Towards a dynamic code generator for run-time self-tuning kernels in embedded applications

Fernando A. Endo, Damien Couroussé and Henri-Pierre Charles, Univ. Grenoble Alpes, F-38000 Grenoble, France / CEA, LIST, MINATEC Campus, F-38054 Grenoble, France

To reduce energy consumption and ensure ISA back-compatibility, high-performance embedded processors are growing with unprecedented complexity. While self-tuning code approaches have been statically used to address desktop- and server-class computing complexity, online approaches are rare. Existing run-time self-tuning tools are only adapted to scientific and data-center workload, in order to amortize the overheads.

This article goes toward the implementation of the first dynamic code generator that enables run-time self-tuning of computing kernels in embedded-class applications. We evaluate `deGoal`, a dynamic code generator, in ARM microprocessors. We discuss and give examples of self-tuning possibilities. The results highlight the potentiality to use `deGoal` to self-tune code in embedded applications.

General Terms: Dynamic code generation, embedded systems, online self-tuning code

ACM Reference Format:
Int. Workshop on Dynamic Compilation V, N, Article A (January 2015), 11 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

High-performance embedded processors are evolving with unprecedented grow in complexity. ISA back-compatibility and energy reduction techniques are among the main reasons. In the past, embedded applications were compiled to only one target architecture, with simpler ISAs compared to current ones. Now, applications do not necessarily run in only one target, one binary code may run in processors from different manufacturers and even in different cores inside a SoC.

Iterative compilation and auto-tuning techniques have been used to address the complexity of desktop- and server-class systems. They show moderate to high performance gains compared to classic compilation, because default compiler options are usually based on the performance benefits observed on generic hardware when they are applied to generic benchmarks. Self-tuning tools have been used to automatically find the best compiler/algorithm optimizations for a given source code and target CPU. Usually, such tools need long space exploration times to find quasi-optimal machine code. Previous work addressed self-tuning at run-time [Voss and Eigenmann 2000] [Tiwari and Hollingsworth 2011] [Chen et al. 2012], however their techniques are only adapted to applications that run for several hours or even days, such as scientific or data center workload, in order to pay off the space exploration overhead and overcome static compilation.

In contrast, embedded applications usually execute during a relatively small amount of time. If the exact target architecture is not know in advance, which is becoming a common place, traditional JIT compilers may not compensate the space exploration overhead at run-time.

This article presents the ongoing work of porting a dynamic code generator, `deGoal`, for high-performance ARM microprocessors. `deGoal` is a lightweight code generator mainly used for dynamic code specialization [Charles et al. 2014]. Because of its low overhead, our tool could also be used to implement online auto-tuning kernels or as a run-time iterative code generator. A preliminary performance validation is presented, by comparing 8 configurations of computing kernels implemented with `deGoal` against static compilation and manual vectorization (PARSEC 3.0 [Bienia 2011] and

ISBN: 978-1-4503-3294-1

PARVEC [Cebrian et al. 2014] benchmark suites). The results highlight the potentiality to use `deGoal` to self-tune computing kernels of embedded-class applications.

2. RELATED WORK

Charles et al. presented the `deGoal` tool and related dynamic code generators [Charles et al. 2014]. This section presents related work of compilation/optimization, iterative compilation and auto-tuning performed at run-time.

Voss and Eigenmann implemented an automatic optimization framework (ADAPT), which could dynamically optimize a running application, without prior user intervention [Voss and Eigenmann 2000]. The dynamic compilation was performed in an external processor or in a free processor inside the computing system. In two heavy workload, running more than one hour on an uniprocessor, they observed speedups between 0 and 40 %, while for a small workload (5 minutes) the dynamic compilation showed a slowdown of more than 10 %.

Tiwari and Hollingsworth presented a run-time compilation and tuning framework for parallel programs (Active Harmony) [Tiwari and Hollingsworth 2011]. By defining a new language, programmers can express tunable parameters and their range, which are explored at run-time. In their experiments, performed in cluster of computers, the code generation is deployed in idle machines, in parallel to the running application. They observed average speedups between 1.11 and 1.14, in two scientific applications run with various problem sizes and in three different platforms.

Chen et al. proposed a framework of iterative optimization for data centers (IODC) [Chen et al. 2012], which is transparent to the user. Their strategy is to control the intensity of space exploration (recompilations and training runs) accordingly to the savings achieved since the beginning of the execution, because the execution time of the workload is assumed to be unknown. For compute intensive workload, IODC achieved an average speedup of 1.14.

Nuzman et al. presented a JIT technology for C/C++ [Nuzman et al. 2013]. Their approach consists in delivering executable files along with an Intermediate Representation (IR) of the source code. In other words, an initial binary code is provided to avoid the overhead of generating it at run-time. Then, a JIT compiler is charged to profile the running code and recompile hot functions in idle cores. In their main experiment, they considered the case where the benchmarks are compiled with moderate optimization levels, arguing that this is a realistic context in the real world. In average their approach reduced the execution time of CINT2006 benchmarks by 7 %, running in a Power7 processor.

Cohen and Rohou propose processor virtualization and split compilation to provide performance portability over heterogeneous multicore embedded systems [Cohen and Rohou 2010]. They believe that those techniques combined can provide a generic solution to allow run-time iterative compilation.

While previous work addressed the run-time iterative compilation, auto-tuning and optimizations in desktop- and server-class processors and workload, no work focused on embedded-class processors and applications. In order to compensate overheads, previous studies targeted long running applications and run-time recompilation was usually off-loaded to remote or idle resources. While this is acceptable in general computing systems, embedded systems have restricted resources and run smaller workload. In such environment, we believe that a lightweight tool should be employed to explore pre-identified optimizations. Our tool targets computing kernels and defines a low-level programming language, which combined can prune the space exploration to pre-identified optimizations and allow very fast code generation.

```
 1  float dist(Point p1, Point p2, int dim)
 2  {
 3      int i;
 4      float result = 0.0;
 5
 6      for (i = 0; i < dim; i++)
 7          result += (p1.coord[i] - p2.coord[i])*
 8                    (p1.coord[i] - p2.coord[i]);
 9
10      return(result);
11  }
```

Fig. 1. Euclidean distance computation in Streamcluster from PARSEC.

```
float dist(Point p1, Point p2, int dim)
{
    float ret;
    int i;
    _MM_TYPE result, _aux, _diff, _coord1, _coord2;

    result = _MM_SETZERO();

    for (i=0;i<dim;i=i+SIMD_WIDTH) {
        _coord1 = _MM_LOADU(&(p1.coord[i]));
        _coord2 = _MM_LOADU(&(p2.coord[i]));

        _diff = _MM_SUB(_coord1, _coord2);
        _aux = _MM_MUL(_diff,_diff);
        result = _MM_ADD(result, _aux);
    }

    ret = (_MM_CVT_F(_MM_FULL_HADD(result, result)));
    return ret;
}
```

Fig. 2. Euclidean distance computation in Streamcluster from PARVEC. SIMD_WIDTH = 4 for ARM.

3. DEGOAL

deGoal implements a domain specific language for dynamic code generation. It defines a pseudo-assembly RISC-like language, which can be mixed with standard C code. The machine code is only generated by deGoal instructions, while management of variables and code generation decisions are implemented by deGoal pseudo-instructions, optionally mixed with C code. Charles et al. presented a detailed introduction of the deGoal tool [Charles et al. 2014].

3.1. Example of euclidean distance computation: PARSEC, PARVEC and deGoal versions

For illustration, we present an example of (squared) euclidean distance computation implemented in the Streamcluster benchmark, originally found in the PARSEC 3.0 suite, manually vectorized in the PARVEC suite and implemented with deGoal.

Figure 1 shows the original C implementation. *Point* is a structure that contains the *dim* coordinates of a point. Between lines 6 and 8, the difference of respective coordinates of two points is squared and accumulated. Figure 2 presents the same kernel implemented in PARVEC. Here, the vectorization is manually implemented by using compiler SIMD[1] intrinsics. In this implementation, the dimension of the points is considered as a multiple of 4 (not shown in the code). Figure 3 details the kernel generator implemented with deGoal, in this case equivalent to PARVEC. Lines 19, 29 and 30 represent the C-equivalent *for* instruction, but here the loop index is reversed

[1]Single instruction multiple data.

```
 1  Begin code Prelude out = weight1, coord1, assign1,
 2      cost1, weight2_, coord2_, assign2_, cost2_, dim_
 3
 4  Type int_t int 32
 5  Alloc int_t i
 6  Alloc int_t coord2
 7
 8  Type fpvec_t float 32 4
 9  Alloc fpvec_t Vc1
10  Alloc fpvec_t Vc2
11  Alloc fpvec_t Vdiff
12  Alloc fpvec_t Vaux
13  Alloc fpvec_t Vresult
14
15      mv coord2, coord2_
16      mv i, dim_
17      mv Vresult, #(0)
18
19      while_ge i, #(4)
20          lw Vc1, coord1
21          lw Vc2, coord2
22
23          sub Vdiff, Vc1, Vc2
24          mul Vaux, Vdiff, Vdiff
25          add Vresult, Vresult, Vaux
26
27          add coord1, coord1, #(4*4)
28          add coord2, coord2, #(4*4)
29          sub i, i, #(4)
30      whileend
31
32  Type fp_t float 32
33  Alloc fp_t result
34
35      add result, Vresult
36
37      mv out, result
38      rtn
39
40  End
```

Fig. 3. Euclidean distance computation (PARVEC-like) with deGoal.

for simplicity. Lines 20 to 25 perform the difference, squaring and sum of coordinates, exactly as lines 10 to 15 in the PARVEC code (Figure 2). In both cases, the computation is performed over vectors of 4 elements, then the final result is the sum of the elements of the result vector, represented in the lines 18 in Figure 2 and 35 in Figure 3.

3.2. Current code generation options

Here, we present the current code generation options in deGoal. Given the broad range of market targeted by ARM cores, the ARM 32-bit architecture support in deGoal allows static or dynamic configuration of the target core. Static configuration is interesting when the target is fixed and its parameters are well known, as in the case of micro-controllers. The dynamic configuration is useful when the target is not known at compile time or when multiple cores are targeted at run-time.

3.2.1. Instruction scheduling. Instruction scheduling plays a very important role in program performance. deGoal embeds one-pass architecture-specific instruction schedulers. For ARM, the execution (EXE) stage is modeled with a configurable number of pipelines, which can contain one or more functional units (FUs). Each assembly instruction is mapped to one FU and is modeled with an issue and result latency.

3.2.2. Instruction selection. Instruction selection options include SIMD/SISD[2] instruction generation, hardware support of divide instructions and instruction compression/decompression.

3.2.3. Register allocation. `deGoal` provides an abstraction layer for register allocation, without loosing performance at the cost of programming efforts. One `deGoal` register is automatically allocated as a single register or as a vector of registers. However, register spilling is not automatic for efficiency. The programmer must ensure that the code generation never fails or return an error if machine code could not be generated.

4. EXPERIMENTAL SETUP

This section details the experimental environment used to evaluate the performance of `deGoal` in ARM microprocessors.

4.1. Evaluation boards

We choose two evaluation boards. The first embeds out-of-order cores, the Snowball SDK[3] (SB), equipped with a dual Cortex-A9 processor [Calao Systems 2011]. It runs the Linaro 11.11 distribution with a Linux 3.0.0 kernel. The second is the BeagleBoard-xM SDK (BB) [BeagleBoard.org 2010] and has an in-order core (Cortex-A8). It runs a Linux 3.9.11 kernel from the Ubuntu 11.04 distribution. In both platforms, the clock frequency was fixed at 800 MHz, and FP instructions execute in the RunFast mode. This mode is not fully IEEE 754 compliant, however the performance comparison of Advanced SIMD and VFP instructions is fairer. Nevertheless, in this mode FP instructions can not bypass results in the BB [ARM 2010].

4.2. Benchmarks and `deGoal` kernels

We implemented with `deGoal` four kernels of two benchmarks, VIPS and Streamcluster, released in the PARSEC 3.0 suite [Bienia 2011] and manually vectorized and released in the PARVEC suite [Cebrian et al. 2014]. In other words, we implemented and compared eight kernels: half use scalar instructions and the other half use SIMD instructions.

In the VIPS benchmark, three kernels were evaluated and the performance was measured per kernel with performance counters. In the Streamcluster benchmark, one kernel was evaluated, and the measured performance corresponds to the execution time of the benchmark, given that the considered kernel accounts for more the 90 % of its execution. We ran the benchmarks using the Simlarge input set. The execution time of the kernels is between 1.7 and 5 seconds in VIPS to a few minutes in Streamcluster. We consider that in this range of duration `deGoal` is adapted to perform run-time code generation.

We compiled the eight reference kernels and the equivalent `deGoal` versions in an Ubuntu 11.04 ARM distribution with a gcc 4.5.2. All kernels are compiled to the Thumb-2 ISA and the ARMv7-A architecture, and all 32 single-precision registers can be used. Other important compiler options are (default PARSEC flags): -O3 -g -funroll-loops -fprefetch-loop-arrays. The generic instruction scheduler in `deGoal` is parametrized with the Cortex-A8 instruction latencies (using the A9 timing produces very close results), because only the A8 and the A9 have publicly available information about cycle timing. To ensure a fair comparison, our instruction scheduler does not model specific instruction behavior of the chosen cores and `deGoal` did not gener-

[2]Single instruction single data.
[3]Software development kit.

Table I. Speedup of deGoal generated kernels compared to PARSEC and PARVEC

Benchmark	Kernel	Beagleboard		Snowball	
		PARSEC	PARVEC	PARSEC	PARVEC
VIPS	Linear transf.	1.61	1.06	1.00	1.00
	Interpolation	1.14	0.59	0.87	0.68
	Convolution	1.03	1.98	1.04	1.60
Streamcluster	Euclid. dist.	1.18	1.00	1.00	0.95
Geometric mean		1.22	1.05	0.98	1.01

ate special instructions rarely used by compilers. These are advantages of a low-level language, and will be explored in the future.

5. EXPERIMENTAL RESULTS

This section presents the experimental results of `deGoal` in ARM microprocessors. First, we validate the raw performance of `deGoal` kernels compared to the reference benchmarks. Then, we show the transparent vectorization enabled by `deGoal` and the performance of dynamically specialized kernels. Finally, we discuss and illustrate the possibilities to use `deGoal` to self-tune kernels.

5.1. Validation

We present a preliminary validation of `deGoal` for ARM microprocessors. The objective is to evaluate the raw performance of machine code generated by `deGoal`, by translating kernels found in benchmarks into the `deGoal` language and comparing their performance.

Table I presents the speedup of `deGoal` versions compared to the reference kernels. In the BB, `deGoal` is in average 22 % faster than PARSEC, thanks to simple and efficient optimizations such as loop unrolling and ILP, allowed by a low-level language. Compared to PARVEC, `deGoal` is 5 % faster in average. In the SB, the dynamic scheduling pipeline probably hid under-optimized code, and the average `deGoal` performance virtually matched those of PARSEC and PARVEC.

In the following, we detail the code and performance differences of each kernel.

5.1.1. Linear transformation. This kernel applies a linear transformation on a buffer. The original code is implemented with two nested loops, the first iterating over the buffer elements and the second over the image layers. In the PARSEC version, the inner loop is partially unrolled for different factors, while with `deGoal` we partially unroll three times (common assumption of a three-layer image being processed). The excess instructions generated by the compiler to partially unroll for different factors explain why `deGoal` is 61 % faster in the BB. On the other hand, in the SB, there's no performance difference between the PARSEC and `deGoal` versions. It's likely that the out-of-order execution could hide the latency of the extra instructions by executing them during cycles that otherwise would be stalled in an in-order pipeline. The PARVEC version specializes the number of image layers to completely unroll the inner loop three times. Even comparing to a manually unrolled loop, `deGoal` is 6 % faster in the BB and matches the PARVEC performance in the SB.

5.1.2. Interpolation. The interpolation kernel is a small piece of code which performs a bilinear interpolation of an image point from four neighbor pixels. There's only one loop that iterates over the image layers. In the PARSEC version, the loop is also partially unrolled for different factors, which explains again the 14 % of speedup of `deGoal` in the BB. However, in the SB, `deGoal` slows down the execution by 13 %, which is explained by function inlining performed by gcc. If we force the no-inlining of the this small kernel, `deGoal` matches the performance of PARSEC in the SB. Regarding the PARVEC comparison, besides the performance degradation from the function inlining that `deGoal` can not perform, the manually vectorized version uses a trick to improve performance: in some vectors of three elements, it loads one extra element out-of-bound. This optimization reduces the number of load instructions. Given that `deGoal` can not perform such optimization yet, it contributes to a slowdown of 41 and 32 % in the BB and SB, respectively.

5.1.3. Convolution. The convolution kernel is a separable integer convolution. Instead of performing a full 2D convolution, this technique first applies a 1D mask and then the same mask rotated by 90°. In our evaluation, only the first step is compared, which is a kernel by itself with two nested loops. In PARSEC, the inner loop is unrolled by using a technique called *Duff's Device* [Duff 1983]. It partially unrolls a loop 16 times by using a switch-case statement in C. The main advantage of employing this technique is that there's no need to process leftover iterations in a separate loop. `deGoal` does not support such implementation, then we had to implement a main (partial-unrolled) and a leftover loop. The only advantage over the Duff's Device is that instructions may be better scheduled in the main loop. The results show that `deGoal` is only 3 to 4 % faster than the C version. In PARVEC, the technique of partial unrolling and leftover processing is used. Surprisingly, `deGoal` is 1.60 to almost 2 times faster than the manually vectorized code, but this performance comes from an under-optimized vectorization: this kernel processes integer elements, however in PARVEC the actual computation is performed in floating-point precision, which results in performance degradation because of integer to FP conversion, and vice-versa.

5.1.4. Euclidean distance. The PARSEC implementation of this kernel is shown in Figure 1. As explained in section 3.1, it computes the euclidean distance of two points using FP precision. The only loop iterates over the dimension of the points which is a benchmark parameter. As in the previous kernels, gcc partially unrolls the loop up to eight times. Without any information about the dimension of the points, with `deGoal` we decided to partially unroll the loop four times, in order to match it with the ARM SIMD width. This value will allow us to demonstrate the capability of the transparent vectorization provided by `deGoal`, explored in the following sections. Another difference is that the vector-like processing in `deGoal` increases the ILP[4]. While gcc generates a code that reuses one register to accumulate the partial result, with `deGoal` we accumulate those results in different registers. In the BB, this produces a speedup of 18 %, specially because the increased ILP compensates the impossibility to bypass FP results (c.f. section 4.1), while in the SB no speedup was observed, because the bypass restriction does not apply to the Cortex-A9 FP pipeline. Comparing with PARVEC, `deGoal` shows no speedup in the BB and a slow-down of 5 % in the SB. Here, the generic loop unrolling performed by gcc shows its advantage: with `deGoal` we did not unroll the loop, given that no information about the dimension was known, while gcc partially unrolls the loop up to eight times.

5.2. Transparent vectorization: SISD vs SIMD code generation

In this section, we demonstrate the transparent vectorization enabled by `deGoal`. Since we implemented kernel generators with vector-like processing, it's possible to generate SISD or SIMD instructions just by toggling a code generation flag. In other words, the PARSEC-like kernels written with `deGoal` in Section 5.1 can transparently generate SIMD instructions.

Table II shows the speedup of the SIMD generated kernels compared to their SISD versions. We compare to the SISD versions because no extra programming effort is

[4]Instruction-level parallelism.

Table II. Speedup of `deGoal` PARSEC-like kernels when generating SIMD code compared to SISD code generation

Benchmark	Kernel	Beagleboard	Snowball
VIPS	Linear transf.	1.22	0.89
	Interpolation	1.78	1.14
	Convolution	0.96	1.05
Streamcluster	Euclid. dist.	1.99	1.33
Geometric mean		1.43	1.09

Table III. Parameters dynamically specialized by `deGoal`

Benchmark	Kernel	Parameter	Optimizations
VIPS	Linear transf.	Number of image layers.	Completely unroll the inner loop and partially unroll the outer loop.
	Interpolation	Number of image layers.	Completely unroll the main loop.
	Convolution	Convolution mask size.	Depending on the number of registers available, try to keep mask coefficients in registers.
Streamcluster	Euclid. dist.	Dimension of points.	Uses as many registers as possible to unroll loops and conditionally generates a leftover loop, if needed.

Table IV. Speedup of `deGoal` specialized kernel versions over non-specialized ones

Benchmark	Kernel	Beagleboard		Snowball	
		SISD	SIMD	SISD	SIMD
VIPS	Linear transf.	1.14	1.88	1.55	1.64
	Interpolation	1.16	1.25	1.05	1.27
	Convolution	1.22	1.18	1.26	1.11
Streamcluster	Euclid. dist.	1.10	0.98	1.22	1.00
Geometric mean		1.15	1.28	1.26	1.23

Table V. `deGoal` run-time overhead (% of kernel run-time) to generate the dynamic specialized kernels

Benchmark	Kernel	Beagleboard		Snowball	
		SISD	SIMD	SISD	SIMD
VIPS	Linear transf.	0.0061	0.0092	0.0103	0.0084
	Interpolation	0.0085	0.0147	0.0121	0.0171
	Convolution	0.0064	0.0053	0.0058	0.0048
Streamcluster	Euclid. dist.	0.0004	0.0006	0.0005	0.0005

needed to vectorize the code. In average, the SIMD versions are 43 % faster in the BB, and only 9 % in the SB. The observed speedup differences and slowdowns are discussed in the section 5.4.1.

5.3. Dynamic code specialization

One of the main usage of `deGoal` is as a fast dynamic code specializer [Charles et al. 2014]. To demonstrate this capability, we dynamically specialize parameters of the four benchmarks studied. Those parameters depend on the inputs of the benchmarks, but they are constants during the whole execution. Table III describes the specialized parameters and the optimizations performed.

The implemented kernel generators are parametrizable in two aspects: they not only specialize code given a kernel parameter, but also maximize vector sizes and loop unrolling factors, by taking into account the number of available registers. For example, in the euclidean distance kernel (shown in Figure 1), we specialize the dimension of the point, which is 128 in the Simlarge input set. We need at least three vectors: two to hold the coordinates of the two points and one to hold the partial sum. The BB and SB have 16 quadword (total of 64 single-precision elements) and 32 single-precision registers. Then, in this case, when SISD code generation is activated, the length of the allocated vectors is 10, while for the SIMD code generation, it's 20. These lengths are automatically calculated and adapted to the dimension parameter.

The speedups of the dynamic specialized versions compared to the generic PARSEC-like `deGoal` versions are shown in Table IV. In average, the dynamic code specialization provides speedups of 15 % and 28 % in the BB for SISD and SIMD codes, respectively. In the SB, the mean speedups are 26 % (SISD) and 23 % (SIMD). The run-time code generation overhead is negligible as Table V shows. The observed speedups comes from the reduced number of instructions generated, better ILP and instruction scheduling, thanks to loop unrolling.

5.4. Towards a dynamic code generator for self-tuning kernels

In this section, we discuss and illustrate the possibilities to use `deGoal` to self-tune kernels at run-time. As shown in Table V, `deGoal` has a very low run-time overhead. For instance, if we limited the code regeneration to 1 % of the kernel execution time, in the worst case, it would be possible to generate 58 different interpolation kernel versions (this kernel executes during only 1.8 second). For Streamcluster, the most favorable case, `deGoal` could generate 2500 versions. It's worth observing that these measured overheads are per kernel and per core, which highlights the scalability of our approach to multi-threaded applications in multicore systems. In the following, we discuss and give examples of code generation options that `deGoal` could automatically explore. One auto-tuning example is also presented.

5.4.1. SIMD vs SISD. ISA-compatible cores must ensure instruction compatibility, but how instructions are implemented and what performance they deliver may not be available to programmers and compilers. For instance, the performance of vectorized code compared to non-vectorized one depends on various factors, including the target architecture and machine-specific code optimizations. For example, in Table II the linear transformation kernel has a speedup of 22 % in the BB, but a slowdown of 11 % in the SB. SIMD performance depends heavily on micro-architectural features, such as the width of the data bus connecting the L1-D cache to the pipeline. In the BB, this bus is 128-bit wide for NEON and 64-bit for the ARM pipeline, while in the SB both are only 64-bit wide, which explains why the BB shows greater SIMD speedups in average. The only exception is the convolution kernel, which had a slowdown of 4 % in the BB, but a speedup of 5 % in the SB. In this case, it's possible that some SIMD instructions are less efficiently implemented in the BB.

5.4.2. Auto-tuning example. In the specialized version of the euclidean distance kernel, we modified the kernel generator to make the vector length parametrizable. For SIMD instructions, we can vary them from 4 to 20, in steps of 4. In both platforms, we observed that the best length is 16: this granularity provided an extra speedup of 8.0 % in the BB and 4.1 % in the SB, over the speedups shown in Table IV. No dynamic decision is taken here, `deGoal` in this example behaves as a code generator for off-line auto-tuning.

5.4.3. Instruction scheduling. In high-performance processors, full bypass logic can be costly in area and energy dissipation and can put pressure in the cycle time [Ahuja et al. 1995]. In consequence, usually only partial or critical operand bypasses are implemented. While out-of-order pipelines can dynamically reorder instructions and avoid pipeline stalls caused by missing bypass paths, in-order pipelines must rely on compiler scheduling to avoid such stalls. However, if we consider the growing complexity of embedded processors, detailed models are rarely published and, even if they were, using them in static compilers can lead to unaffordable compilation times. Differently from server- and desktop-class processors, the trend in embedded processors is to take advantage of in-order cores to reduce the energy consumption. If we also take into account the increasing diversity of ISA-compatible embedded core implementations, it's clear the statically compiled binaries only provide sub-optimal instruction scheduling. To better take advantage of core heterogeneity, we believe that dynamically configuring and eventually auto-tuning the instruction scheduler at run-time can provide energy savings, specially in in-order cores.

6. CONCLUSION

In this article, we evaluated `deGoal`, a dynamic code generator, for ARM microprocessors. We showed the performance gains when `deGoal` is used as a dynamic code specializer and discussed the possibility to use our tool to self-tune computing kernels at run-time in embedded environment.

In this work, `deGoal` generated the same code for two platforms, because they support the same ISA and have similar EXE stage configurations. However, future embedded processors will likely have various heterogeneous cores, with different pipeline configurations, accelerators and ISA-specific instructions, not to mention the different designs of different manufacturers. In this scenario, machine-specific code optimizations and self-tuning may play a very important role, specially if the target core is only known at run-time.

REFERENCES

AHUJA, P. S., CLARK, D. W., AND ROGERS, A. 1995. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. MICRO 28. IEEE Computer Society Press, Los Alamitos, CA, USA, 36–45.

ARM. 2010. *Cortex-A8 Technical Reference Manual*. Revision: r3p2.

BEAGLEBOARD.ORG. 2010. *BeagleBoard-xM Rev C System Reference Manual*. Revision 1.0.

BIENIA, C. 2011. Benchmarking modern multiprocessors. Ph.D. thesis, Princeton University.

CALAO SYSTEMS. 2011. *SKY-S9500-ULP-CXX (aka Snowball PDK-SDK) Hardware Reference Manual*. Revision 1.0.

CEBRIAN, J., JAHRE, M., AND NATVIG, L. 2014. Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. 66–75.

CHARLES, H.-P., COUROUSSÉ, D., LOMÜLLER, V., ENDO, F. A., AND GAUGUEY, R. 2014. deGoal a tool to embed dynamic code generators into applications. In *Compiler Construction*, A. Cohen, Ed. Lecture Notes in Computer Science Series, vol. 8409. Springer Berlin Heidelberg, 107–112.

CHEN, Y., FANG, S., EECKHOUT, L., TEMAM, O., AND WU, C. 2012. Iterative optimization for the data center. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. ACM, New York, NY, USA, 49–60.

COHEN, A. AND ROHOU, E. 2010. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Proceedings of the 47th Design Automation Conference*. DAC '10. ACM, New York, NY, USA, 102–107.

DUFF, T. 1983. Tom Duff on Duff's Device.

NUZMAN, D., ERES, R., DYSHEL, S., ZALMANOVICI, M., AND CASTANOS, J. 2013. JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Archit. Code Optim. 10,* 4, 59:1–59:25.

TIWARI, A. AND HOLLINGSWORTH, J. K. 2011. Online adaptive code generation and tuning. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. IEEE Computer Society, Washington, DC, USA, 879–892.

VOSS, M. J. AND EIGENMANN, R. 2000. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*. ICPP '00. IEEE Computer Society, Washington, DC, USA, 163–.