Analysis and Simulation of a Common Bus Multiprocessor

J. R. Parker

University of Calgary, Department of Computer Science
Calgary, Alberta, Canada T2N-1N4

I. Introduction

In order that a multiprocessor system be considered successful, an improvement in performance over a single processor system must be evident. This improvement will probably be emphasized in the particular application area for which the multiprocessor was designed: image processing, geophysics, robotics, and so on. Indeed, performance may be degraded significantly when the multiprocessor is used for a task that is inappropriate for the particular architecture. It is important to be able to predict, with some confidence, the performance measures of interest for a particular system acting on a particular problem.

The VISTA Multiprocessor [11] was designed as a system for performing image processing operations, especially those that can be performed on sub-images of an image. There are many operations that are commonly performed on digital signals: filtering, transformations, statistical measurements, and so on. Many two dimensional signals, including images, can be treated as collections of one dimensional signals for the purposes of implementing these operations[3,5,6,13]. In other cases an image may be broken up into regions, each of which may be processed in parallel. In the case of the Fourier transform, single rows and columns can be defined as regions. The processing of each region is independent of all of the others, and can be carried out in any order[12,16].

VISTA is designed to operate on regions of images. Each processor has local code and data memory which can be loaded by a central processor called the *master control unit (MCU)*. Image regions are loaded into the processors across a common bus, and a processor operates only on data in its local memory. Control of bus transfers centrally offers a number of advantages, the most important of which is the elimination of contention, as such. The processors do not request data; the MCU sends a region into local data memory and then allows the processor to proceed. The MCU then repeats this process with the next processor, and so on. When the first processor has finished, it sets a bit that the MCU can see; there are now two possibilities for how to proceed. Either the MCU can load all processor memories and then start over again, looking for set completion bits, or it can start over again when the first processor sets its completion bit. In the latter case, it can be seen that no more processors will actually be used than the MCU can reasonably deal with, and this number will depend on the region size and algorithm being implemented.

One other modification can be made to improve things further. When a processor has completed its task, the MCU must unload the local data memory into the global Frame Buffer and load a new region into the local memory. While this is happening, the processor involved is idle. If two buffers are used, and both are loaded initially, then the MCU simply swaps buffers and starts the processor immediately, then unloads the processed buffer while the processor works on the other. This doubles the local memory requirement while reducing the idle time of the processors to nearly nil.

When the system is started there is an initialization time during which processors will be idle. This corresponds to a pipeline startup time. There is a similar time when all processors have finished, again corresponding to the time taken to unload a pipeline. Other than at these two moments, none of the processors being used will be idle, and the bus will be working at near maximum capacity. This method of handling the local data memories will be referred to as a *distributed cache* memory [1,4,14], and superficially resembles the arrangement found on the FLIP processor[8].

The lack of contention in the Vista system should mean that a model could be easily constructed. However, the real system, once constructed, demonstrated better performance than had been calculated. We then proceeded to do a more thorough analysis and simulation to more fully understand the workings of the system.

II Analysis of the VISTA System

The VISTA system as constructed uses an 8086 microprocessor as the MCU, with each other processor being a TMS32017 Signal Processor chip [10]. VISTA will perform an operation on an image that is broken up into sub-images each of which resides in a large frame buffer. Each TMS processor has local memory: 2048 words of program and 2048 words of data memory; data memory is divided into two 'pages' of 1024 words each.

Definitions of variables that will be used in the analysis are:

Symbol	Meaning
B	Maximum bus transfer rate, bytes per sec.
k	Total number of processors on the bus.
M	Number of columns in the object image.
N	Number of rows in the object image.
N_B	Total number of sub-images.
N_b	Number of bytes in one sub-image.
N_p	Actual number of processor units allocated to this problem.
T_b	Time needed to load one sub-image into a processor cache.
T_R	Total real time needed to complete the entire problem.
$T_{RL}, T_{RR}, T_{U1}, T_{U2}$	Real time needed for a particular phase: (L=Load, R=Reload, U1=Unload phase 1, U2=Unload phase 2)
T_r	Time needed by the MPU to send one byte onto the bus. (Overhead)
T_t	Time needed by one processor unit to process one sub-image.
T_W	Total amount of processing time needed to complete the entire problem.
$T_{WL}, T_{WR}, \ldots$	Amount of processing time done in a particular phase.

Some of the simple results are:

$$N_B = \frac{N*M}{N_b} \text{ (Simplest case)}$$

$$T_b = N_b * T_r + \frac{1}{B}$$

Computation on the VISTA machine can be divided into three phases. First, the processors are loaded with sub-images. This corresponds to the start-up phase of a pipeline, and is called the *load cycle*. For each processor, the first buffer if loaded, then the processor is started, and then the second buffer is loaded. Since each sub-image takes T_b seconds to be loaded, and there are N_p processors each with 2 buffers, the load cycle take $2N_pT_b$ seconds.

Next, the results are removed from buffers and replaced with fresh data to process. For each processor: first the processor is restarted, then the processed data is removed from the full buffer, and then new data is loaded. This continues until all sub-images have been sent, and is called the *reload cycle*. A sub-cycle involves reloading all N_p processors once, and will require $2N_pT_b$ seconds. During the reload

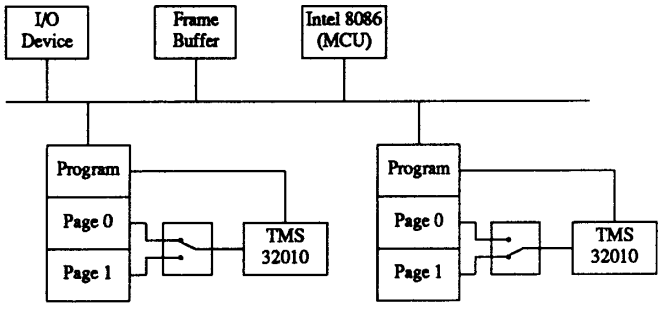


Figure 1 - Outline of the Vista Hardware

cycle, all processors are working.

The final phase is the *unload cycle*, in which the last of the results are collected. As the final page full of data is gathered from a processor it becomes inactive, corresponding to the terminal phase of a pipeline. At first, each processor being unloaded is started before unloading, since it still has a single full buffer remaining unprocessed; then the last buffers are unloaded and the processors go idle. Thus the unload cycle consists of two distinct parts. During the first part, or cycle U1, all N_p processors have a buffer unloaded and are restarted on the final buffer, which was loaded previously. During the second part, or cycle U2, all N_p processors have their final full buffer unloaded and become idle.

It is important to be able to predict how much real time is required to compute a given result on the Vista system. Because of the way in which the processors are connected, this analysis is not obvious. What is clear is that the total real time needed is the sum of the real times needed for the load cycle, reload cycle, cycle U1, and cycle U2. This is the basis for the analysis below.

Assume for a moment a problem that uses three processors, each having two pages. The load and reload cycles can be illustrated as follows:

Time units are integer multiples of T_b.

Operation	Work done so far T1	T2	T3	Real Time (after op)	Total Work (After op)
Load T1 page 0	0	0	0	1	0
Start T1	0	0	0	1	0
Load T1 page 1	1	0	0	2	1
Load T2 page 0	2	0	0	3	2
Start T2	2	0	0	3	2
Load T2 page 1	3	1	0	4	4
Load T3 page 0	4	2	0	5	6
Start T3	4	2	0	5	6
Load T3 page 1	5	3	1	6	9

---------- End of Load Cycle ----------

Reload subcycle A:

Restart T1	5	3	1	6	9
Unload T1 page 0	6	4	2	7	12
Load T1 page 0	7	5	3	8	15
Restart T2	7	5	3	8	15
Unload T2 page 0	8	6	4	9	18
Load T2 page 0	9	7	5	10	21
Restart T3	9	7	5	10	21
Unload T3 page 0	10	8	6	11	24
Load T3 page 0	11	9	7	12	27

reload subcycle B: Repeat, unloading page 1 and then loading again

Continue until all sub-images have been sent.

---------- End of Reload Cycle ----------

The timing of the unload cycle is peculiar, and will be dealt with later.

By definition, it can be seen that the work done by processor 1 in the Load phase is T_t processor seconds (PS). By inspecting the diagram above, it can be seen that the work done by processor 2 is $T_t - 2T_b$, and so on for all N_p processors. In general, the load phase work done by processor i is:

$$T_{Li} = T_t - 2(i-1)T_b$$

Thus, the total work done in the load phase, considering all N_p processors, is:

$$\sum_{i=1}^{N_p}(T_t - 2(i-1)T_b) = \sum_{i=1}^{N_p}(T_t - 2iT_b + 2T_b) = \sum_{i=1}^{N_p}T_t - \sum_{i=1}^{N_p}2iT_b + \sum_{i=1}^{N_p}2T_b$$

$$= N_p T_t + 2N_p T_b - 2T_b\sum_{i=1}^{N_p} i = N_p T_t + 2N_p T_b - 2T_b(\frac{N_p(N_p+1)}{2}) = N_p T_t + 2N_p T_b - T_b N_p{}^2 - T_b N_p$$

$$T_{WL} = N_p(T_t - T_b(N_p-1))$$

This amount of work is done in real time $T_{RL} = 2N_p T_b$, where work is in PS units.

Computation of the work done in the reload cycle is simpler. Each step in the reload cycle involves one load and one unload operation, which takes $2T_b$ in terms of real time. Since each sub-cycle of the reload cycle involves N_p of these steps, then the real time needed for a complete sub-cycle would be $2N_p T_b$. During this period, each processor is active for T_t seconds, and so a total work of $N_p T_t$ is performed.

In most cases it will not happen that C will be an integer. Each subcycle can be broken down into a number of load-unload pairs, and while a full subcycle has N_p of these, the final subcycle can have fewer. This fact causes problems in calculating the duration of the unload cycle. What should be clear is that C is a rational number, consisting of an integer part (I = total number of completed reload sub-cycles) and some fractional part (J = number of unload-load operations in the partial cycle), or

$$C = I + \frac{J}{N_p}$$

and the real time needed to complete C subcycles is

$$2N_p T_b C = 2N_p T_B(I+\frac{J}{N_p}) = 2N_p T_b I + 2T_b J$$

The reload cycle has ended after the last block has been loaded into a processor. At this point, all processors have both buffers loaded, and until now the MPU has been constantly active (no idle time). A total of $2N_p T_b + 2N_p T_b C$ seconds of real time has passed and a total of $2N_p + N_p C$ blocks have been loaded into processors. Since all buffers have been loaded, we have

$$N_B = 2N_p + CN_p = N_p(2+C)$$

and thus $C = \frac{N_B}{N_p} - 2$.

It is possible to find the times at which the processors will start execution. By inspection, it can be seen that processor T_i will be started for cycle j, j = 1,2... at time

$$T(i,j) = \begin{cases} (2i-1) & \text{if } j = 1 \\ (j-1)2N_p T_b + 2(i-1)T_b & \text{if } j > 1 \end{cases}$$

where cycle 0 is the load cycle. Unload cycle start times are not computable in this way. It is also possible to compute values for I and J. Since $C=(I+\frac{J}{N_p})$,

$$I + \frac{J}{N_p} = \frac{N_B}{N_p} - 2$$

$$J = N_B \bmod N_p, \qquad I = N_B \operatorname{div} N_p - 2$$

Assume that processors are loaded in ascending numeric order. Then we know that processors numbered 1 through J will be loaded exactly once more than will the processors from J+1 through N_p, and will

therefore be started one extra time. The number of starts of a processor i is:

$$S(i)=\begin{cases} I+2 & \text{if } i \le J \\ I+1 & \text{if } i > J \end{cases}$$

This counts starts in the load and reload cycles only. All of these facts are needed to determine the MCU idle time during the unload cycle.

Unload cycle U1 starts at real time $2T_bN_p(C+1)$. At this time, does the MCU have to wait before unloading the next processor, the first in the unload cycle? If the MCU does wait, it will wait for processor number $J+1$ first. T_{J+1} will finish its second last buffer at time T(J+1, S(J+1))+T_t. That is, at the last start time + the time needed to complete processing. The current time should now be T(J, S(J))+$2T_b$, or the time at which the last processor in the final reload cycle was started + the time needed to unload and reload that processor.

If the time at which processor J+1 will finish is greater than the current time, then the MCU must wait for J+1 to finish before unloading it. If the MCU does not wait, it unloads processor J+1, taking time T_b to do so, and then checks processor J+2. Again, the MCU will either wait for J+2 or unload it directly, and so on for all N_p processors. Each processor after J+1 will be ready for unloading at intervals of $2T_b$ seconds, but it only takes T_b seconds to unload a processor. Thus, the MCU will wait for J+1 if

$$X = T_t - 2T_bN_p > 0$$

The MCU will wait for J+2 if

$$X = T_t + T_b - 2T_bN_p > 0$$

In general, the MCU will wait for processor J+1+q (modulo N_p) if there exists a value q such that

$$X = T_t + qT_b - 2T_bN_p > 0$$

where

$$0 \le q < N_p$$

If the wait time is negative then the MCU does not wait, and X is set to 0.

If the MCU waits for a processor in the first half of the unload cycle, then it will also wait for half of the rest of the processors remaining to be loaded, and will wait T_b seconds for each of them. This is because a processor needs $2T_b$ seconds to be serviced in the reload cycle, but only T_b seconds in the U1 cycle. We can then say that

$$Y = \left\lceil \frac{N_p - q - 1}{2} \right\rceil T_b$$

The total time that the MCU spends waiting during the U1 cycle is X+Y, and so the total real time spent in the U1 cycle is

$$T_{RU1} = N_pT_b + X + Y$$

The second half of the unload cycle, U2, may also involve waiting.

The second half of the unload cycle, U2, may also involve waiting. Cycle U2 begins the moment that the last buffer has been unloaded in cycle U1. Before that processor was unloaded, T_b seconds before the end of U1, it was started on its last full buffer. This means that cycle U2 will end when that processor has finished work on its buffer and has itself been unloaded. This requires time $T_t + T_b$ seconds in all, and so cycle U2 must always require $T_t+T_b-T_b = T_t$ seconds.

The total real time needed by Vista to solve a given problem can now be calculated. It is

$$T_R = T_{RL} + T_{RR} + T_{RU1} + T_{RU2}$$

$$=2N_pT_b + 2CN_pT_b + (N_pT_b+X+Y) + T_t$$

The conditions for which this formula is correct are:

1. $C \ge 1$

2. $T_t - 2T_b N_p \leq 0$

3. $k \geq \left\lceil \frac{T_t + T_b}{2T_b} \right\rceil$

These would be the normal conditions of operation for the Vista processor in any case. In plain English, these conditions are:

1. There are many more blocks of data to be processed than there are processors.
2. The time needed by a processor to complete one calculation is significantly greater than the time needed to transmit one block to the processor.
3. The Vista processor has a minimum number of processors available to dedicate to the problem.

III. Removing Simplifications

The expression $N_B = \frac{NM}{N_b}$ is a simplification that assumes the image can be broken up into equal sized sub images exactly. In many cases this cannot be done; there will be a region of overlap between sub-images. An example would be edge enhancement using a 3x3 template, in which case the boundary row and column would need to be retransmitted with later sub-images.

If a is the number of rows in a sub-image, and b is the number of columns in a sub-image then we can break up a row into K sections of b bytes each, where each section except the first and last overlaps by one byte on each end. The first and last have a one byte overlap on one end only. In this case $M = K(b-1) + R_m$ where R_m is the number of bytes left over ($R_m < b$). The best value of b causes R_m=0, which occurs when $K = \frac{M}{b-1}$ exactly. Now the number of bytes transmitted per row is actually $M+K = M+\frac{M}{b-1}$. For example, if M=64 we could have b=17, and a total of 68 bytes per row would be sent. The same is done for columns, and instead of NM bytes being transmitted we get $(M+\frac{M}{b-1})(N+\frac{N}{a-1})$ bytes. The extra bytes are overhead caused by breaking up the problem into smaller pieces. Dividing this more precise byte count by N_b gives a much better value for N_B.

IV. Simulation

The expression that was derived above for determining the real time needed by Vista to solve a given problem is only accurate when the three conditions are satisfied. There are at least seven other cases to be examined in detail and new expressions could be derived for each case. Rather than do this a Simula [18] simulation of Vista was constructed that would predict execution times in all cases. An instruction level model was also built so that actual Vista code could be executed [10].

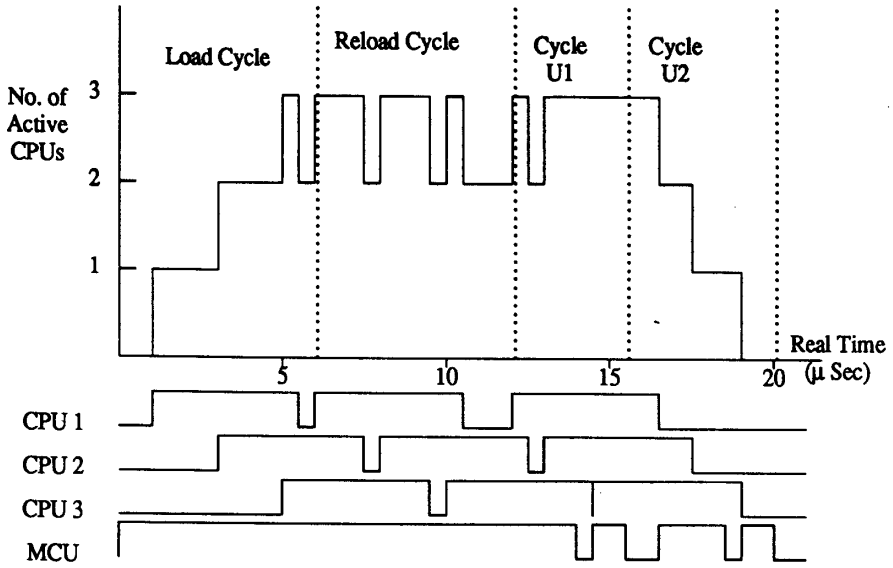
The simulation consisted of 368 lines of Simula, and was conducted along conventional lines. There was an array of 200 Simula process class items each representing a TMS32010 processor, and each containing two arrays to be used as the data buffers. There was also another process class item that represented the MCU. Its job was to load (and unload) data from a large central array (frame buffer) into the TMS local memories. Concurrency was managed by the Simula coroutine facilities.

An initial version of this simulation was written before construction of the Vista prototype was begun, and was used to test design issues before they were actually built. As a result there were no major hardware changes made to repair faulty design decisions. Also, it turned out that the Simula code written as the MCU model was close enough to the real code needed for the real MCU that it was translated into 8086 assembler and used in the real system.

The simulation model and the analytic model have agreed with each other in predicting execution times for all situations that were tried.

V. Results

The Vista system actually constructed consists of one MCU, one processor, and one 500K frame buffer. No DMA device is used, so the MCU must perform the actual data reads and writes. This is not ideal, but is sufficient for assessment purposes. The benchmark program is a two dimensional FFT on a



Execution trace of a 3 processor Vista system with T_b=1, T_t=4.5, and N_B=9.

The Phases of VISTA Execution

256 by 256 image.

The Vista system performs the FFT in 12.6 seconds. This does not seem impressive until compared with the time taken on a VAX 11/780: 92 seconds. A second processor would cut the time in half, to 6.3 seconds. Our prototype would allow four processors at most to be fully utilized, at which point the FFT would take 3.2 seconds.

The simulation model in this paper would predict a time of 11.28 seconds to perform the FFT. The difference may be explained, in part, by the fact that the TMS processors has several instructions that cannot be accurately timed. These instructions cause the internal pipeline to be cleared, increasing the execution unpredictably by up to a factor of four. Because the prototype Vista has only one TMS processor, the analytic model could not be applied. Note that a two dimensional transform is actually two separate steps, one for rows and the other for columns. Transforming columns has a larger MCU overhead due to greater complexity in indexing an element; the array is stored in row major order.

Much of the problem with the prototype is caused by the lack of DMA capability on the bus used (Multibus [7]). Assuming a bus rate DMA controller as part of the MCU, the figures become more impressive. Each processor can transform a row or column in 0.022 seconds, and transmission of the entire image across the bus and back requires 0.05 seconds. Thus, the best that Vista can do is to use 112 processors and compute the FFT in 0.1 seconds. This is an effective computation rate of 350 MIPS using a common bus.

There is another feature of Vista that makes it interesting when applied to image processing. Algorithms intended for use on systolic [9] pipeline, or pyramid [2] architectures can be run on Vista with no hardware change. The software for processor page swapping would be modified to conform to the data transfer method of the architecture being simulated. A Vista simulating a systolic array would not perform as well as would the actual systolic array, but the performance would still be adequate, and the performance loss is compensated for by flexibility.

Further work is proceeding on the Vista system. Future processors will be based on the TMS32030 processor, which permits much more memory and is twice the speed of the TMS32010. A custom DMA controller is also being designed for use on a different bus. Finally, much of the control logic for the processor, currently requiring one entire Multibus card, is being redesigned for a VLSI implementation.

References

1. Abu-Sufah, W., Husmann, H., Kuck, D., *On Input/Output Speedup in Tightly Coupled Multiprocessors*, IEEE Trans. on Computers, Vol C-35 No. 6, June 1986.
2. Ahuja, Narendra, *Multiprocessor Pyramid Architectures For Bottom Up Image Analysis*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol PAMI-7 No 1, January 1985.
3. Armstrong, J.L., *Programming a Parallel Computer for Robot Vision*, The Computer Journal, Vol. 21 No. 3, Aug. 1978.
4. Bhuyan, L., *On the Performance of Loosely Coupled Multiprocessors*, Proc. 11th Annual Symposium on Computer Architecture, Ann Arbor, Michigan, June 1984.
5. Briggs, F.A., Fu, K.S., Hwang, K., Patel, J.H., *A Shared Resource Multiple Microprocessor System For Pattern Recognition And Image Processing*, in 'Special Computer Architectures for Pattern Processing', Fu, K.S. and Ichikawa, I. editors. CRC Press, 1982.
6. Gannon, D.B., Van Rosendale, J., *On The Impact Of Communications Complexity On The Design Of Parallel Numerical Algorithms*, IEEE Trans. on Computers, Vol. C-33 No. 12, December 1984, Pp 1180-1194.
7. IEEE, *Proposed Microcomputer System Bus Standard*, IEEE Computer Society Subcommittee Microcomputer System Bus Group, Oct. 1980.
8. Luetjen, K., Gemmer, P., *FLIP: A Flexible Multiprocessor System for Image Processing*, Proc. Fifth International Conf. on Pattern Recognition, Miami. December 1980.
9. Ni, L.M., Jain, A.K., *A VLSI Systolic Architecture For Pattern Clustering*, IEEE PAMI Vol PAMI-7 No 1, January 1985.
10. Parker, J.R., *The TMS 32010 Emulation System*, University of Calgary Dept. of Computer Science Report.
11. Parker, J.R., Ingoldsby, T.R., *Common Bus Multiprocessor Design for Signal Processing Applications*, University of Calgary Dept. of Computer Science Report.
12. Pavlidis, T., *Algorithms for Graphics And Image Processing*, Computer Science Press, Rockville, MD. 1982
13. Preston, K. Jr. and Uhr, L. (eds), *Multicomputers and Image Processing*, Academic Press, N.Y., 1982.
14. Rudolph, L., Segall, Z., *Dynamic Decentralized Cache Schemes for MIMD Parallel Processors*, Proc. 11th Annual Symposium on Computer Architecture, Ann Arbor, Michigan, June 1984.
15. Selfridge, D.B., Mahakian, S., *Distributed Computing For Vision: Architecture And Benchmark Test*, IEEE Trans. on Pattern Recognition and Machine Intelligence Vol. PAMI-7 No. 5 September 1985, pp 623-626.
16. Siegel, L.J., Siegel, H.J., Feather, A.E., *Parallel Processing Approaches To Image Correlation*, IEEE Transactions on Computers Vol C-31 No. 3 March, 1982. Pp 208-217.
17. Texas Instruments, *TMS32010 User's Guide*, Texas Instruments Ltd, Dallas Texas, 1983.
18. Unger, B.W., Parker, J.R., *An Operating System Simulation and Implementation Language*, Conf. on Simulation, Measurement, and Modeling of Computer Systems, Boulder Co. 1979.