instructions), sum of absolute differences (1 instruction), halfword multiply with addition (1 instruction), multiplication (5 instructions), maximum, minimum and average (6 instructions), shifts and bitwise operations(22 instructions), comparison (9 instructions). The miscellaneous instructions that are of our interest are shuffle instructions and pack/unpack instructions. They can be used to implement more complicated SIMD operations that include exchanging of vector elements.

Finally, the NEON extension is similar to SSE. Vectors can be either 64-bit or 128-bit wide and can contain signed or unsigned 8-bit, 16-bit, 32-bit, 64-bit integers, or single precision float numbers. The operations that are supported include standard arithmetic and logical operations, comparison operations, memory operations and shuffling operations. The more specialised operations which we do not take into consideration include table lookup and complicated mathematical operations, like reciprocal square-root estimate.

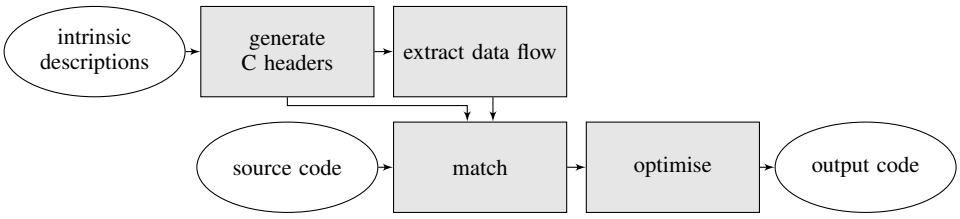

Fig. 4: Stages of execution of the FREE RIDER tool

2.2. Benchmark Kernels and Application

There are no readily available standard benchmarks, which make explicit use of intrinsic functions due to the resulting undesirable restriction to a single platform. Therefore, we use a compute-intensive, open-source application extensively used in the academic, hobby and industrial communities. This application is PX4 [Meier et al. 2012] – a high-end autopilot for unmanned aerial vehicles (UAV) using computer vision algorithms – jointly developed by researchers from the Computer Vision and Geometry, the Autonomous Systems and the Automatic Control Labs at ETH Zurich (Swiss Federal Institute of Technology). PX4 has been developed and optimised for an ARM CORTEX M4F CPU and, in particular, the optical flow module makes extensive use of SIMD intrinsics. Among the most computational intensive functions in the computer vision component are those for the calculation of the Hessian at a pixel location, the average pixel gradient of all horizontal and vertical steps, the SAD distances of subpixel shift of two 8×8 pixel patterns, the SAD of two 8×8 pixel windows, and the pixel flow between two images. We have extracted these functions and use them in isolation (to avoid system benchmarking involving the whole UAV) for our empirical evaluation. A single function (*absdiff*) is written entirely using ARM assembly, for which we provide a portable C implementation.

In addition, we use a number of benchmarks extracted from the popular OPENCV computer vision library [Bradski 2000]. This provides with reference implementations, manually ported and optimised by a independent third party, supporting both ARM and INTEL through platform-specific intrinsics. We use these benchmarks to evaluate the performance and capabilities of our FREE RIDER retargeting tool in comparison to a manual effort.

3. FREE RIDER METHODOLOGY

3.1. Overview

The FREE RIDER tool performs four major transformation steps as shown in the overview diagram in Figure 4: header generation, data-flow extraction, graph matching, and source-level code transformation.

Initially descriptions of the source and target intrinsics are taken as inputs and emulation C header files (in the style of [Zhislina 2014]) are generated. These header files declare and define portable C inline functions for the intrinsics of the source platform. We show in Section 4 that the use of these "emulated" intrinsics results in portable code, but yields a low performance level of only 82% of a plain C implementation of the corresponding functionality on the target platform. This means that emulation of intrinsics through inline C functions provides compatibility, but results in a performance penalty.

In a second step the header files are used as input to the next stage, in which we generate data flow graphs for each intrinsic (see Figure 3 for example). These graphs, annotated with the types of inputs and outputs, serve as intermediate representation. Nodes of the graphs are also annotated with the operations performed, for example vector addition or vector sum reduction. We will use these data flow graphs for graph based pattern matching.

In the next step the C header files, the data flow graphs, and the source code of the program under consideration are all fed to the matching stage of FREE RIDER. It employs a greedy subgraph

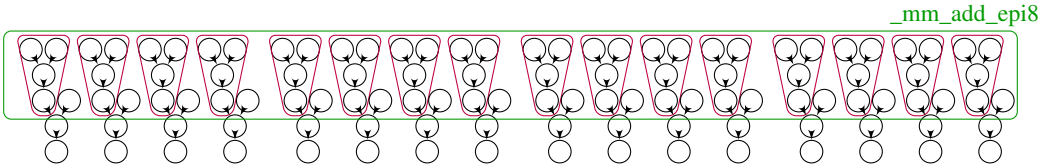

Fig. 5: Matching of four __UADD8 intrinsics (in red) and one _mm_add_epi8 intrinsic (in green) resulting from subgraph isomorphism detection, loop unrolling and dead variable/code elimination. Redundant flag setting computations have no counterpart in SEE and either require additional scalar C code or can be eliminated if there are no further uses of the flags (see also Figure 3).

isomorphism algorithm (similar to [Lipets et al. 2009]) to match the data flow graph of each intrinsic encountered in the source code with data flow graphs of target intrinsics. The graphs of two target intrinsics can connect into a single graph, by connecting the output nodes of one of them to the input nodes of the other by an assignment edge. In this way multiple target intrinsics can cover completely or partially a source intrinsic. For source intrinsics, which can only be partially covered by target intrinsics, scalar C code is generated for the remaining, non-covered parts of the data flow graphs. The output of this stage is C code of the target application with its source intrinsics partly or fully replaced with those of the target platform, wherever possible, and additional plain C code where a direct match is not possible.

Finally, the resulting code after substituting source intrinsics with target intrinsics is further optimised. Checks are performed to remove dead computations and variables (e.g. introduced as part of the flag setting operations in ARM intrinsics, see also Figure 5). Additionally, loop unrolling might be performed to adjust the possibly different SIMD word sizes of the two platforms (also shown in Figure 5).

3.2. Description of Intrinsics

Intrinsics are described by the user in a high-level, human readable format. The description comprises the following items: name of native platform, list of operand names and types, output name and type, and the behaviour (a snippet of restricted C code). An abbreviated example of such a description is provided in Figure 6, which shows the specification of the ARM UADD8 intrinsic.

Operand and result types can be standard C types, or vector types which should also be described in a format comprising the name of the native platform, total size in bits, and the type of a single element (atom type). While this allows for nested types of vectors of vectors, this feature is not used as there are no instructions that operate on such complex types. Thus, the atom type is a standard C type.

Behaviours of intrinsics, i.e. semantic actions, are expressed in a restricted subset of the C language. During generation of header files behaviours are used as the function body of the generated inline function for the intrinsic.

Finally, platform-specific special registers can be described if they are used as part of the side effects of an intrinsic function execution. An example of such register is the ARM CORTEX-M4 APSR (Application Program Status Register), which is used e.g. for indicating arithmetic overflow.

3.2.1. FREE RIDER *Description Language (*FDL*).* Figure 9 shows the FDL grammar in BNF form. As described earlier, the intrinsic definition includes a platform declaration, operand types, result type, and behaviour description.

The architecture, also called platform in our description, is specified by a unique identifying string, for example ARM_CORTEX_M4. As with the following constructs, an example of this is shown in Figure 6.

The types of the operands and the output of the intrinsic are either: 1. default C types, 2. types defined in the stdint.h header file (uint8_t, int16_t, etc.), or 3. user defined types. Most of the time, the last would be used, as there are no standardized vector types in the C programming

```
define intrinsic UADD8
{
    platform ARM_CORTEX_M4
    operands val0:uint8x4_t@32, val1:uint8x4_t@32
    result   res:uint8x4_t@32
    behaviour {
        uint8x4_t res;

        // Load data and cast to prepare for
        // main operation
        uint16_t a0 = (uint16_t)UINT8X4_T_READ(val0,0);
        uint16_t a1 = (uint16_t)UINT8X4_T_READ(val0,1);
        uint16_t a2 = (uint16_t)UINT8X4_T_READ(val0,2);
        uint16_t a3 = (uint16_t)UINT8X4_T_READ(val0,3);

        uint16_t b0 = (uint16_t)UINT8X4_T_READ(val1,0);
        uint16_t b1 = (uint16_t)UINT8X4_T_READ(val1,1);
        uint16_t b2 = (uint16_t)UINT8X4_T_READ(val1,2);
        uint16_t b3 = (uint16_t)UINT8X4_T_READ(val1,3);

        // Perform additions
        // Need 16-bit intermediate results
        // to determine overflow flags
        uint16_t c0 = a0 + b0;
        uint16_t c1 = a1 + b1;
        uint16_t c2 = a2 + b2;
        uint16_t c3 = a3 + b3;

        // Assign results, casting to 8-bit
        UINT8X4_T_WRITE(res,0,(uint8_t)c0);
        UINT8X4_T_WRITE(res,1,(uint8_t)c1);
        UINT8X4_T_WRITE(res,2,(uint8_t)c2);
        UINT8X4_T_WRITE(res,3,(uint8_t)c3);

        // Flag setting, depending on
        // 16-bit intermediate results
        if (c0 >= 0x100) APSR_GE_SET(0); else APSR_GE_RESET(0);
        if (c1 >= 0x100) APSR_GE_SET(1); else APSR_GE_RESET(1);
        if (c2 >= 0x100) APSR_GE_SET(2); else APSR_GE_RESET(2);
        if (c3 >= 0x100) APSR_GE_SET(3); else APSR_GE_RESET(3);

        // Return result
        return res;
    }
}
```

Fig. 6: Example showing the high-level description of the ARM `UADD8` intrinsic.

language, however the other two options are necessary for supporting particular operations e.g. operations involving accumulator variables. Alignment of the variables can be specified, by appending `@<align-size>` after the types.

The descriptions of the intrinsics are provided in a simplified code in the C programming language. In our research we followed the following guidelines to writing the behaviours.

Behaviour descriptions should begin with reading of the separate elements of each of the argument vectors into separate variables. The names of these variables should be `a0`, `a1`, etc. for the elements of `val0`; `b0`, `b1`, etc. for the elements of `val1` and so on. Operations should then be performed on each pairs of elements consequtively, storing the results in a new variable every time. When all operations are performed, the results need to be stored in the different elements of the

```
define register APSR
{
    platform ARM_CORTEX_M4
    width    32
    fields   { N[31], Z[30], C[29], V[28], Q[27], GE[19..16] }
}
```

Fig. 7: Example showing the high-level description of the ARM APSR register.

output vector res. Afterwards flag setting can be performed if needed and a single return statement should be present.

While this is a rather restrictive way of representing the behaviours of the intrinsics, there are several advantages to it. Firstly, it is relatively easy to analyse the resulting C code and build a control-flow graph from it. Furthermore, due to the repetition of operations for each element, the graph can be manipulated to a control-flow graph of vector operations, rather than single element operations. At the same time the code representing the behaviour can be directly compiled by using any standard C compiler and thus no further modification of the code is needed to achieve portability. Furthermore, the format allows many default optimizations to be performed, which would eliminate the performance overhead of repetative code. Finally, since the behaviour description is very explicit and does not include obscure statements involving multiple operations or potentially confusing pointer operations then it is easier to verify the correctness just by looking at the code and before using more detailed and robust verification mechanisms.

It is important to note that by no means these restrictions are imposed by our methodology, but are merely guidelines. The level of complication that is allowed in the behaviour description is limited only by the C language analysis sub-module of FREE RIDER, which is independent of the rest of the system and can be easily improved or replaced.

Moving on to defining a custom register, in order to do so one needs to specify its platform, bit width, and fields, each corresponding to a sequence of bits within the register. The platform declaration requirement is the same as for intrinsics and the bit width is a valid integer - usually a power of 2. Fields are specified as a list of named elements, each followed by its bit position within the register. If a field spans multiple bits then this can be specified by $\texttt{field}[\texttt{bit}_{\texttt{start}} \ldots \texttt{bit}_{\texttt{end}}]$. Figure 7 shows an example description of a register.

To conclude the section on FDL, platform specific vector types are defined using the type declaration construct. Users need to specify the platform, the total bit width of the vector, the type of a single element from it, and the mapping from bit ranges to vector elements. The platform declaration agrees with that of intrinsics and registers. The total bit width of the vector is the amount of memory that is required to store all of its elements. As an example, a vector of 8 elements, 8-bit wide each, has a total bit width of 64 bits.

The type of the single element of a vector type is called atom type in the description file. It must be one of the default types of the C programming language or a type defined in the stdint.h header file.

Finally, in the mapping field, one needs to specify the vector width as w in y[w]. It should agree with the total bit width divided by the width of the atom type. Each element k is mapped by a declaration of the form

$$\texttt{x}[\texttt{start}_{\texttt{x}} \ldots \texttt{end}_{\texttt{x}}] \rightarrow \texttt{y}[\texttt{k}][\texttt{start}_{\texttt{y}} \ldots \texttt{end}_{\texttt{y}}]$$

where an index $= 0$ represents the least significant bit. This allows for conversions of endianness and elements split over multiple locations. Figure 8 shows an example of a vector type definition.

3.3. Generation of C Header Files

After the intrinsic descriptions are provided they are used to generate one C header file per platform. Definitions of the custom types are output first together with macro functions to read and write

```
define type uint8x4_t
{
    platform ARM_CORTEX_M4
    bitwidth 32
    atomtype uint8_t
    mapping (x:y[4]) {
        x[7..0]     -> y[0][7..0]
        x[15..8]    -> y[1][7..0]
        x[23..16]   -> y[2][7..0]
        x[31..24]   -> y[3][7..0]
    }
}
```

Fig. 8: Example showing the high-level description of the ARM type for a vector of 4, byte-sized unsigned integers.

```
<fdl-file> ::= ("define" (<intrinsic> | <register> | <typedef>))*

<intrinsic> ::= "intrinsic" <name> "{"
                    "platform" <name>
                    "operands" <operands>
                    "result" <variable>
                    "behaviour" <behaviour>
                "}"
<operands> ::= <variable> ("," <variable>)*
<variable> ::= <name> ":" <name> "@" <number>
<name>     ::= "[a-zA-Z_][a-zA-Z0-9_]*"
<number>   ::= "[1-9][0-9]*"

<register> ::= "register" <name> "{"
                    "platform" <name>
                    "width" <number>
                    ("fields" "{" <field> ("," <field>)* "}")*
               "}"
<field> ::= <name> "[" <number> (".." <number>)? "]"

<typedef> ::= "type" <name> "{"
                    "platform" <name>
                    "bitwidth" <number>
                    "atomtype" <name>
                    "mapping" "(" "x" ":" "y" "[" <number> "]" ")" "{"
                          <bit-mapping>
                          <bit-mapping>*
                    "}"
              "}"
<bit-mapping> ::= "x" "[" <number> ".." <number> "]" "->"
                  "y" "[" <number> "]" "[" <number> ".." <number> "]"
```

Fig. 9: The formal grammar of the FREE RIDER Description Language in BNF. A `<behaviour>` is a plain C function body and parsed using an embedded C parser.

```
typedef struct {
    uint8_t _[4];
} uint8x4_t;

#define UINT8X4_T_READ(v, i) (v._[i])
#define UINT8X4_T_WRITE(v, i, x) (v._[i] = x)
```

Fig. 10: Example showing the representation of the ARM type for a vector of 4, byte-sized unsigned integers. This representation is generated by FREE RIDER.

separate elements of a vector. Then, special registers are implemented as bit field structures and access macros for them are generated.

After this supporting code has been created the implementation of the intrinsic functions as inline C functions is generated. Signatures are generated using the type information for the operands and the result, and the body of the functions are copied from the behaviour descriptions.

Using the generated header files, a data flow graph is derived for each intrinsic using standard data flow analysis techniques. These data flow graphs, together with the input/output type information of each intrinsic are used in the matching stage as descriptions of the intrinsics.

3.3.1. Structure of the generated files. Each generated header file describes all the information FREE RIDER has about the corresponding platform. Types are implemented as C `structs` that contain an array representing the vector. The type of the array is the same as the atom type in the definition, while its size is the total bitsize of the vector divided by the bitsize of an individual element.

In order to access the elements of a vector type `READ` and `WRITE` access macros are generated as part of the definition. These macros implement the switch of endiannes if there is one, and splitting and combining of fragmented elements, if there are any. In their simplest form, they just use the elements of the underlying C array as the elements of the vector. Figure 10 shows an example of a generated type.

As mentioned earlier, platform specific registers are implemented as bitfields, with each flag given its required number of bits. Access macros are generated for each flag, so that they set and reset in an implementation independent manner. Flags that cover multiple bits of the register have more complicated access macros, that take the index of the bit to be modified as an argument. Figure 11 shows an example of a generated register together with its access macros.

Finally, the C functions implementing the intrinsics are generated. The return type is copied from the type of the `result` variable in the description, and the arguments have the types and names of the specified operands. The body of the function is directly pasted from the behaviour block in the description. Figure 12 shows an example of a generated function emulating an intrinsic, with the body of the function omitted, as it is exactly the same as the one shown in 6.

3.4. Graph Matching and Source-level Transformation

The process of matching intrinsics is outlined in Figure 13. Given the data flow graph of a source intrinsic, an intrinsic from the target platform is searched for using the VF2 graph-subgraph isomorphism algorithm and library described in [Cordella et al. 2004].

Since graph matching is an NP-complete problem, but one with great importance to many areas in computer science, there are multiple approaches to efficiently find a solution. The one we are hinging on employs a deterministic algorithm and a State Space Representation (SSR) of the problem, to iteratively build a solution. There are five feasibility rules that are applied to pairs of nodes from the graphs being matched and help prune the search tree quickly. A total order relation guarantees that the applied Depth-First Search (DFS) does not reach the same state of the SSR twice, but via different paths. All of these techniques add up to an efficient graph-subgraph isomorphism algorithm. For more details, please refer to the original article [Cordella et al. 2004].

```
typedef struct {
    int _  : 23;
    int GE :  4;
    int Q  :  1;
    int V  :  1;
    int C  :  1;
    int Z  :  1;
    int N  :  1;
} _APSR;

_APSR APSR;

#define APSR_N_SET() (APSR.N = 1)
#define APSR_Z_SET() (APSR.Z = 1)
#define APSR_C_SET() (APSR.C = 1)
#define APSR_V_SET() (APSR.V = 1)
#define APSR_Q_SET() (APSR.Q = 1)
#define APSR_GE_SET(i) (APSR.GE |= (1 << i))

#define APSR_N_RESET() (APSR.N = 0)
#define APSR_Z_RESET() (APSR.Z = 0)
#define APSR_C_RESET() (APSR.C = 0)
#define APSR_V_RESET() (APSR.V = 0)
#define APSR_Q_RESET() (APSR.Q = 0)
#define APSR_GE_RESET(i) (APSR.GE &= ((1 << 4) - (1 << i) - 1))
```

Fig. 11: Example showing the representation of the ARM `APSR` register. This representation and access macros are generated by FREE RIDER.

```
uint8x4_t UADD8(uint8x4_t val0, uint8x4_t val1) {
    ...
}
```

Fig. 12: Example showing the implementation of the ARM `UADD8` intrinsic. The body of the function is equivalent with the behaviour block in Figure 6.

The overhead that is added to the compilation time by using this algorithm is immeasurably small for all our test cases. Its authors evaluate that it can match graphs up to 2000 nodes in under a tenth of a second. Since the graphs that we use to represent the intrinsics are quite small in comparison (under 20 nodes for the most complicated intrinsics) we are not concerned about a potential added overhead.

When a structural and operational match is found, the type information of the found operation is compared with the type information at the source location of the match. If the vector widths of the operands and the result match, the matching part of the graph is directly replaced with the found intrinsic and the process is repeated until no further unmatched parts of the source dataflow graph can be found or there are no more target operations that can cover the remaining graph.

There are two reasons for possible mismatches of vector widths: (a) The target intrinsic is too narrow (i.e. it contains fewer operands than the corresponding source intrinsic), or (b) it is too wide (i.e. it contains more operands than the source intrinsic). In the first case, the target intrinsic can be invoked as many times as it takes to match the width of the source vector (e.g. using four 4-element additions to implement one 16-element addition). In the second case, loop unrolling is required in order to enable a match (e.g. unroll a loop containing a 4-element addition in order to use a 16-element addition to implement four 4-element additions from four iterations).

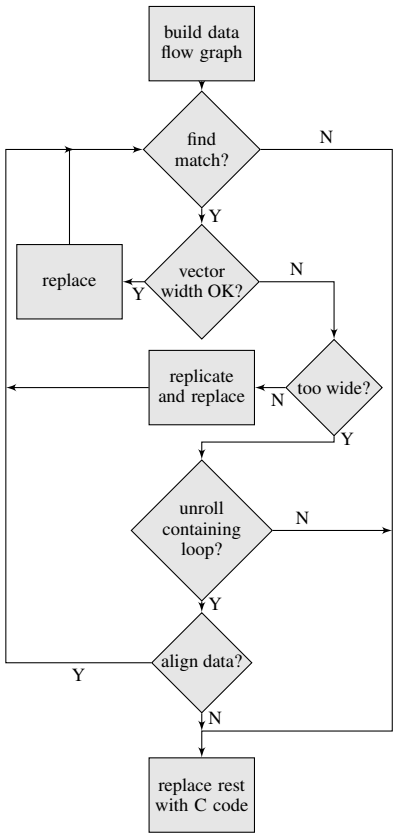

Fig. 13: High-level algorithm for matching source and target intrinsics including loop unrolling for adjusting different SIMD word sizes and aligmnents.

In case loop unrolling is required it is performed alongside further data alignment. The latter might be necessary if arrays are not accessed in order, but some elements are skipped over. If either the unrolling or the data alignment steps fails, the matching fails and the default replacement with plain C code is performed to ensure correctness of the result. However, if they succeed the whole process is repeated again, until no further matches can be found.

Substitution of intrinsics as well as optimisation (loop unrolling, alignment, dead code/variable elimination) are implemented as source-level transformations. This means that C code enhanced with target intrinsics is generated, which can be compiled with the standard compiler for the target platform.

3.5. Limitations

As described in Section 2, intrinsics available for one platform cannot always be expressed by intrinsics available on another platform, or even in standard C code. Examples of such intrinsics are cache-ability and synchronisation operations. We limit our approach to standard data processing operations and do not consider complex intrinsics whose behaviours cannot be expressed in C.

Benchmark	Summary
calib	Calibrates a camera given a sequence of images.
bgfg	Split of background and foreground of video.
edge	Canny edge detection on an image.
align	Automatic alignment of an image.
polar	Polar transformation on a video.
segm	Automatic segmentation of objects in a video.
stitch	Stitching multiple images into a mosaic.
vstab	Automatic stabilisation of a video.

Table I: OPENCV benchmark applications.

4. EMPIRICAL EVALUATION

4.1. Evaluation Methodology

For our evaluation we have applied the FREE RIDER methodology to automatically retarget ARM-specific benchmarks and applications to an INTEL SSE enabled platform. The system used for performance evaluation is an INTEL EDISON module running at 500 MHz. The available physical memory is 1GB and the operating system is YOCTO LINUX, kernel version 3.10.17. All the benchmarks run on a single processor core.

In addition, we have retargetted INTEL SSE-optimised applications to an ARM NEON enabled platform. The system that we used for this part of the evaluation is a PANDABOARD device running at 1.2 GHz. The available physical memory is 1GB and the operating system is UBUNTU LINUX, version 12.04. For this part, all benchmarks run on a single processor core too.

Performance is measured by using the UNIX program `time` to retrieve the total execution time of each benchmark. This is repeated up to 100 times and the reported times are recorded in a log. This log is then analysed to verify that the values roughly fit in a normal distribution, supporting the model that the difference in the measurements is just gaussian white noise. The average of all runtimes per benchmark is considered to be the representative runtime for that benchmark. Error bounds are not included, as they are too small to plot (less than 0.5 % for all benchmarks).

The OPENCV benchmarks are selected from the default sample programs that are provided with the OPENCV library, version `3.0.0-beta`. We have prepared them by removing the user interaction and substituting it with command line arguments and `stdout` messages. Our eight benchmark programs each contain a significant part ($>$ 10% of CPU time) executing vectorisable code. This was computed by compiling OPENCV with and without the included manual vector optimisations and comparing the runtimes of each benchmark for the two cases. Each of the benchmarks makes heavy use of functions provided by the OPENCV library. Many of these OPENCV functions have been manually ported and optimised for different target architectures, including ARM and INTEL.

For the first part of our evaluation we take the ARM ports of these functions, automatically retarget them to INTEL and then evaluate the performance of these automatically retargeted codes in comparison to the manual INTEL port provided with OPENCV. For the second part, we do the opposite: we take the INTEL ports of the functions, automatically retarget them to ARM-optimised code and then evaluate their performance in comparison to the manual ARM port provided with OPENCV.

Table I provides descriptions of the benchmarks. All of these benchmarks are real-world examples of programs from the computer vision domain.

The SSE intrinsics that we implemented include 14 arithmetic operations, 15 logical and comparison operations, 16 memory and initialisation operations, 4 conversion operations, and 8 shuffling operations. These correspond roughly to the NEON intrinsics that we implemented which include 21 arithmetic operations, 13 logical and comparison operations, 19 memory and initialisation operations, 12 conversion operations, and 8 shuffling operations. The greatest discrepancy is in the

amount of conversion operations. There are more NEON conversion operations because NEON allows for two vector widths (64- and 128-bit), and can convert between them, adding 8 operations.

The arithmetic operations comprise different versions of additions, subtractions, and multiplications, in addition to a single operation for division, maximum, minimum, and square root. The logical operations comprise different versions of logical ands, ors, xors, and shifts, whereas the comparison operations are comparisons for equality and strict inequalities.

The implemented memory operations comprise different loads and stores, whereas the initialisation operations are generating vectors, all depending on the data type of the given argument. The conversion operations convert the elements of the vector between different datatypes. Finally, the shuffling operations comprise instructions that reorder the elements of a vector in different ways.

On the target INTEL system we use the CLANG/LLVM compiler to produce executable binaries. For comparison, we also have conducted an experiment where plain C sources are presented to the compiler for auto-vectorisation of the PX4 application.

When targeting ARM we used CLANG/LLVM for compilation again, this time however cross-compiling from an INTEL host.

4.2. Benchmark Performance Results

Figure 14 shows the results from running the automatically ported OPENCV ARM benchmarks on the INTEL evaluation system. While "native" SSE code delivers a speedup of 1.26 over a plain C baseline, the FREE RIDER ports approach this performance delivering a speedup of 1.21. On average, FREE RIDER produced code that delivers a performance of about 96% of manually ported and optimised INTEL SSE implementation. For every single benchmark the FREE RIDER port outperforms its plain C baseline, despite attempts of the compiler to auto-vectorise this plain C code. Even for the worst performing benchmark (`polar`) the automatically retargeted implementation outperforms the plain C baseline and delivers a speedup just about 6% lower than that of the manual port.

Simlarly, Figure 15 shows the results from running the automatically ported OPENCV INTEL benchmarks on the ARM evaluation system. The manually optimized NEON code delivers a speedup of 1.31 over a plain C baseline, the FREE RIDER-ported applications perform comparatively good, with an average speedup of 1.26. Thus, FREE RIDER produced code that again delivers a performance of about 96% of the manually ported and optimised ARM NEON implementation.

Closer inspection of the generated code revealed that the remaining performance differences between the manual and the auto-generated ports are mainly due to additional code restructuring performed by the expert programmers and optimisations to the scalar code surrounding the intrinsics.

For the evaluation of the impact on code size we compared the executable segments of the compiled OPENCV library before and after optimization. We observed that the difference between the two versions was negligible, with the optimized version less than 1% larger than before. The reason for this relatively small impact on code size is that the volume of code transformation is miniscule in comparison to the whole code base of the library. At the same time the resulting optimized code is contained in performance hotspots and contributes substantially to the overall runtime performance.

4.3. Application Performance Results

Next we focus on porting a larger application – the PX4 computer vision system – from ARM CORTEX-M4 to INTEL SSE. Figure 16 shows a set of results from running the target application in different configurations on the INTEL evaluation system.

The first configuration is a plain C baseline derived from the application's original sources. All further results are relative to this baseline configuration. As shown in Figure 16 the compiler fails to automatically vectorise this application, hence performance levels with and without compiler vectorisation are the same.

If ARM intrinsics are emulated by inline C functions, following an approach outlined in [Zhislina 2014], performance suffers resulting in a drop of execution speed of about 18%. Again, the LLVM compiler's auto-vectoriser fails to exploit any vectorisation opportunities.

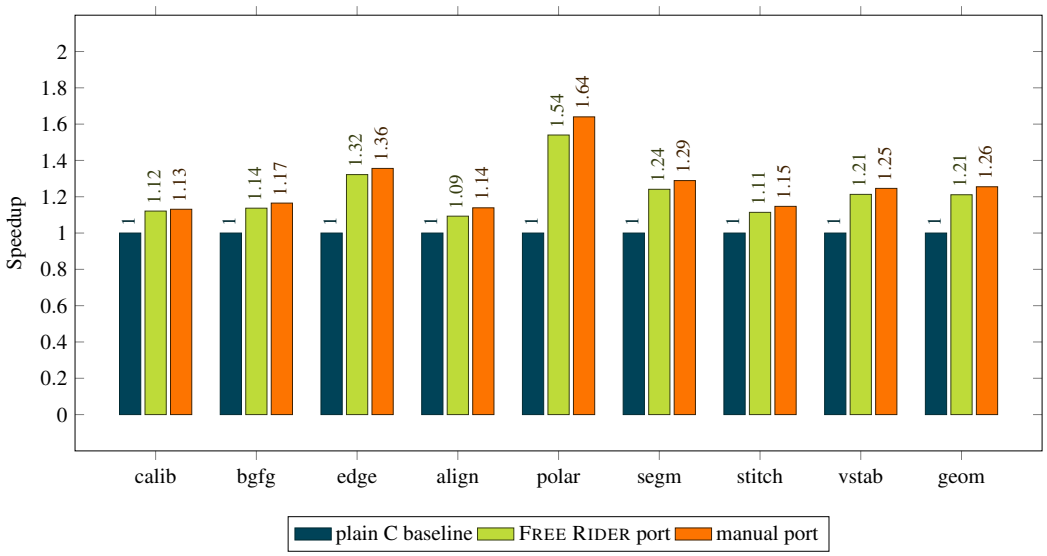

Fig. 14: These OPENCV applications have been ported automatically from ARM NEON to INTEL SSE. Each bar reports the speedup per benchmark over a plain C baseline. In each case, the automatically retargeted version produced by FREE RIDER outperforms the plain C baseline. In fact, the performance of the auto-generated ports approaches that of the manually ported OPENCV applications, tuned extensively by the OPENCV community developers. On average, we achieve 96% of the speedup of a manual port, without paying the cost involved in manual code rewriting.

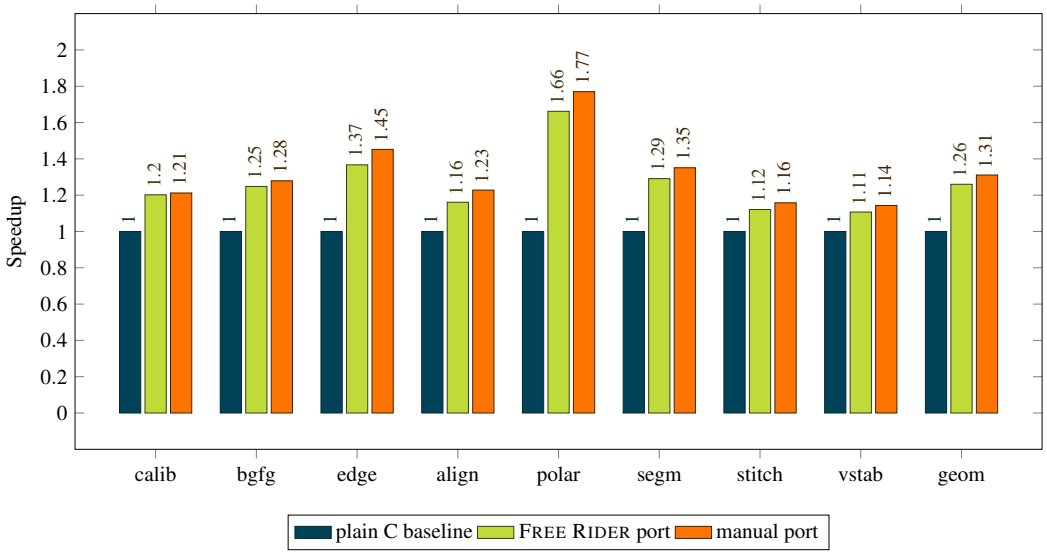

Fig. 15: The same OPENCV applications as in Figure 14, this time ported automatically from INTEL SSE to ARM NEON. Expectedly, the results are similar to those shown in the previous Figure. We still achieve 96% of the speedup of a manual port, without paying the cost involved in manual code rewriting.

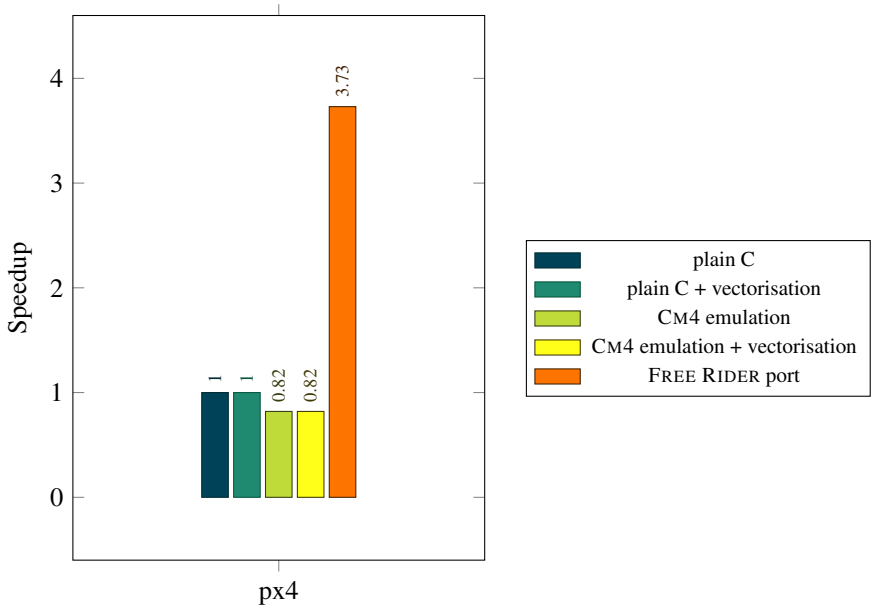

Fig. 16: Relative performance of the ported PX4 application on the INTEL platform relative to a plain C baseline. Compiler vectorisation is not effective as it fails to detect and exploit vectorisation opportunities. Emulation of ARM intrinsics through inline functions, implemented in plain C, eases portability, but degrades performance with and without compiler vectorisation efforts. The automatically generated FREE RIDER port is capable of exploiting SIMD parallelism on the INTEL platform and delivers an almost four-fold speedup over the plain C baseline.

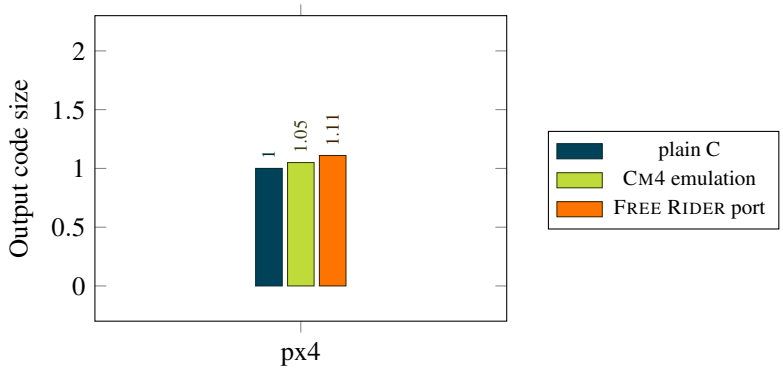

Fig. 17: Relative codesize of the ported PX4 application on the INTEL platform relative to a plain C baseline. In this example, the baseline application is optimized for code size and the emulation version is 5% larger. This is expected, as instructions which would include intrinsics in the original program are implemented as inline functions in the emulated one. The ported program is even larger, at 11% code size increase from the baseline. This is the case, because the translation of the intrinsics was produced after an aggressive transformation that results in multiple target intrinsics. This size growth is justified by the performance benefit discussed earlier.

Intrinsic	Frequency
__USADA8	36
__UHADD8	24
__UADD8	3
__USAD8	2
vld*	50
vadd*	38
vsh*	35
vdupq_n_*	27
vget_*	26
vmov*	26
vst*	24
vqmov*	22
vand*	16
vmul*	16
vcombine_*	15
vceq*/vcgt*/vcge*/vlt*/vle*	8
vsub*	7
vmin*/vmax*/vabs*	5
vqdmulhq_s16	4
vmla*	4
vbslq_f32	3
vzip_s16	2
vsqrt*	2

Table II: Frequency of occurrence of ARM CORTEX-M4 and ARM NEON intrinsics in our benchmarks and application. Each line represents a single or multiple intrinsics. In the latter case the name ends with an asterisk to indicate that there are different variants available. The number of multiple intrinsics per line varies from 2 to 5.

In contrast, the port produced by FREE RIDER substantially outperforms the baseline implementation. The reason for this is that even though FREE RIDER fails to exploit some optimisation opportunities due to irregularity of data accesses, it manages to vectorise the code in the most critical loop of the program and achieves a 3.73 performance speedup. It clearly demonstrates that FREE RIDER is capable of exploiting the source platform's intrinsics and mapping them onto corresponding intrinsics of the target platform. For the INTEL target platform used in this study this is close to the ideal four-fold speedup attainable using four-way SIMD processing.

Finally, Figure 17 illustrates how code size of the program is affected by the transformation performed by FREE RIDER tool. The net effect is a 11% increase in code size, while performance increases by 273% over a plain C implementation. The increase in code size can be largely attributed to the source level transformations applied to the programme, in particular loop unrolling.

4.4. Coverage and Frequency of Intrinsics

Tables II and III list those intrinsics that were encountered in the translation process of the benchmarks (ARM NEON to INTEL SSE) and the larger application (ARM CORTEX-M4 to INTEL SSE). In addition, for each intrinsic we list its frequency of occurrence in the benchmark sources. The first part of each table lists the intrinsics involved in the translation of the target application, while the second part lists the intrinsics involved in the translation of the benchmarks.

Some of the lines in the table represent multiple intrinsics, which is indicated by the names ending in an asterisk. An example is the _mm_add* line from the SSE table, which represents four intrinsics: _mm_add_epi16, _mm_add_epi32, _mm_add_ps, and _mm_adds_epi16.

Intrinsic	Frequency
_mm_srli_epi32	40
_mm_add_epi16	40
_mm_loadu_si128	34
_mm_store_si128	27
_mm_and_si128	24
_mm_set1_epi32	22
_mm_sub_epi16	20
_mm_abs_epi16	20
_mm_unpackhi_epi8	20
_mm_unpacklo_epi8	20
_mm_add_epi8	7
_mm_or_si128	6
_mm_load*	66
_mm_sll/sra/srl*	51
_mm_pack/unpack*	43
_mm_store*	40
_mm_add*	39
_mm_mul*	30
_mm_set*	27
_mm_and*	17
_mm_xor*	17
_mm_cmp*	16
_mm_cvt*	13
_mm_sub*	11
_mm_andnot_si128	7
_mm_sqrt_ps*	2
_mm_min/max_ps	2
_mm_div*	1
_mm_or_si128	1
_mm_movemask_epi8	1

Table III: Frequency of occurrence of INTEL SSE intrinsics in the retargeted benchmarks and application. Asterisks are used similarly to Table II. The first part of the table summarises statistics for the automatic translation of the target application, whereas the second part summarises statistics for the benchmarks.

Also, some of the lines represent a group of instructions separated by a forward slash, for example `vceq*/vcgt*/vcge*/vlt*/vle*`.

We note a couple of interesting observations in Tables II and III. Firstly, there are a lot more occurrences of INTEL intrinsics when compared to ARM intrinsics. This is because we were targeting INTEL SSE as a destination platform during our experiments and a single source intrinsic might be mapped to one or more destination intrinsics, so we end up with more target intrinsics than source intrinsics after the translation. This is more pronounced in the translation between ARM CORTEX-M4 and INTEL SSE compared to the translation between ARM NEON and INTEL SSE since the first pair of platforms is more dissimilar (as explained in Section 2.1) than the second. Secondly, the tables show that FREE RIDER is capable to cover wide range of intrinsics, which enable us to automatically process not only isolated benchmarks, but complex real-world applications resulting in performance levels approaching those of manual retargeting and optimisation.

5. RELATED WORK

Handling of intrinsic functions by the compiler has found little attention in the academic community, possibly due to their normally straight-forward, but target- and compiler-specific implementation. Among the few publications dealing with various aspects related to intrinsic functions are the following.

Compilation for multimedia instructions has been an active research area for over a decade [Sreraman and Govindarajan 2000; Krall and Lelait 2000; Pokam et al. 2004; Jiang et al. 2005; Tenllado et al. 2005; Guelton 2010]. Krall and Lelait [Krall and Lelait 2000] describe basic compilation techniques for multimedia processors. They compare classical vectorisation, borrowed from the age of the vector supercomputers, to using loop unrolling for vectorisation. The mentioned classical vectorisation employs a dependency analysis and might fail if the operations within the loop are not vectorisable. Loop unrolling is more likely to succeed, as the operations of consecutive loop iterations are the same – thus vectorisable. The only reason for failure there might be loop carried dependencies. The authors also explore the problem of unaligned memory accesses. FREE RIDER allows alignment to be specified in the description of architectures and honours it during translation.

Pokam et al. propose SWARP [Pokam et al. 2004] – a retargetable preprocessor for multimedia instructions that is extendable by the user. Their work allows taking advantage of vector operations, without the programmer specifying that intention in the source code, i.e. the input provided to SWARP is plain C and it generates C code, which uses SIMD extensions. A flexible idiom recognition phase eases the retargeting of the system to new machines without changing SWARP itself. Our approach is different in that we are retargeting platform-specific intrinsics. We leverage the expertise already invested in optimising the application to one platform and try to maintain this information when translating to another platform. The idiom recognition is replaced by our matching phase, and flexibility is achieved by the intrinsic description language, which is used to describe different targets.

Similar approaches, all operating on plain C input and trying to extract superword level parallelism within an optimising or vectorising compiler are described in [Tenllado et al. 2005; Sreraman and Govindarajan 2000; Jiang et al. 2005; Guelton 2010]. Whilst these techniques are useful for the initial identification of vectorisation opportunities in C code, they fail to process applications, which have already been vectorised for a particular platform using intrinsics.

A graph based instruction selection technique has been developed in [Murray and Franke 2012], where the compiler targets automatically generated instruction set extensions, where instruction patterns are not tree shaped, but highly irregular and sometimes larger (up to 12 inputs and 8 outputs) than typical multimedia instructions. The graph pattern matching approach used in this article is somewhat comparable to that in [Murray and Franke 2012], however, the purpose of our work is to aid the user retargeting an application optimised for a platform other than the current target platform, whereas graph pattern matching is used in [Murray and Franke 2012] to match highly idiosyncratic instructions.

Modelling of instruction semantics in ADL processor descriptions for C compiler retargeting has been presented in [Ceng et al. 2006]. The focus of this work is more on generating a basic compiler using an architecture description, rather than retargeting of existing, optimised code.

Implementation of intrinsic functions for a DSP compiler is subject of [Batten et al. 2000]. This article proposes and implements a new approach to intrinsic functions where the programmer targets a compiler's intermediate representation rather than the assembly language of a particular processor.

A general introduction to intrinsics for vector processing in the GCC compiler is provided in [Koharchik and Jones 2012].

Possibly most relevant to the work presented in this article is [Zhislina 2014], where a set of hand-coded inline functions compatible with ARM NEON intrinsics is provided for an INTEL platform with SSE. The result is a similar "emulation" layer providing portability for a particular combination of intrinsics (ARM NEON to INTEL SSE), but unlike FREE RIDER this is not automated and

retargetable to any platform, but the result of a major manual implementation effort for one specific pair of platforms.

6. SUMMARY & CONCLUSIONS

In this article we have developed a new methodology for retargeting platform-specific intrinsics from one platform to another. We use a description language to specify signatures and semantics of intrinsics of both platforms. These descriptions are processed by our FREE RIDER tool, which performs subgraph isomorphism checking to substitute one set of intrinsics with one or more intrinsics of the target platform, plus additional scalar code wherever needed. In addition, FREE RIDER performs source-level loop unrolling in order to account for differences in SIMD word sizes and alignment, and dead variable/code elimination to remove artefacts introduced by the substitution of intrinsics. We have evaluated our methodology by automatically porting OPENCV benchmarks optimised for ARM NEON and a compute-intensive application optimised for the ARM CORTEX-M4 processor to SSE4.2 enabled INTEL X86 processors (and vice-versa). We demonstrate that FREE RIDER can take advantage of intrinsics, and that automatically retargeted code delivers performance levels comparable to manually optimised code for the target platform. We achieve a speedup of up to 3.73 over a plain C baseline on an INTEL EDISON module for the target application, and on average 96.0% of the speedup of manually ported and optimised versions of the benchmarks.

REFERENCES

ARM Ltd. 2010. *Cortex^{TM}-M4 Devices Generic User Guide*. ARM Ltd. http://http://infocenter.arm.com

D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy. 2000. A new approach to DSP intrinsic functions. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*. 10 pp. vol.1–. DOI: http://dx.doi.org/10.1109/HICSS.2000.926967

G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

Jianjiang Ceng, Weihua Sheng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. 2006. Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting. *Journal of VLSI signal processing systems for signal, image and video technology* 43, 2-3 (2006), 235–246. DOI: http://dx.doi.org/10.1007/s11265-006-7273-3

L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26, 10 (Oct 2004), 1367–1372. DOI: http://dx.doi.org/10.1109/TPAMI.2004.75

Serge Guelton. 2010. SAC: An Efficient Retargetable Source-to-Source Compiler for Multimedia Instruction Sets. (2010).

Weihua Jiang, Chao Mei, Bo Huang, Jianhui Li, Jiahua Zhu, Binyu Zang, and Chuanqi Zhu. 2005. Boosting the Performance of Multimedia Applications Using SIMD Instructions. In *Compiler Construction*, Rastislav Bodik (Ed.). Lecture Notes in Computer Science, Vol. 3443. Springer Berlin Heidelberg, 59–75. DOI: http://dx.doi.org/10.1007/978-3-540-31985-6_5

George Koharchik and Kathy Jones. 2012. An Introduction to GCC Compiler Intrinsics in Vector Processing. Linux Journal. (September 2012). http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing

Andreas Krall and Sylvain Lelait. 2000. Compilation Techniques for Multimedia Processors. *Int. J. Parallel Program.* 28, 4 (Aug. 2000), 347–361. DOI: http://dx.doi.org/10.1023/A:1007507005174

V. Lipets, N. Vanetik, and E. Gudes. 2009. Subsea: an efficient heuristic algorithm for subgraph isomorphism. *Data Mining and Knowledge Discovery* 19, 3 (2009), 320–350. DOI: http://dx.doi.org/10.1007/s10618-009-0132-7

Stanislav Manilov, Björn Franke, Anthony Magrath, and Cedric Andrieu. 2015. Free Rider: A Tool for Retargeting Platform-Specific Intrinsic Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded*

Systems 2015 CD-ROM (LCTES'15). ACM, New York, NY, USA, Article 5, 10 pages. DOI: http://dx.doi.org/10.1145/2670529.2754962

Lorenz Meier, Petri Tanskanen, Lionel Heng, Gim Hee Lee, Friedrich Fraundorfer, and Marc Pollefeys. 2012. PIXHAWK: A micro aerial vehicle design for autonomous flight using onboard computer vision. *Autonomous Robots* (2012), 1–19. http://dx.doi.org/10.1007/s10514-012-9281-4 10.1007/s10514-012-9281-4.

Gaurav Mitra, Beau Johnston, Alistair P. Rendell, Eric McCreath, and Jun Zhou. 2013. Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW '13)*. IEEE Computer Society, Washington, DC, USA, 1107–1116. DOI: http://dx.doi.org/10.1109/IPDPSW.2013.207

Alastair Murray and Björn Franke. 2012. Compiling for Automatically Generated Instruction Set Extensions. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 13–22. DOI: http://dx.doi.org/10.1145/2259016.2259019

Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 151–160. http://dl.acm.org/citation.cfm?id=2190025.2190062

Dorit Nuzman and Ayal Zaks. 2008. Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 2–11. DOI: http://dx.doi.org/10.1145/1454115.1454119

Gilles Pokam, Stéphane Bihan, Julien Simonnet, and François Bodin. 2004. SWARP: a retargetable preprocessor for multimedia instructions. *Concurrency and Computation: Practice and Experience* 16, 2-3 (2004), 303–318.

N. Sreraman and R. Govindarajan. 2000. A Vectorizing Compiler for Multimedia Extensions. *Int. J. Parallel Program.* 28, 4 (Aug. 2000), 363–400. DOI: http://dx.doi.org/10.1023/A:1007559022013

Christian Tenllado, Luis Piñuel, Manuel Prieto, Francisco Tirado, and F. Catthoor. 2005. Improving Superword Level Parallelism Support in Modern Compilers. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '05)*. ACM, New York, NY, USA, 303–308. DOI: http://dx.doi.org/10.1145/1084834.1084909

Victoria Zhislina. 2014. From ARM NEON to Intel SSE – the automatic porting solution, tips and tricks. Intel Developer Zone. (February 2014). http://software.intel.com/en-us/blogs/2012/12/12/from-arm-neon-to-intel-mmxsse-automatic-porting-solution-tips-and-tricks