Dynamic Code Generation: an Experiment on Matrix Multiplication

Damien Couroussé Henri-Pierre Charles
CEA-LIST, Lastre laboratory
firstname.surname@cea.fr

Abstract

In this paper we detail the implementation of a typical CPU-bounded processing kernel: matrix multiplication. We used `deGoal`, a tool designed to build fast and portable binary code generators. We were able to outperform a traditional compiler: we obtained a speedup factor of 2.22 and 1.86, respectively for integer and floating-point multiplication with 256×256 matrices. Furthermore, code specialization on the data to process allows us to further increase the performance of the multiplication kernel by a factor of more than 20 in favorable conditions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Translator writing systems and compiler generators; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures—Parallel processors

General Terms Performance, Design, Algorithms

Keywords dynamic code generation, run-time optimization, embedded systems, parallel computing

1. Introduction

Since the early beginning of computer history, one has needed programming languages as an intermediary translation between algorithms and machine-readable instructions. Typically, from a simple viewpoint, running an algorithm on a computer requires the following steps: (1) the developer translates the algorithm into a source file containing programming language instructions, (2) a compiler translates these programming language instructions into machine code, (3) 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, it is not possible to produce machine code on the basis of knowledge of the execution context, which can be only known at run-time. This means that one has either to assume about the characteristics of the execution context (and to provide verification mechanisms), or to add extra instructions to adapt the program behavior. The other way to deal with this problem is to generate the program's machine code at run-time, after the execution context is known. This can be achieved by instruction translation or compilation at run-time [1]. 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 does the match with the machine instructions supported by the target architecture.

Run-time compilation is also useful for large-scale parallel computer systems, where an application component can be populated on a lot of processing elements. 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. In this case, 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.

`deGoal` was designed with the two issues described above in mind 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* [2]. In Just-In-Time compilers (JITs) *all* the application code is generated at run-time, which allows to perform optimizations covering the whole scope of the application, but also incurs a strong performance overhead. 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 in a software application. Each code generator is specialized to produce the machine code of one application kernel. This enables the production of very fast code generators (10 to 100 times faster than common JITs).

The rest of this paper is organized as follows: section 2 introduces the core idea of `deGoal` and how this tool can be integrated in a larger-scale application, section 3 details the use of our tool on matrix multiplication for the processors of a MPSoC, section 4 details the results achieved, and section 5 presents related works.

2. Overview of `deGoal`

2.1 Kernels and compilettes

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

Kernel A kernel is a small portion of code, which is part of a larger application, and which is most of the time under strong performance constraints; our technique focuses on the optimization at run-time of these small parts of a larger application in order to improve the kernel's performance. In the context of this paper, good performance is understood as low execution time and/or low memory footprint.

LCTES 2012 June 12–13, 2012, Beijing, China

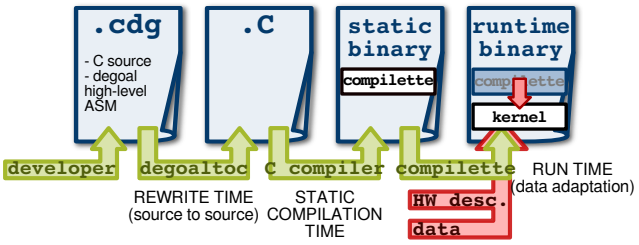

Figure 1. `deGoal` workflow: from the writing of application's source code to the execution of a kernel generated at run-time

Compilette A compilette is designed to generate the code of kernels at run-time. It can be understood as a small compiler that is executed at application's run-time. We use the term *compilette* to underline the fact in order to achieve very fast code generation, this small run-time compiler does not embed all the optimization techniques usually carried out by a static compiler. The binary code of a compilette is generated during the static compilation along with the rest of the application.

Compilettes are described using a mix of standard C and of a high-level ASM language [2], which describes the instructions that will be generated at run-time. However, on the contrary to common ASM languages, it is possible to parametrize these instructions with values known at run-time, and to use vector variables. More precisely, it is possible to manipulate vectors of registers, whose size will be determined at the time of code generation, when the use of registers in the programming context is known.

2.2 Workflow of code generation

The building of an application using `deGoal` is illustrated in figure 1 and explained below:

Writing the source code (application development time) This task is handled by the application developer, and/or by high-level tools. The source code of compilettes is written in specialized `.cdg` files, while the rest of the application software components are written using a standard programming language, such as C.

Generation of C source files (rewrite time) This step consists in a source-to-source transformation: the `.cdg` source files mixing high-level ASM instructions and standard C are translated into standard `C` source files by `degoaltoc`, which is one of `deGoal` tools. At this phase architecture-dependent features can be introduced in the C source files generated, for example register allocation and vectorization support.

Compilation of the application (static compilation time) 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.

Generation of kernel's binary code (run-time) At run-time, 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 1 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.

```
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]
        }
    }
}
```

Figure 2. Reference implementation of the matrix multiplication (in pseudo C code)

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

/* compute matrix multiplication */
clear(C)
for (y=0; y < n; y++) {
    for (i=0; i < p; i+= v) {
        kernel(y, i)
    }
}
```

Figure 3. optimized implementation of the matrix multiplication using `deGoal` (in pseudo-code)

Kernel execution (run-time) The program memory buffer filled by the compilette is run on the target processor (not shown in figure 1).

3. 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.

3.1 Reference implementation

Our aim is to perform matrix multiplication as described in equation 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 $n \times q$:

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

The reference implementation of this algorithm is illustrated in figure 2. We used it as a reference implementation for our experimental measurements.

3.2 First implementation in a compilette

A simplified overview of our implementation of the matrix multiplication using `deGoal` is illustrated figure 3. `compilette` is the code generator that produces an optimized kernel function `kernel`, which encompasses the inner-most loop from figure 2: it performs a vector multiplication between a row in `A` and a column in `B`, and accumulates the result into the corresponding element of `C`. 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, the only parameters needed by `kernel` are the row and column numbers of matrix `C`.

```
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);
} }
```

Figure 4. Implementation of the matrix multiplication (pseudo-code) with code specialization on matrix values

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.
- loops are reordered to minimize the number of memory loads. Considering the reference implementation of figure 2, we rearranged the loops to minimize memory loads for matrix `A`: the loop on `x` in done internally in `kernel`, and that the loop on `i` is raised one level up (figure 3). In other words, this means that once a line in matrix `A` is loaded, we compute all the related elements in matrix `C`.

As we will show in the results section, these improvements alone already contribute to a good improve performance.

3.3 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 (figure 4). This time, the code generation is split in two phases: `template_gen` generates the global structure of the processing kernel that is not likely to change upon 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 for code generation because the kernel's code at each step in the innermost loop, but, as we will show below, this overhead can be compensated very quickly.

4. Experimental results

4.1 Target architecture

We target in this work the embedded platform called Platform 2012 (P2012) [6], under development by STMicroelectronics and CEA. It is composed of multiple clusters connected through an asynchronous network-on-chip allowing each cluster to have its own voltage and frequency domain. Each cluster aggregates 16 cores dedicated to processing, plus one extra core dedicated to task management. All of the cluster processors are STxP70-4 cores from STMicroelectronics.

We have added support for the STxP70 to `deGoal`. The STxP70-4 processor is a 32-bit RISC core. It comes with a variable-length instruction encoding and a dual VLIW architecture allowing two instructions to be issued and executed at each cycle. 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 P2012 SDK is delivered with a full toolchain for compiling, debugging, profiling and simulation in functional and cycle-accurate modes. Our experiment is based on the platform's toolchain and on the cycle-accurate simulator of the STxP70 core.

4.2 Experimental setup

We have evaluated our optimized version of the matrix multiplication against the reference implementation described in section 3.1.

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 was obtained with an implementation close to the pseudo code described in figure 2.

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.

4.3 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 25 to 80 cycles per instruction generated. The speed of code generation varies significantly, mainly because of instruction bundling, and because of the extra computations done at the end of code generation, for example computing the jump addresses. The best results are achieved for unrolled loops 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 15 to 20 % of the execution time for a multiplication of 16×16 matrices, and less than 0,01 % for 256×256 matrices.

4.4 Performance of the processing kernels

Figure 5 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 compilette brings a good overall performance improvement: when the matrix size is 256×256 elements, we achieve a reduction of the execution time of 2.22 times for integer multiplication, and of 1.86 times for floating-point multiplication.

Figure 6 illustrates the speedup factor measured when using code specialization on the data of matrix A, as presented in section 3.3. 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

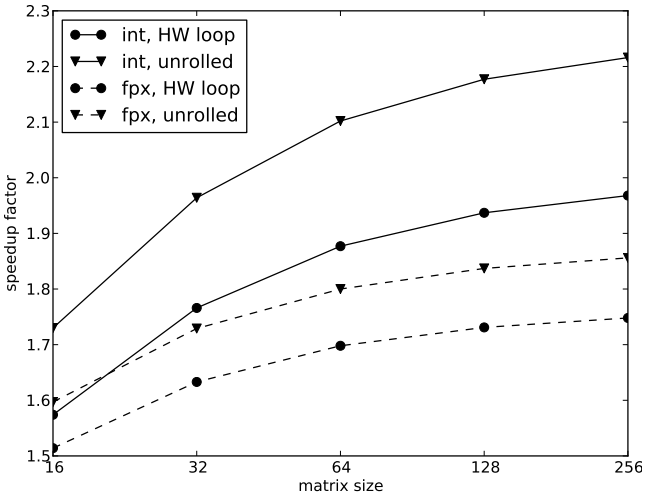

Figure 5. Speedup factor measured, for integer multiplication (plain line) and floating-point multiplication (dashed line), according to the implementation described in section 3.2.

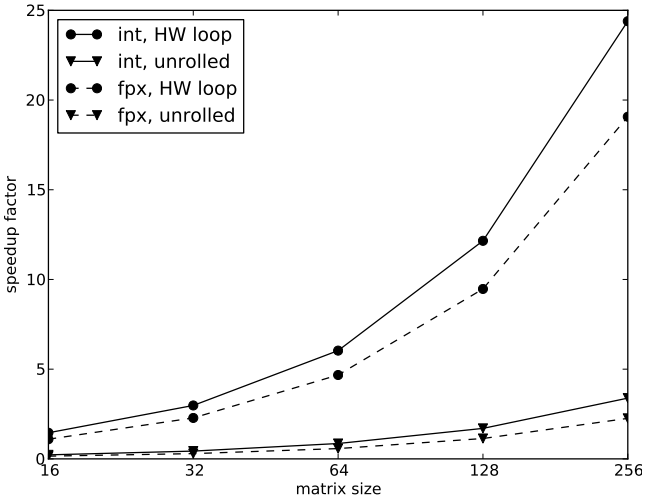

Figure 6. Speedup factor measured, for integer multiplication (plain line) and floating-point multiplication (dashed line), according to the implementation described in section 3.3.

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.

5. Related work

There is an extensive amount of literature about dynamic compilation, mainly related to Just-In-Time compilers (JITs) [1]. JITs dynamically select the parts of the program to optimize without a priori knowledge on the input code. This usually requires to embed a large amount of intelligence in the JIT framework, which means a 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 [4], but the binary code produced often presents a lower performance because of the smaller amount of optimizing intelligence embedded in the JIT compiler [5].

The approach chosen in `deGoal` is similar to partial evaluation techniques [3], 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 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 [2].

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.

6. Conclusion

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 in favorable conditions the performance increase can be huge.

In this paper, we have illustrated the benefits of using `deGoal` to optimize processing kernels. Because `deGoal` is related to the generation of machine binary instructions, its scope 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, the STxP70, and other RISC processors under NDA.

Acknowledgments

The authors wish to acknowledge the support of the EU Commission under the SMECY project (ARTEMIS Joint Undertaking under grant agreement number 100230) in part funding the work reported in this paper.

References

[1] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35: 97–113, June 2003.

[2] H.-P. Charles. Basic infrastructure for dynamic code generation. In H.-P. Charles, P. Clauss, and F. Pétrot, editors, *workshop "Dynamic Compilation Everywhere", in conjunction with the 7th HiPEAC conference*, Paris, France, january 2012.

[3] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 145–156, 1996.

[4] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE '06*, pages 144–153, New York, NY, USA, 2006. ACM.

[5] N. Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Java VM'02*, pages 119–126, Berkeley, CA, USA, 2002. USENIX Association.

[6] STMicroelectronics and CEA. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. In *CMC Research Workshop on STMicroelectronics Platform 2012*, 2010.