EFFICIENCY OF RADIX-*K* TRANSFORMS ON COMPUTERS WITH CACHE

Ryszard M. Stasinski[1]

Hoegskolen i Narvik
Lodve Langes gt. 2
N-8501-Narvik, Norway

ABSTRACT

In the paper impact of the most critical part of the up-to-date computer memory hierarchy, the cache, on the efficiency of fast transform algorithms (e.g. FFT, DCT/DST, DWHT, including multidimensional generalizations) is analyzed. Cache misses can severely deteriorate efficiency of a computer program, and indeed, it is shown that for large data vectors a modification of a fast transform algorithm realization may change their number dramatically. Several memory managing problems are pointed out, and suggestions for their amelioration are given. Formulae on the minimum of data-related cache misses numbers for radix-2^s transform realizations are given, s is an integer.

1. INTRODUCTION

The most exact measure of algorithm performance on a computer is execution time of a program that is its realization. This measure is however not very practical for general comparison of algorithms, as it is difficult to asses what features decide that one program is faster than the other: algorithm dependent or algorithm independent. That is why algorithms are compared on the basis of numbers of instructions that are critical for an appropriate computer model. For sequential computers the commonly accepted model is the *RAM machine* with *arithmetical* operations to be counted [1]. The model was adequate in fiftieth, then its appropriateness diminished as advanced computer features like multiple internal processor registers, pipelining, and caches become more and more popular. Today the features are ubiquitous [2], hence the RAM machine model should be used rather cautiously.

As in some digital signal processing applications fast transform algorithms are processing huge data structures, their computation can be severely affected by memory organization, in particular by caches. This is the main topic of the paper being covered in section 4, the section contains also some hints how to minimize the negative effects, as well as formulae on the minimal numbers of cache misses. An overview of memory organization, and cache types in particular is given in section 2. Many of the fast transform algorithms are built up of radix-*K* 'butterflies' (usually radix-2 ones, e.g. one- and multidimensional FFT, DWHT, DCT, DHT...), section 3. It should be noted that radix-*K* transforms consists of many independently computed modules, and they can be very efficiently realized on processors with pipelining of virtually any depth, hence, pipelining effects need not be considered in the corrected RAM machine model for them.

2. MEMORY HIERARCHY AND CACHES

One of aims of computer technology is to give the user an impression that a computer has unlimited memory all words of which are equally easy accessible. This principle has been reflected in the *RAM machine* model [1], which states that from the computational point of view a computer consists of a sequential processor with no more than one internal register and the random-access memory. Because of technological efforts done for supporting the illusion, for problems that do not require excessive amounts of memory the RAM machine model is still a fairly good approximation of even the most sophisticated up-to-date sequential computers.

Nevertheless, if the compared algorithms have similar performance, the RAM machine model could fail. This happened when linear code FFT and WFTA programs were compared on computers having several floating-point accumulators [3], [4]. Internal processor registers introduce memory hierarchy, we have large and slow main memory, and fast but small memory inside a processor. There is also a change in philosophy of writing optimal computer programs. Instead of steady flow of memory-to-memory, or memory-to-accumulator instructions the program brakes down into phases of register-to-register computations separated by phases of memory-to-register operations being necessary for updating the register contents (in competition winning RISC architectures the only memory-to-register operations are loads and stores [2]). The conclusion of [3], [4] has been that for multi-register processors the traditional arithmetical complexity algorithm measure should be extended by the count of main memory addressing operations (loads and stores in the case of RISC computers).

Since that times a new level of memory hierarchy has become popular, the *cache* [2]. The cache is a buffer memory between processor registers and main computer memory. Its main purpose is to reduce the time of main memory loads and stores, as read and write times of today processors are many times shorter than those for dynamic RAM (DRAM) chips. Caches are faster, but smaller (more expensive) than main memories, but not to the extent the internal processor registers are, they form an intermediary memory hierarchy level (or levels [2]) between them. Caches are invisible for a programmer, they store copies of memory words that are likely to be used by a program, or loaded as program instructions.

The simplest cache control is based on *direct mapping* of memory addressing space into cache addressing space. When a

[1] On leave from Poznan University of Technology, Institute of Electronics and Telecommunications.

datum is required by a processor for the first time, apart from loading it to the processor register it is also stored in the cache at the address obtained by computing the residue of datum memory address modulo the size of the cache:

Cache_address = *memory_address* modulo *cache_size*

In fact, the address computation can be somewhat more complicated, as in some caches data are stored in blocks. Next read or write of the datum will take place between processor and cache, hence, it will be much faster. Notice also that if data or program form a contiguous cluster in memory, then new data/instruction load does not erase old ones (no *conflict*), unless the cluster is greater than the cache. The cache concept works as typical programs and associated with them data indeed exhibit *temporal* and *spatial locality* [2], i.e. data and instruction locations are reused several times, and form concentrated clusters in memory.

When the spatial locality of a program is poor and highly structured, the probability of data/instruction conflict is quite high as the address mapping formula above is too rigid. The remedy is to use the *associative* cache, for which any main memory location can be placed anywhere in the cache. Such caches require, however, quite big circuitry for searching requested data in the cache, hence they are usually small. The compromise solution is to use the *m-way associative* cache, *m* is small. This is the variant of direct-mapped cache composed of associative caches of size *m*, the address mapping formula is:

Cache_address = *memory_address* modulo *(cache_size / m)*

When a new block stored in the cache has the same cache address as an old one, then usually no conflict occurs, as there is a place for *m* blocks having the same cache address.

If searched data is in the cache, then we have a *cache hit*, in the opposite case a *cache miss* take place. In the fastest computers a cache miss may result in program delays lasting even several hundreds of clock cycles [2], i.e. significantly more than the execution time of a floating-point arithmetical operation.

3. RADIX-*K* TRANSFORMS

In this paper the name *radix-K transforms* is given to fast transform algorithms that are built up from *K*-point operations called *butterflies*:

$$\begin{aligned} x(n) &\leftarrow f_0[x(n), x(n+offset_1), \ldots, x(n+offset_{K-1})], \\ x(n+offset_1) &\leftarrow f_1[x(n), x(n+offset_1), \ldots, x(n+offset_{K-1})], \\ &\ldots\ldots\ldots\ldots \\ x(n+offset_{K-1}) &\leftarrow f_{K-1}[x(n), x(n+offset_1), \ldots, x(n+offset_{K-1})], \end{aligned} \quad (1)$$

where $f_i[.]$, $offset_i$ depends on the transform type, $i=0,1,\ldots,K-1$. We are interested only in data addressing schemes of the transforms, including addressing of constants involved in computation of functions $f_i[.]$. We can distinguish two basic types of addressing schemes, the FFT-like one, and the DCT-like one. In the first type the offsets are given by a highly regular relation $offset_i = i\ M/K$, where M is the number of data samples processed in a given transform substructure, $n=0,1,\ldots,M/2-1$. For example, the simplest radix-2 butterfly that exists defines the fast DWHT algorithm, and is given by the following relation:

$$\begin{aligned} x(n) &\leftarrow x(n) + x(n+M/2), \\ x(n+M/2) &\leftarrow x(n) - x(n+M/2). \end{aligned} \quad (2)$$

In the second addressing scheme the processed data samples are located symmetrically with respect to the data vector center. For example, the unnamed DCT-like counterpart of DWHT algorithm (2) is defined as follows:

$$\begin{aligned} x(n) &\leftarrow x(n) + x(M-1-n), \\ x(M-1-n) &\leftarrow x(n) - x(M-1-n). \end{aligned} \quad (3)$$

Without loss of generality we can assume that an N-point radix-K transform consists of $\log_K N$ stages, and that in the first stage $M=N$, in the second $M=N/K$, then $N/(K^2),\ldots,K^2,K$. The samples of M-point substructures occupy contiguous areas of memory, e.g. for $M=N/4$, their indices are $n=0,1,\ldots,N/4-1$; $n=N/4,\ldots,N/2-1$; $n=N/2,\ldots,3N/4-1$; and $n=3N/4,\ldots,N-1$.

It is worth noting here that usually higher-radix structures as in (1) can be broken down into smaller ones. For example, computations inside butterflies of radix-2^s FFTs can be defined in terms of two-point structures, s is an integer, hence, the algorithms can be treated as radix-2 ones with variable butterfly definition. Conversely, we can agglomerate several low-radix butterflies into 'superstructures' forming in this way high-radix realizations of algorithms. Both techniques can be combined for optimization of algorithm realization, see [5] for minimization of the split-radix FFT load/store counts. The conclusion is that we have a freedom in realization of a radix-K algorithm, it could be radix-K, radix-K^s, or even radix-$\sqrt[s]{K}$ one.

They are many FFT-like radix-K transforms [6], [7]: FFT, including split-radix FFT, fast algorithms for DWHT, DHT (its flowgraph is reversed, i.e. $M=2,4,\ldots,N$), slant transform (with slight irregularities in its flowgraph), and separable multidimensional generalizations of all of them. Some circular convolution algorithms for N being a power of K are computed using two radix-K structures: forward followed by the reversed one (i.e. firstly M values grow, then diminish). The DCT-like transforms include different types of DCT and DST with their multidimensional generalizations. Rader DFT algorithms for prime numbers are constructed using circular convolution algorithms of sizes $N-1$, but because of input/output data permutations they belong rather to the DCT-like transform class. The prime factor and Winograd Fourier transform algorithms are mixed-radix transforms, i.e. transforms with variable K value. Finally, polynomial transforms of sizes being powers of K also consists of butterflies, but the addressing scheme for them is somewhat more complicated than for FFT or DCT.

4. RADIX-*K* ADDRESSING AND CACHES

Radix-K transforms are both spatially and temporally local. Nevertheless, in scientific computations, and especially in solving multidimensional or multivariable problems sizes of the processed data vectors may be larger than the cache. Moreover, even small but not spatially local while highly regular tasks (e.g. transforming of a sub-block of multidimensional data) may cause problems with cache conflicts. In all such situations we can expect severe degradation of software performance.

4.1. Simplest case, basic DWHT algorithm

This is the radix-2 algorithm of size $N=2^r$ defined by (2), r is an integer. As can be seen, no constants are involved in evaluation of functions $f_0[.]$, $f_1[.]$. For very large transform sizes numbers $M/2$ for some first transform stages are multiplicities of *cache_size*, which results in the following conflict problem with direct-mapped caches:

```
r1 ← x(M/2+n)     # cache miss, r0, r1, r2 are registers;
r0 ← x(n)         # cache miss and conflict, x(M/2+n) erased;
r2 ← r0 + r1      # function f0[.];
r1 ← r0 - r1      # function f1[.];
x(n) ← r2         # cache hit;
x(M/2+n) ← r1     # cache miss and conflict, x(n) erased.
```

As can be seen, we have here three cache misses, a great burden for a program consisting of only two arithmetic operations. In consecutive loop iterations index n is incremented by 1, hence, burst transmission from DRAM may help here a lot, but only if the control manage to read two data flows, one for consecutive $x(n)$, and the other for $x(M/2+n)$. Notice that the conflicts are avoided for associative caches, and the number of misses diminish to two, the 2-way associative cash suffices. Notice also that there are no immediate conflicts for DCT-type transform defined by (3), as the distance between samples $M-1-2n$ is an odd number.

Another problem is the minimization of cache miss number for the whole transform computation. Notice that after execution of the loop embracing above program the direct-mapped cache contains last data samples $x(M-1)$, $x(M-2)$,... $x(M-cache\_size)$. It is then reasonable to run the loop in the following stage backwards starting with loads of data samples in the cache. In this way we diminish the number of misses by *cache_size*. This implies two guidelines for a program realization:

- the program control should go depth-first inside the algorithm structure, as it is in recursive call algorithm realizations;
- the directions of loops execution interleave, forward and backward.

For associative caches the guidelines are the same, the only difference is that the cache contains two data sequence: $x(M/2-1)$, $x(M/2-2)$,... $x(M/2-cache\_size/2)$; and $x(M-1)$, $x(M-2)$,... $x(M-cache\_size/2)$, hence, the number of saved cache misses in the next stage is $cache\_size/2$. The rules for DCT-like transforms are analogous.

A totally different approach to consider here consists in use of fast matrix transposition based algorithms originally devised for processing of externally stored data [6]. In this method we exploit the high DRAM burst read/write rate for minimizing the total miss penalty rather than the number of isolate cache misses.

Let us calculate the minimal number of cache misses for DWHT of size $N=4\cdot cache\_size$ on a computer with an associative cache. In the first stage all loads are cache misses, i.e. their count is N. Then, lower half of samples is processed with $N/2-cache\_size/2=3N/8$ cache misses, after which the third quarter of data vector is processed with initial $N/8$ misses and then no misses till the end of algorithm flowgraph. Then the program control returns to process the fourth quarter of data samples, and there are $N/4$ cache misses. Finally, the first half of data vector is processed with $N/2+N/8$ initial misses, and additional $N/4$ misses when the control comes back to process the first quarter of data vector (or the second, it depends on actual realization). Summarizing, in total we have $2\frac{5}{8}N$ cache misses, in general,

$$(\log_2 k+1)N - (k-1)cache\_size/2=(\log_2 k+1/2)N+cache\_size/2,\quad k=N/cache\_size,\ k\geq 1.$$

If the guidelines are ignored, this number could be $N\log_2 N$ (N for each stage), as for the simple stage-after-stage strategy, but it is only N for the first stage misses when $N\leq cache\_size$ whatever the strategy is.

4.2 Higher-radix DWHT algorithm realizations

As it has been pointed out, we can agglomerate radix-2 DWHT butterflies into larger structures making in this way higher-radix DWHT algorithm realization. For example, if we combine four butterflies for data samples $x(n)$, $x(n+M/4)$, $x(n+M/2)$, and $x(n+3M/4)$ from two consecutive algorithm stages, then its radix-4 realization is formed. Then, combination of two such radix-4 butterflies with four radix-2 ones in the following stage forms the radix-8 algorithm realization, and so forth. Analysis of the realizations goes the same way as that of the radix-2 one. If the direct-mapped cache size is M/K or less, $K=2^s$, s is an integer, then every load and store of the i-th datum $x(n+iM/K)$ removes the previous one $x[n+(i-1)M/K]$ from the cash; in general for cache size M/L the load or store of variable $x(n+M/L)$ erases the variable $x(n)$. This is a conflict during computation of a K-point butterfly, the same that has been described for the radix-2 realization, K is a multiplicity of number 2. To avoid it at least 2^s-way associative caches should be applied. We can also see here the key advantage of high-radix algorithm realizations, while the number of arithmetical operations inside a radix-K^s butterfly is growing as $0(sK^s)$, the number of cache conflicts increases as $0(K^s)$, only.

The two guidelines for radix-2 transform program realizations apply to that of the radix-K transforms, too. However, the calculation of the number of cache misses for an L-way associative cache becomes somewhat more complicated. Notice that in the case when $M/L\geq cache\_size$ if $L\geq K$ then there are K data sequence that remain in the cache after execution of a stage of an algorithm, and L ones if $L<K$: $x(M/J-1)$, $x(M/J-2)$,... $x(M/J-cache\_size/J)$; $x(2M/J-1)$, $x(2M/J-2)$,... $x(2M/J-cache\_size/J)$; ...; $x(M-1)$, $x(M-2)$,... $x(M-cache\_size/J)$, J is equal either to K or to L. Additionally, if $M/K<cache\_size$ then the K value should be replaced by the $M/cache\_size$ one. Taking this into account for $L\geq K$ we receive the following formula on the minimal number of cache misses when k is an integer power of K, $K=2^s$:

$$(\log_K k+1)N - \frac{k-1}{K-1}\,cache\_size/K, \qquad (4)$$

in general, instead of k the greatest integer power of K lower than k should be taken, and a term $N-\frac{cache\_size}{k \bmod K}$ added. In case when $L<K$ the cache conflicts during computation of K-point butterflies should be considered, hence, the formula is even more complicated.

4.3 Constants addressing

Tables of constants (usually multipliers) add another dimension to the radix-K transform addressing analysis. Except for convolution algorithms the table sizes need not be greater than N: for FFT and DCT/DST samples from a quarter of sine period suffices, which means $N/4$, and $N/2$ samples, respectively, however, availability of samples from half period of sine and cosine functions ($3N/4$ constants) makes FFT programming simpler. If a complex multiplication and analogous operation of DCT/DST is computed using 3 real multiplications and 3 real additions, then two tables of such sizes are needed. Note that each constant is stored in the cache at least once causing at least one cache miss, hence, necessity of two tables undermines the reason for computing complex multiplications by the 'fast' 3-multiplication algorithm. On the other hand, sizes of constants tables for multidimensional transforms, and hence one-dimensional prime factor FFTs, form a small fraction of the total data samples number. If the constants table is designed for an algorithm of size N_{max}, then an N-point algorithm uses multipliers for indices: $offset$, $offset+N_{max}/N$, $offset+2N_{max}/N$, ..., $offset+N_{max}-N_{max}/N$; $offset=0$ for FFT and $offset=1/2N_{max}/N$ for DCT/DST. This means that the whole table addressing range is always used, even for the smallest transform sizes, hence, the program may appear to be not spatially local.

The above observations mean that they are no cache misses problems with radix-K transforms for $N \le cache\_size/2$, only; for direct-mapped caches data and constants should be located in such a way that they are mapped to different cache address spaces. Starting with $N=cache\_size$, however, the butterfly final data stores in first transform stages 'blow up' constants out of the cache, at the same time loading from an oddly defined constants table may cause erasing of data samples in pseudo-random cache locations. If the size of constants table is greater than $cache\_size/2$, then constants-data (and even constant-constant) conflicts occur even if $N \le cache\_size/2$.

It is then reasonable to write two versions of radix-K transform programs, 'normal' for $N \le cache\_size/2$, and special software for first stages of very large transforms. Constant-related cache conflicts can be almost totally avoided, in fact. Firstly, in first transform stages constants can be recursively generated in processor registers accordingly to the prescription from [8]. Unfortunately, we have here a contradiction between constant re-use [8], and the data-conflict minimization strategies; if the program is written in accordance with guidelines from the previous section the constants should be generated N/M times for each M-point part of a stage separately, in the opposite case we can 'spare' only $(\log_K k-1)cache\_size/K$ associative cache misses, and not $[(k-1)/(K-1)]cache\_size/K$ as in (4).

Secondly, a prime factor algorithm can be implemented, in which at least some small modules need so few constants that they can be stored in processor registers. For example, 32 floating-point registers is enough for storing constants and executing 15-point DFT or DCT/DST modules as the first transform structures; then, if N is divisible by 15 and $N/15 \le cache\_size/2$, the following algorithm structures can be performed with no more than $2N/15$ constant-related cache misses. For direct-mapped cache two constants tables are needed, one maps into the upper cache addresses space: $cache\_size/2$, $cache\_size/2+1$, ..., $cache\_size-1$ and is used with transforms processing data mapping into lower one: $0,1,...,cache\_size/2-1$; then the second table is used for data mapping into the upper cache address space. For associative caches only one table is needed, hence, the number of constant-related cache misses can be halved, but only under the condition that cache block replacement logic [2] can be forced to keep constants in the cache all the time.

5. CONCLUSION

In the paper influence of multilevel computer memory organization and caches in particular on the performance of radix-K transforms (e.g. one- and multi-dimensional FFT, DCT, DWHT) is analyzed. It is shown how to organize computations for obtaining minimal numbers of data cache misses, and formulae for their numbers are given. It is also suggested how to minimize numbers of constant-related cache misses in FFT and DCT/DST algorithms. In all cases new program writing guidelines concern processing the most problematic data vectors of sizes greater than half of the computer cache size.

An image that emerges here is that caches do not simplify writing of the fastest possible digital signal processing software, in fact, in some cases they can preclude application of potentially the best solutions. Namely, the highly regular structure of a fast transform may interfere with cache mapping strategy. A much more flexible and adequate here are the *local memory blocks* [9], unfortunately, they are usually not implemented in general purpose computers.

6. REFERENCES

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, ''The design and analysis of computer algorithms'', Addison-Wesley, 1974.

[2] D.A. Patterson, J.L. Hennessy, ''Computer organization and design, the hardware/software interface'', 2nd edition., Morgan Kauffmann Publ. Co., 1997, and Internet resources at *http://www.mkp.com/cod2e.htm*.

[3] H. Nawab, J.H. McClellan, '' on 'Bounds on the minimum number of data transfers in WFTA and FFT programs'', IEEE Trans. Acoust., Speech, Signal Proces., vol. ASSP-28, pp. 480-481, 1980.

[4] R. Stasinski, ''Comments on 'Bounds on the minimum number of data transfers in WFTA and FFT programs'', IEEE Trans. Acoust., Speech, Signal Proces., vol. ASSP-32, pp. 1255-1257, 1984.

[5] R. Stasinski, ''Minimization of non-arithmetical complexity of split-radix FFTs'', XV KKTOiUE (Circuit Theo. Conf.), Warszawa, pp. 363-368, 1992.

[6] H.J. Nussbaumer, ''Fast Fourier transform and convolution algorithms'', Springer-Verlag, 1981.

[7] A.K. Jain, ''Fundamentals of digital image processing'', Prentice-Hall, 1989.

[8] R.C. Singleton, ''An algorithm for computing the mixed-radix fast Fourier transform'', IEEE Trans. Audio Electroacoust., vol. AU-17, pp. 93-103, 1969.

[9] P. Faraboschi, G. Desoli, J.A. Fisher, ''The latest word in digital and media processing'', Signal Processing Mag., vol. 15, No. 2, pp. 59-85, 1998.