MIXED-RADIX FFT FOR IMPROVING CACHE PERFORMANCE

Ryszard Stasiński, Jacek Potrymajło

Institute of Electronics and Telecommunications, Poznań University of Technology
Piotrowo 3A, PL-60195 Poznań, Poland (Europe)
phone: +48 61 6652 631, fax: +48 61 6652 572
email: rstasins@et.put.poznan.pl, jpotrym@et.put.poznan.pl

ABSTRACT

The increasing difference between processors and dynamic memories speeds causes that the problem of cache performance is an issue of ever growing importance. In the paper it is proposed to increase cache performance in FFT programs by implementing mixed-radix FFTs for $N=2^rK^s$, where K is an odd number, and K^s is close, but smaller than some power of 2. This guarantees that data samples processed in one FFT butterfly have different cache addresses, hence, the number of cache conflicts is substantially diminished. Computer simulations for $K^s=243$, and 125 show that this is the case, indeed, and that in spite of higher numbers of arithmetical operations for conservative cache miss delays the mixed-radix FFTs do perform better than the radix-2 FFT.

1. INTRODUCTION

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, hence, memory reads and writes do not influence algorithms complexity. This has been reflected in the RAM machine model [1], which states that a computer consists of a sequential processor 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 is caused by the fact that firstly, modern computers have hierarchical organization of memory, secondly, even in one-processor machines computations are divided into parallel streams [2]. In general, FFT algorithms are very well suited for parallel computations. On the other hand, they are strongly affected by sophisticated memory organization, and in particular, by cache addressing schemes. This problem has been analyzed in papers [3], [4], where it has been pointed out that for obtaining the best results FFT computation should be performed depth-first; in [4] the use of recursive FFT function has been suggested. Another idea has been presented in [5], where special data alignment consisting in inserting empty records among data blocks has been proposed. The alignment causes that samples processed in one FFT butterfly have different cache addresses, which minimizes cache conflicts, for explanation see section 3.

In this paper an approach to minimization of cache misses in FFT computation based on the use of mixed-radix FFT [6] is proposed. Namely, the most critical first stages of FFT are computed using radix-K FFT for odd K, the number of such stages s is chosen in such a way that K^s is close but smaller than some power of 2, 243- and 125-point FFTs are considered. In section 2 general description of caches is given. Section 3 describes why large radix-2 FFT algorithms are prone to cache misses, and why radix-K are not. Simulations of algorithms are reported in section 4. It is shown that indeed, in spite of higher arithmetical complexity for conservative cache miss penalties mixed-radix FFTs are more time-efficient than the radix-2 FFT, which is due to important reduction of cache misses.

2. CACHE

The cache is a buffer memory between processor registers and main computer memory [2]. 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]). 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 datum is required by a processor, or program instruction loaded for the first time, apart from moving it to the processor register it is also stored in the cache at the address obtained by computing the residue of datum (or instruction) memory address modulo the size of the cache:

$$\textit{cache address} = \textit{memory address} \text{ modulo } \textit{cache\_size} \quad (1)$$

In fact, the address computation can be somewhat more complicated, as in majority of caches loading is done in blocks. Next read or write of the datum (or instruction) 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 tem-

poral 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. Theoretically, 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 very small. They are several compromise solution [2], the widely used one, often serving as a benchmark, is 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, 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. FFT AND DATA CACHE

The radix-2 FFT is built up from 2-point operations called *butterflies*, which for decimation-in-frequency FFT are as follows:

$$x(n) \leftarrow x(n) + x(offset+n),$$
$$x(offset+n) \leftarrow [x(n) - x(offset+n)]W(n),$$

where $W(n)$ is the rotation factor, *offset* depends on the stage of algorithm. Namely, an N-point radix-2 transform consists of $log_2 N$ stages, and in the first stage $offset=N/2$, in the second $offset=N/4$, then $N/8,...,2,1,$ of course, they are $N/2$ butterflies in each stage. Notice that as N is a power of 2, *offset* is also a power of 2.This means that for sufficiently large transform sizes in the first steps of the algorithm $x(n)$ and $x(offset+n)$ have the same address in direct mapping cache (1). This may cause severe degradation of algorithm performance due to cache misses. Consider computation of a simplified butterfly (complex arithmetic and multiplication by rotation factor not implemented for clarity):

```
r1 ← x(offset+n) # cache miss, r0, r1, r2 are registers;
r0 ← x(n)        # cache miss and conflict, x(offset+n)
                 # erased;
r2 ← r0 + r1     # first butterfly operation;
r1 ← r0 – r1     # second operation;
x(n) ← r2        # cache hit;
x(offset+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. Notice that the conflicts are avoided for associative caches, and the number of misses diminish to two, the 2-way associative cash suffices.

It is then proposed to compute these first critical stages of FFT not using the radix-2 butterflies, but radix-K ones for some odd K. This means that we are implementing the mixed-radix FFT algorithm [6] of size $N=2^rK^s$. In radix-K FFTs for odd K index offsets are multiplicities of the number K, which are mutually prime with the cache size. This means that cache addresses of FFT butterfly samples are pseudo-random, hence, they are no conflicts. There is a drawback of such a solution, the sub-transform size K^s does not match precisely the cache size. That is why it is proposed to use FFT modules of size K^s close, but slightly smaller than some powers of 2, the most obvious choices are for K=3: $3^5=243<256=2^8$, and for K=5: $5^3=125<128=2^7$. The second proposal results in "decimal" sizes of the computed transform, they are equal to multiplicities of N=1000. It should be noted that the radix-3 and radix-5 FFTs require somewhat more arithmetical operations to be computed than the radix-2 one of comparable size, For very large transform sizes the radix-3 requires approximately 35% more arithmetical operations than the radix-2 one, while for the radix-5 FFT there is approximately 17% more such operations.

4. SIMULATIONS

We choose Simics simulator for our research [7]. Simics is an efficient, system level, instruction set simulator. This software simulates the target system at the level of individual instructions which are executed one at a time. This is lowest level of hardware that software can acces. Simulating at this level allows simulator to be system level simulator. This term means that simulator models a target computer at the level of operating system. Simics models interface to buses, interrupt controllers, disks etc. Those possibilities allow to check different concepts before implementing them in hardware. This is a very efficient tool for computer architecture research or operating system development.

For our research we choose cache parameters as follow: data cache line was 32 bytes. We could not use shorter cache line because original cache simulation method in Simics requires 32 bytes data cache line. Because of this it has been necessary to make a little change in structure representing a complex number. We had to fill structure to 32 bytes size with two 8 byte numbers. This change was only for ensuring that each cache line contains only one complex number. Our cache is direct-mapped and has 256 lines, which conforms with chosen radix-3 FFT module size, 243 points. Write strategy is write-back. We choose a model for the processor 266 MHz Pentium II, one cache level for easy interpretation of results. To obtain time of algorithm execution it was necessary to assume some access time to cache in case of hit and some access time to main memory in case of cache miss. Read miss was set to 53 cycles similarly as write miss. Read hit penalty was set to 5 cycles and write hit was set to 3 cycles, which is similar to cache parameters of Pentium II processor. In our system there was instruction cache too, but it was not important for our research.

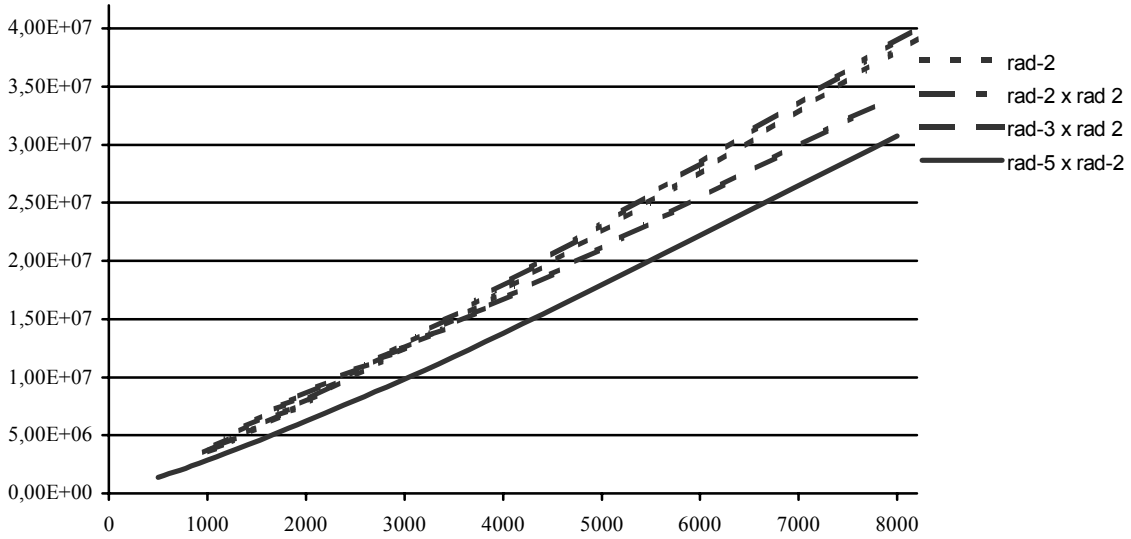

Figure 1 Number of processor cycles as a function of FFT size

To ensure that operating system had not have influence on result, we tested algorithm without operating system. Our program consists of some functions that initiate environment and main function which call function implementing algorithm. Compiled and linked program is loaded to memory using GRUB operating system bootloader.

Simulation results for radix-2 FFT are presented in Table 1, while those for mixed-radix FFTs of size $N=243\cdot 2^r$, and $N=125\cdot 2^r$ in Tables 2 and 3, respectively, and summarized in Figure 1. In Table 4 data for radix-2 FFT version which first 8 stages are realized as 256-point radix-2 FFT are provided. The latter algorithm has exactly the same logical organization as mixed-radix FFTs, only 243-point (or 125-point) module is replaced by the 256-point one. The FFTs are decimation-in-frequency algorithms, each one is followed by digit-reverse function for restoring natural order of data. As can be seen, indeed, the use of non-radix-2 butterflies in first FFT stages resulted in important reduction of cache misses, and hence, in improved performance of the algorithm, with possible exception of FFT sizes close to 2^{11} for $N=243\cdot 2^r$. It should be underlined that this result has been obtained in spite of higher arithmetical complexity of the radix-3 and radix-5 FFT, if compared to the radix-2 FFT one. Moreover, the second version of the radix-2 FFT shows that mixed-radix organization of an FFT program results in its inferior performance, if compared to the straightforward one, which makes the result even more convincing.

5. CONCLUSION

A new technique for improving cache performance in FFT programs has been proposed in the paper. It consists in replacing first, the most critical stages of radix-2 FFT with stages of a radix-K, one, where K is an odd number. In this way there are no cache conflicts between samples of an FFT butterfly, hence, an important source of cache misses in radix-2 FFT avoided. The number of stages s is chosen in such a way, that blocks of FFT samples almost fill the cache, i.e. K^s is close but smaller than a power of 2. Simulation results show that indeed, mixed-radix FFTs have lower time complexities than the radix-2 FFT, mainly due to important reduction of cache misses. This result is obtained in spite of the fact that the used radix-K modules: 243-point, and 125-point ones, require more arithmetical operations per data sample than the 256-point, and 128-point ones. This is a one more proof for the statement that in times of hierarchical computer memories the concept of algorithm complexity should be rethought [2].

REFERENCES

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

[2] J.L. Hennessy, D.A. Patterson, ‘’Computer Architecture, a Quantitative Approach’’, 3nd edition., Elsevier, Morgan-Kaufmann, 2003, see also Internet resources linked with the book.

[3] R. Stasiński, “Efficiency of radix-K Transforms on computer with cache”, Proc. ICASSP'99, Phoenix AZ, vol. III, pp. 1525-1528, 1999.

[4] M. Frigo, Ch. E. Leiserson, H. Prokop, S. Ramachandran, „Cache-oblivious algorithms“, Proc. 40th Annual Symp. on Foundations of Comp. Science (FOCS), pages 285-297, 1999.

[5] P.R. Panda, H. Nakamura, N. D. Dutt, A. Nicolau, „A data alignment technique for improving cache performance”, Proc. ICCD’97, Austin, TX, 1997.

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

[7] P. Magnusson, B. Werner, “Efficient Memory Simulation in SimICS”, 28th Proc. Annual Simulation Symp., Santa Barbara, CA, pp. 62-73, 1995, Internet link: http://www.simics.net.

N	rm	r	%rm	wm	w	%w	cycles
1024	19790	323030	6,13	4098	91491	4,48	3605423
2048	44832	705403	6,36	11304	199319	5,67	8066526
4096	100325	1530752	6,55	28871	431771	6,69	17864207
8192	221664	3299013	6,72	70213	929023	7,56	39154254

Table 1Cache performance for radix-2 FFT – rm-is the number of read misses, r is the number of read transactions, wm-is the number of read misses, w is the number of read transactions.

N	rm	r	%rm	wm	w	%w	cycles
243x4	8606	367596	2,34	3552	96228	3,69	3501991
243x8	36102	802009	4,50	5192	206420	2,52	8369743
243x16	40841	1758072	2,32	3406	447580	0,76	16182218
243x32	56402	3778819	1,49	7205	953714	0,76	33374235

Table 2 Cache performance for mixed radix FFT (radix-3 x radix-2).

N	rm	r	%rm	wm	w	%w	cycles
125x4	3704	135668	2,73	1553	37961	4,09	1357687
125x8	6283	305703	2,06	1774	83038	2,14	2880089
125x16	11026	690648	1,60	1575	183942	0,86	6254498
125x32	24940	1516413	1,64	3529	398000	0,89	13800735
125x64	61551	3326318	1,85	8946	864116	1,04	30733768

Table 3 Cache performance for mixed-radix FFT (radix-5 x radix-2).

N	rm	r	%rm	wm	w	%w	cycles
256x4	11772	387050	3,04	3429	110842	3,09	3691977
256x8	27902	844494	3,30	8993	236270	3,81	8216613
256x16	660033	1851286	3,57	23633	509142	4,64	18413113
256x32	147176	3979302	3,70	55739	1079974	5,16	40026545

Table 4 Cache performance for mixed-radix FFT (radix-2 x radix-2).