for image coding. This algorithm jointly searches for the best spatial segmentation and the best frequency decomposition to use for each segment. The main advantage of these adaptive representations is their versatility: They can adapt to a wide variety of image classes having varying space-frequency characteristics by searching efficiently through a very large library of tree-structured bases. Numerically, on a SPARC 5, calculating a 4-level wavelet transform of a 512×512 image took 1.08 s, while calculating the best single-tree basis (from among 4.9×10^{19} bases) took 5.65 s, and calculating the best double-tree basis (from among 5.6×10^{78} bases) took 21.18 s.

REFERENCES

[1] C. Herley, J. Kovacevic, K. Ramchandran, and M. Vetterli, "Tilings of time-frequency plane: Construction of arbitrary orthogonal bases and fast tiling algorithms," *IEEE Trans. Signal Processing*, vol. 41, no. 12, pp. 3341–3360, Dec. 1993.

[2] A. K. Jain, *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[3] H. S. Malvar, *Signal Processing With Lapped Transforms*. Norwood, MA: Artech, 1992.

[4] I. Daubechies, "Orthonormal bases of compactly supported wavelets," *Commun. Pure Appl. Math.*, vol. XLI, pp. 909–996, 1988.

[5] S. G. Mallat, "A theory for multiresolution signal decomposition: The wavelet decomposition," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 11, pp. 674–693, 1989.

[6] R. Coifman and V. Wickerhauser, "Entropy-based algorithms for best basis selection," *IEEE Trans. Inform. Theory*, vol. 38, pp. 713–718, Mar. 1992.

[7] K. Ramchandran and M. Vetterli, "Best wavelet packet bases in a rate-distortion sense," *IEEE Trans. Image Processing*, vol. 2, no. 2, pp. 160–176, Apr. 1993.

[8] K. Asai, K. Ramchandran, and M. Vetterli, "Image representation using time-varying wavelet packets, spatial segmentation and quantization," in *Proc. CISS*. Baltimore, MD: Johns Hopkins Univ., Mar. 1993.

[9] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform," *IEEE Trans. Image Processing*, vol. 1, no. 2, pp. 205–221, Apr. 1992.

[10] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Processing*, vol. 41, no. 12, pp. 3445–3463, Dec. 1993.

[11] Z. Xiong, K. Ramchandran, and M. Orchard, "Joint optimization of scalar and tree-structured quantization of wavelet image decomposition," in *Proc. Asilomar Conf.*, Pacific Grove, CA, Nov. 1993, vol. 2, pp. 891–895.

[12] Z. Xiong, K. Ramchandran, M. T. Orchard, and K. Asai, "Wavelet packets-based image coding using joint space-frequency quantization," in *Proc. ICIP'94*, Austin, TX, Nov. 1994, vol. III, pp. 324–328.

Cache Write Generate for Parallel Image Processing on Shared Memory Architectures

Craig M. Wittenbrink, Arun K. Somani, and Chung-Ho Chen

***Abstract*—We investigate *cache write generate*, our cache mode invention. We demonstrate that for parallel image processing applications, the new mode improves main memory bandwidth, CPU efficiency, cache hits, and cache latency. We use register level simulations validated by the UW-Proteus system. Many memory, cache, and processor configurations are evaluated.**

I. INTRODUCTION

Cache memories play an important role in achieving higher performance in modern uni- and multiprocessors. When a high percentage of reads and writes are made to the cache, the effective bandwidth of the memory is that of the cache. Many prior studies have focused on read caching [7]. Here, we focus on write caching. Write buffers [7], write allocate [4], and write through [1], [5] do not address the removal of unnecessary traffic. To prevent unnecessary reads, many systems provide software control of cache write updating [6]. Word validate has been used by [1], [4], and [5], and write allocate has been used by [4].

C. M. Wittenbrink, in [9], investigated the effect of directly updating the line when it was known in advance that the line is to be written by using trace analysis. In this paper, we further investigate the cache write technique *cache write generate*. Cache write generate directly updates the cache on write misses, without reading from memory. We show that for a class of applications, the overall performance improvement is significant. We performed the analysis using hardware description language (HDL) simulations and performance measurements of each cache write technique.

Cache write generate (CWG) is defined as cache write validation on a write miss. The cache line is updated with the write and the cache line tag is modified to the address of the write. Writes that benefit from CWG are computed or initialized by the processor. Examples include dynamically allocated memories, stack segments, static memory segments, and temporary buffers. In image processing and vision applications [3], [8], these memory areas are easy to identify through explicit declaration or by the compiler. CWG is done only on memory areas denoted as generate, and a cache line in a generate memory area may lose its CWG ability to insure memory consistency. We have developed several schemes to provide self consistency, but do not discuss them due to space constraints. See our paper [10] for details.

II. SIMULATION MODELS AND HARDWARE SYSTEM

A. Cache Modes, Sizes, and Memory Timings

We compare the relative efficiency of the cache write generate policy to existing write caching controls, using single and multipro-

Manuscript received February 18, 1995; revised September 27, 1995. This work was supported by the Navy Coastal Systems Center. The associate editor coordinating the review of this correspondence and approving it for publication was Prof. A. M. Tekalp.

The authors are with the Department of Electrical Engineering, University of Washington, Seattle, WA 98195 USA.

C. M. Wittenbrink is currently at the Baskin Center for Computer Engineering, University of California, Santa Cruz, CA 95064 USA (e-mail: craig@cse.ucsc.edu).

Publisher Item Identifier S 1057-7149(96)03172-7.

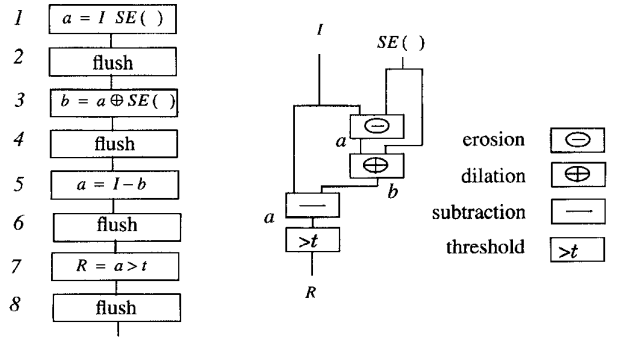

Fig. 1. Bright feature detection morphological algorithm.

TABLE I
INSTRUCTION COUNTS, *os* PROGRAM

Type	Number	Percentage
Loads/Stores	173,545	26%
Delayed Branches	56,797	8%
Other	444,308	66%
Total	674,650	100%

cessor shared memory systems. We developed a detailed model of the Intel i860 RISC processor, a custom external cache, and a main memory using register transfer level simulation language, ISP-prime, used with the N.2 simulator. The simulation models were developed for architectural investigation while designing the UW-Proteus system [2], [3], [8]. The UW-Proteus cluster hardware (currently 32 i860's) was used for verification and performance measurement. The simulation models are shared memory multiprocessors, where we varied memory timings, cache sizes, cache control, number of processors, and external or no external caches. For performance comparisons, we consider the following three scenarios for caching. Case N, normal mode, is where the application is run with normal write back caching, with write around on miss. Read misses are cached. Case A, allocate mode, is write caching on write misses in addition to read caching. Cache lines are first fetched from main memory and then updated in the cache. Case G, generate mode, is write caching without fetches on write misses for generate areas. For nongenerate areas, we use the normal mode (write allocate can also be used). To investigate the added complexities of performance with thrashing, we simulated two cache sizes 64k and 256k bytes. We also simulated systems with no external caches where the caching behavior of the i860 on-chip cache was modified to include cache generate. For the first level cache study, we use the same size cache as the i860, an 8k byte data cache.

To investigate the effect of write posting (a nonblocking operation) and replacement policies, we have also simulated two fundamentally different external caches, cache *x* and cache *y*. Cache *x* uses no write posting, no wraparound fills, and no posted replacement or other enhancements. The simplest control is used to see how these devices may have influenced the relative performance of CWG. Cache *y* (used in the implemented digital hardware of the UW-Proteus system) uses posted writes, wraparound fills, and posted replacements.

In all systems, on chip cache, secondary caches, and main memory operate at progressively slower speeds. Let t_p be processor clock time. The secondary cache time is $k_s t_p$ and the main memory time is $k_m t_p$. In our simulations, for the secondary cache we have $k_s = 2$. For the main memory we have three models, fast $k_m = 2(4)$, medium $k_m = 4(5)$, and slow $k_m = 20(21)$ where the numbers in parentheses are those for the first bus cycle in a burst mode. For the i860, $t_p = 25$ ns. The one-level cache system also uses these main memory timings.

TABLE II
HIT RATIOS FOR EXTERNAL CACHES, *os* PROGRAM

Size	Mode	Read	Write	Combined
64k	N	0.7558	0.0364	0.2918
	A , G	0.7552	0.8750	0.8325
256k	N	0.8285	0.4243	0.5678
	A , G	0.9420	0.9375	0.9391

TABLE III
EXTERNAL CACHE REQUESTS, *os* PROGRAM

Cache	Line Fills	External Requests
Data Cache	8,987	35,948
Inst. Cache	35	137
Write Buffer		65,536
Total		101,621

B. Workload

We benchmarked our cache variations with image processing applications using a mathematical morphology algorithm of bright feature detection shown in Fig. 1. I is the input image operated on by the structuring element $SE()$. Memory is used most efficiently by using temporary images a and b, and processing in the eight-step program shown along the left side of Fig. 1. Additionally, we optimized the algorithm to use a minimal amount of memory shown by the buffers labeled in Fig. 1 by I, a, b, and R. Buffer a is reused, which helps caching. Flushes are only necessary for the parallel version SPSD, below. To execute the task graph of Fig. 1, it is partitioned using two types of parallelism. SPMD (*nd* variant), single programming multiple data, the data for each task is strictly partitioned; and SPSD (*os* variant), single programming single data, each function is computed by all of the processors. This uses finer grained sharing. Data are split for processing, each part given to a separate processor. For four processors, each processor works on $1/4$ of the job. In the UW-Proteus system, we use 1M cache and 256×256 images. So, for 128×128 images, 256K cache was the chosen scaling.

III. SIMULATION AND RESULTS

We have grouped the results into three different cases: i) simulation results when the processor's on-chip cache model remains the same but the secondary cache uses different modes; ii) simulation results when only one level (on-chip) cache is used; and iii) the UW-Proteus system measurements where the secondary cache is programmed to use generate, allocate, or normal write caching.

A. Secondary Cache Results

With the mix of instructions given in Table I, the on-chip cache behavior of the i860 is the same regardless of secondary cache modes. For the external cache, allocate and generate give exactly the same hit ratios for reads and writes (see Table II). Allocate and generate are differentiated by read and write miss penalties. This affects program performance, which can be seen through the number of bus cycles they use, the number of load stalls, and the run time.

B. Bus Cycles

The external cache uses generate to reduce the number of bus cycles to main memory. To illustrate, we present all of the cycles in the system for this program. The on-chip cache loads, cache stores, and instruction cache misses create read, write, line fill, and line flush requests on the bus outside of the processor. These requests are serviced by the external cache. Since the on-chip (i860) behavior for all modes in the secondary cache is the same, the number of external requests are the same for all three modes and these are summarized in Table III for the *os* program. For this program, there

TABLE IV
SHARED MEMORY BUS CYCLES, *os* PROGRAM

Cache	Mode	Writes	Burst Writes	Burst Reads	Total
64k	N	63,152	298	8,812	99,592
	A	0	7,936	17,024	99,840
	G	0	7,936	8,832	67,072
256k	N	37,728	0	6,188	62,480
	A	0	0	6,188	24,752
	G	0	0	2,092	8,368

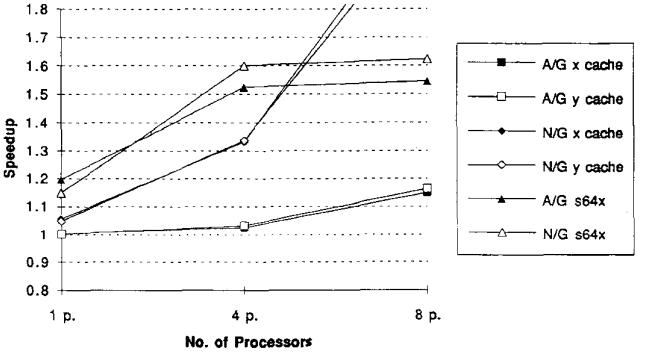

Fig. 2. *os* program speedup of generate versus normal and allocate, with one, four, and eight processors in two-level cache system.

are no replacements, and the number of memory references by the on-chip execution unit is 848 195. About 88% of the memory requests are satisfied on chip, and therefore, the number of external cycles is only 101 621 for data cache line fills, instruction cache line fills, and writes.

Table IV is the most concise demonstration of the difference between normal, allocate, and generate cache modes. In this table, we show the number of bus cycles due to single write, burst writes, and burst reads (to fill cache lines). The final column in Table IV is the total number of external reads and writes to main memory by one processor. In a multiprocessor system with n processors, there will be n times as many bus cycles (external memory read and write cycles). Generate has fewer bus cycles than allocate or normal.

C. Load Stalls

In varying our external cache model, the number of stalled loads varies. A load stall is a memory load operation that cannot be satisfied in a single processor cycle, because the data is not available on chip, and/or there is bus contention. The 64k caches in all modes have 19 852 stalled loads, or 18.38% of all loads. The 256k caches are large enough for no replacements and are more efficient than 64k caches. The number of stalled loads for 256k caches is N (= 18 424), A and G (= 17 777). Because the generate mode writes are more efficient, fewer loads are stalled for that cache. Increasing the cache size reduces stalled loads by 7.19%. Increasing the cache size and using A or G reduces stalled loads by 11.67%, an improvement of 3.64% over 64k cache N mode.

D. Performance

For our two application programs, the speedup of mode G over mode A (A/G in Figs.) and mode G over mode N (N/G in Figs.) in external caches are shown in Figs. 2–5. We use speedup as defined by the following: speedup = (execution $\text{time}_{\text{original}}$)/(execution $\text{time}_{\text{enhanced}}$). The speedups for 256K cache *os* program, 64K cache *os* program, and 64K cache *nd* program are also given in Tables V-VII.

The results in Fig. 2 are speedups for 256K cache fast memory with respect to the number of processors. The speedups show that both

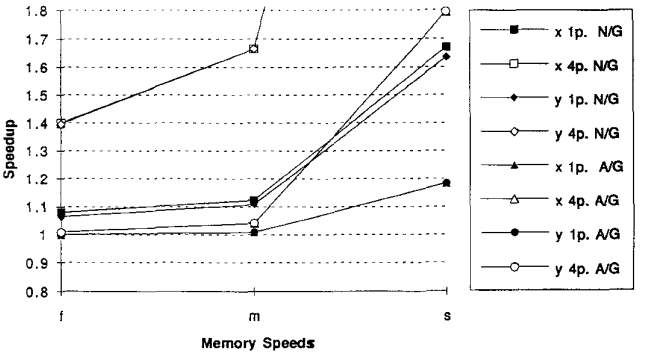

Fig. 3. *os* program speedup of generate versus normal and allocate, fast, medium, and slow memories in two-level cache is 256k.

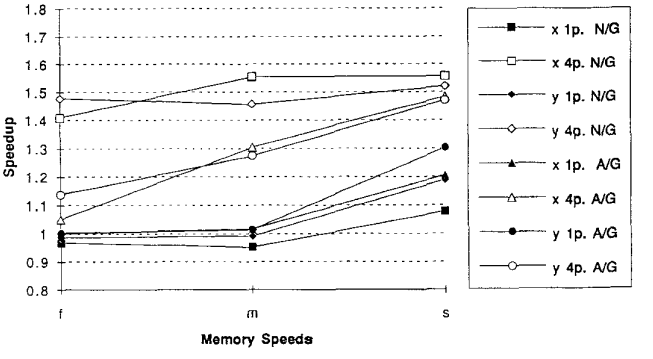

Fig. 4. *os* program speedup of generate versus normal and allocate, fast, medium, and slow memories in two-level cache is 64k.

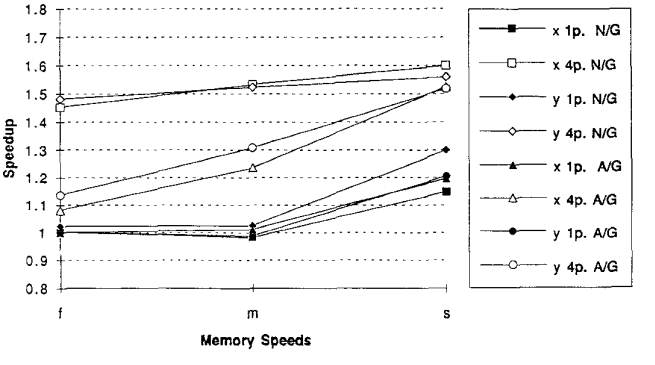

Fig. 5. *nd* program speedup, generate versus normal and allocate, fast, medium, and slow memories in two-level cache is 64k.

allocate and generate perform the same using up to four processors. Beyond that, generate outperforms allocate due to fewer bus cycles and, hence, less contention on the bus. This shows that generate is an effective technique for increasing the number of processors on a shared bus. Generate performs better in comparison to the normal caching mode even when only four processors are sharing the bus.

Figs. 3–5 (and Tables V–VII) show the speedups with respect to the memory speed (f, m, or s) for different cache sizes (256K or 64K), different cache implementations (x or y) and different programs (*os* or *nd*). These figures show that generate improves performance by a greater amount when the memory is slower. This is as expected, because we are incrementally improving a small percentage of the program—the writes. Generate yields a significant speedup over mode A for all systems with slow memories in the single-processor case. For the more sophisticated y cache model, we achieve a greater

TABLE V
SPEEDUP FOR THE *os* PROGRAM 256K SECONDARY CACHE

No. Proc.	1 p						4 p						8 p	
Memory	f		m		s		f		m		s		f	
Cache	x	y	x	y	x	y	x	y	x	y	x	y	x	y
N/G	1.08	1.06	1.12	1.11	1.67	1.64	1.40	1.40	1.66	1.67	4.13	4.13	2.39	2.39
A/G	1.00	1.00	1.01	1.01	1.18	1.18	1.01	1.01	1.04	1.04	1.79	1.79	1.08	1.08

TABLE VI
SPEEDUP FOR THE *os* PROGRAM 64K CACHES

No. Proc.	1 p						4 p					
Memory	f		m		s		f		m		s	
Cache	x	y	x	y	x	y	x	y	x	y	x	y
N/G	0.97	0.99	0.95	0.99	1.08	1.19	1.41	1.48	1.55	1.46	1.56	1.52
A/G	1.00	1.00	1.02	1.01	1.21	1.30	1.05	1.14	1.30	1.27	1.48	1.47

TABLE VII
SPEEDUP FOR THE *nd* PROGRAM 64K CACHES

No. Proc.	1 p						4 p					
Memory	f		m		s		f		m		s	
Cache	x	y	x	y	x	y	x	y	x	y	x	y
N/G	1.00	1.02	0.98	1.02	1.15	1.30	1.45	1.48	1.53	1.52	1.60	1.56
A/G	1.00	1.00	1.01	0.99	1.20	1.21	1.08	1.14	1.24	1.31	1.52	1.52

speedup, which shows that generate is improved even with fills and flush postings.

E. Single-Level Cache Memory

We also performed simulations using only one level on-chip cache by modifying the i860 cache to support generate. We did not change the size of the on-chip cache (it remained 8K for data cache and 4K for instruction cache), simulated two memory speeds, fast (f) and medium (m), and varied the number of processors sharing a single bus from one to eight. The speedups for the *nd* program are shown in Fig. 6 and Table VIII, and the run times are shown in Fig. 7. Generate has a lower number of load stalls and external bus cycles than allocate. It is interesting to note that the normal mode performs best with one processor, but is taken over by generate with four processors and then by generate and allocate with eight processors. Generate achieves 17% speedup for the fast memory and 32% speedup for the medium-speed memory over normal caching when four processors are sharing the bus. The corresponding speedups over allocate are 22% and 27%. With eight processors sharing the bus, speedups achieved by generate are even better. These results, again, show how generate improves performance on a shared bus.

F. UW-Proteus Performance Results

Lastly, we ran the *nd* program for 256 × 256 32-b integer images and an optimized matrix multiplication assembly program to multiply two 256 × 256 floating point matrices on UW-Proteus. We used normal, generate, and allocate caching modes for the secondary cache with one and four processors. Measured speedups of generate over normal were 18% for the *nd* program. The speedup was only 0.8% for the matrix multiplication. The write frequency is less than 0.4% for the matrix multiplication, and not much speedup is as expected for one processor with the fast memory sytem used to implement UW-Proteus. Note that we implemented CWG in hardware and validated our simulation models.

Table IX shows how simulated performance results compare to measured UW-Proteus performance on the *os* program. The program

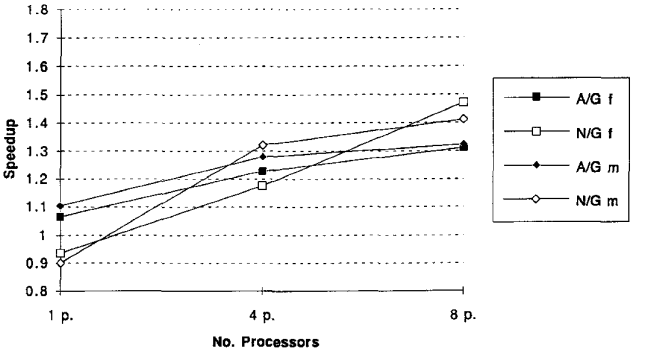

Fig. 6. *nd* program speedup, generate versus normal and allocate, with one, four, and eight processors in one-level cache system.

TABLE VIII
SPEEDUP FOR THE *nd* PROGRAM SINGLE LEVEL CACHE

No. Proc.	1 p		4 p		8 p	
Memory	f	m	f	m	f	m
N/G	0.94	0.90	1.17	1.32	1.47	1.41
A/G	1.06	1.10	1.23	1.28	1.31	1.32

was parallelized for two and four processors, and contained synchronization code for barriers between the respective operations. The speedup of generate over allocate is 3% for timings taken from a four-processor program running on UW-Proteus. As predicted the performance impact is small; the simulation showed speedups of 1% and 4% on a four-processor system. Results are duplicated here from Tables V and VI. The timings help verify the proper modeling of the system.

IV. CONCLUSIONS

Generate improves performance of multiprocessor applications significantly over allocate and write back caching modes by altering

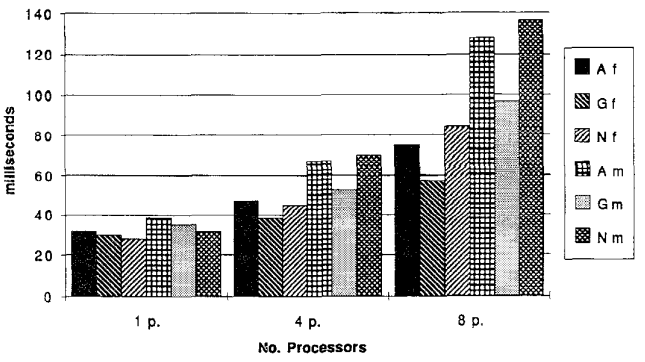

Fig. 7. *nd* program timings of generate, normal, and allocate, with one, four, and eight processors in one-level cache system.

TABLE IX
COMPARISON OF UW-PROTEUS MEASURED SPEEDUP FOR THE *os* PROGRAM TO SIMULATED SPEEDUP WITH y (POSTING, LOCKUP FREE) CACHES

No. Proc.	1 p	2 p	4 p
A/G	1.0079	1.0065	1.03046
Simulation 64k cache A/G fast mem.	1.00	-	1.14
Simulation 256k cache A/G fast mem.	1.00	-	1.01
Simulation 256k cache A/G medium mem.	1.01	-	1.04

the second-level cache. When generate is incorporated on chip, even greater improvements are achieved. Allocate and generate have the same hit ratio, but generate significantly reduces the number of bus cycles by making write misses more efficient. For our image processing application, generate reduced the number of bus cycles in a multilevel cache system by 33% to 66%. Performance is weakly coupled with trace results in the case of write behaviors, and the improvement in generate over allocate averaged about 20%. The reduction in bus cycles is achievable without rewriting programs, and allows shared bus systems to use greater numbers of processors. Program performance can be improved in single-processor systems as well, where the contention between loads, stores, branches, and instructions is reduced by decreasing cache write miss service times.

REFERENCES

[1] W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 processor memory element," in *Proc. 1985 Int. Conf. Parallel Processing,* Aug 1985, pp. 782–789.

[2] S. B. Choi *et al.*, "Reconfigurable multiprocessor system having circuit-switched routing, cache write generation, and fault tolerant synchronization," U.S. pat. pend., 1992.

[3] R. M. Haralick *et al.*,"UW-Proteus: a reconfigurable computational network for computer vision," in *J. Machine Vision Applic.*, vol. 8, no. 2, pp. 85–100.

[4] N. P. Jouppi, "Cache write policies and performance," in *20th Annual Int. Symp. Comput. Architect.*, San Diego, CA, May 1993, pp. 191–201.

[5] *MC68030 Enhanced 32-bit Microprocessor User's Manual,* Motorola Inc, 1987.

[6] G. Radin, "The 801 minicomputer," in *Proc. Symp. Architect. Support Program. Lang. Operating Syst.*, Palo Alto, CA, Mar. 1982, pp. 39–47.

[7] A. J. Smith. "Cache memories,"*Comput. Surveys*, vol. 14, no. 3, Sept. 1982.

[8] A. K. Somani *et al.*, "UW-Proteus system architecture & organization," in *Proc. 5th Int. Parallel Processing Symp.*, Anaheim, CA, Apr.–May, 1991, pp. 276-284.

[9] C. M. Wittenbrink, "Directed data cache for high performance morphological image processing," Master's thesis, University of Washington, Dept. of Elect. Eng., Oct. 1990.

[10] C. M. Wittenbrink, A. K. Somani, and C.-H. Chen, "Cache write generate for parallel image processing on shared memory architectures," tech. rep. TR-EE-95-17, University of Washington, Dept. of Elect. Eng., Jan. 1995.

The Application of the Gibbs–Bogoliubov–Feynman Inequality in Mean Field Calculations for Markov Random Fields

Jun Zhang

***Abstract*—The Gibbs–Bogoliubov–Feynman (GBF) inequality of statistical mechanics is adopted, with an information-theoretic interpretation, as a general optimization framework for deriving and examining various mean field approximations for Markov random fields (MRF's). The efficacy of this approach is demonstrated through the compound Gauss–Markov (CGM) model, comparisons between different mean field approximations, and experimental results in image restoration.**

I. INTRODUCTION

Recently, there has been considerable interest in the mean field theory (MFT) for Markov random fields (MRF's) and its application in image processing and computer vision (e.g., see [1]–[4]).[1] In MRF-related optimization problems, the MFT can provide "SA-like" performance at "ICM-like" computational cost.[2] It does this by approximating the mean of a MRF, which is optimal in the sense of least mean square error (LMSE) [1]. Most previously proposed MFT schemes belong to two classes, the physically motivated local mean field energy (LMFE) [1]–[2] and the Gibbs–Bogoliubov–Feynman (GBF) inequality [4]. Although the efficacy of the MFT has been demonstrated in applications, it is often unclear how the various schemes relate to each other and how their (mostly) statistical mechanics motivations translate into signal processing ones.

In this paper, we adopt the GBF as a framework for mean field approximation. We will give it an information-theoretic interpretation that could be understood more easily in signal processing terms (Section II). As an illustration, the GBF scheme is applied to an important and fairly complex MRF, a compound Gauss–Markov (CGM) model (with line field), and compared with the LMFE (Section IV).

II. THE GBF INEQUALITY

We begin with the concept of the MRF. Let $\mathbf{S}$ be a finite extent two-dimensional (2-D) lattice with a neighborhood system and its associated cliques (see Fig. 1). Let $\mathbf{x} = \{x_i, i \in \mathbf{S}\}$ be a collection of random variables. For the sake of simplicity, assume for the moment

Manuscript received November 24, 1993; revised September 17, 1995. This work was supported by the National Science Foundation under Grants DIRIS-9010601 and IRI-9315193. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. R. L. Lagendijk.

The author is with the Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, WI 53201 USA (e-mail: junzhang@ee.uwm.edu).

Publisher Item Identifier S 1057-7149(96)04539-3.

[1] More references can be found in a longer version of this paper [5].

[2] SA: simulated annealing; ICM: iterative conditional mode.