Chapter 6
Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform

Damien Couroussé, Victor Lomüller, and Henri-Pierre Charles

6.1 Introduction

Since the early beginning of computer history, one has needed programming languages as an intermediate representation between algorithms description and machine-readable instructions. In broad outline, running an algorithm on a computer requires the following steps: (1– software development, implementation) the developer transcribes the algorithm into a source file containing programming language instructions, (2– compilation) a compiler translates these programming language instructions into machine code and performs adaptations to the original code for optimized fit to the target execution support, (3– execution) the processor reads and executes the machine instructions, loads the input data and produces the data results.

Because compilation is performed *before* the program is run, the execution context and run-time data are not known at the time of code generation (Figure 6.1a). This means that, in order to leverage such information in code optimizations, one has either to assume about the characteristics of the execution context (and to provide verification mechanisms), to add extra instructions to adapt the program behavior depending on runtime data, which is known as code specialization, or to generate the program's machine code at run-time, after the execution context is known.

Dynamic code generation can be achieved by interpretation or compilation at runtime [78]. In classical frameworks, the aim is to provide a generic infrastructure for code generation, bounded by the syntactic and semantic definition of a programming language. The generality of such solutions comes at the expense of an important overhead in code generation, both in terms of memory footprint and computing

Damien Couroussé
CEA, LIST, DACLE/LIALP, F-38054 Grenoble, France e-mail: damien.courousse@cea.fr

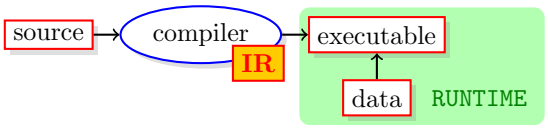
Victor Lomüller
CEA, LIST, DACLE/LIALP, F-38054 Grenoble, France e-mail: victor.lomuller@cea.fr

Henri-Pierre Charles
CEA, LIST, DACLE/LIALP, F-38054 Grenoble, France e-mail: henri-pierre.charles@cea.fr

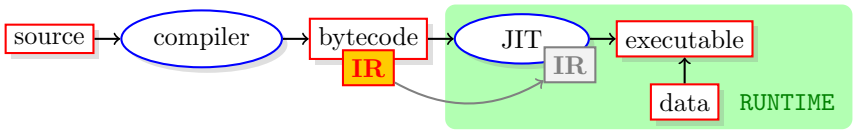
power. A well-known example is the Java programming language, designed to enhance application portability: Java source code is written without a priori knowledge of the platform that will execute the final machine code, thanks to a virtual machine that relies on an intermediate representation, the Java bytecode (Figure 6.1b). At runtime, the bytecode is either interpreted or compiled into machine code as soon as the overhead of code generation can be amortized by repeated calls of the generated code [79]. Despite the fact that a virtual machine has all required information to perform data-dependent optimizations, interesting values are difficult to use for such systems owing to an already high code generation cost [78]

Code optimization from run-time information is also useful for large-scale parallel computing systems, where an application component can be populated on a lot of processing elements. This application component has to be parametrisable so that its behaviour can be adapted to the processing element where it is instantiated. To do so, one would need either (1) a generic implementation that one can parametrize at instantiation but that will suffer from the performance overhead brought by a generic implementation, or (2) to modify and re-compile the component dynamically at run-time after one knows where it will be finally executed. Being able to specialize the executed code for each of the computing elements is likely to provide performance improvements, as long as the cost for such optimization remains modest. This issue is applicable to all large-scale multi-processor platforms: from High Performance Computers in data centers to multiprocessor Systems-on-Chip (MPSoCs) in future embedded devices. Due to the distributed nature of computing and memory resources in many-core platforms, it becomes challenging to bring dynamic compilation capabilities to such platforms. Moreover, because of the non-negligible memory footprint of the frameworks for Just-In-Time compilation (JITs), the limited size of the local memory in embedded many-core platforms becomes another important bottleneck in this context.

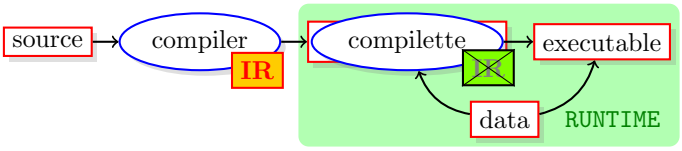
deGoal was designed to provide application developers the ability to implement application kernels tunable at run-time depending on the execution context, on the characteristics on the target processor, and furthermore on the *data to process*: their characteristics and their values [80]. Usually in processing applications, most of the execution time is spent in a very small portion of the whole application source code, which is most of the time a computation-intensive task also called *kernel*. We assume that improving the performance of kernels can leverage the overall application performance. Therefore, the idea using deGoal is to embed *ad hoc* run-time code generators, called *compilettes*, in a software application. Each compilette is specialized to produce the machine code of one application kernel. On the contrary to dynamic compilation, in our solution we embed at runtime only the necessary processing intelligence to perform code optimizations that can exploit the properties of the data to process, but no analysis of the intermediate representation or a subset such as bytecode (Figure 6.1c). As a consequence, this enables the production of very fast code generators (10 to 100 times faster than typical frameworks for runtime code interpretation or dynamic compilation). As such, deGoal provides a lightweight solution for dynamic code generation applicable to massively parallel systems. The compilettes offer a low memory footprint and very fast code gener-



(a) static compilation



(b) dynamic compilation (Java, etc.)



(c) runtime code generation with deGoal compilettes

Fig. 6.1 Illustration of the static and dynamic compilation schemes, and comparison with the runtime code generation with compilettes. *IR* stands for *Intermediate Representation*

ation. Furthermore, deGoal was designed to provide very large portability, which makes it easily applicable to heterogeneous platforms: The compilettes are compiled from ANSI-C source code after source-to-source code transformations. This the code generation process that we propose here can target a large number of platform architectures, which is only limited by the availability of a C compiler for the processor that will perform the code generation at runtime.

In this paper, we present an approach to describe a specialized code generator. The aim is to build a system that:

- Minimizes the generation overhead compared to classical JIT systems.
- Allows more flexibility over the generated function application domain. Specifically, we want to be able to select the data-type at run-time.
- Brings gain in performance, or at least similar performances, by removing dead code, unused loads or by constant propagation...

Our main contribution in this paper is:

- The presentation of a way to describe how a code generator should behave for a key part of an algorithm.
- To illustrate that taking into account run-time environment for auto-tuning is possible, and how it offers a performance improvement.
- The illustrate the use of specialized code generation for the STHORM platform

The rest of this paper is organized as follows: section 6.2 introduces the core idea of deGoal and data-dependent code optimization, section 6.3 details the use of our tool on matrix multiplication for the processors of a MPSoC, and the results achieved. We end this chapter by providing an overview of the related works in section 6.4.

6.2 Overview of deGoal

6.2.1 Kernels and compilettes

The two categories of software components around which our code generation technique is organised are called *kernels* and *compilettes*.

Kernel A kernel is a small portion of code, which is part of a larger application, and which is the target of our runtime code generation setup. Our technique focuses on the optimisation at runtime of these small parts of a larger application in order to improve the kernel's performance. In the context of the typical use of deGoal, good performance is understood as one or several criteria among low execution time, low memory footprint and low energy consumption.

Compilette A compilette is designed to generate the code of *one* kernel at runtime. A compilette can be understood as an *an hoc* small code generator that is executed at application runtime. We use the term *compilette* to underline the fact that, in order to achieve very fast code generation, this small runtime generator does not embed all the optimisation techniques usually carried out by a static compiler, but only the required ones considering the target kernel to optimize.

In order to target computing architectures that include domain-specific accelerators and to raise the level of abstraction of the source code, compilettes are described using a mix of standard C and of a dedicated high-level ASM language: `Cdg` [80]. This language has demonstrated its ability to achieve performance improvements in comparison with highly optimised static code [81]. We have chosen to stay with an assembler-like language in order to stay as close as possible to the final run-time model: an instruction-set processor. Our aim is furthermore to allow the direct use of multimedia arithmetics and to provide flexible and easy support to vectors and complex data sets.

The main paradigm shift relies in the fact that `Cdg` instructions describe code to be generated instead of code to be executed. On the contrary to common ASM languages, it is possible here to parametrise these instructions with values known at runtime, and to use vector variables. The variables manipulated are vector registers, whose size will be determined at the time of code generation, when the use of the physical registers in the programming context is known. It is also possible to map the assembly instructions to vector instructions when they are available on the target processor, and to map the assembly instructions to different arithmetic operators

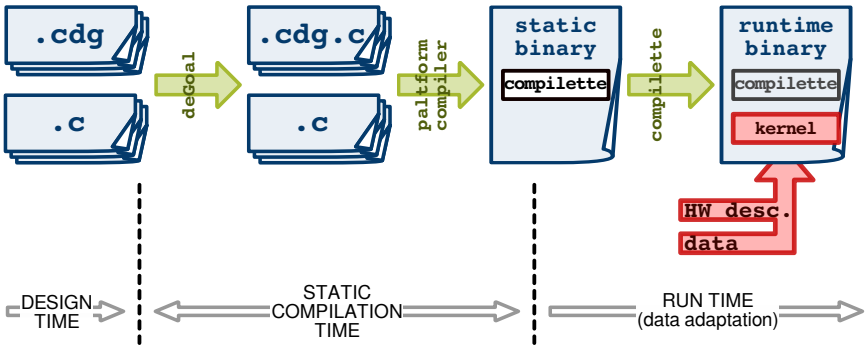


Fig. 6.2 deGoal workflow: from the writing of application's source code to the execution of a kernel generated at runtime

depending on the data values to process. As we will illustrate in section 6.2.3.2, it is possible to mix `C` instructions and `Cdg` instructions. In this case, the C source code will control the code generation done in the `Cdg` instructions.

The instruction set includes:

A variable length register set The instruction set uses vectorial registers with variable width and a variable number of elements. i.e. the programmer could define `VectorType f float 64 8`, to use any register of type `f` as a vector of 8 elements of 64 bit floating point values.

Classical arithmetic instructions `add`, `sub`, `mul`, `div`, but also instructions specific to the multimedia domain such as `sad` (sum of absolute differences), `mma` (matrix multiply and add) and FFT butterfly. These instructions can work on registers of variable length and type.

Load and store This family of instructions supports stride description. This permits the description of complex memory access patterns.

Using this high-level instruction set, deGoal can generate the corresponding instructions for processors which have native support, or generate optimised code for processors without support. In both cases the code generation is fast and produces efficient code.

6.2.2 Workflow of code generation

The building and the execution of an application using deGoal consists of the following steps: writing the source code; compiling the binary code of the application and the binary code of compilettes using static tools; generating the binary code of kernels by compilettes; running the kernels. These steps are illustrated in Figure 6.2 and are explained below:

Application development time: writing the source code This task is handled by the application developer, and/or by high-level tools. The source code of compilettes is written in specialized `.cdg` source files that allow for the mix of `C`dg and `C` languages, while the rest of the application software components are written using a standard programming language, such as C.

Rewrite time: generation of C source files This step consists in a source-to-source transformation: the `.cdg` source files are translated into standard `C` source files by `degoaltoc`, which is one of deGoal tools.

Static compilation time: compilation of the application The source code of the application now consists in a set of standard C source files, including the source code of the compilettes. The binary code of the application is produced by a standard C compiler. This step is the same as in the development of a standard C application.

Runtime: generation of kernel's binary code At runtime, the compilette generates optimized binary code for the kernel(s) to optimize. This task can be executed on a processor that is different of the processor that will later run the kernel. Furthermore, the compilette's processor and the kernel's one do not necessarily need to have the same architecture. A compilette can be run several times, for example as soon as the kernel needs to be regenerated for new data to process. We have detailed on figure 6.2 two particular inputs of the compilette: data and hardware description. The originality of our approach indeed relies in the generation of a binary code optimized for a particular set of application data. At the same time, the code generation is able to introduce hardware-specific features.

Runtime: kernel execution The program memory buffer filled by the compilette is run on the target processor (not shown in figure 6.2).

6.2.3 A Tutorial Example

Our tutorial example illustrates how to handle simple kernels for scalar multiplication using deGoal (Figure 6.3). We introduce the main concepts of deGoal with the trivial example of the multiplication of two integer variables. We then elaborate on vector multiplication. For the purpose of illustrating how code generation is performed, our examples are based on the STxP70 processor, described in section 6.3.2.1. However, the source code of the compilettes illustrated here could be applied straightforward to other processor architectures.

6.2.3.1 Simple multiplication

We want to perform the runtime specialization of the generic function `genericMul` that multiplies two integers (Figure 6.3a). After specialization, the function will be replaced by a function that multiplies by a constant known at runtime, i.e. that specializes the `val` parameter of `genericMul`. However, this parameter can only be

```
1  int genericMul (int param, int
       val)
2  {
3    return (param*val);
4  }
```

(a) Generic code in C

```
1  void mulCompile(cdgInsnT *code,
       int val)
2  {
3    #[
4      Begin code Prelude in
5      mul out, in, #(val)
6      rtn
7      End
8    ]#;
9  }
```

(b) Compilette code (in Cdg)

```
      PUSHRL 0x4000 ;;          1
G7? MAKE32 R12, 3 ;;            2
G7? MP R0, R0, R12 ;;           3
      POPRL 0x4000 ;;           4
G7? RTS  ;;                     5
```

(c) assembly code (val=10)

```
      PUSHRL 0x4000 ;;          1
G7? SHL R0, 1;;                 2
      POPRL 0x4000 ;;           3
G7? RTS  ;;                     4
```

(d) assembly code (val=2)

```
G7? MAKE32 R12, 10 ;;           1
G7? MP R0, R0, R12 ;;           2
G7? RTS  ;;                     3
```

(e) assembly code (val=10) for a leaf kernel

Fig. 6.3 A tutorial example: dynamic specialisation of multiplication.

known at runtime: at the initialization time of the process or during the program execution. Furthermore, it is likely to change multiple times.

The compilette `mulCompile` is a standard C function that includes elements of the `Cdg` language at lines 3 to 8 between `#[` and `]#` (Figure 6.3b). Line 4 marks the moment where the code generation actually begins. `Prelude` states that this block needs stack and register management: in the generated code, we only save and restore the R14 register (link register) because R0 and R1 are defined as scratch registers in the ABI (Application Binary Interface) of the STxP70. `code` is the pointer to the code cache, and finally `Prelude` comes with one parameter: `in`, which means that the generated kernel will take one parameter named `in`. According to the ABI of our target processor, `in` will be allocated by default on `R0`.

Finally, the `rtn` instruction is the return instruction that ends the kernel routine and inserts the return instruction. `End` ends the code generation: during code generation, the evaluation of this instruction triggers the computation of branch locations and the flushing of internal data.

Line 5 performs the multiplication between register `in` and a C r-value (written inside `#()`) and stores the result in `out`. In this case, the r-value is simply `val`. The compilette, when called at runtime, produces a binary kernel for the architecture selected at compilation time (Figures 6.3c and 6.3d, respectively when `val` equals to 10 and 2). The two dotted arrows highlight the locations where the runtime value `val` is evaluated and integrated into the produced code as a constant. In this tutorial example we illustrate a simple data-dependent optimization: the compilette generates either a kernel that uses the standard multiplication instruction (Figure 6.3c),

```
1 void vector_mul(int * A, int A_len, int alpha, int * B) {
2     for (int i=0; i<A_len; i++) {
3         B[i] = alpha * A[i];
4     }
5 }
```

Fig. 6.4 A trivial implementation of vector multiplication in C

or the shift left instruction (Figure 6.3d) depending on the value taken by `val` at runtime.

The source code of the compilette (Figure 6.3b) is statically processed by deGoal. The specialized code generator is then generated and dumped into a C file, that is statically compiled by the compiler of the target platform. This approach removes any direct intermediate representation manipulation which needs complex code generation. This way, we reduce the required computation time to the minimum.

6.2.3.2 Vectorial multiplication

Now that we have introduced the main elements of deGoal for the building of code generators, we can safely introduce an important feature of our tool: vectorial registers. To do so, we will extend our previous example to vector multiplication. Our aim is to compute $[B] = \alpha \times [A]$, where $[A]$ is the input vector, α a scalar known at the time of code generation, and $[B]$ is the result vector.

Using standard C, we could write vector multiplication as in figure 6.4. With dynamic code generation, we will specialise the kernel according to the memory location of A, its length, and the value of α. As a consequence, the kernel generated by the compilette will need only one invocation parameter: the address of vector B (assuming that it has the same length than A).

There are several possibilities to implement such a code generator, and we will illustrate two of them here: (1) with the vector support of deGoal instructions (figure 6.5), and (2) by mixing `cdg` instructions with plain C to control the code generation and loop over the vector elements (figure 6.6). The disassembled binary code that will be produced for these two generated kernel are illustrated in figures 6.7a and 6.7b, respectively. To use float arithmetic instead of integer, one would simply need to replace `int` by `float` at lines 5 and 6 in figure 6.5 and at line 4 in figure 6.6, when declaring the types used for scalar arithmetics.

In the compilette illustrated in figure 6.5, each of the elements of the vector register `v` will be mapped to a physical register, as long as there are enough registers available on our target processor. In figure 6.5, one can see in the generated code that `v` has been mapped on registers `R2` to `R9`. `v` being a vector register of 8 elements, the instruction `lw v, tmp` will actually generate 8 successive memory loads with a stride of 1 word from the address contained in the register variable `tmp`, mapped to `R1`. The code generator proceeds similarly for the `mul` and `sw` instructions. The

```
1  void compilette(cdgInsnT* code, int * A_addr, int A_len, int
       alpha) {
2  #[
3    Begin code Prelude B_addr
4    Type  int32       int 32
5    Type  vectorInt32 int 32 8
6    Alloc int32       tmp
7    Alloc vectorInt32 v
8
9    mv tmp, #(A_addr)
10   lw v, tmp
11   mul v, v, #(alpha)
12   sw B_addr, v
13   rtn
14
15   Free tmp, v
16   End
17 ]#;
18 }
```

Fig. 6.5 Implementation of a compilette for vector multiplication using vector registers. For the sake of simplicity, we assume that we have enough registers available to allocate vectors *A* and *B* at once.

multiplication (`MP`) instruction of the STxP70 processor only works with two register arguments and not with an indirect memory address as an argument. Thus, the instruction `make32 R12, 104876` instruction at line 2 of figure 6.7a is automatically generated by `mul` to store the address `A_addr` in the scratch register `R12`.

Figure 6.7a also demonstrates the capability of our instruction scheduler to deal with instruction latency and register dependencies. We will illustrate this point on one example: on the STxP70 processor, the LW instructions have a latency of 3 cycles. This means that, to avoid cycle stalls, the MP instruction on R2 (line 6) must come 3 cycles after the instruction `LW R2` (line 3).

The main difference of C-controlled kernel (figure 6.6) with the vectorized kernel (figure 6.5) comes from the use of the same register variable `tmp`, mapped on the physical register `R5`. `tmp` successively stores each of the memory loads from vector *A* (`A_addr`) and is then used to store the result of the multiplication by α (`alpha_r`). Because the same physical register `R5` is used to perform all of the store and multiplication operations for each of the vector elements, our instruction scheduler is not able to bundle the instructions generated in this kernel because of register dependencies. As a consequence, the binary code generated from this kernel (figure 6.7b) is far less compact than the code in figure 6.7a.

To give an idea of the level of optimisation enabled in this example, we compare the execution times of the kernels in figures 6.7a and 6.7b and of the C version illustrated in figure 6.4. The compilation is performed with the `-O3` optimization flag, and the execution times are measured using the simulator of the STxP70 pro-

```
1  void compilette(cdgInsnT* code, int * A_addr, int A_len, int
       alpha) {
2  #[
3    Begin code Prelude B_addr
4    Type  scalar32_t int  32
5    Type  addr_t     uint 32
6    Alloc scalar32_t alpha_r
7    Alloc addr_t     A_addr_r
8    Alloc scalar32_t tmp
9
10   mv alpha_r, #(alpha)
11   mv A_addr_r, #((unsigned int)A_addr)
12 ]#;
13 for (int i=0; i<A_len; i++) {
14   #[
15     lw tmp, @(A_addr_r + #(i))
16     mul tmp, tmp, alpha_r
17     sw @(B_addr + #(i)), tmp
18   ]#;
19 }
20 #[
21   rtn
22   Free tmp, A_addr_r, alpha_r
23   End
24 ]#;
25 }
```

Fig. 6.6 A compilette for vector multiplication controlled. The code genation is controlled by C statements

cessor in CAS mode, presented later in section 6.3.2.1. The kernels execute respectively in 51, 71 and 80 cycles for two vectors containing 8 elements. The binary code of the C version counts 18 instruction bundles. This code is even smaller than our kernel in figure 6.7a because the C kernel uses the hardware loop instructions of the STxP70. We could help the C compiler with hints about vectorisation (e.g. `#pragma unroll`), but unrolling vectorial multiplication at the time of static compilation is difficult because the vector lengths are not known. On the contrary, at runtime, it becomes easy to perform loop unrolling knowing the lengths of the vector that our kernel will process. For larger loops where unrolling would incur a loss in performance, we could as well use branch instructions and loop structures. This is not shown in this paper for the sake of brevity.

```
1  PUSHRL 0x43F8;;
2  MAKE32 R12, 1048576; MAKE32 R1, 16176;;
3  LW R2, @(R1+0x0);;
4  LW R3, @(R1+0x4);;
5  LW R4, @(R1+0x8);;
6  LW R5, @(R1+0xC);  MP R2, R2, R12;;
7  LW R6, @(R1+0x10); MP R3, R3, R12;;
8  LW R7, @(R1+0x14); MP R4, R4, R12;;
9  LW R8, @(R1+0x18); MP R5, R5, R12;;
10 LW R9, @(R1+0x1C); MP R6, R6, R12;;
11 SW @(R0+0x0), R2;  MP R7, R7, R12;;
12 SW @(R0+0x4), R3;  MP R8, R8, R12;;
13 SW @(R0+0x8), R4;  MP R9, R9, R12;;
14 SW @(R0+0xC), R5;;
15 SW @(R0+0x10), R6;;
16 SW @(R0+0x14), R7;;
17 SW @(R0+0x18), R8;;
18 SW @(R0+0x1C), R9;;
19 POPRL 0x43F8;;
20 RTS;;
```

(a) kernel generated from figure 6.5

```
1  PUSHRL 0x4038;;
2  MAKE32 R4, 16152; MAKE32 R1, 1048576;;
3  LW R5, @(R4 + 0x0);;
4  MP R5, R5, R1;;
5  SW @(R0 + 0x0), R5;;
6  LW R5, @(R4 + 0x4);;
7  MP R5, R5, R1;;
8  SW @(R0 + 0x4), R5;;
9  LW R5, @(R4 + 0x8);;
10 MP R5, R5, R1;;
11 SW @(R0 + 0x8), R5;;
12 LW R5, @(R4 + 0xC);;
13 MP R5, R5, R1;;
14 SW @(R0 + 0xC), R5;;
15 LW R5, @(R4 + 0x10);;
16 MP R5, R5, R1;;
17 SW @(R0 + 0x10), R5;;
18 LW R5, @(R4 + 0x14);;
19 MP R5, R5, R1;;
20 SW @(R0 + 0x14), R5;;
21 LW R5, @(R4 + 0x18);;
22 MP R5, R5, R1;;
23 SW @(R0 + 0x18), R5;;
24 LW R5, @(R4 + 0x1C);;
25 MP R5, R5, R1;;
26 SW @(R0 + 0x1C), R5;;
27 POPRL 0x4038;;
28 RTS;;
```

(b) kernel generated from figure 6.6

Fig. 6.7 Binary code (disassembled) of the kernel generated by the compilettes for vector multiplicaiton. For the sake of simplicity, guard registers are not shown here.

6.3 An experiment on matrix multiplication

6.3.1 Implementation of matrix multiplication

This section describes the implementation of a processing kernel for matrix multiplication in order to illustrate the use of deGoal. We describe first a reference implementation, which is statically compiled with the platform's compiler. We then describe two improved implementations using deGoal: the first exploits matrix properties such as matrix size, element size, and memory addresses; the second exploits the values of matrix elements.

6.3.1.1 Reference implementation

Our aim is to perform the standard matrix multiplication as described in equation 6.1, where a, b and c stand respectively for elements of matrices $[A]$, $[B]$ and $[C]$ of sizes $n \times p$, $p \times q$ and $n \times q$:

$$\forall i \in \{1,\ldots,n\}, \forall j \in \{1,\ldots,q\}, c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{6.1}$$

The reference implementation of this algorithm is illustrated in Figure 6.8. We used it as a reference implementation for our experimental measurements.

```
clear(C)
for (y=0; y < n; y++) {
    for (x=0; x < q; x++) {
        for (i=0; i < p; i++) {
            C[x,y] = C[x,y] + A[i,y] * B[x,i]
        }
    }
}
```

Fig. 6.8 Reference implementation of the matrix multiplication (in pseudo C code)

6.3.1.2 First implementation in a compilette

A simplified overview of our implementation of the matrix multiplication using deGoal is illustrated in Figure 6.9. `compilette` is the code generator that produces an optimized kernel function `kernel`, which encompasses the inner-most loop from Figure 6.8. The code generated for `kernel` depends on the properties of matrices `A`, `B` and `C` : row and column sizes, memory alignment and address of the data in memory. These values are precomputed and propagated into the instructions of `kernel` at code generation time. In consequence, `kernel` does not need invocation parameters.

This implementation of `kernel` is very similar to the reference implementation introduced above, at the exception that all the constants describing matrix properties, which are known at code generation time, have been propagated into the generated code. As we will show in the results section, these improvements alone already contribute to a good performance improvement.

```
/* generation of the kernel's code */
(kernel, v) = compilette(A, B, C)

/* compute matrix multiplication */
clear(C)
kernel();
```

Fig. 6.9 optimized implementation of the matrix multiplication using deGoal (in pseudo-code)

6.3.1.3 Kernel specialization on matrix values

If the matrices to process are sparse or contain remarkable data values, it is possible to further increase performance by specializing the generated code depending on the element *values* of the matrix to process. We illustrate the data-dependent specialization of our processing kernel with a naive algorithm for sparse matrices. Usually, applications that involve the processing of sparse matrices will move to different processing algorithms and to a dedicated representation of data. However, our aim is to illustrate here how, thanks to the use of data-dependent optimizations with runtime code generation, it is possible to drastically improve the performance of our base algorithm.

The code generation is split in two phases (Figure 6.10): `template_gen` generates the global structure of the processing kernel that is independent of data values in *A*. At each processing loop, `data_gen` fills the kernel's code upon data values in the row vector to process in *A*. When there is nothing to execute (for example, all matrix values in the current row in *A* are null), `data_gen` returns `NULL` and we immediately move to the next loop step.

This technique involves an extra overhead because the kernel's code is regenerated at each step in the innermost loop. However, as we will show below, this overhead can be compensated very quickly for sparse matrices.

```
clear(C)

/* generate the kernel's structure */
(kernel_templ, v) = template_gen(A, B, C);

/* process matrix multiplication */
for (y=0; y < n; y++){
  for (i=0; i < p; i+=v){
    /* specialize instructions on matrices' data */
    kernel = data_gen(kernel_templ, A, y, i);
    if (NULL != kernel)
      kernel(y, i);
} }
```

Fig. 6.10 Implementation of the matrix multiplication (pseudo-code) with code specialization on matrix values

6.3.2 Experimental results

6.3.2.1 Target architecture

We target in this work the embedded platform called STHORM (formerly Platform P2012), jointly developed by STMicroelectronics and CEA [55].

The STxP70-4 processor is a 32-bit RISC core. It comes with a variable-length instruction encoding and a dual issue VLIW architecture. Two sets of hardware loop counters are provided to enable loop execution at maximum speed without cycle overheads due to software control. The core processor contains an internal extension for integer multiplication, and an optional single-precision floating point extension used in this experiment.

The STHORM SDK is delivered with a full toolchain for compiling, debugging, profiling and simulation in functional and cycle-accurate modes. Our experiments are based on the platform's toolchain and on the ISS simulator of the STxP70 core in CAS (cycle-accurate) mode. In this mode, the simulator models all the latencies that can occur in the processor pipeline : instruction latency, CPU stalls and register dependencies. The latencies of memory accesses are not taken into account by this mode. All our experiments are however using the scratchpad memories (TCDM and TCPM) of the processor, which lowers the effect of this limitation of the simulator in our experiments.

6.3.2.2 Port of deGoal for the STxP70 processor

deGoal handles by default register allocation and a simple mechanism for instruction scheduling. A simple scheduler allows for the optimization of instruction scheduling with regards to instruction latencies and register dependencies.

However, as compared to standard RISC processors, code generation for the STxP70 processor is a bit more challenging, especially when moving to runtime code generation. Thenceforth, we extended the port of deGoal for this architecture with VLIW support: optimizing the dual issue and the construction of instruction bundles. Also the floating-point support comes as an extension and uses a separate register file of 16 32-bit registers. Our port of deGoal supports all of these features of the STxP70.

6.3.2.3 Experimental setup

We have evaluated our optimized version of the matrix multiplication against the reference implementation described in section 6.3.1.1.

The reference implementation is compiled in `-O3`. Loop unrolling, support of hardware loop counters and of the floating-point extension are also enabled. The best performance for the reference implementation was eventually obtained with an

implementation close to the pseudo code described in Figure 6.8, with the addition of `#pragma unroll 8` on top of the innermost loop.

The code generated by deGoal's compilette does not depend on compiler optimizations, because it is generated at run-time by the compilette. Hence whatever the compiler optimizations selected, the execution time of the generated kernel remains constant. Compiler optimizations have however an effect on the performance of the compilette, because it is statically compiled as a standard application component. In our performance measurements, we have used the same compiler options to compare the reference implementation and our implementation using deGoal.

We have also exploited the VLIW extension of the STxP70-v4 core, using the appropriate compilation flags. On the compilette's side, VLIW support is integrated in the `cdg` pseudo-ASM language of deGoal. As a consequence, it is not exposed to the developer and the compilette is tailored to automatically exploit this feature as soon as the processor supports it.

6.3.2.4 Measure of the code generation time

We have instrumented the compilette to measure the time spent in code generation at run-time: code generation takes from 150 to 300 cycles per instruction bundle generated. The variation of the average speed of code generation per instruction bundles is due to the computation of instruction bundling, the scheduling of instructions according to register dependencies and the extra computations done at the end of code generation, for example computing the jump addresses. The best code generation speed is achieved for unrolled code without instruction bundling.

The code generation time is not taken into account in the speedup results presented below, because it is not necessary to regenerate the code for each matrix multiplication. As an indicator, code generation represents 100 % of the execution time for a multiplication of 16×16 matrices, and less than 0,1 % for 256×256 matrices.

6.3.2.5 Performance of the processing kernels

Figure 6.11 illustrates the performance improvements achieved using deGoal as compared to the reference implementation compiled with full optimization, for two cases of code generation: using the hardware loop counters provided by the STxP70 core (`HW loop`), and fully unrolling the kernel's code (`unrolled`). The speedup factor s represents the reduction factor of the execution duration of our implementation as compared to the reference implementation. We calculate it as follows: $s = \frac{t(\text{ref})}{t(\text{degoal})}$, where $t(\text{ref})$ measures the time execution of the reference implementation, $t(\text{degoal})$ the time execution of the generated kernel. Our implementation using compilette brings a good overall performance improvement: when the matrix size is 256×256 elements, we achieve a reduction of the execution time of 21 % for integer multiplication, and of 17 % for floating-point multiplication.

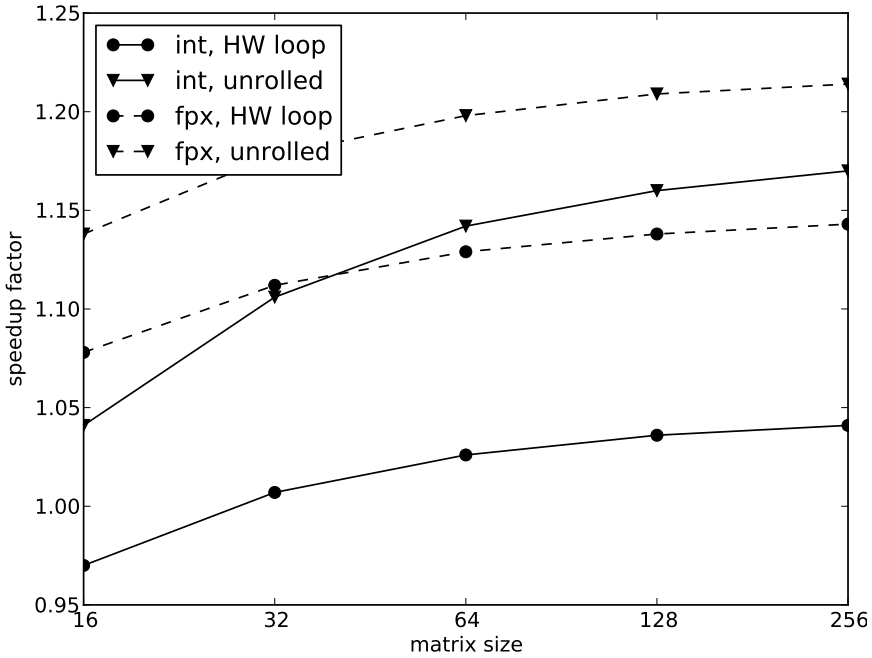


Fig. 6.11 Speedup factor measured, for integer multiplication (plain line) and floating-point multiplication (dashed line), according to the implementation described in section 6.3.1.2.

Figure 6.12 illustrates the speedup factor measured when using code specialization on the data of matrix A, as presented in section 6.3.1.3. We illustrate here the most favorable case where matrix A is the identity matrix. In this case, the looped implementation shows a huge speedup because of the instructions removed from the kernel when null values are met in matrix A. The unrolled version is not efficient, considering the favorable experimental conditions, because a part of the code generation is performed *during* kernel's execution, and code unrolling requires a lot more instructions to be generated.

6.4 Related work

There is an extensive amount of literature about approaches related to our work with deGoal.

Dynamic compilation and interpretation are most of the time used together in Just-In-Time compilers (JITs) [78]. JITs use interpretation for the parts of the program that are run seldom, and dynamic compilation is reserved for hostpots, which are identified by tracing the application activity at runtime. Such techniques usually require to embed a large amount of intelligence in the JIT framework, which means a

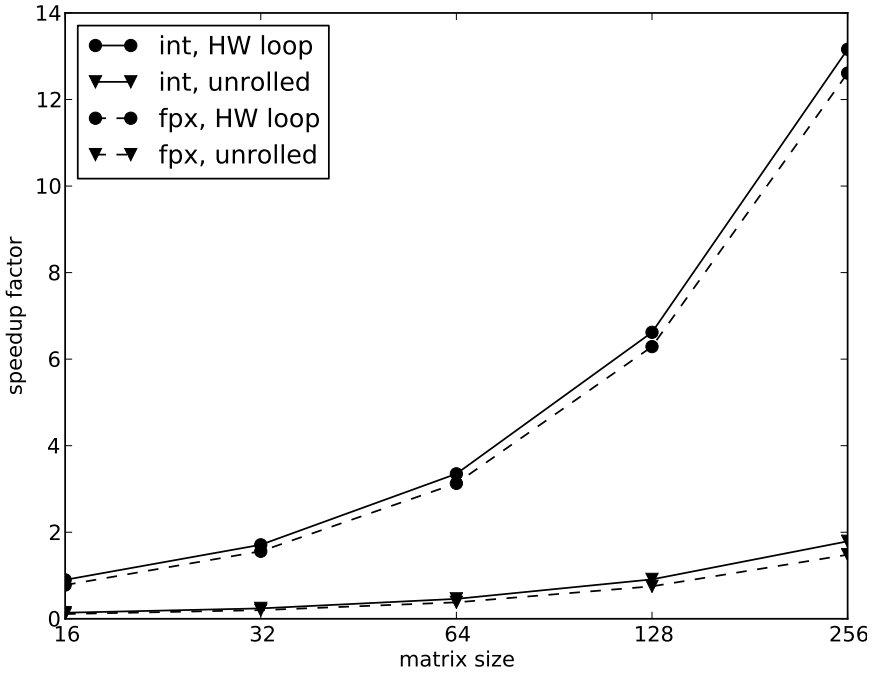


Fig. 6.12 Speedup factor measured, for integer multiplication (plain line) and floating-point multiplication (dashed line), according to the implementation described in section 6.3.1.3.

large footprint and a significant performance overhead. In order to target embedded systems, some research works have tried to tackle these limitations: memory footprint can be reduced to a few hundreds of KB [82], but the binary code produced often presents a lower performance because of the smaller amount of optimizing intelligence embedded in the JIT compiler [83].

In deGoal, the objective is to reduce the cost incurred by runtime code generation. Our approach allows to generate code at least 10 times faster than traditional JITs: JITs hardly go below 1000 cycles per instruction generated while we obtain 25 to 80 cycles per instruction generated on the STxP70 processor. Our approach is similar to partial evaluation techniques [84, 85], which consists in pre-computing during the static compilation passes the maximum of the generated code to reduce the run-time overhead. At run-time, the finalization of the machine code consists in: selecting code templates, filling pre-compiled binary code with data values and jump addresses. Using deGoal we compile statically an *ad hoc* code generator (the compilette) for each kernel to specialize. The originality of our approach relies in the possibility to perform run-time instruction selection depending on the *data* to process [80].

Code specialization is a technique similar to partial evaluation. Specialization can be done statically, at compilation time, or dynamically. C++ templates might

be the most well known static specializer. The developer writes a function that is parametrized by a list of types or integer-constant parameters. When the template is used, the user indicates missing parameters and the compiler automatically generates the new function according to those parameters. This brings more flexibility during the development process at the expense of a fast growing binary size (for each set of parameters, a new function is generated). However, the template parameter values have to be known at compile time, which strongly limits the number of code optimizations applicable.

Equivalent systems that operate at run-time are less used and include a larger diversity of approaches, which can be regrouped into different categories. Fully-manual approaches rely on the user to describe what should be generated at run-time [86, 87]. With this approach, the user has a fine control over the generated code. Semi-manual approaches rely on the user to annotate parameters that should be specialized [88]. Fully-automatic approaches try to detect, at compile-time, the code parts that could benefit from run-time code generation [89, 90, 91]. Each approach avoids an explosion in code size while maintaining a larger spectrum in which it can be used. A major advantage is the capability to cover the whole function application domain without having to speculate on parameter values. The major drawback is, of course, that a part of the compilation cost has to be paid at run-time and has to be amortized in one way or another.

Various optimizations can be performed in the various times of application life-time, for instance during static compilation (where most optimizations are usually performed), during link time [92, 93] or during installation time like ATLAS [94]. Some tools use complex schemes like FFTW [95], where multiple code variants are generated, compiled and then evaluated at installation time. Then, during program initialization the codelets are selected by a planner that is parametrized by the size of the DFT to be computed. Another example that use a similar approach is ATLAS [94]. Some other optimizations are staged across several times. For instance, iterative compilation accumulates information through different times. With enough information, it rolls back to an earlier time to perform new optimizations. Profile-guided compilation (PCG) uses execution traces obtained by running the application with a learning data-set to perform new optimizations. But few tools are able to perform optimization based on the run-time environment.

Late code specialization is very close to our approach. Generally speaking, these approaches pre-compile statically a template version of the application code, which is completed at runtime by a code specializer. `C [87] extends the C syntax by adding syntactic elements like ` or @ to describe parts of code that will be generated at run-time. The compilation phase transforms `C expressions into an Intermediate Representation (IR). At runtime the IR is assembled and compiled via simplified compiler back-end. DyC [88] is a tool that creates code generators from an annotated C code. Like `C, it adds some tokens such as @ to evaluate C expressions and inject the results as an immediate value into the machine code. Calpa [91] uses profile-guided compilation to detect functions that could benefit from runtime code specialization and generate the code generator using DyC. Tempo [90] works on an unannotated subset of C. It analyses the source code to detect parts of the code that

could benefit from constant propagation, and creates a binary template from it. At run-time, the template is filled with missing values and executed. Fabius [89, 96] uses a similar approach to Tempo, applied to ML. Our approach differs from those tools targeting late code specialization by using:

- a low-level code representation with vector description,
- no manipulation of byte-code at runtime,
- the capability to control the code generation,
- the capability to perform cross-architecture code generation.

Approaches for multi-core architectures mostly use a classical JIT. Our approach tries to avoid the use of byte-code manipulation to focus on specialization using run-time informations. LLVM [97] (Low Level Virtual Machine) is a compilation framework that can target many architectures, including x86, ARM or PTX. One of its advantages is the unified internal representation (LLVM IR) that encode a virtual low-level instruction with some high-level information embedded on it. Various tools were built on top of it, starting with clang, a C/C++/Objective-C compiler. With the release of the CUDA toolkit 4.1, the Nvidia compiler compiler is based on LLVM. The driver loads a textual representation of the assembly language targeting the GPU, and then dynamically compiles it to a binary representation. This technique is here mainly used to hide the implementation details of Nvidia GPUs, and not in the purpose of runtime optimizations.

6.5 Conclusion

In this paper, we have introduced deGoal as a tool for runtime code generation thanks to the integration of compilettes in a binary application, and have illustrated the benefits of using deGoal to optimize processing kernels.

We have shown that deGoal can easily compete with a highly optimized code produced by a static compiler with little effort: the code produced has better performance than a code statically compiled with full optimization, and furthermore the quality of the code produced with deGoal is consistent and does not depend on compiler's options. deGoal also allows to specialize the code of a processing kernel for a particular set of run-time data, which is not possible using a static compiler. We have shown that for processing kernels with a high dependency on the data to process the performance increase can be huge.

Because deGoal is related to the generation of machine binary instructions, its scope of application is actually restricted to the processor. In order to use these optimization techniques in large scale platforms, e.g. MPSoCs or HPC clusters, one must rely on tools of higher level for the parallelization of an application on multiple processing elements. Future work will present how it is possible to integrate kernels optimized with degoal's compilettes in large scale applications.

deGoal is currently under active development. It is able to produce code for multiple platforms: Nvidia GPUs, ARM processors (Jazelle, SIMD, Thumb, NEON), the STxP70, and other RISC processors under NDA.