A GENERALIZED CACHED-FFT ALGORITHM

Bevan M. Baas

Department of Electrical and Computer Engineering
University of California, Davis

ABSTRACT

Fast Fourier Transform (FFT) algorithms are typically designed to minimize the number of multiplications and additions while maintaining a simple form. Few FFT algorithms are designed to take advantage of hierarchical memory systems, which are easy to include in special-purpose processors, and nearly universal in modern programmable processors. We present a new generalized algorithm, called the *cached-FFT,* which is designed explicitly to operate on a processor with a hierarchical memory system. By taking advantage of a small and fast cache memory, the algorithm enables higher clock frequencies (for special-purpose processor applications), reduced data communication energy, and increased energy-efficiency—since smaller memories require lower energy per access and can be positioned closer to the processor.

1. INTRODUCTION

A distinguishing characteristic of the cached-FFT is that it isolates the high-speed and smaller-memory portion of the processor from the large main memory. The algorithm allows repeated accesses of data from a faster and smaller level of the memory hierarchy. This characteristic offers several advantages over methods which do not exploit the use of data caches, such as increased performance and energy-efficiency. The algorithm also presents several disadvantages, including the addition of new functional units (caches) if they are unavailable, and added controller complexity.

First fully described in 1999 [1], the cached-FFT algorithm has been used to develop at least three custom FFT processors: a 1024-point FFT chip [2], a 512-point 2D FFT chip [3], and a programmable 64–2048-point FFT chip [4]. The 1024-point chip has an energy-efficiency 16 times greater and a clock rate 2.6 times higher than other known FFT processors at the time of publication.

An algorithm which exploits a hierarchical memory system must specify its memory access patterns, as the order of memory accesses strongly effects performance. This paper presents the butterfly addresses and W_N exponent control signals independent of radix and in such a way that memory access patterns can be rearranged while maintaining correct operation and maximum reuse of data in the cache. Because the simple and regular structures of radix-r and "in-place" algorithms make their use attractive, the cached-FFT we present is also radix-r and in-place.

This work was supported by an NSF Graduate Fellowship, NASA GSRP Fellowship NGT-70340, Intel Corporation, NSF Grant CCF 430090, and a UCD Faculty Research Grant.

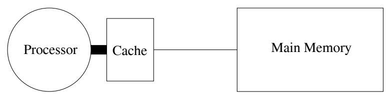

Fig. 1. Cached-FFT processor block diagram

2. OVERVIEW OF THE CACHED-FFT

2.1. Basic Operation

The cached-FFT algorithm utilizes an architecture with a small cache memory positioned between the processor and main memory, as shown in Fig. 1. The C-word cache has significantly lower latency and higher throughput than the main memory since normally, $C \ll N$, where N is the length of the FFT, and the main memory has at least N words.

The FFT caching algorithm operates with the following procedure:

1. N input data are loaded into main memory.
2. C of the N words are loaded into the cache.
3. As many butterflies as possible are computed using the data in the cache.
4. Processed data in the cache are flushed to main memory.
5. Steps 2–4 are repeated until all N words have been processed once.
6. Steps 2–5 are repeated until the FFT has been completed.

The FFT algorithm does not naturally map to a hierarchical memory structure as can be seen in the dataflow diagram of a common 64-point radix-2 FFT in Fig. 2. The 64 memory locations are indicated along the vertical axis and computation flows from left to right. The diagram shows the global nature of the algorithm where every output depends on every input.

2.2. Definitions

To aid in the introduction of the cached-FFT algorithm, we define several new terms:

An *epoch* (E) is the portion of the cached-FFT algorithm where all N data words are loaded into a cache, processed, and written back to main memory *once*. Normally, $E \geq 2$. Steps 2–5 in the list of Sec. 2.1 comprise an epoch.

A *group* is the portion of an epoch where a block of data is read from main memory into a cache, processed, and written back to main memory. Steps 2–4 in the list of Sec. 2.1 comprise a group.

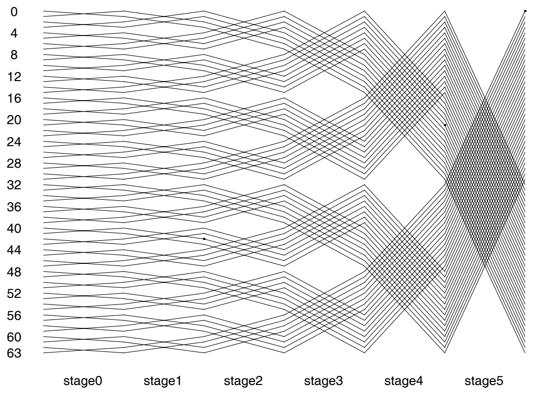

Fig. 2. Dataflow diagram for a 64-point radix-2 FFT

A *pass* (P) is the portion of a group where each word in the cache is read, processed with a butterfly, and written back to the cache *once*.

A cached-FFT is *balanced* if there are an equal number of passes in the groups from all epochs. Balanced cached-FFTs do not exist for all FFT lengths. The transform length of a balanced cached-FFT is constrained to values of r^{EP}, where r is the radix of the decomposition.

3. FFT ALGORITHMS SIMILAR TO THE CACHED-FFT

In 1966, Gentleman and Sande [5] proposed an FFT algorithm that makes use of a "hierarchical store." The method they propose selects a transform length N that has two factors: $N = AB$. The factor A is the "largest Fourier transform that can be done in the faster store." Their decomposition can also be viewed as a single radix-A or single radix-B decimation of the input sequence. Published in 1967, Singleton's approach [6] is perhaps of limited use for modern processors because of its intended application which uses "serial-organized memory files, such as magnetic tapes or serial disk files." In 1969, Brenner [7] proposed two algorithms—one is better suited for cases where the length of the transform is not much longer than the size of the fast memory, and the second for cases where the transform length is much longer. Rabiner and Gold [8] discuss a method they call "FFT computation using fast scratch memory." Other similar algorithms have been published by a number of other researchers [1, Ch. 4]. All of these previously-published algorithms can be viewed as two-epoch cases of the more general cached-FFT, providing less insight into how a balanced epoch structure can be achieved or how the memory accesses, cache accesses, and processor design can be optimized.

4. THE CACHED-FFT ALGORITHM

To simplify its presentation, we initially consider only *balanced* cached-FFTs. Furthermore, we consider only DIT decompositions, and note that the development of DIF algorithms follows a similar procedure.

Operation of the cached-FFT requires several counters: *epoch*, *group*, *pass*, and *butterfly*. Table 1 shows one arrangement of the *group* and *butterfly* counters that allows a subset of the FFT to be calculated in $r^{\log_r(C)} = C$ memory locations. Other analogous mappings are possible. The W_N coefficients are generated using bits within the *group* and *butterfly* counters as shown.

Although the radix-2 approach is easiest to visualize, the description given by the table applies to algorithms with other radices as well. For use with radices other than two, the counter digits (*e.g.,* $g_1, *, b_0, \ldots$) are interpreted as base-r digits instead of binary digits ($\in [0, 1]$) as in the radix-2 case. Higher-radix algorithms generally require more than one W_N coefficient, so although Table 1 gives only one W_N, it serves as a base factor where other coefficients are normally multiples of the given value.

The asterisks represent digit positions where different values in that position address the r inputs and r outputs of the butterflies. For example, in the radix-2 case, a "0" in place of the asterisk addresses one input and one output of the butterfly, and a "1" in the asterisk's position addresses the other input and output.

The tables do not specify a particular order in which to calculate the FFT. The subscripts of the counter digits g and b show only the relationship between address digits and W_N-generating digits. The digits can be incremented in any order or pattern—the only requirements are that all N/r butterflies are calculated once and only once and that epochs must be calculated in order.

In every cached-FFT, the following are true: (i) Across any epoch, the positions and values of the *group* counter digits are constant. (ii) Within each epoch, the memory address pattern is identical for the $\log_r(C) - 1$ address digits not connected to the *group* counter. These digits are the $\log_r(C) - 2$ *butterfly* digits plus the one "$*$" digit.

4.1. Implementing the Cached-FFT

As with any FFT, the length (N) and radix (r) must be specified. The cached-FFT also requires the selection of the number of epochs (E). Although the computed values presented below are derived from $N, r,$ and E—and are therefore unnecessary—we introduce new variables to clarify the implementation of the algorithm.

For a balanced cached-FFT, the number of passes per group is $\log_r(N)/E$ and the cache size is $C = \sqrt[E]{N}$. For an unbalanced cached-FFT, the cache size is $C = r^{\left\lceil \frac{\log_r N}{E} \right\rceil}$. For balanced cached-FFTs, the number of groups per epoch is N/C and the number of butterflies per pass is C/r. In some cases, the cache size and the transform length are fixed, and the number of epochs will then be $E = \left\lceil \frac{\log_r N}{\lfloor \log_r C \rfloor} \right\rceil$.

Without a cache, data are read from and written to main memory $\log_r N$ times. With a cache, data are read and written to main memory only E times. Therefore, the reduction is memory traffic is $\log_r(N)/E$.

Software implementations of the cached-FFT algorithm on programmable processors (with fixed cache sizes) can yield significant performance and energy improvements. Details of a cached-FFT implementation on a Mitsubishi D30V DSP processor that resulted in a 32% reduction in computation time, or a $1.47\times$ speedup have been reported [1].

Although the description of the cached-FFT given here is sufficient to generate a wide variety of cached-FFT algorithms, additional variations are possible. Many alternatives are developed by varying the placement of data words in main memory and the cache. Partitioning the main memory and/or cache into multiple banks will increase memory bandwidth and alter the memory address mappings.

Epoch number	Pass number	Butterfly Addresses ($\underline{\text{x}}$ = one base-r digit)				W_N exponents
0	0	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$	$000 \cdots 000$
	1				$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$	$b_0 00 \cdots 000$
	$\vdots$				$\vdots$	$\vdots$
	$\log_r(N)/E-1$				$\underline{*} \cdots \underline{b_2}\ \underline{b_1}\ \underline{b_0}$	$b_{\log_r(C)-2} \cdots b_1 b_0 0 \cdots 00$
1	0	$\underline{g_{\log_r(N/C)-1}} \cdots \underline{g_{\log_r(N/C)-\log_r(C)}}$	$\cdots$	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$	$g_{\log_r(C)-1} \cdots g_1 g_0 0 \cdots 00$
	1			$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$		$b_0 g_{\log_r(C)-1} \cdots g_1 g_0 0 \cdots 00$
	$\vdots$			$\vdots$		$\vdots$
	$\log_r(N)/E-1$			$\underline{*} \cdots \underline{b_2}\ \underline{b_1}\ \underline{b_0}$		$b_{\log_r(C)-2} \cdots b_1 b_0 g_{\log_r(C)-1} \cdots$ $\cdots g_1 g_0 0 \cdots 00$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$E-1$	0	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{b_0}\ \underline{*}$	$\cdots$	$\underline{g_{2\cdot\log_r(C)-1}} \cdots \underline{g_{\log_r C}}$	$\underline{g_{\log_r(C)-1}} \cdots \underline{g_1}\ \underline{g_0}$	$g_{\log_r(N/C)-1} \cdots g_1 g_0 0 \cdots 00$
	1	$\underline{b_{\log_r(C)-2}} \cdots \underline{b_1}\ \underline{*}\ \underline{b_0}$				$b_0 g_{\log_r(N/C)-1} \cdots g_1 g_0 0 \cdots 00$
	$\vdots$	$\vdots$				$\vdots$
	$\log_r(N)/E-1$	$\underline{*} \cdots \underline{b_2}\ \underline{b_1}\ \underline{b_0}$				$b_{\log_r(C)-2} \cdots$ $\cdots b_1 b_0 g_{\log_r(N/C)-1} \cdots g_1 g_0$

Table 1. Memory addresses for an N-point, balanced, radix-r, DIT cached-FFT. The variables g_k and b_k represent the k^{th} digits of the *group* and *butterfly* counters respectively. Asterisks ($*$) represent digit positions where the r possible values address the r butterfly inputs and r outputs. Note the values of digits in the *group* counter are constant within epochs.

Epoch number	Pass number	Butterfly address digits ($\underline{x}$ = one base-r digit)	W_N butterfly coefficients
0	0	$\underline{g_2}\ \underline{g_1}\ \underline{g_0}\ \underline{b_1}\ \underline{b_0}\ \underline{*}$	W_{64}^{00000}
	1	$\underline{g_2}\ \underline{g_1}\ \underline{g_0}\ \underline{b_1}\ \underline{*}\ \underline{b_0}$	$W_{64}^{b_0 0000}$
	2	$\underline{g_2}\ \underline{g_1}\ \underline{g_0}\ \underline{*}\ \underline{b_1}\ \underline{b_0}$	$W_{64}^{b_1 b_0 000}$
1	0	$\underline{b_1}\ \underline{b_0}\ \underline{*}\ \underline{g_2}\ \underline{g_1}\ \underline{g_0}$	$W_{64}^{g_2 g_1 g_0 00}$
	1	$\underline{b_1}\ \underline{*}\ \underline{b_0}\ \underline{g_2}\ \underline{g_1}\ \underline{g_0}$	$W_{64}^{b_0 g_2 g_1 g_0 0}$
	2	$\underline{*}\ \underline{b_1}\ \underline{b_0}\ \underline{g_2}\ \underline{g_1}\ \underline{g_0}$	$W_{64}^{b_1 b_0 g_2 g_1 g_0}$

Table 2. Addresses and W_N coefficients for a 64-point, radix-2, DIT, 2-epoch cached-FFT

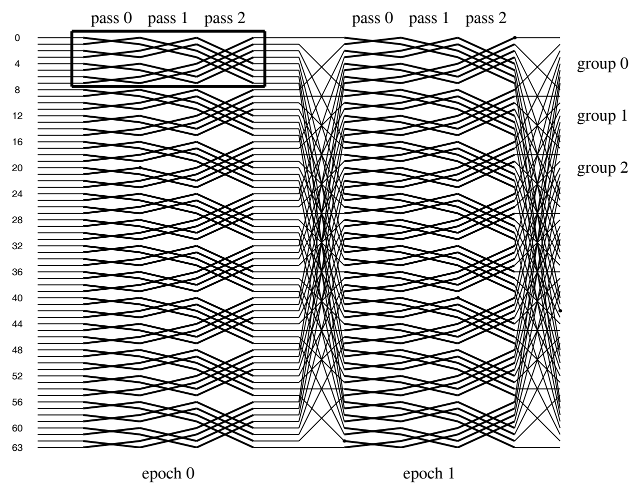

Fig. 3. Dataflow diagram for a 64-point, radix-2, 2-epoch cached-FFT with an 8-word cache

4.2. Example: $N = 64$, $E = 2$, Radix-2 Cached-FFT

To illustrate how the cached-FFT works, we now consider a cached-FFT implementation of a $N = 64$, radix-2, DIT FFT. We choose two epochs ($E = 2$) for this cached-FFT example, which implies a cache size of $C = \sqrt[E]{N} = \sqrt[2]{64} = 8$ words. Table 2 provides the address digit positions and W_N coefficients for a 64-point cached-FFT.

Two epochs of three passes each use three *group* counter digits (g_2, g_1, g_0) that are fixed across both epochs. The *butterfly* counter and asterisk digit ($b_1, b_0, *$) positions are the same in both epochs.

Figure 3 shows the flow graph of the 64-point cached-FFT. Radix-2 butterflies are drawn with heavier lines, and loads and stores between main memory and the cache—which involve no computation—are drawn with lighter-weight lines. A box encloses an 8-word group to show butterflies calculated together from the cache.

5. SUMMARY

The generalized cached-FFT algorithm is suitable for FFT transforms of any length and radix, and is presented in a form that simplifies simultaneous optimization of the cache size and the FFT algorithm. Significantly increased levels of performance and energy-efficiency are possible with both hardware and software implementations.

6. REFERENCES

[1] B. M. Baas, *An Approach to Low-Power, High-Performance Fast Fourier Transform Processor Design*, Ph.D. thesis, Stanford University, Stanford, CA, USA, Feb. 1999.

[2] B. M. Baas, "A low-power, high-performance, 1024-point FFT processor," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 3, pp. 380–387, Mar. 1999.

[3] N. Miyamoto, L. Karnan, K. Maruo, K. Kotani, and T. Ohmi, "A small-area high-performance 512-point 2-dimensional FFT single-chip processor," in *European Solid-State Circuits Conference*, Sept. 2003, pp. 603–606.

[4] J.-C. Kuo, C.-H. Wen, and A.-Y. Wu, "Implementation of a programmable 64–2048-point FFT/IFFT processor for OFDM-based communication systems," in *IEEE International Symposium on Circuits and Systems*, May 2003, vol. 2, pp. 121–124.

[5] W. M. Gentleman and G. Sande, "Fast fourier transforms—for fun and profit," in *AFIPS Conference Proceedings*, Nov. 1966, vol. 29, pp. 563–578.

[6] R. C. Singleton, "A method for computing the fast fourier transform with auxiliary memory and limited high-speed storage," in *IEEE Transactions on Audio and Electroacoustics*, June 1967, vol. AU-15, pp. 91–98.

[7] N. M. Brenner, "Fast fourier transform of externally stored data," in *IEEE Transactions on Audio and Electroacoustics*, June 1969, vol. AU-17, pp. 128–132.

[8] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975.