Automatic Performance Tuning in the UHFFT Library

Dragan Mirković[1] and S. Lennart Johnsson[1]

Department of Computer Science
University of Houston
Houston, TX 77204
mirkovic@cs.uh.edu, johnsson@cs.uh.edu

Abstract. In this paper we describe the architecture–specific automatic performance tuning implemented in the UHFFT library. The UHFFT library is an adaptive and portable software library for fast Fourier transforms (FFT).

1 Introduction

The fast Fourier transform (FFT) is one of the most popular algorithms in science and technology. Its uses range from digital signal processing and data compression to numerical solution of partial differential equations. The importance of the FFT in many applications has provided a strong motivation for development of highly optimized implementations. The growing complexity of modern microprocessor architectures with multi-level memory hierarchies and instruction level parallelism has made performance tuning increasingly difficult. In particular, FFT algorithms have a number of inherent properties that make them very sensitive to the memory hierarchy mapping. These include the recursive structure of the FFT algorithm, its relatively high efficiency ($O(n \log n)$) which implies a low floating–point v.s. load/store instruction ratio, and the strided data access pattern. Besides that, the unbalance in the number of additions and multiplications reduces some of the advantages of modern superscalar architectures.

The need for the FFT codes has forced many application programmers to manually restructure and tune their codes. This is a tedious and error prone task, and it requires the expertise in computer architecture. The resulting code is less readable, difficult to maintain and not easily portable.

One way to overcome these difficulties is to design codes that adapt themselves to the computer architecture by using a dynamic construction of the FFT algorithm. The adaptability is accomplished by using a library of composable blocks of code, each computing a part of the transform, and by selecting the optimal combination of these blocks at runtime. A very successful example that uses this approach is the FFTW library [1]. We have adopted similar approach in the UHFFT library and extended it with a more elaborate installation tuning and richer algorithm space for execution.

For this approach to be efficient the blocks of code (*codelets* in FFTW lingo) in the library should be highly optimized and tuned to the specific architecture

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 71–80, 2001.

and the initialization should be fast and inexpensive. These goals can be achieved by performing the time consuming tasks required by the optimization during the installation of the library. In UHFFT, we first use a special–purpose compiler to generate and tune the codelets in the library. Second, we do most of the time consuming performance measurements during the installation of the library. The major novelty in the UHFFT library is that most of the code is automatically generated in the course of the installation of the library with an attempt to tune the installation to the particular architecture. To our knowledge this is the only FFT library with such capability. Although several other public domain libraries make use of automatic code generation techniques similar to ours, their code is usually pregenerated and fixed for all platforms. Even if they allow for possible modifications of the generated code, this modifications are cumbersome and not at all automatic. On the other hand our code is generated and optimized at the time of installation.

In this paper we give an overview of the automatic performance tuning techniques incorporated in the UHFFT library. The rest of the paper is organized as follows. Section 2 gives the basic mathematical background for the *polyalgorithmic* approach used both to build the library of codelets and to combine them during the execution. Section 3 describes the automatic optimization and tuning methodology used in the UHFFT.

2 Mathematical Background

The Fast Fourier Transform (FFT) is a method used for the fast evaluation of the Discrete Fourier Transform (DFT). The DFT is a matrix–vector product that requires $O(n^2)$ arithmetic operations to compute. Using the FFT to evaluate the DFT reduces the number of operations required to $O(n \log n)$. In this chapter we give a short list of the algorithms used in the UHFFT library. We refer the reader to [2], [3], and [4] for the more detailed description of the algorithms. In particular, the notation we use here mostly coincides with the notation in [2].

Let $\mathbf{C}^n$ denote the vector space of complex *n–vectors* with components indexed from zero to $n-1$. The Discrete Fourier Transform (DFT) of $\mathbf{x} \in \mathbf{C}^n$ is defined by

$$\mathbf{y} = W_n \mathbf{x} \tag{1}$$

where $W_n = (w_{lk})_{l,k=0}^{n-1}$ is the DFT matrix with elements $w_{lk} = \omega_n^{lk}$, and $\omega_n = e^{-2\pi i/n}$ is the principal nth root of unity. The fast evaluation is obtained through factorization of W_n into the product of $O(\log n)$ sparse matrix factors so that (1) can be evaluated as

$$W_n \mathbf{x} = (A_1 A_2 \dots A_r)\mathbf{x} \tag{2}$$

where matrices A_i are sparse and $A_i \mathbf{x}$ involves $O(n)$ arithmetic operations. The factorization (2) for given n is not unique, and possible variations may have properties that are substantially different. For example, it can be shown that when $n = rq$, W_n can be written as

$$W_n = (W_r \otimes I_q) D_{r,q} (I_r \otimes W_q) \Pi_{n,r}, \tag{3}$$

where $D_{r,q}$ is a diagonal *twiddle–factor* matrix, $D_{r,q} = \bigoplus_{k=0}^{r-1} \Omega_{n,q}^k$, $\Omega_{n,q} = \bigoplus_{k=0}^{q-1} \omega_n^k$, and $\Pi_{n,r}$ is a mod–r sort permutation matrix. The algorithm (3) is the well known Cooley–Tukey [5] mixed–radix splitting algorithm. In this algorithm a non–trivial fraction of the computational work is associated with the construction and the application of the diagonal scaling matrix $D_{r,q}$. The *prime factor* FFT algorithm (PFA) [6,7,8] removes the need for this scaling when r and q are relatively prime, i.e., $\gcd(r, q) = 1$. This algorithm is based upon splittings of the form:

$$W_n = P_1(W_r \otimes I_q)(I_r \otimes W_q)P_2 = P_1(W_r \otimes W_q)P_2 = P^T(W_r^{(\alpha)} \otimes W_q^{(\beta)})P, \quad (4)$$

where P_1, P_2 and P are permutations and $W_r^{(\alpha)} = (w_{lk}^{(\alpha)})_{l,k=0}^{n-1}$ is the *rotated* DFT matrix with elements $w_{lk}^{(\alpha)} = \omega_n^{\alpha lk}$.

If q is not a prime number the above algorithms can be applied recursively, and this is the heart of the fast Fourier transform idea. In some cases the splitting stages can be combined together and, with some simplifications, the result may be a more efficient algorithm. The well known example is the *split–radix* algorithm proposed by Duhamel and Hollmann [9], which can be used when n is divisible by 4. Assume that $n = 2q = 4p$ and let $\mathbf{x} \in \mathbf{C}^n$. By using (3) with $r = 2$ we obtain

$$W_n = (W_2 \otimes I_q)D_{n,q}(I_2 \otimes W_q)\Pi_{n,2}. \quad (5)$$

The *split–radix* algorithm is obtained by using the same formula again on the second block of the block–diagonal matrix $I_2 \otimes W_q = W_q \oplus W_q$, and rearranging the terms such that the final factorization is of the form

$$W_n = B(W_q \oplus W_p \oplus W_p)\Pi_{n,q,2}. \quad (6)$$

Here, B is the split–radix butterfly matrix and $\Pi_{n,q,2}$ is the split–radix permutation matrix, $\Pi_{n,q,2} = (I_q \oplus \Pi_{q,2})\Pi_{n,2}$. The efficiency of the split–radix algorithm follows from simplifications of the butterfly matrix

$$B = (W_2 \otimes I_q)D_{n,q}[I_q \oplus (W_2 \otimes I_p)D_{q,p}], \quad (7)$$

which, after some manipulations, can be written as

$$\begin{aligned} B &= (W_2 \otimes I_q)[I_q \oplus (SW_2 \otimes I_p)](I_q \oplus \Omega_{n,p} \oplus \Omega_{n,p}^3) \\ &= B_a B_m, \end{aligned} \quad (8)$$

where $S = 1 \oplus -i$; $B_a = (W_2 \otimes I_q)[I_q \oplus (SW_2 \otimes I_p)]$ is the additive and $B_m = (I_q \oplus \Omega_{n,p} \oplus \Omega_{n,p}^3)$ is the multiplicative part of the butterfly matrix B.

When n is a prime, there is a factorization of W_n proposed by Rader [10,3, 2] involving a number–theoretic permutation of W_n that produces a circulant or a skew–circulant submatrix of order $n - 1$. The indexing set $\{0, \ldots, n-1\}$ for prime n is a field with respect to addition and multiplication modulo n, and all of its nonzero elements can be generated as powers of a single element called a *primitive root*. The permutation induced by the powers of the primitive root r

$$\mathbf{z} = Q_{n,r}\mathbf{x}, \quad z_k = \begin{cases} x_k & if \quad k = 0, 1 \\ x_{<r^{k-1}>_n} & if \quad 2 \le k \le n-1 \end{cases} \quad (9)$$

is called the *exponential permutation* associated with r. It can be shown that

$$W_n = Q_{n,r}^T \begin{pmatrix} 1 & \mathbf{1}^T \\ \mathbf{1} & C_{n-1} \end{pmatrix} Q_{n,r^{-1}} = Q_{n,r}^T \begin{pmatrix} 1 & \mathbf{1}^T \\ \mathbf{1} & S_{n-1} \end{pmatrix} Q_{n,r}, \tag{10}$$

where $\mathbf{1}$ is a vector of all ones and C_{n-1} and S_{n-1} are circulant and skew–circulant matrices respectively, generated by the vector $\mathbf{c} = (\omega_n, \omega_n^r, \ldots, \omega_n^{r^{n-2}})^T$. Both C_{n-1} and S_{n-1} can be diagonalized by W_{n-1},

$$C_m = W_m^{-1}\mathrm{diag}(W_m\mathbf{c})W_m \quad \text{and} \quad S_m = W_m\mathrm{diag}(W_m^{-1}\mathbf{c})W_m,\ m = n-1. \tag{11}$$

This effectively reduces the prime size problem to a non–prime size one for which we may use any other known algorithm. Asymptotically optimal algorithms for prime n can be obtained through full diagonalization of C_{n-1} and S_{n-1}. When n is small, though, a partial diagonalization with $(W_2 \otimes I_{(n-1)/2})$ may result in a more efficient algorithm [3].

The list of possible FFT algorithms by no means ends with the four basic algorithms described in this section, but these four algorithms and their variants are used as the basic building blocks in the UHFFT library.

3 Performance Tuning Methodology

The optimization in the UHFFT library is performed on two levels and a coarse flowchart of the performance tuning methodology is shown in Figure 1. The first (high) level optimization consists of selecting the optimal factorization of the FFT of a given size, into a number of factors, smaller in size, for which an efficient DFT codelet exists in our library. The optimization on this level is performed during the initialization phase of the procedure, which makes the code adaptive to the architecture it is running on.

The second (low) level optimization involves generating a library of efficient, small size DFT codelets. Since the efficiency of the code depends strongly on the efficiency of the codelets themselves, it is important to have the best possible performance for the codelets to be able to build an efficient library. The code generation and the architecture specific tuning is a time consuming process and it is performed during the installation of the library. We have a small number of installation options that can be specified by the user. At this moment these options are restricted to the range of sizes and dimensions for which the library should be optimized. We are planning to extend the range of options to include the interface, data distribution and parallelization methods in the future releases. We may also include some application specific options like known symmetries in the data, restrictions on the size of the library and the memory used by the code. The extent of the additional options will strongly depend on the feedback we get from the users. The idea is to exploit the flexibility of code generation and optimization tools which are built in the library for the benefit of the user and to allow for a significant and simple customization of the library.

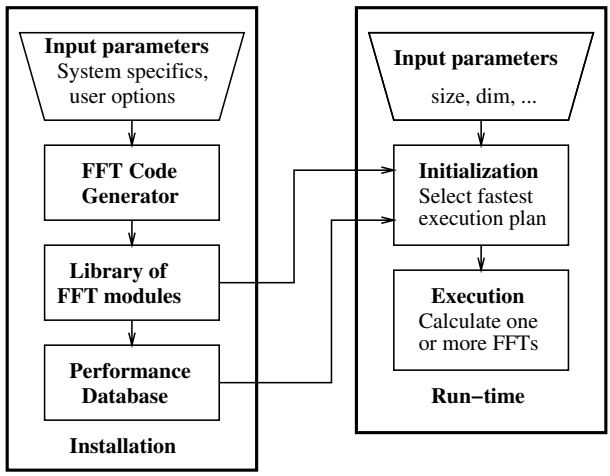

Fig. 1. Performance tuning methodology in UHFFT.

3.1 Execution Plan Generation

Given the parameters of the problem, the initialization routine selects the strategy in terms of execution time on the given architecture. This selection involves two steps. First, we use a combination of the mixed–radix, split–radix and prime factor algorithm splittings to generate a large number of possible factorizations for a given transform size. Next, the initialization routine attempts to select a strategy that minimizes the execution time on the given architecture.

The basis for generating execution plans are the library of codelets and two databases: the *codelet database* storing information about codelet execution times, and the *transform database* that stores information about the execution times for entire transforms. The codelet database is initialized during installation of the library as a part of the benchmarking routine.

The transform database stores the best execution plan for different size transforms. The transform database is initialized for some of the popular FFT sizes during installation (such as power of 2 and PFA sizes).

For transform sizes that are not in the database, an execution plan must be created and this can be done in two different ways.

The first method is to empirically find the execution plan that minimizes the execution time by executing all possible plans for the given size, and choose the plan with the best performance. This method ensures that the plan selected will indeed result in the smallest execution time for all choices possible within the UHFFT library, but the time required to find the execution plan may be quite large for large size FFTs. So, unless many transforms of a particular size are needed this method is not practical.

The second method is based on estimating the performance of different execution plans using the information in the codelet database. For each execution plan feasible with the codelets in the library the expected execution time is

derived based on the codelets being used in the plan, the number of calls to each codelet, and the codelet performance data in the codelet database. The estimation algorithm also takes into account the input and output strides and transform direction (forward or inverse). It also accounts for the twiddle factor multiplications for each plan as the number of such multiplications depend on the execution plan.

For large transform sizes with many factorizations, the estimation method is considerably faster than the empirical method. The quality of the execution plan based on the estimation approach clearly relies heavily on the assumption that codelet timings can be used to predict transform execution times, and that the memory system will have a comparable impact on all execution plans.

The adaptive approach used by UHFFT is very similar to the one used by the FFTW library [1]. The main difference is the set of algorithms used generate the collection of possible execution strategies. FFTW uses the mixed–radix and Rader's algorithm while we currently use mixed–radix, split–radix and prime factor algorithm. While we are still planning to include the Rader's algorithm, its significance at the execution level is to have an asymptotically optimal code for all transform sizes (including the prime sizes and sizes containing prime factors not included in the library of codelets). The performance that can be achieved for these sizes, though, is relatively low when compared to the neighboring (non–prime) transform sizes. For example the transforms for sizes 32 and 128 are approximately ten times faster then the transforms for sizes 31 and 127 respectively. On the other hand, both the split–radix and the prime factor algorithm provide for the richer and more efficient algorithm space covered by the library. We illustrate that by comparing the performance of UHFFT versus FFTW for the PFA transform sizes in Figure 2. Here UHFFT uses the PFA to combine the codelets, while FFTW uses the mixed–radix algorithm.

3.2 Library of FFT Modules

The UHFFT library contains a number of composable blocks of code, called *codelets*, each computing a part of the transform. The overall efficiency of the code depends strongly on the efficiency of these codelets. Therefore, it is essential to have a highly optimized set of DFT codelets in the library. We divide the codelet optimization into a number of levels. The first level optimization involves reduction of the number of arithmetic operations for each DFT codelet. The next level of optimization involves the memory hierarchy. In current processor architectures, memory access time is of prime concern for performance. Optimizations involving memory accesses are architecture dependent and are performed only once during the installation of the library.

The codelets in our library are generated using a special purpose compiler that we have developed. The FFT code generation techniques have been in use for more than twenty years (see an overview given by Matteo Frigo in [11]). Most of the existing implementations are restricted to complex transforms with a predetermined generation algorithm. A notable exception is the FFTW generator *genfft*, which not only uses a flexible set of algorithms, but also deals

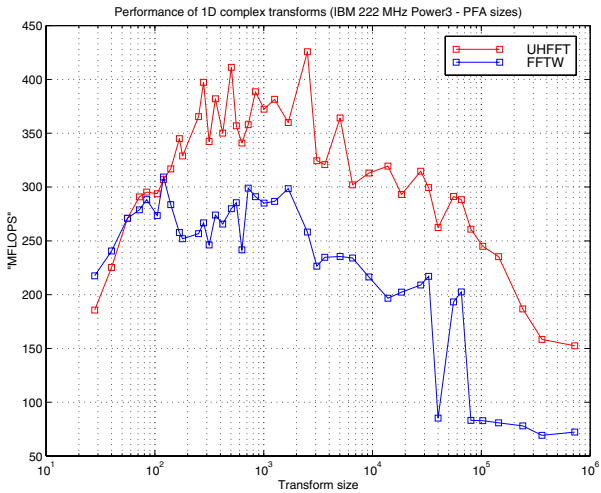

Fig. 2. Graph of the performance of UHFFT versus FFTW on a 222 MHz IBM Power 3 processor for selected transform sizes that can be factored into mutually prime powers of 2, 3, 5, 7, 11, and 13. The peak performance of 409 MFLOPS achieved by the UHFFT PFA plan for $n = 2520$ is not only higher than the FFTW performance for the same size (258 MFLOPS), it is also higher than the performance of FFTW for any size we tested on this processor. The peak performance achived by FFTW was 397 MFLOPS for $n = 64$.

with optimization and scheduling problems in a very efficient way. The FFTW code generator is written in Objective Caml [12], a powerful and versatile dialect of the ML functional language. Although the Caml capabilities simplify the construction of the code generator, we find the dependence on a large and nonstandard library an impediment in the automatic tuning of the installation. For that reason we have decided to write the UHFFT code generator in C. This approach makes the code generation fast and efficient and the whole library is more compact and ultimately portable. We have also built the enough infrastructure in the UHFFT code generator to match most of the functionalities of *genfft*. Moreover, we have added a number of derived data types and functions that simplify the implementation of standard FFT algorithms. For example, here is a function that implements the mixed–radix algorithm (3).

```
/*
*       FFTMixedRadix()       Mixed-radix splitting.
*       Input:
*               r       radix,
*               dir     direction of the transform,
*               rot     rotation of the transform,
*               u       input expression vector.
*/
ExprVec *FFTMixedRadix(int r, int dir, int rot, ExprVec *u)
```

```
{
        int     m, n = u->n, *p;

        m = n/r;
        p = ModRSortPermutation(n, r);
        u = FFTxI(r, m, dir, rot,
                TwiddleMult(r, m, dir, rot,
                IxFFT(r, m, dir, rot, PermuteExprVec(u, p))));
        free(p);
        return u;
}
```

The functions `FFTxI()` and `IxFFT` correspond to the expressions $(W_r \otimes I_m)$ and $(I_r \otimes W_m)$ respectively, `TwiddleMult()` implements the multiplication with the matrix of twiddle factors $D_{r,q}$; the action of the mod-r sort permutation matrix $\Pi_{n,r}$ is obtained by calling the function `PermuteExprVec(u, p)`, where the permutation vector `p` is the output of the function `ModRSortPermutation(n, r)`.

The UHFFT code generator can produce DFT codelets for complex and real transforms of arbitrary size, direction (forward or inverse), and rotation (for PFA). It first generates an abstraction of the FFT algorithm by using a combination of Rader's algorithm, the mixed–radix algorithm, the split–radix algorithm and the PFA. The next step is the scheduling of the arithmetic operations such that memory accesses are minimized. We make effective use of temporary variables so that intermediate writes use the cache instead of writing directly to memory. We also use blocking techniques so that data residing in the cache is reused the maximum possible number of times without being written and re–read from main memory. Finally, the abstract code is unparsed to produce the desired C code. The output of the code–generator is then compiled to produce the executable version of the library. The structure of the library is given in Figure 3.

The performance depends strongly on the transform size, input and output strides and the structure of the memory hierarchy of a given processor. Once the executables for the library are ready, we benchmark the codelets to test its performance. These benchmark tests are conducted for various input and output strides of data. The results of these performance tests are then stored in a database that is used later by the execution plan generator algorithm during the initialization phase of an FFT computation. In Figure 4 we show a typical performance map on two different processors.

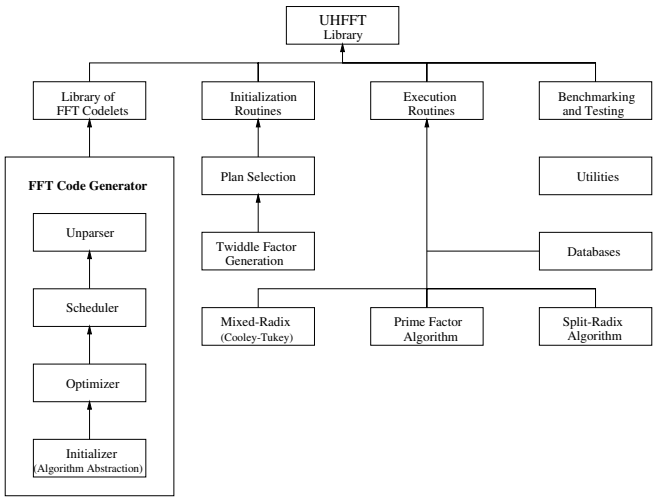

Fig. 3. UHFFT Library Organization.

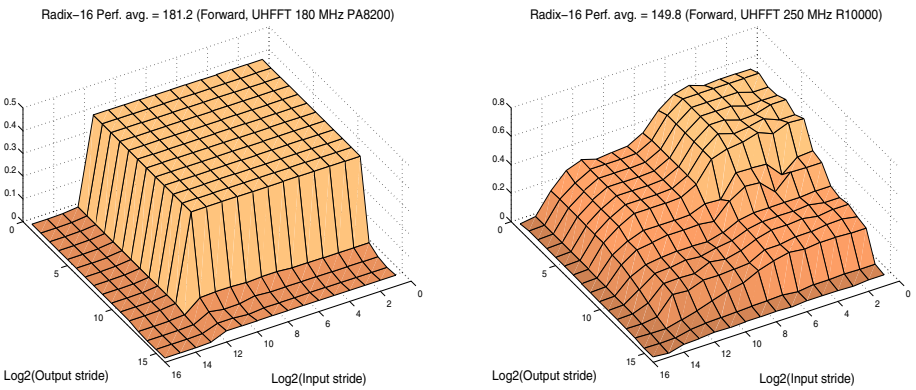

Fig. 4. Two examples of the size 16 codelet performance on 250 MHz SGI R10000 and 180 MHz HP PA8200. The SGI R10000 processor has peak performance of 500 MFlops. This processor has as primary caches a 32 KB two way set associative on chip instruction cache and a 32 KB two way set associative, two way interleaved on chip data cache with LRU replacement. It also has a 4 MB two way set associative L2 secondary cache per CPU. The SGI R10000 has 64 physical registers, each 64 bits wide. The HP PA8200 RISC microprocessor operating at 180 MHz is capable of 720 MFlops peak. It has a 1 MB direct mapped data cache and a 1 MB instruction cache. The performance is plotted as a fraction of the peak achivable performance. The di erence is typical for one versus two levels of cache.

References

[1] Matteo Frigo and Steven G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, MIT, 1997.

[2] Charles Van Loan. *Computational frameworks for the fast Fourier transform.* Philadelphia:SIAM, 1992.

[3] Richard Tolimieri, Myoung An, and Chao Lu. *Algorithms for Discrete Fourier Transforms and Convolution.* Springer–Verlag, New York, 1 edition, 1989.

[4] P. Duhamel and M. Vetterli. Fast Fourier Transforms: A Tutorial Review and a State of the Art. *Signal Processing*, 19:259–299, 1990.

[5] J.C. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series. *Math. Comp.*, 19:291–301, 1965.

[6] I.J. Good. The interaction algorithm and practical Fourier Analysis. *J. Royal Stat. Soc., Ser. B*, 20:361–375, 1958.

[7] L.H. Thomas. Using a computer to solve problems in physics. In *Application of Digital Computers.* Ginn and Co., Boston, Mass., 1963.

[8] C. Temperton. A Note on Prime Factor FFT Algorithms. *Journal of Computational Physics*, 52:198–204, 1983.

[9] P. Duhamel and H. Hollmann. Split Radix FFT Algorithms. *Electronic Letters*, 20:14–16, 1984.

[10] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56:1107–1108, 1968.

[11] Matteo Frigo. A Fast Fourier Transform Compiler. *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, 1999.

[12] Xavier Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Technical Report 2721, INRIA, November 1995.