DYNAMICALLY GENERATING FFT CODE ON MOBILE DEVICES

Anthony M. Blake

Department of Computer Science
The University of Waikato
Hamilton, New Zealand

ABSTRACT

This paper characterizes the benefits of dynamic code generation when computing the discrete Fourier transform (DFT) on mobile devices. A library called FFTS has recently been shown to be faster than FFTW, Intel IPP and Apple vDSP, partly due to the use of program specialization and dynamic code generation. However, dynamic code generation is prohibited on some mobile platforms for security reasons. In this work, FFTS was modified to avoid dynamic code generation, while making every effort to maximize performance, and the results of benchmarks on Apple A4, A6, Nvidia Tegra3 and Samsung Exynos4 based devices show that disabling dynamic code generation in FFTS decreases performance by as much as 25%, depending on the device and the parameters of the transform.

Index Terms— Mobile Computing, Fast Fourier Transforms, Digital Signal Processing, Dynamic Compiler

1. INTRODUCTION

Recent benchmark results have shown that a library called FFTS ("The Fastest Fourier Transform in the South") represents the state-of-the-art when computing the discrete Fourier transform (DFT) on ARM based mobile devices [1]. FFTS has been benchmarked on a range of recent Intel x86 and ARM machines, and is, in almost all cases, faster than self-tuning libraries such as FFTW ("The Fastest Fourier Transform in the West"), and even vendor-tuned libraries such as Intel Integrated Performance Primitives (IPP) and Apple vDSP (ibid.). Furthermore, FFTS binaries are smaller than those of other libraries; compared to FFTW, two orders of magnitude smaller.

The performance and size of FFTS is, at least in part, due to the use of program specialization and dynamic code generation. When FFTS initializes a specific transform at run time, e.g., an audio processing application might require a forward real-valued 1024 point 1-D transform, it performs the calculations that depend on the parameters which have just been fixed, and generates specialized machine code to compute the remainder of the calculations which depend on the data. The idea of specializing program was formulated and proven by Kleene more than 50 years ago [2].

Despite the benefits afforded by dynamic code generation, Apple and Microsoft's mobile app stores have a policy of rejecting apps that use dynamically generated code, ostensibly to improve security and prevent certain exploits. Android apps, in contrast, do not have the same restrictions.

In this work, FFTS is modified to work without the use of dynamic code generation, thus allowing it to be used on platforms with restrictive security features. When modifying the code, every effort was made to maximise performance. The resulting code is compared to the original FFTS in benchmarks, and the difference in performance illustrates the performance cost of security features which prohibit dynamic code generation.

2. BACKGROUND

Self-tuning libraries such as FFTW [3, 4, 5] and UHFFT [6] employ a planner to search the space of all possible factorizations of several highly parameterized FFT algorithms to find a plan that has the smallest execution time. Each plan is composed of references to blocks of straight-line code, called codelets, which are automatically generated and optimized at a low level during compilation. FFTW has a library of over 150 codelets, each corresponding to a subtransform which may be as large as 128 points.

FFTW and UHFFT employ dynamic programming [7] to search the space of possible factorizations, exploiting the fact that each plan is divided into subproblems and each of the subproblems considered during the search is essentially the same. The primary difference between FFTW and UHFFT is that UHFFT performs some calibration and initializes a database of execution times during installation, while FFTW only performs calibration at run time.

More recently, it has been demonstrated that the performance of self-tuning libraries such as FFTW is primarily due to the use of program specialization and highly optimized codelets, rather than machine-specific calibration [8]. In these experiments, specialized code for a range of transforms was statically generated, using several automatically generated codelets, and performance of the resulting code was, in almost all cases, faster than FFTW and even vendor-tuned libraries on a range of x86 and ARM machines (ibid.).

However, although the experiments with statically generated code demonstrated that the relationship between machine-specific calibration and performance was somewhat tenuous, there were some limits to the practicality of statically generating specialized code for specific transforms. First, the parameters of the transforms required by an application had to be known at compile time, and second, if a large range of transforms was required, or even just one very large transform, the size of the binary became an issue.

In previous work, the limitations of statically generated code were overcome by performing program specialization at run time and dynamically generating specialized machine code, and the resulting library was called FFTS [1]. Several small, hand optimized codelets coded in assembly are dynamically modifed and used to compose specific transforms at run time, and at least some FFTS's performance can be attributed to the quality of these small codelets, independent of the fact that the code is being dynamically specialized.

In this work, FFTS is further developed so as to have the option of disabling dynamic code generation, in a way that has minimal impact on performance. Benchmark experiments on two iOS devices

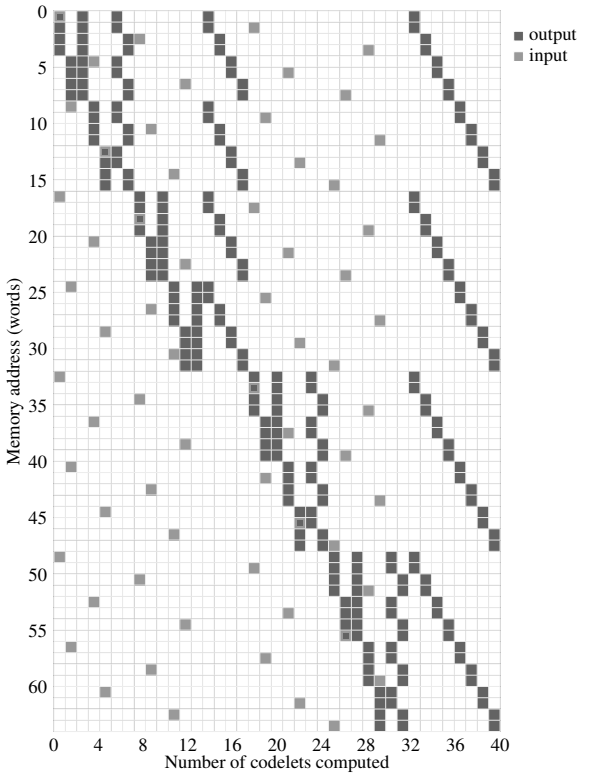

Fig. 1. Memory access pattern of a size 64 decimation-in-time depth-first recursive conjugate-pair FFT operating out-of-place. The loops have been vectorized. Figure 2 shows the same transform following modifications to increase memory locality.

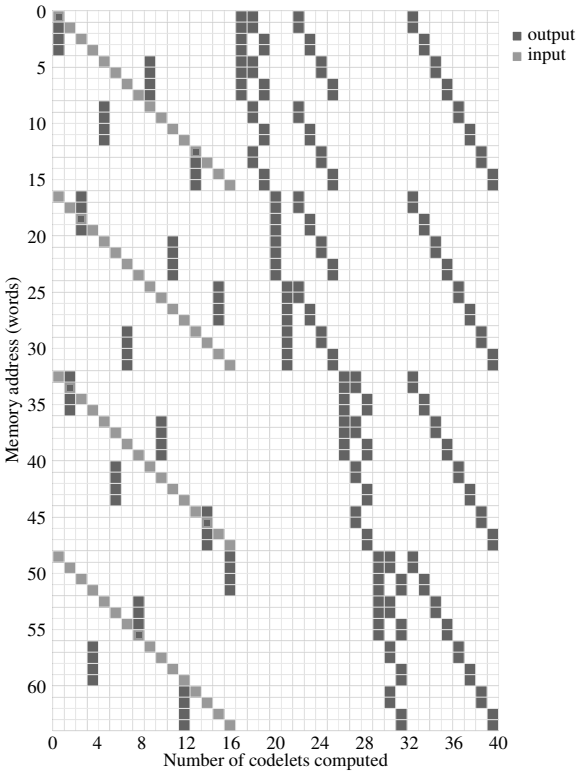

Fig. 2. Memory access pattern of a modified size 64 depth-first recursive conjugate-pair FFT, where the leaves of the computation are sorted to improve spatial locality and computed before the rest of the transform. The leaves of the computation can now be computed iteratively with three loops, which are easily vectorized. Figure 1 shows the same transform without modification.

and two Android devices are performed to evaluate the performance cost of the changes.

3. THE FASTEST FOURIER TRANSFORM IN THE SOUTH

FFTS uses a variant of the split-radix algorithm [9] called the conjugate-pair algorithm [10]. It was first described with the claim that it had set a new record for the minimum number of floating point operations required to compute the FFT, but this was later shown to be incorrect [11, 12, 13]. More recently, James van Buskirk used it as the basis for the Tangent FFT, which does actually reduce the operation count [14].

Rather than computing the FFT iteratively, as most traditional implementations do, FFTS uses a depth-first recursive implementation, which has theoretical advantages in terms of cache utilization, and has been shown to be asymptotically optimal [15].

A point of difference between FFTS and other depth-first recursive implementations is that FFTS first computes the sub-transforms at the leaves of the computation iteratively, before computing the rest of the transform, in order to improve the spatial locality of memory accesses [1]. Figure 1 shows the memory access pattern of an out-of-place decimation-in-time (DIT) algorithm computed recursively, while Figure 2 shows the memory access pattern of the same transform if the leaves are first computed iteratively such that the accesses to the input data are contiguous.

When initializing a specific transform at run time, FFTS emits code for computing the leaves of the transform iteratively, and then emits a sequence of function calls to compute each of the loops in the rest of the transform.

Because the conjugate-pair algorithm decomposes a transform into smaller sub-transforms of two different sizes, two different sizes of sub-transforms occur at the leaves of the computation. When two adjacent sub-transforms of the smaller size are combined, and all the sub-transforms are sorted according to the input addresses they access, the two types of leaf sub-transform are partitioned into three groups of approximately equal size. Thus these sub-transforms can be computed iteratively with three loops.

For example, in Figure 2 the first 16 codelets correspond to the 16 sub-transforms at the leaves of the recursion. The first 6 codelets compute size 4 sub-transforms, the next 5 codelets consist of two size 2 codelets combined, and the final 5 codelets are again size 4.

After emitting code for the leaves of the transform, the rest of the computation is flattened into a sequence of function calls, each corresponding to the loop of sub-transforms computed at each node of the recursion.

Where possible, constants are pushed into the immediates of instructions, and the resulting dynamically generated function for a specific transform only uses the stack to save registers in the prologue and epilogue, and is not used inside the function.

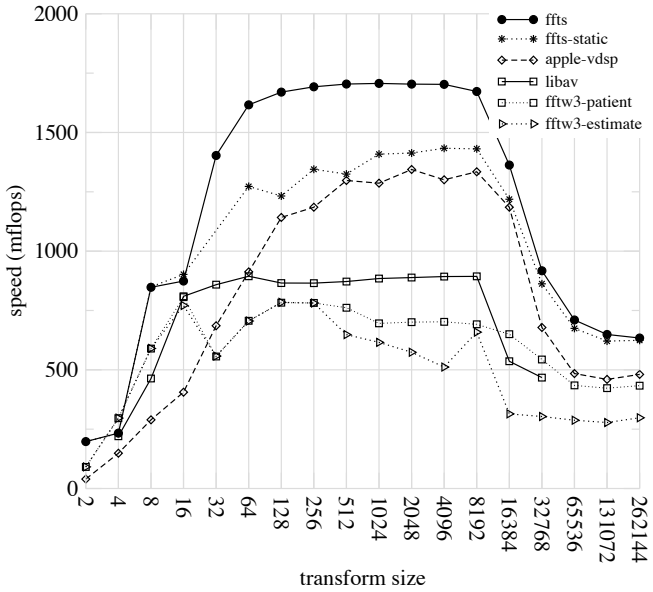

Fig. 3. Performance of FFT code on an iPod Touch 4G, which uses the Apple A4 single-core SoC.

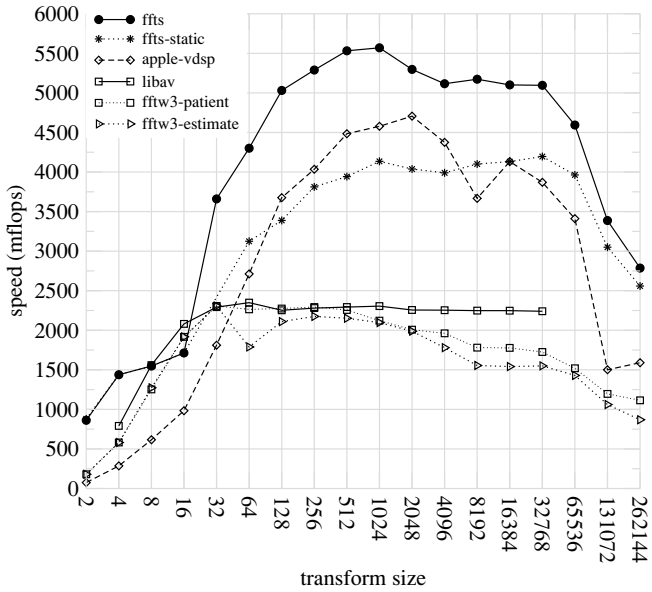

Fig. 4. Performance of FFT code on an iPhone 5, which uses the Apple A6 dual-core SoC.

4. DISABLING DYNAMIC CODE GENERATION

Dynamic code generation was disabled by replacing the generation of specialized functions at run time with the general functions. As described in the previous section, FFTS computes the transform in two parts: first, the sub-transforms at the leaves of the computation are computed iteratively, and second, the rest of the transform is computed recursively.

The sign of the primitive root-of-unity is a parameter which frequently occurs within the body of the codelets, and it has a negative impact on performance if it is a parameter implemented with either conditional logic or as a variable that is unconditionally exclusive or'ed with data to change the sign. Because there are only two possible assignments for the sign, it was determined to be an acceptable performance vs. code size tradeoff to implement two copies of all the code: one copy which is hardcoded for forwards transforms, and the other which is hardcoded for inverse transforms. During initialization, the sign is used to determine which code path should be assigned to a function pointer.

Due to the skewed nature of the conjugate-pair decomposition, the counts of the two types of sub-transforms found at the leaves of the recursion oscillate as the size of the transform is increased. This effects the relative placement of a residual sub-transform which doesn't evenly divide into the loops (when using SIMD, several iterations are computed in parallel). In this case, the use of conditional logic was again avoided in favour of two code paths in order to maximise performance.

5. RESULTS

Two versions of FFTS were benchmarked on four ARM based mobile devices, along with a few other libraries for reference, using the benchmark methods described in [8]. The benchmarks presented here are configured to use NEON and be single threaded, out of place, complex to complex and are in one dimension.

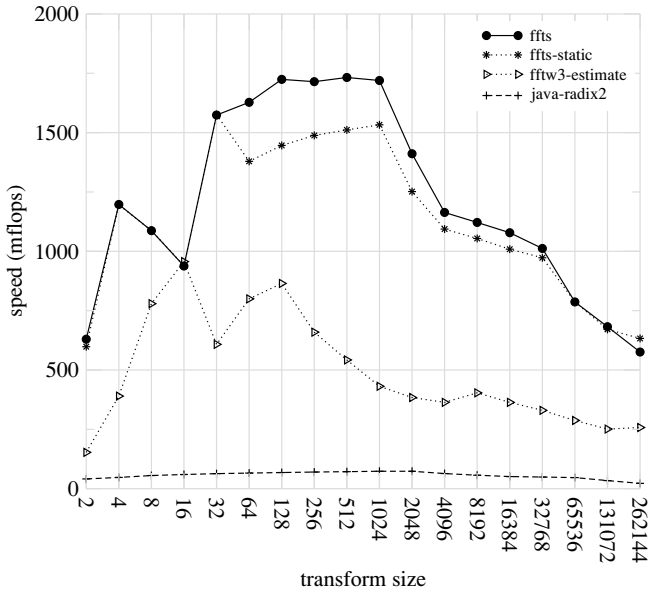

Fig. 5. Performance of FFT code on an Asus Eee Pad TF201, which uses the Nvidia Tegra 3 quad-core SoC.

The code which uses dynamic code generation is labeled as 'ffts', while the code described in Section 4 is labeled as 'ffts-static'.

Figures 3 and 4 plot the performance of code running on two Apple devices. Applications which use dynamic code generation can be developed and tested on iOS based devices, however Apple's policy is to reject these apps if they are submitted to the App Store for distribution.

It can be noted that FFTS running on the A6, either in static or dynamic mode, can compute large transforms faster than some workstations, for example, Intel IPP running on an Intel Core 2 Duo P8600 only manages about 2500 MFLOPS for a size 262144 trans-

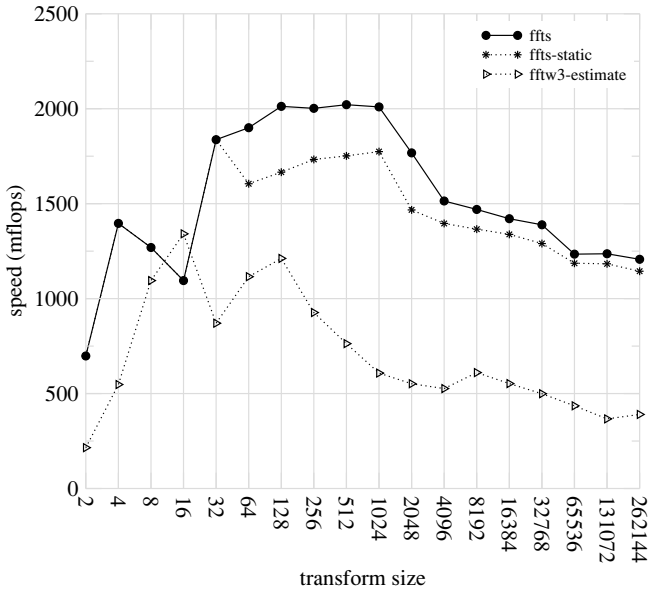

Fig. 6. Performance of FFT code on a Samsung Galaxy S III, which uses the Exynos 4 quad-core SoC.

Name of FFT library	Size (kilobytes)
FFTW3	2260
FFTS	33
FFTS-static	44

Table 1. Size of FFT libraries compiled for Android.

form.

Figures 5 and 6 plot the performance of code running on two Android based devices. FFTW in patient mode did not run correctly on these devices, and was taking several hours to create plans larger than a few hundred points.

Comparing Figures 4 and 6, it can be seen that single thread performance on the iPhone 5 is much faster than the Samsung Galaxy S III, however it should be noted that the Exynos 4 SoC in the Galaxy S III is quad core.

More code is required for the static implementation of FFTS, as shown in Table 1. However, compared to FFTW3, the static implementation is still two orders of magnitude smaller.

The fact that the static implementation of FFTS is faster than other libraries suggests that FFTS has an algorithmic advantage.

6. CONCLUSIONS AND FUTURE WORK

In this work, FFTS was modified to support operation on platforms where dynamic code generation is prohibited for security reasons, which includes iOS and Windows Phone 8. The results of benchmark experiments show that removing dynamic code generation decreases performance by as much as 25%, depending on the device and the size of the transform.

FFTS currently computes complex, real-valued and multi-dimensional transforms, but only for power-of-two sizes, and future work will apply the techniques described in this paper to arbitrary sized transforms and multi-threaded transforms.

FFTS has been released as open source code under a permissive license,[1] and currently runs on Linux, OS X, iOS, Android and Windows Phone 8, on the Intel x86 and ARM architectures.

7. ACKNOWLEDGMENT

I am grateful to Ian Witten and Craig Taube-Schock for allowing me the use of their smartphones to run the benchmarks in this paper.

8. REFERENCES

[1] A.M. Blake, I.H. Witten, and M.J. Cree, "The fastest Fourier transform in the south," *IEEE Trans. on Signal Processing*, (submitted to).

[2] S.C. Kleene, NG de Bruijn, J. de Groot, and A.C. Zaanen, "Introduction to metamathematics," 1952.

[3] M. Frigo and S.G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*. IEEE, 1998, vol. 3, pp. 1381–1384.

[4] M. Frigo and S.G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[5] S. G. Johnson and M. Frigo, "Implementing FFTs in practice," in *Fast Fourier Transforms*, C. S. Burrus, Ed., Connexions, chapter 11. Rice University, Houston TX, September 2008.

[6] D. Mirkovic and L. Johnsson, "Automatic performance tuning in the UHFFT library," *Lecture notes in computer science*, pp. I–71, 2001.

[7] R.L. Rivest and C.E. Leiserson, "Introduction to algorithms," 1990.

[8] A.M. Blake, *Computing the fast Fourier transform on SIMD microprocessors*, Ph.D. thesis, University of Waikato, New Zealand, 2012.

[9] R. Yavne, "An economical method for calculating the discrete Fourier transform," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 1968, pp. 115–125.

[10] I. Kamar and Y. Elcherif, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 25, pp. 324, 1989.

[11] R. A. Gopinath, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 25, pp. 1084, 1989.

[12] A. M. Krot and H. B. Minervina, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 28, pp. 1143, 1992.

[13] H-S. Qian and Z-J. Zhao, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 26, pp. 541, 1990.

[14] D. Bernstein, "The tangent FFT," *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pp. 291–300, 2007.

[15] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[1] Available at `http://www.cs.waikato.ac.nz/~ablake/ffts`