Relating FFTW and Split-Radix*

Oleg Kiselyov[1] and Walid Taha[2]

[1] Monterey, CA 93943. (oleg@okmij.org)
[2] Department of Computer Science, Rice University. (taha@rice.edu)

Abstract. Recent work showed that staging and abstract interpretation can be used to derive correct families of combinatorial circuits, and illustrated this technique with an in-depth analysis of the Fast Fourier Transform (FFT) for sizes 2^n. While the quality of the generated code was promising, it used more floating-point operations than the well-known FFTW codelets and split-radix algorithm. This paper shows that staging and abstract interpretation can in fact be used to produce circuits with the same number of floating-point operations as each of split-radix and FFTW. In addition, choosing between two standard implementations of complex multiplication produces results that match each of the two algorithms. Thus, we provide a constructive method for deriving the two distinct algorithms.

1 Introduction

Hardware description languages are primarily concerned with resource use. But except for very high-end applications, verifying the correctness of hardware systems can be prohibitively expensive. In contrast, software languages are primarily concerned with issues of expressivity, safety, clarity and maintainability. Software languages provide abstraction mechanisms such as higher-order functions, polymorphism, and general recursion. Such abstraction mechanisms can make designs more maintainable and reusable. They can also keep programs close to the mathematical definitions of the algorithms they implement, which helps ensure correctness. Hardware description languages such as VHDL [7] and Verilog [14] provide only limited support for such abstraction mechanisms. The growing interest in reconfigurable hardware invites us to consider the integration of the hardware and software worlds, and to consider how verification techniques from one world can be usefully applied in the other. Currently, programming reconfigurable hardware is hard [1]: First, software developers are typically not trained to design circuits. Second, specifying circuits by hand can be tedious, error prone, and difficult to maintain. The challenge in integrating both hardware and software worlds can be summarized by a key question:

> *How can we get the raw performance of hardware without giving up the expressivity and clarity of software?*

Program generation [4, 3] provides a seed for an answer to this question: it gives us the full power of a high-level language to generate descriptions of hardware (or any other

* Supported by NSF ITR-0113569 "Putting Multi-stage Annotations to Work" and Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers."

kind of resource-bounded computation). Each generator represents a *family* of circuits. The practical benefit is a high-level of flexibility and reuse. The research challenge lies in finding analysis and verification techniques that can check and reason about this full family of circuits just by looking at the generator, and without having to generate all possible circuits. Resource-aware Programming (RAP) languages [12, 5] are designed to address these problems by providing:

1. A highly expressive untyped substrate supporting features such as dynamic data-structures, modules, objects, and higher-order functions.
2. Constructs that allow the programmer to express the *stage distinction* between computation on the development platform and computation on the deployment platform.
3. Advanced static type systems to ensure that computations intended for execution on resource-bounded platforms are both type-safe and resource-bounded *without* generating all possible programs.

Developing a program generator in a RAP language proceeds as follows:

1. Implement the input-output behavior in an expressive, type-safe language such as OCaml [6]. For FFT, this step is just implementing the Cooley-Tukey recurrence.
2. Verify the correctness of the input-output behavior of this program. Because we used an expressive language, this step reduces to ensuring the faithful implementation of the textbook algorithm and the correct transformation of the program into a monadic style. The monadic transformation is well-defined and mechanizable [8].
3. Determine which parts of the computation can be done on the development platform, and which must be left to the deployment platform (cf. [11]).
4. Add staging annotations. In this step, staging constructs (hygienic quasi-quotations) ensure that this is done in a semantically transparent manner. Staging a program turns it into a program generator. A two-level type system understands that we are using quasi-quotations to generate programs (cf. [13]) and can guarantee that there are no inconsistent uses of first- and second-stage inputs. A RAP type system [12, 5] goes further and can ensure that second-stage computations only use features and resources that are available on the target platform. The source code of the resulting *generator* is often a concise, minor variation on the result of the first step.
5. Use abstract interpretation techniques [2] to improve *generated code*, by shifting more computations from the generated code to the generator.

1.1 Contributions

The use of abstract interpretation in the RAP process (Step 5 in the above) was only recently proposed and investigated [5]. It provides a safe, systematic means for augmenting the generator with knowledge of domain-specific optimizations, and is a key difference between RAP and previous approaches to program generation including those used in the widely popular FFTW [3]. Despite the merits of this approach (see [5]), the number of arithmetic operations in RAP generators for FFTW circuits only approached those in implementations generated by FFTW.

This paper shows that the technique of [5] can in fact be used to produce circuits with the same number of floating-point operations as in FFTW. In addition, choosing a different standard implementation of complex multiplication gives us circuits with

the same number of floating-point operations as in split-radix FFT, another optimal FFT algorithm [9]. Thus, we provide a new, constructive method to deriving the two different classes of algorithms.

2 Matching FFTW and Split-radix

Previous work [5] uses staging and abstract interpretation to construct a concise generator of combinatorial FFT circuits. The starting point is the textbook decimation-in-time FFT $F(N,x)_k$ of the N-point complex-valued sample x_j

$$\begin{array}{ll} F(1,x)_0 & = x_0 \\ F(N,x)_k & = F(\frac{N}{2},E(x))_k + F(\frac{N}{2},O(x))_k \cdot w_N^k \\ F(N,x)_{k+\frac{N}{2}} & = F(\frac{N}{2},E(x))_k - F(\frac{N}{2},O(x))_k \cdot w_N^k \end{array} \text{ where } \begin{array}{l} E(x)_j = x_{2j} \\ O(x)_j = x_{2j+1} \end{array}$$

where $w_N^k = e^{-i2\pi k/N}$ is the Nth root of unity. E and O split the input x into even and odd parts, respectively. The transform is first applied to both halves, recursively, and the result is combined to yield the two halves of the transform sequence. Given the sample size N, the generator code literally follows this recursive algorithm. But instead of performing multiplications and additions, staging constructs [11] are used to generate a circuit that performs these computations. Observing that circuits generated this way are not always efficient, abstract interpretation is used to improve the quality of the generated code. In essence, abstract interpretation is used to enrich the generator with knowledge about several specific identities of real numbers, namely:

$$\begin{array}{rr} r \cdot 0 = 0 & sin(0) = 0 \\ r + 0 = r & cos(0) = 1 \\ r \cdot 1 = r & sin(\frac{\pi}{4}) = cos(\frac{\pi}{4}) \\ r + (-1 \cdot r') = r - r' & sin(t + \frac{\pi}{2}) = cos(t) \\ f \cdot r + f \cdot r' = f \cdot (r + r') & cos(t + \frac{\pi}{2}) = -sin(t) \end{array} \quad (1)$$

With these optimizations, the quality of the generated code in terms of number of addition and multiplication operations came very close to that of FFTW codelets [3]. But the resulting circuits had more floating point operations than in the corresponding FFTW codelets for sample sizes larger than 8. Whether it is possible to reach the same numbers as FFTW or other optimal algorithms remained open. This paper reports on three modifications to the abstract interpretation step that yield code with the same number as operations as FFTW:

1. Exploiting identities of complex roots of unity rather than their floating-point representations,
2. Switching from decimation in time (DiT) to decimation in frequency (DiF),
3. Exploiting the pattern of additions and subtractions in the algorithm.

FFT deals with complex-valued operations. Analysis of the algorithm shows that factors known at generation time are not arbitrary. In particular, they are never zero, and are always roots of unity ($e^{i2\pi j/n}$). If instead of representing such constants as floating

points we represent them as rational numbers $\frac{j}{n}$, we can implement the identities on these values *exactly*. In particular, we are able to exploit the following identity:

$$e^{i2\pi j/n} \cdot e^{i2\pi j'/n'} = e^{i2\pi(\frac{j}{n}+\frac{j'}{n'})}$$

When roots of unity are represented as a pair of floating-point numbers, such equivalences do not hold except for trivial cases.

The decimation-in-frequency definition of FFT is as follows:

$$\begin{array}{ll} F(1,x)_0 & = x_0 \\ F(N,x)_{2k} & = F(\frac{N}{2}, E(x))_k \\ F(N,x)_{2k+1} & = F(\frac{N}{2}, O(x))_k \end{array} \text{ where } \begin{array}{l} E(x)_j = x_j + x_{j+\frac{N}{2}}, \\ O(x)_j = (x_j - x_{j+\frac{N}{2}}) \cdot w_N^j \end{array}$$

We have already noted that all complex multiplications in the FFT algorithm multiply a root of unity with a linear combination of input values. As a result, complex additions and complex subtractions in FFT always have the form $w_1 \cdot c \pm w_2 \cdot d$ where w_1 and w_2 are roots of unity. Such patterns can be computed in two different ways. The first is to do the multiplications by $w_1 \cdot c$ and $w_2 \cdot d$ first, and use the result for the final addition and the subtraction. The second approach is to re-write the expression as $w_1(c \pm w_2/w_1 \ \cdot d)$. The second approach is useful when the multiplication either by w_1 or by w_2/w_1 is trivial. The trivial multiplication is the one by ± 1, $\pm i$.

These are all the optimizations needed to match FFTW operation count.

2.1 Split-Radix FFT

Split-radix FFT is a particular FFT algorithm that aims to compute FFT with the least number of multiplications. In the general case, complex multiplication can be computed with four real multiplications and two real additions

$$(a + ib) \cdot (c + id) = (ac - bd) + i(ad + bc) \tag{2}$$

or, with three multiplications and five addition/subtractions

$$(a + ib) \cdot (c + id) = (t_1 - t_2) + i(t_1 + t_3) \text{ where } \begin{cases} t_1 = a(c + d) \\ t_2 = d(b + a) \\ t_3 = c(b - a) \end{cases} \tag{3}$$

The benefit of that particular formula among others with the same operation count is that when the factor $a + ib$ is known at generation time, two of the required additions/subtractions, namely, $b + a$ and $b - a$, can be computed at that time, leaving the other three additions and three multiplications to the run-time of the generated code.

Choosing equation (2) gives us the code that matches FFTW in operation count, whereas choosing equation (3) gives us the code that matches split-radix.

2.2 Experiments

The following table summarizes our measurements of the effect of abstract interpretation for FFT. The first column gives the size of the FFT input vector. The second column

gives the number of floating-point multiplications/additions in the code resulting from direct staging. The column "RAP DiT" reproduces previous results [5]. The column, "RAP DiF 1" demonstrates the improvement of the more precise abstract interpretation described here. The next column shows the number of multiplications/additions in code generated by FFTW for the various problem sizes[3]. The column "RAP DiF 2" is "RAP DiF 1" but with complex multiplication with three real multiplies and five real additions, two of which are done at code generation time. The last column is the data for a split-radix algorithm with the complex input [9, Table II].

Size	Direct staging	RAP DiT	**RAP DiF 1**	FFTW [3]	**RAP DiF 2**	Split Radix
4	32/32	0/16	0/16	0/16	0/16	0/16
8	96/96	4/52	4/52	4/52	4/52	4/52
16	256/256	28/150	24/144	24/144	20/148	20/148
32	640/640	108/398	84/372	84/372	68/388	68/388
64	1536/1536	332/998	248/912	248/912	196/964	196/964
128	3584/3584	908/2406	660/2164	$\approx$ 752/2208	516/2308	516/2308
256	8192/8192	2316/5638	1656/5008	$\approx$ 2016/5184	1284/5380	1284/5380

We have used our FFT generator to generate all the circuit descriptions summarized by above table. To check the correctness of these implementations, we have translated them into C programs and checked their results and performance against FFTW. The following table shows the performance of the algorithms above as measured by the FFTW benchmark v3.1 on a Pentium IV 3GHz computer. The numbers show reported MFLOPS relative to FFTW for double-precision, complex, in-place, forward FFT. All code was compiled with GCC 3.2.2. The performance numbers show that on a Pentium IV, the floating-point multiplications are about just as fast as floating-point additions, and on modern super-scalar CPUs, the performance depends on many other factors (such as caching, pipelines stalls, etc.) rather than merely the floating-point performance. On DSP, FPGA and other similar circuits/processors, floating-point performance is usually the bottleneck.

Size	4	8	16	32	64	128	256
RAP DiF 1	335%	162%	97%	96.1%	83.1%	77.7%	68.8%
RAP DiF 2	323%	162%	102%	88.0%	79.2%	78.6%	69.6%
FFTW	100%	100%	100%	100%	100%	100%	100%

3 Conclusions

With systematic improvements to the domain-specific optimizations used, we found that staging and abstract interpretation can generate FFT circuits that match both FFTW and the split-radix algorithm in terms of operation count. Furthermore, to generate circuits that match each of the two algorithms, all that is needed was to chose between two different definitions of complex multiplication.

[3] The numbers for FFTW are obtained from its codelets. FFTW does not have codelets for sample sizes 128 and 256. For those and larger sizes, FFTW uses the composition of smaller FFTW transforms. For those sample sizes, the operation counts in the table are estimates based on the counts for smaller sizes and on the Cooley-Tukey recurrences for the power-of-2 FFT algorithm.

Unlike FFTW, we know precisely where savings are coming from, and which particular equivalences contribute to which improvements in the code. We do not search for optimal code using extensive low-level optimizations at the level of real-valued terms. Rather, we use a small number of optimizations at the level of complex-numbers. Complex numbers are *the* domain-specific type for this application. We do not attempt to apply optimizations after generation, but rather, during generation. As such, our experience provides further evidence that abstract interpretation is a promising tool for expressing domain-specific optimizations in a program generation system.

Acknowledgements: We would like to thank Anthony Castanares, Emir Pašalić, and Abd Elhamid Taha for comments on this manuscript.

References

1. W. Boehm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. In *Supercomputing*, number 21, pages 117–130, 2002.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
3. Matteo Frigo. A Fast Fourier Transform compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 169–180, 1999.
4. C. S. Burrus I. W. Selesnick. Automatic generation of prime length FFT programs. In *IEEE Transactions on Signal Processing*, pages 14–24, Jan 1996.
5. Oleg Kiselyov, Kedar Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *the International Workshop on Embedded Software (EMSOFT '04)*, Lecture Notes in Computer Science, Pisa, Italy, 2004. Springer-Verlag. To appear.
6. Xavier Leroy. Objective Caml, 2000. Available from http://caml.inria.fr/ocaml/.
7. R. Lipsett, E. Marschner, and M. Shaded. VHDL - The Language. In *IEEE Design and Test of Computers*, pages 28–41, April 1986.
8. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.
9. M.T.Heideman and C.S.Burrus. On the number of multiplications necessary to compute a length-2^n DFT. *IEEE Trans. ASSP*, ASSP-34(1):91–95, February 1986.
10. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html.
11. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [10].
12. Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Imperative, Heap-Bounded Programs in a Functional Setting. In *Proceedings of the Third International Conference on Embedded Software*, Philadelphia, PA, October 2003.
13. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
14. Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 3rd edition, 1996.