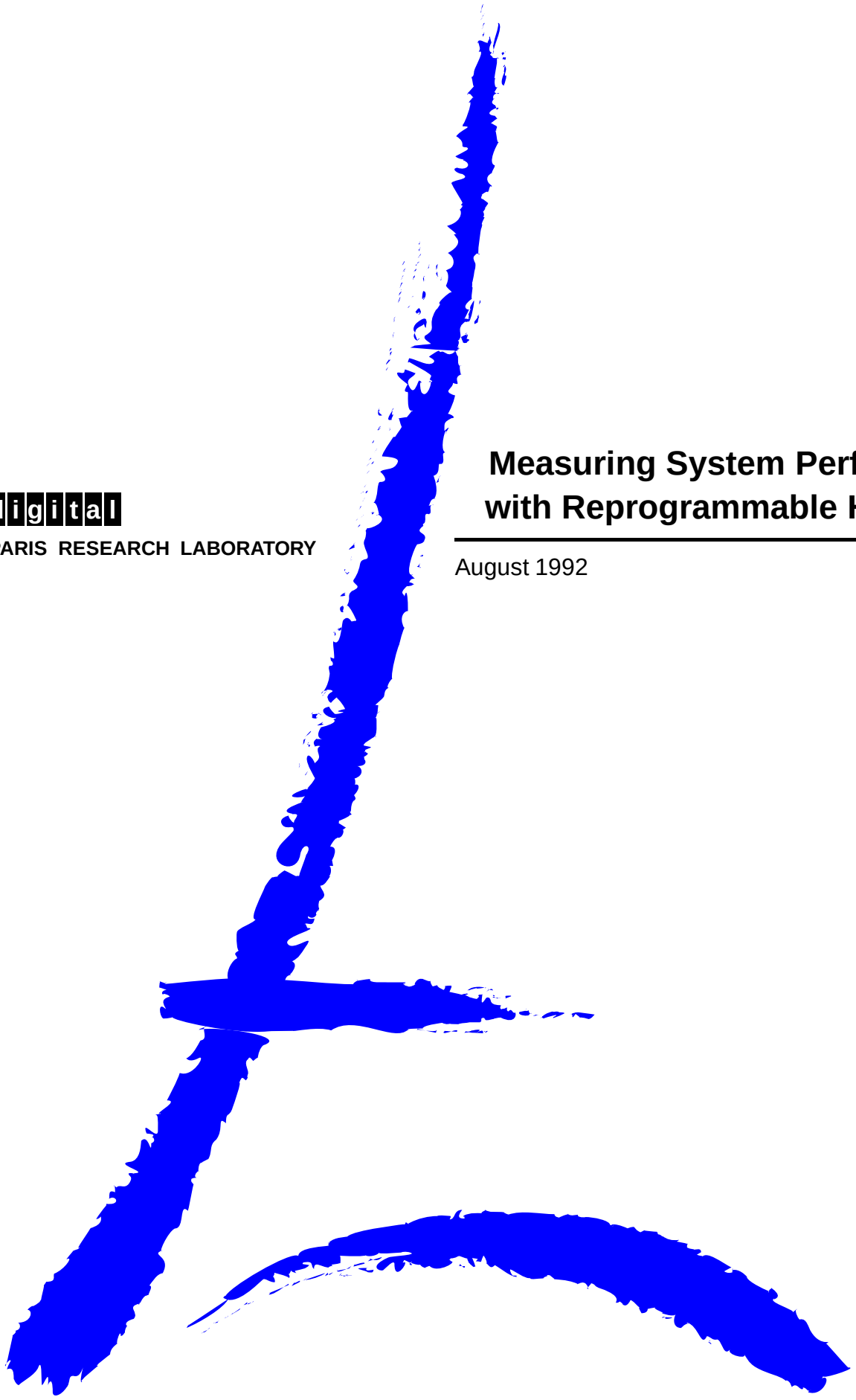
19

Measuring System Performance with Reprogrammable Hardware

digital
PARIS RESEARCH LABORATORY

August 1992

Mark Shand



19

Measuring System Performance with Reprogrammable Hardware

Mark Shand

August 1992

Publication Notes

An early version of this report appeared in AUUG'91, the Australian Open Systems User's Group conference, Sydney, 25-27 September 1991, under the title "Measuring Unix Kernel Performance".

Abstract

We present accurate, low-level measurements of process preemption, interrupt handling and memory system performance of a UNIX[1] workstation.

To gather this data, we use PAMs (Programmable Active Memories). These are fast, general purpose, bit-level programmable coprocessors based on field-programmable gate arrays. They are mapped to part of the system address space and appear to the CPU as memory, much like memory-mapped I/O devices.

PAMs are primarily aimed at computationally intensive problems, where wide, application specific data-paths can offer large speedups over software. By contrast, in this application we rely on the real-time, concurrent aspects of a PAM that is relatively modest in terms of computational resources.

Starting with a simple 25 MHz counter, we describe a series of measurement devices built to answer specific questions about low-level system performance. Many of the devices are active in that they provoke the events they seek to measure. Our measurement techniques allow us to construct histograms gradated in CPU clock cycles and cover: frequency and duration of user process preemption, latency from interrupt to kernel handler, DMA throughput and latency, and the effect of other system activity on DMA.

Résumé

Nous présentons les résultats de mesures de bas niveau des performances d'un poste de travail UNIX. Pour obtenir ces données, nous utilisons une PAM (Mémoire Active Programmable). Une PAM est un coprocesseur rapide universel, programmable au niveau du bit, utilisant la technologie des FPGA (matrice de portes reprogrammable). La PAM est accessible depuis le système hôte à travers le mécanisme d'adressage mémoire.

Les PAM sont principalement utilisées comme accélérateurs matériels d'algorithmes nécessitant de larges chemins de données. L'application décrite dans ce rapport est quant à elle modeste en temps de calcul, mais exploite les propriétés temps réels de la PAM.

Nous décrivons une suite de configurations de la PAM qui permettent de répondre à des questions précises d'analyse de performance(s) du système à bas niveau. La plupart de ces configurations forment des systèmes actifs, qui provoquent les événements que l'on cherche à mesurer. Nos techniques nous permettent de déterminer la distribution temporelle, à la précision d'un cycle machine, des événements suivants: la fréquence et la durée des préemptions de processus; le temps de réponse du noyau à une interruption; le temps de réaction et le débit du DMA (accès mémoire direct); et les effets du reste du système sur le performances du DMA.

[1] Unix is a registered Trademark of AT&T.

Keywords

Performance measurement, PAM, reprogrammable hardware, FPGA, memory system, interrupt latency, DMA throughput, TURBOchannel.

Acknowledgements

Jean Vuillemin is the founder of the PAM project and the force behind it. As part of our next generation PAM, Jean Vuillemin, Patrice Bertin and Didier Roncin designed and built the 3mint board and PAM programming tools on which this work depends.

Contents

1 Introduction 1
- 1.1 PAM Technology . . . 1
- 1.2 TURBOchannel . . . 2
- 1.3 Computer System Measurement with PAMs . . . 2

2 Basic Measurement Devices 3
- 2.1 A Simple Timer . . . 3
- 2.2 Process Virtual and Real Timers . . . 4

3 Active Measurement Devices 4
- 3.1 Raising Interrupts . . . 6
- 3.2 Monitoring Critical Regions . . . 9
- 3.3 Summary of Interrupt Measurements . . . 12

4 TURBOchannel DMA 12
- 4.1 DMA Throughput . . . 12
- 4.2 DMA Latency . . . 14

5 Conclusions 17

6 References 19

1 Introduction

In the design of high performance computer peripherals designers must pay attention not only to the basic capabilities of the host hardware, but also to characteristics of the operating system. Parameters such as time to service a device interrupt can fundamentally affect the viability of a proposed design. To some extent such parameters can be determined by analytical models or, in the case of interrupt latency, by simple expedients such as counting instructions in the perceived critical path, but ultimately the most reliable method is measurement.

Unfortunately measuring activity at the lowest levels of computer systems can be extremely difficult. For coarse grained high-level measurements a computer can often be turned on itself and useful statistics can be accumulated by reference to little more than its own real-time clock. In contrast, at lower levels, we find that, the phenomena under measure occur in far less time than the resolution of standard real-time clock devices and involve system components like the cache that are, by their very nature, not visible to the CPU. Recently we have seen much progress in instrumentation of programs and even entire systems by code modification [3, 13], but these methods can greatly perturb the objects being measured and the techniques are by no means easy to implement. The alternative of adding purpose-built measurement hardware [4, 7] tends to be either inflexible, expensive, or both, and may rely on an underlying microcoded implementation.

For the purpose of characterizing the performance of a few parts of the kernel, neither of the above approaches seems justified in terms of the time and effort involved. Even adding a high resolution clock is an excessive burden if new hardware must be built which will find no further use once the measurements are made.

1.1 PAM Technology

PAM technology, introduced by J. Vuillemin [1], is based on a matrix of programmable active bits. It permits the realization, through a downloadable bitstream, of synchronous logic circuits comprising combinatorial logic and registers, each of the registers being updated on each cycle of a global clock signal. The maximum clock speed for such a circuit is directly determined by its critical combinatorial path, which varies from one circuit to another. When implemented through field-programmable gate arrays such as the Xilinx 3000 series [14], it is not difficult to realize circuits that operate at 25 to 30 MHz.

PAM stands for Programmable Active Memory. It is infinitely reprogrammable thanks to the field-programmable gate arrays (FPGA) from which it is built. The rest of the acronym derives from its role as a coprocessor occupying part of a host address space, receiving commands and returning results in response to the address and data transactions the host directs towards it. Being mapped directly into user and kernel address spaces, the PAM may be accessed very cheaply from the CPU, making it suitable for the implementation of fine grain measurement devices. Being reusable and easy to program makes it suitable for the implementation of specialized single use devices.

PAMs are primarily aimed at computationally intensive problems, where wide, application specific data-paths can offer great speedups over software [10]. By contrast, in measurement applications we rely on the real-time, concurrent aspects of a PAM that is relatively modest in terms of computational resources. In fact such a PAM is at our disposal—*3mint*. 3mint is the interface module of DECPeRLe-1, our laboratory's latest computationally oriented PAM [2]. In normal operation 3mint programming is fixed, being loaded out of PROM, and is used to control downloading of and access to DECPeRLe-1, a matrix of 23 Xilinx gate arrays. However, 3mint too uses a Xilinx gate array, and by providing the ability to alter its programming we obtain a small PAM connected directly to the system I/O channel.

1.2 TURBOchannel

The particular I/O channel in question is TURBOchannel [5]. TURBOchannel is a synchronous, asymmetric I/O channel operating at 12.5 to 25 MHz. It connects one *system* module containing CPU and memory to some number of *option* modules. TURBOchannel supports two kinds of transaction: an *I/O transaction* in which the system module can read or write an option module, and a *DMA transaction* in which an option module can read or write the system module.

I/O transactions are relatively straightforward; they are issued in response to CPU memory transactions to the region of the address space to which an option module is mapped. Writes are allowed to proceed asynchronously and may be issued in one cycle; sustained throughput to minimum latency option modules is one word every three cycles. Reads must stall the CPU until the option responds. On the DECstation 5000/200 implementation of TURBOchannel the stall is a minimum of 8 cycles. This asymmetry means that scattered I/O writes place a much lighter burden on the CPU than I/O reads.

DMA transactions offer much higher bandwidth, being able to transfer multiple words in a single transaction, each word taking one cycle. Ignoring startup overheads, a 25 MHz TURBOchannel has a theoretical throughput of 100 megabytes/second. TURBOchannel DMA uses physical addresses in its interactions with the system module. This greatly simplifies the design of option modules, moving the burden of address translation to software. Under Unix however, successive memory pages in a user's virtual address space are not necessarily physically contiguous. Performing large DMA transfers to non-contiguous 4 kilobyte pages of user memory would require an address translation for every 1024 words transferred; roughly once every 40μs. Determining address translations in a user process requires a call to the operating system. These considerations lead naturally to questions of kernel performance.

1.3 Computer System Measurement with PAMs

We present a number of performance measures, each obtained with a specific reprogramming of 3mint and appropriate driving software. The majority of these results are obtained on a DECstation 5000/200 running Ultrix 4.2A with a 3mint board in TURBOchannel option slot 2. In the case of interrupt latency measurements, results from a DECstation 5000/240

and a DECstation 5000/125 are provided for comparison. For our purposes, the important characteristics of these three machines are as follows:

Model	CPU clock frequency (MIPS R3000A)	Memory System and TURBOchannel clock frequency
DECstation 5000/240	40MHz	25MHz
DECstation 5000/200	25MHz	25MHz
DECstation 5000/125	25MHz	12.5MHz

While the results of the measurements are of interest in themselves we avoid detailed analysis, instead choosing to focus on the range of techniques which this approach gives us. This is the main contribution of the work.

2 Basic Measurement Devices

Our first experiments are based on passive devices that run concurrently with the CPU and exploit the high bandwidth, low latency interface between the CPU and the PAM.

2.1 A Simple Timer

For the first measurements the gate array on 3mint is programmed to be a 32 bit counter (based on [12]). It is clocked by the 25 MHz TURBOchannel clock (See Figure 1, TbC in the figures refers to TURBOchannel). This counter runs freely, wrapping around to zero periodically. This presents no problem provided we are only interested in measuring the duration of events that last less than $2^{32} \times 40$ns; roughly 171 seconds.

A user process continually reads this counter from a tight loop. Whenever successive reads differ by more than a threshold of a few tens of cycles the process supposes it has just resumed after being preempted and records the duration of the preemption in a histogram. We present two such histograms in Figure 2. The logarithmic scales used on both axes in this figure allow the histograms to present several phenomena in data that ranges over several orders of magnitude. The solid line show the execution of the measurement process on an otherwise idle machine for a period of 30 minutes during which 474 973 preemptions were observed. The dashed line shows execution with one competing compute bound process and is gathered over the same period.

Summing the total number of preemptions recorded in the histogram, we observe that the idle system records approximately 260 preemptions per second, that 65% last between 20μs and 25μs and all but 5% take less than 100μs. Almost all of this activity can be explained by the 3.906ms system clock on the DECstation 5000/200, generating 256 interrupts per second. Such base level system activity consumes about 1.5% of the CPU.

The busy system shows a significant peak at one tenth of a second. This is the duration of

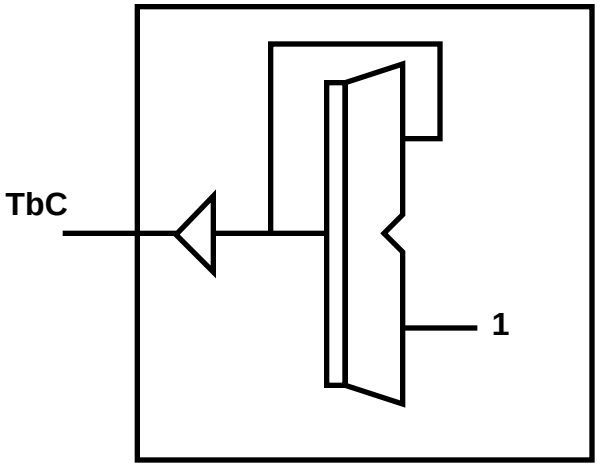


Figure 1: A Simple Timer

the time slice normally given to the competing process.

2.2 Process Virtual and Real Timers

Adding to our previous design an additional counter that is stoppable and loadable, and modifying the kernel to restore and save this second counter on context entry and exit provides a high resolution timer that runs in process virtual time (see Figure 3). Instruction level profiling systems such as *pixie* developed by MIPS[2] Computer Systems [9] assume a perfect memory model. Discrepancies between estimated cycle counts and actual counts obtained from a high resolution timer running in process virtual time can point to memory bottlenecks caused by cache or TLB misses [8]. Recognizing the importance of such measurement techniques, new computer architectures such as Digital Equipment Corporation's Alpha include cycle counters as standard architectural features [6].

3 Active Measurement Devices

Only so much of the system can be observed by monitoring a passive device from user mode. To measure interrupt latencies we need, in addition to cycle counters, a method of provoking interrupt activity.

[2]MIPS is a Trademark of MIPS Computer Systems

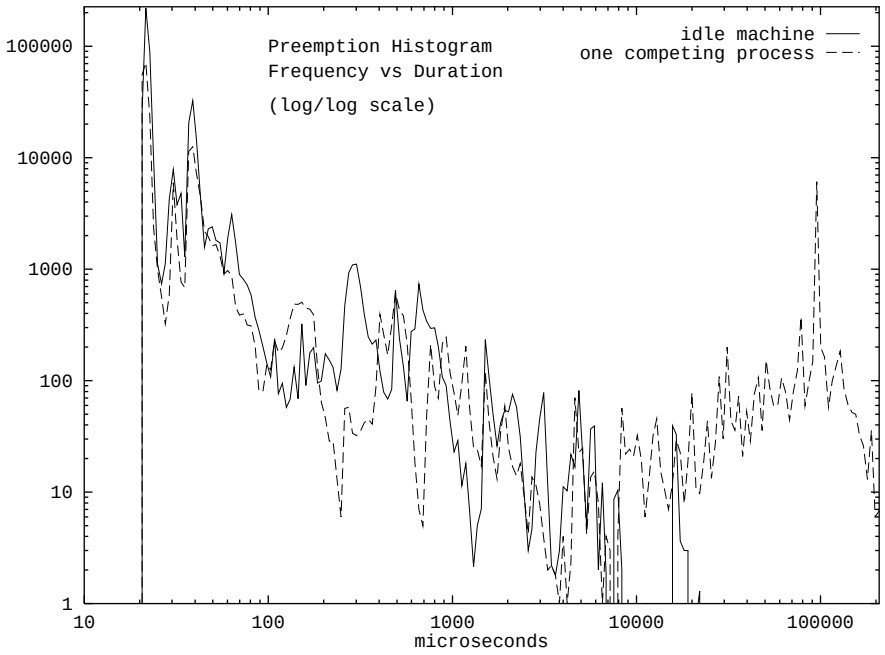


Figure 2: Preemption of User Processes (Durations in μs)

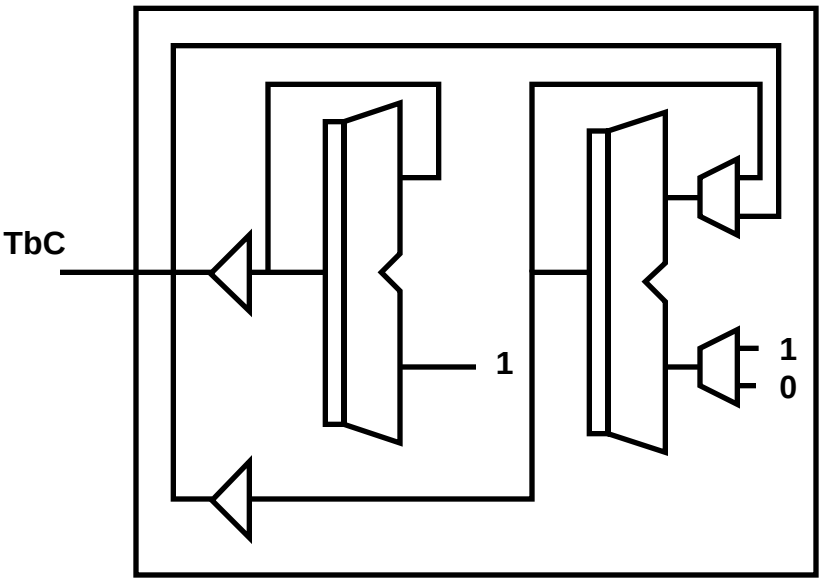


Figure 3: A Combined User Virtual and Real Timer

3.1 Raising Interrupts

3mint is a fully fledged TURBOchannel device, and therefore can raise interrupts. We modify the simple timer to raise an interrupt each time bit 21 of the counter goes high—roughly 5 times a second. On the DECstation 5000/200, such interrupts cause the processor to trap to the *general exception trap* entry point. From there, kernel code interrogates various status registers to discriminate the actual cause of the exception, eventually calling a device specific interrupt handling routine. We modify the kernel to include a routine specific to our interrupt raising design. On entry this routine stores the current time from 3mint and increments a count of interrupts taken. The time value and interrupt count are held in memory that is shared with a user process. By comparing the stored time to the next smaller time at which a bit 21 transition could have occurred user code can determine the time from hardware interrupt to handler and store this in a histogram. Provided this time is never greater than 200ms no ambiguity results because no new interrupt will have occurred. Likewise we can determine the time taken from handler back to user code by reference to the value recorded by the kernel and the current time value.

With this experimental set-up, observations of cycle count histogram from hardware interrupt to handler show some curious artifacts when viewed at single cycle resolution. The histogram contains a series of large peaks, each followed by smaller peaks spread over about 10 cycles (Figure 4).

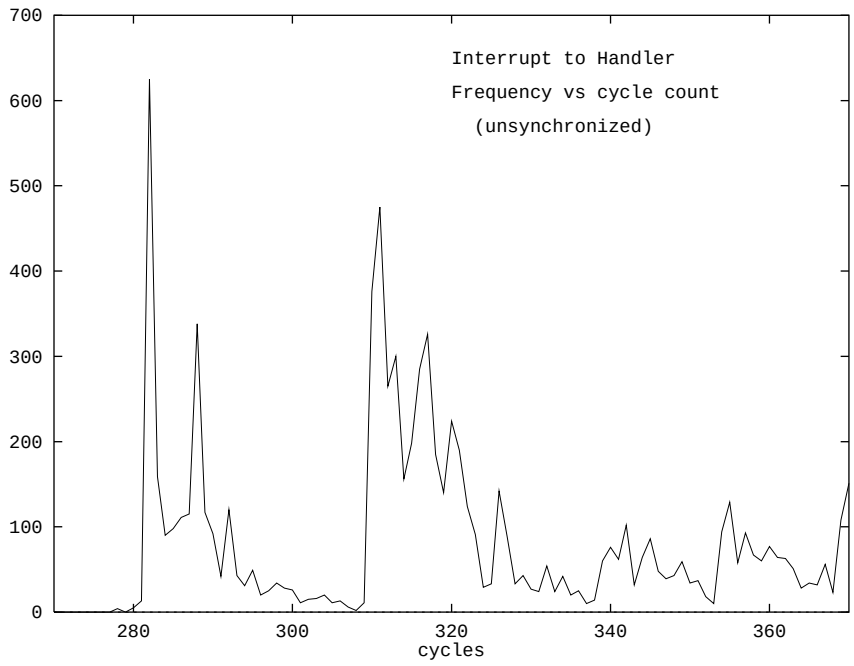


Figure 4: Portion of Histogram of Unsynchronized Interrupt Raising Design

Careful inspection of the inner loop of the user process that records interrupts reveals an

inner loop of 8 instructions, one which issues an I/O read to the 3mint. The smaller trailing peaks represent interrupts that are raised at some time during the 10 cycles that the processor stalls during the I/O read. Since exception handling starts between one and ten cycles after the interrupt is raised, these counts are spread in time. The large peak represents interrupts that are issued during the other loop instructions when the processor can immediately start exception handling.

We first modify the interrupt raising design to *schedule* an interrupt on the transition of bit 21 and issue it when it believes the processor is not currently performing or about to perform an I/O read. It makes this deduction by observation of its SELECT input and by knowledge of the length of the loop that user code executes.

This new design eliminates most of the trailing peaks but still exhibits a secondary peak 5 cycles after the primary peak. This secondary peak was already apparent in the unsynchronized interrupt raising design. Understanding its origin reveals some quite subtle aspects of the DECstation 5000/200 memory system. This machine schedules a DRAM refresh of 5 cycles every 195 cycles. The shortest time from interrupt to handler that we are observing is roughly 280 cycles. Thus we expect one or two DRAM refreshes during execution of the code sequence under test, with one refresh being slightly more common.

We further refine the interrupt raising design to raise an interrupt when the delay between successive I/O reads is just longer than the minimum time around the inner loop of the sampling program. Such longer delays occur when a DRAM refresh occurs between two reads of the timer. Thus the raising of interrupts is now synchronized with DRAM refresh and the secondary 5 cycle delayed peak is eliminated from our histograms.

Since the design exists only to be used in conjunction with a single measuring program, such extreme customization is not unreasonable. When the scheduled interrupt is finally issued, the 3mint stores the current time in a dedicated register for later interrogation by the interrupt handler. This design appears in Figure 5.

Synchronized interrupts show clear peaks in the histogram of interrupt to handler. In one run, spanning one hour and processing 21 342 interrupts from 3mint, 35% took exactly 283 cycles. Data from this run appear in Figure 6. The figure exhibits marked peaks every 15 cycles starting at 283. Curiously, while all runs exhibit the same 15 cycle peaks, the detailed structure varies. In the example cited the main peak is at 283 cycles with a secondary peak at 313 cycles, exactly 30 cycles later. In other runs the secondary peak may be 45 cycles later, or, occasionally, the main peak may appear at 298 or 313 cycles.

We believe the peaks are cache artifacts. Their spacing correlates with the cache line load latency of the DECstation 5000/200. The variation is caused by cache aliasing of the user process code with kernel interrupt handling code, whereby different main memory locations are mapped to the same cache location and compete to occupy it.

To demonstrate aliasing we run eight different tests. Each test executes a different 8 kilobyte block of instructions after each 3mint-generated interrupt is detected. The blocks of instructions are organized to cover the entire 64 kilobyte direct mapped instruction cache of the

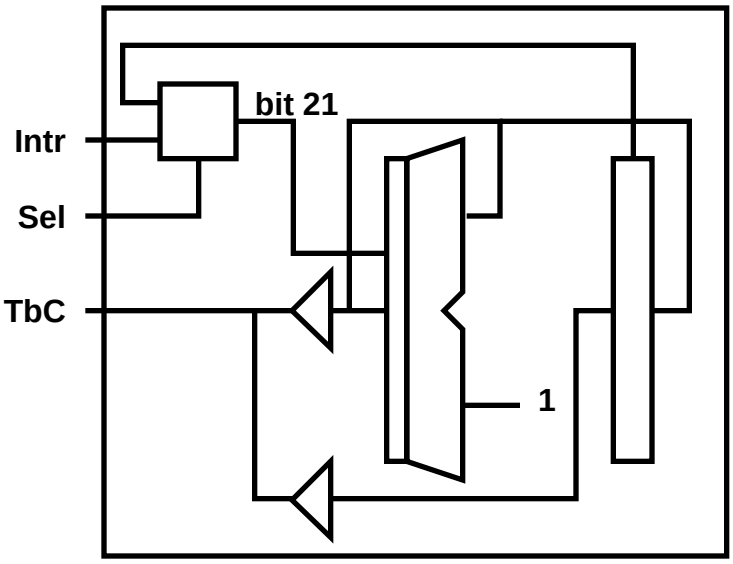


Figure 5: Synchronized Interrupt Raising Design

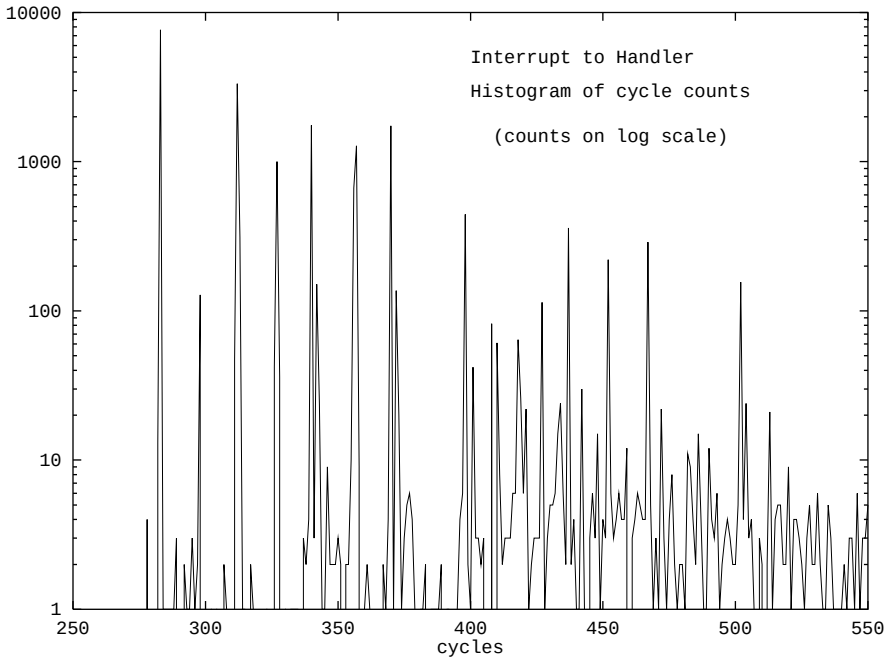


Figure 6: Histogram of Synchronized Interrupt Raising Design

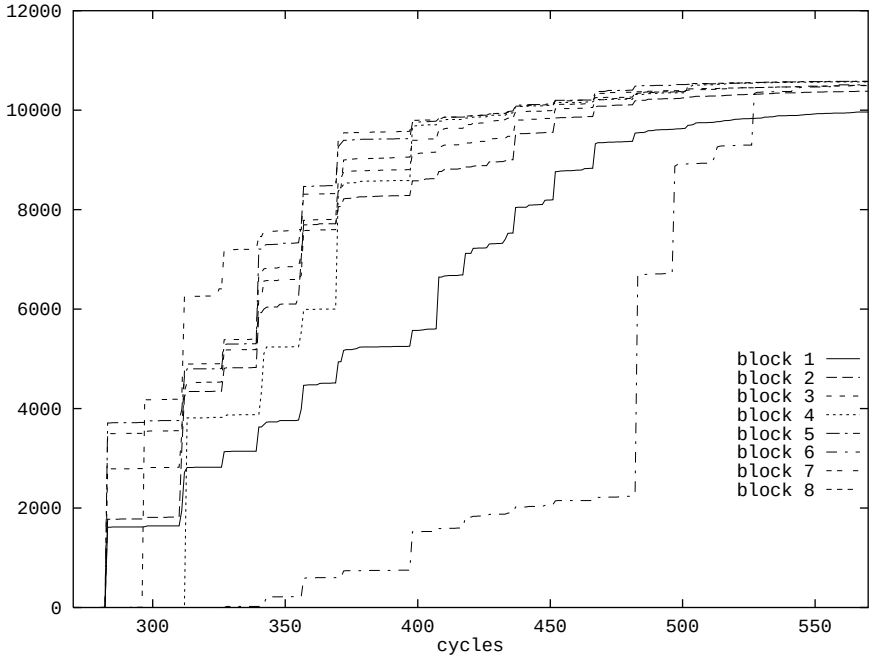


Figure 7: Cumulative histograms showing cache aliasing between user and kernel code

DECstation 5000/200. Cumulative histograms of number of cycles from interrupt to handler appear in Figure 7. Execution of the instructions in block 6 results in considerably poorer interrupt response indicating that this block of instructions aliases with many instructions on the path from the kernel's general exception trap entry point to the specific interrupt handler for TURBOchannel option 2.

3.2 Monitoring Critical Regions

As a final test of the cache artifact hypothesis we develop a new 3mint design and kernel instrumentation that permits us to record the entry time of each of the half dozen routines in the path from the general exception trap entry point to the device specific interrupt handler. This is sensitive code. At the trap entry point itself all but two processor registers contain values from the process running at the time of the exception; values which must be saved before these registers can be reused. In all the routines we must avoid adding extra overhead that will perturb our measurements. Recalling that scattered I/O writes are much cheaper than reads (Section 1.2) we add a small fifo to our 3mint interrupt raising design. On writes to a region of its address space the 3mint now pushes the 29 low bits of the timer plus 3 bits from the target address into this fifo. Writing to this fifo costs just two cycles and requires only one free register—that in which the target address is loaded. When the interrupt handler is reached the fifo contents are read and stored in the shared buffer for later analysis by a user process. The new design appears in Figure 8.

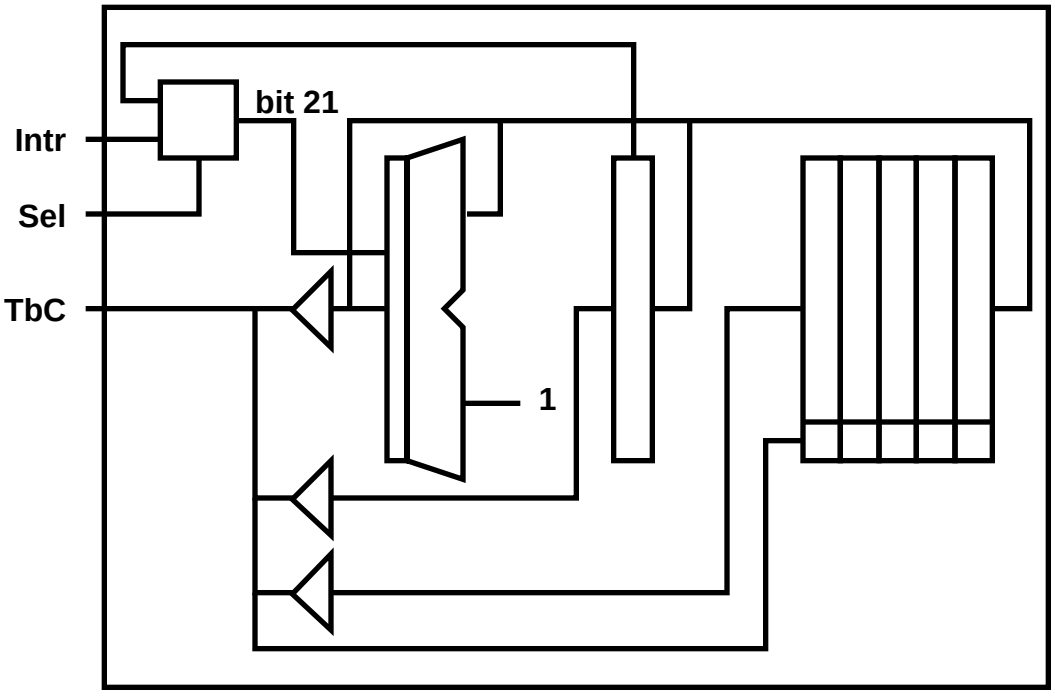


Figure 8: Synchronized Interrupt Raising Design with Event Fifo

We place six probes in the path to the interrupt handler.

Except The earliest point at which we could add instructions after the exception entry point.

Vec1 The start of main interrupt vectoring routine. Called after setup of kernel stacks and differentiation of interrupts from other exceptions.

Vec2 Before saving user state.

Vec3 After saving user state.

K1 First C language routine called. The beginning of interrupt type discrimination.

K2 Discrimination between different TURBOchannel options.

Pam 3mint specific interrupt handler where probe results are collected and saved.

Figure 9 shows results obtained with this design. The graphs are cumulative histograms. A sharply rising curve turning abruptly to a flat plateau indicates that almost all observations were of the same duration. A gently rising curve shows great variation among the observed values. As we look further down the interrupt path the variations in cycle counts increases in keeping with the predictions of our cache miss hypothesis. The early stages of the path are shared by all exceptions and are therefore more likely to be in the cache.

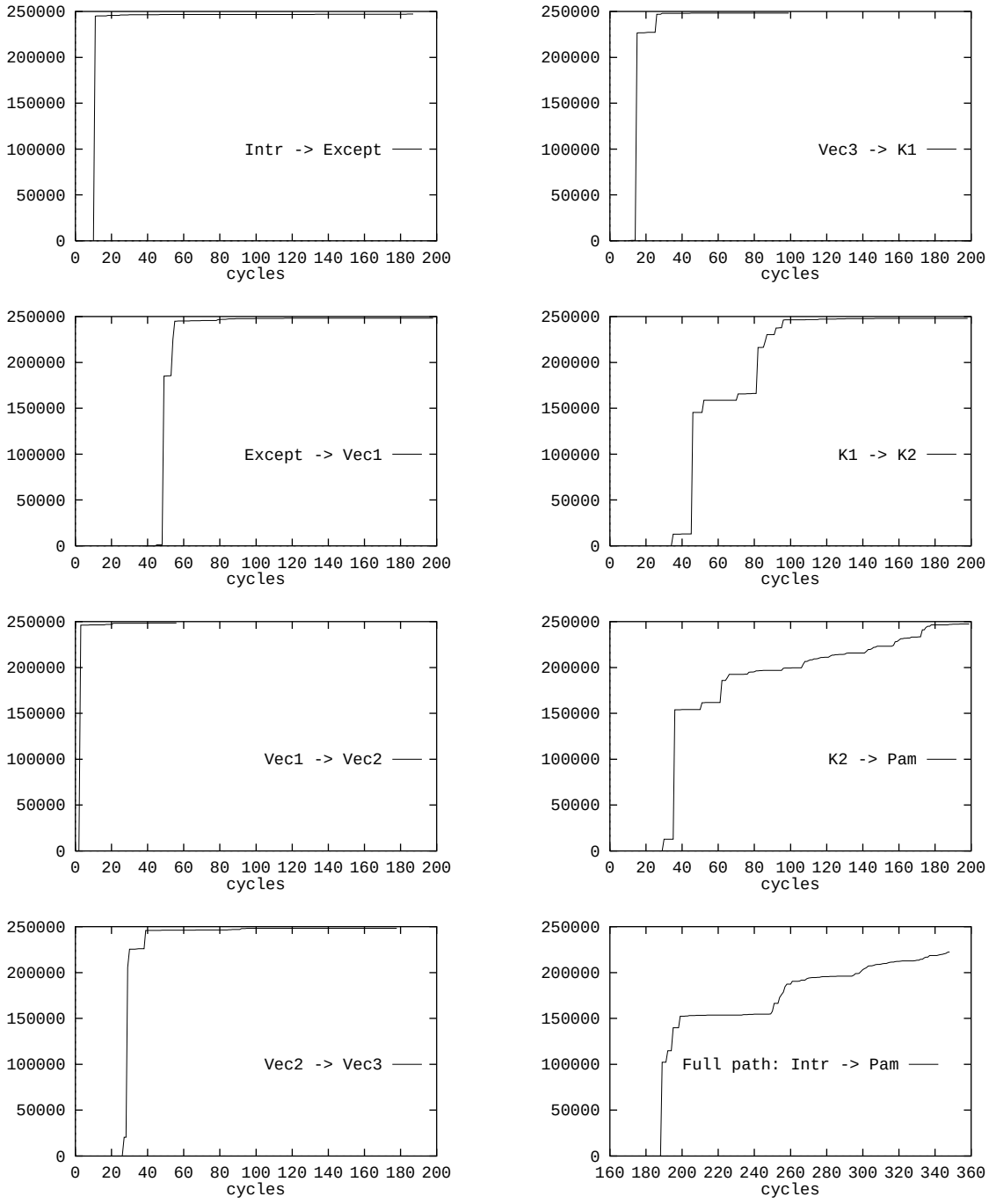


Figure 9: Cumulative Cycle Count Histograms of Steps in the Path to Interrupt Handler

3.3 Summary of Interrupt Measurements

In 70% of cases TURBOchannel interrupts start to get service from their handler with 14μs. Within 21μs, 98% have service. This variation appears to be due to cache misses. Much longer delays sometimes occur and are almost certainly due to interrupts being masked while other critical operations are performed. Delays of over 1ms have been observed.

Figure 11 shows cumulative histograms of DECstation 5000/200 interrupt latency under four different workloads:

Idle Machine running a process with minimal cache requirements.

DBusy Machine running a process that fills the data cache.

IBusy Machine running a process that fills the instruction cache.

IDBusy Machine running a process that fills both data and instruction caches.

Figures 10 and 12 show these same measurements made on DECstation 5000/240 and DECstation 5000/125. To facilitate easy visual comparison the scale of the *cycles* axis in Figure 10 is chosen so that it covers the same period of real time as that used in Figures 11 and 12 in spite of the faster CPU of the DECstation 5000/240.

4 TURBOchannel DMA

Interrupts allow a TURBOchannel option to request service from the CPU. DMA allows the option to work autonomously from the CPU. The rate at which such autonomous work can be carried out is determined by DMA throughput and latency.

4.1 DMA Throughput

TURBOchannel limits DMA to relatively short bursts, thus simplifying the design of the memory system and helping to ensure fair service even with fixed priority scheduling. TURBOchannel guarantees to support DMA transfers of at least 64 words. The implementation on the DECstation 5000/200 supports bursts of up to 128 words. There is a fixed overhead in starting a DMA that is a amortized over the length of the transfer. With 3mint reprogrammed to perform long sequences of DMA at a user specified block-length to contiguous memory, we obtain the throughput results of Figure 13. Our design does not exercise DMA as aggressively as it might, it waits 7 cycles between repeated requests, nevertheless for a block-length of 128 it achieves 91 megabytes/second for DMA writes and 86 megabytes/second for DMA reads, against a theoretical 100 megabytes/second if overheads are ignored.

Running our DMA design continuously while using interactive applications, we find that qualitatively the system remains quite useable even in the face of such heavy traffic. However,

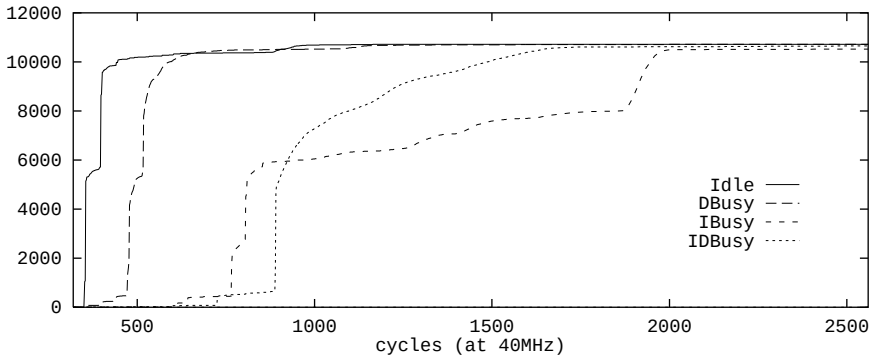


Figure 10: Cumulative histograms of interrupt latency for DECstation 5000/240

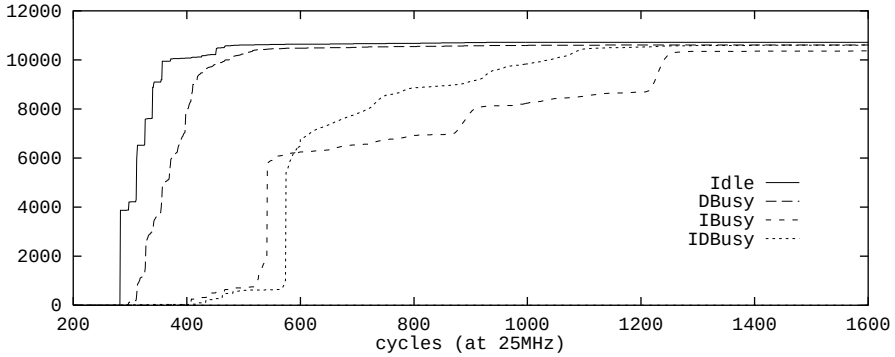


Figure 11: Cumulative histograms of interrupt latency for DECstation 5000/200

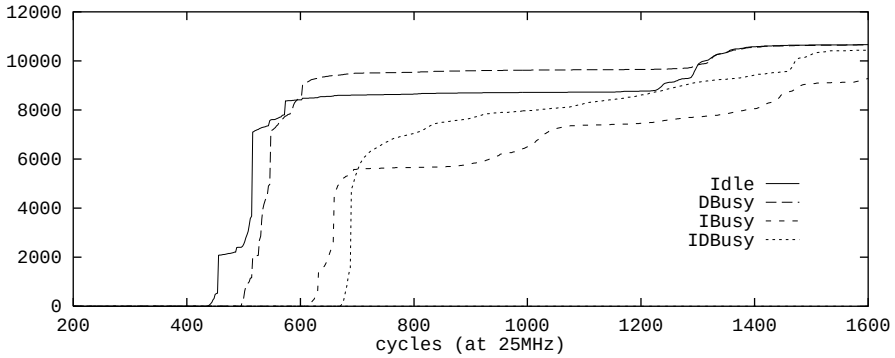


Figure 12: Cumulative histograms of interrupt latency for DECstation 5000/125

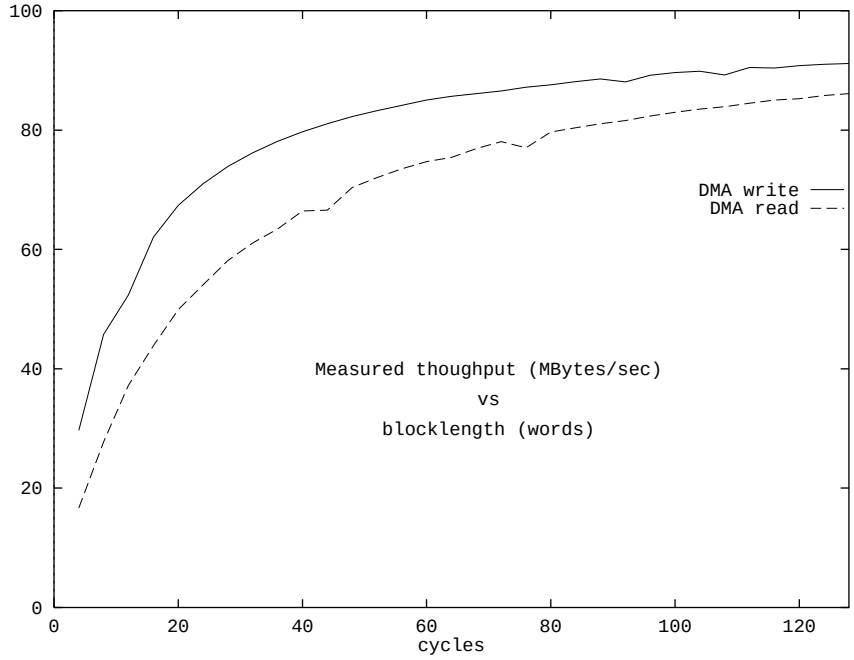


Figure 13: DMA throughput vs. block-length

turning to considerations of large DMAs to non-contiguous user memory, we observe that, at these rates, a 4096 byte DMA (the system page size) takes 45μs; about four times the interrupt latency that we have measured and about one half the time to take and return from an interrupt. It seems unlikely that the system will remain useable if interrupts are used to feed DMA on a per page basis.

Instead 3mint, when programmed as the DECPeRLe-1 interface, uses a second level of DMA to fetch successive physical page addresses from the system. This approach adds only 10 cycles in 1000 for each 4096 bytes transfer by DMA.

4.2 DMA Latency

In the previous design 3mint transferred counter values. We performed one final reprogramming of 3mint making it an interface to a Videk high resolution digital camera [11], as shown in Figure 14.

The Videk Camera outputs a 1.5 megabyte image at a fixed rate of 10 megabytes/second. This data rate, while is well within the capabilities of TURBOchannel DMA, is difficult to handle by other means on a DECstation 5000/200. Our 3mint design buffers camera output and DMAs it directly into system memory. In fact the Xilinx 3000 series programmable gate arrays are rather poor at implementing memory so we are only able to fit 12 words of buffering in the design. Data arrives from the camera and is reformatted at the rate of one word ever

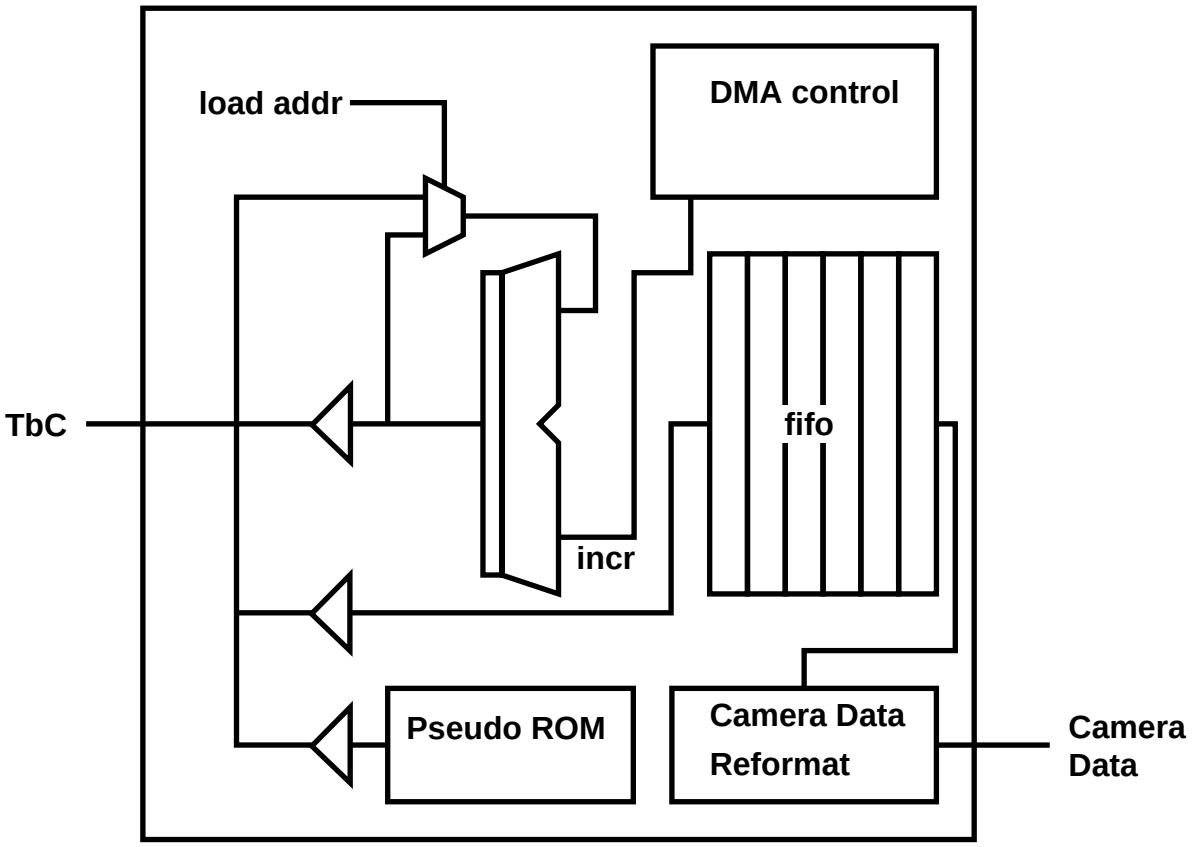


Figure 14: Videk Camera Interface

ten cycles. As a consequence, in order not to lose data, a DMA request must be granted once every 120 cycles. When this is not the case, the design detects an overrun and inserts zero valued pixels for the lost data. Usually the images are transferred reliably, but occasionally the transfer suffers a burst of many overruns. To understand these overruns we investigate the latencies in the granting of DMAs.

Reverting to our original DMA design we DMA the value of a free running counter into system memory and record the time differences between the end and start of successive blocks in a histogram. We run the test on systems under varying workloads. Generally DMAs are granted after a very short delay. Occasionally the request experiences a delay of up to 200 cycles. Of all workloads measured, heavy ethernet traffic accentuates these long delays the most. Figure 15 shows a histogram of measured DMA latency comparing an idle system with one experiencing heavy ethernet traffic. There is a significant peak at 195 cycles. These delays are caused when the DECstation 5000/200 ethernet subsystem blocks a CPU access while it is accessing a shared buffer. They are eliminated entirely when the ethernet is disconnected.

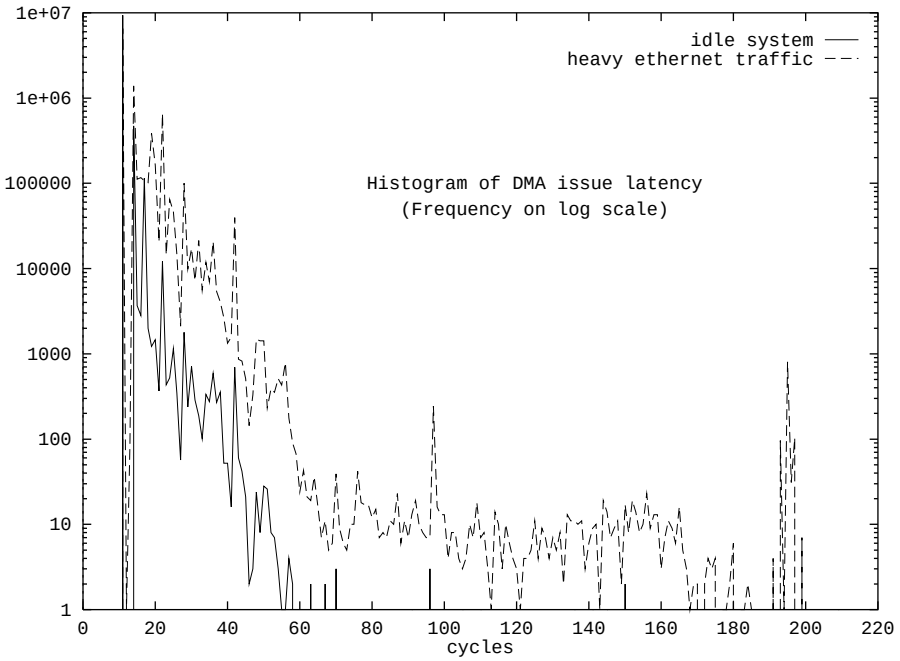


Figure 15: Histogram of DMA Latency under Different Workloads

Unfortunately on 3mint we are just a little short of the necessary storage to ensure lossless buffering. Nevertheless, 3mint provides a workable interface to the Videk camera—a quite unexpected application of the 3mint board. In Figure 16 we see the 3mint milliseconds before it transferred the pixels of this image into the memory of a DECstation 5000/200.

5 Conclusions

PAMs prove to be remarkably versatile at performing low-level system measurements. These measurements would be very difficult to obtain by other means. They give unexpected insights into the foundations of a computer system and the lowest levels of the operating system.

In practical terms we have new data on interrupt handling under Unix and guidance in developing a peripheral that will tax the limits of the DECstation 5000/200's memory system. And surprisingly we have a camera interface too.

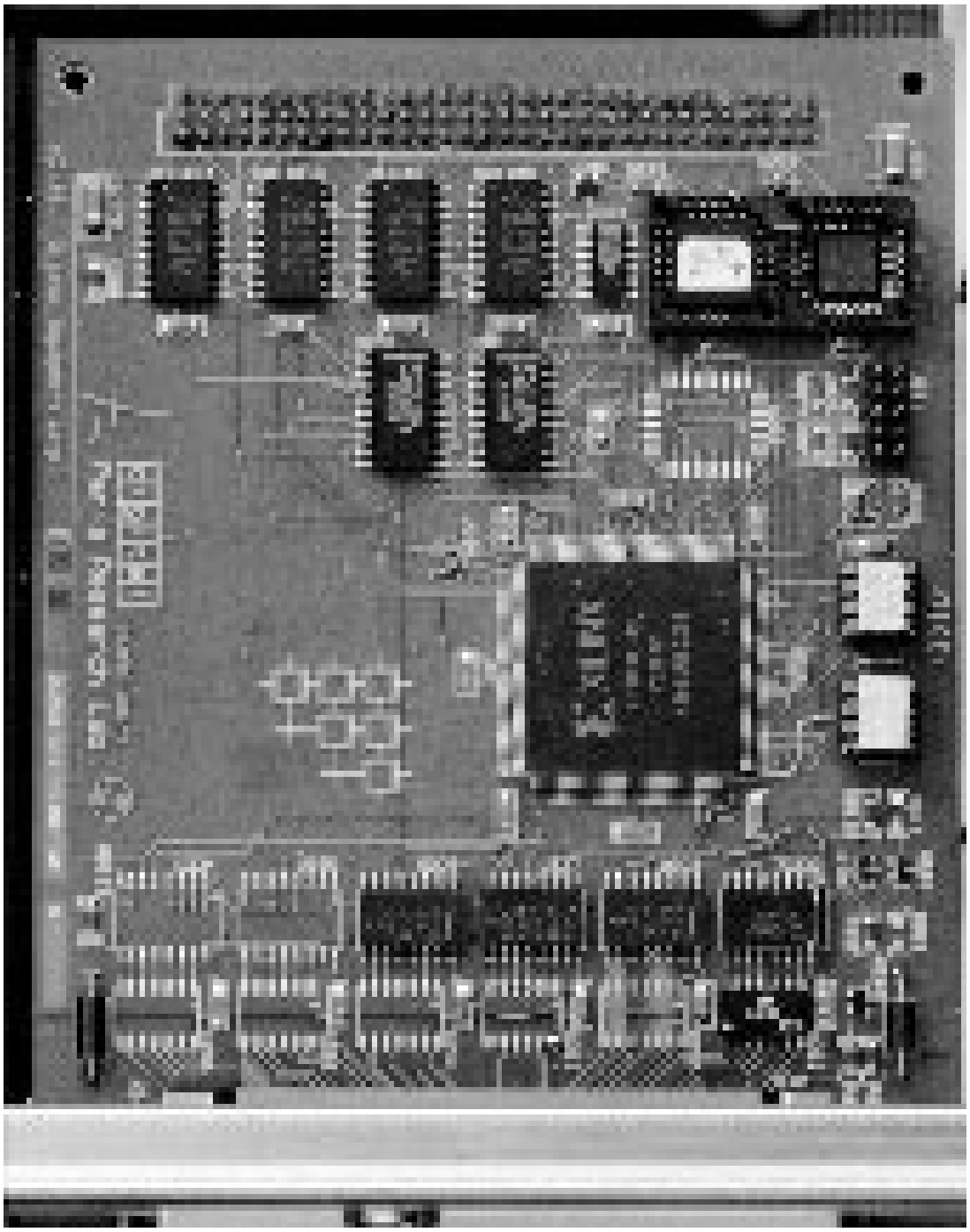


Figure 16: 3mint Taking its own Photo

6 References

1. P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. *Systolic Array Processors*. J. McCanny, J. McWhirter, and E. Swartzlander Jr., editors, pages 301-309, Prentice Hall (1989). Also available as *PRL Research Report 3*, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France.

2. P. Bertin, D. Roncin, and J. Vuillemin. Programmable Active Memories: A Performance Assessment. In *FPGA'92, Proc. of the 1st ACM/SIGDA Workshop on Field Programmable Gate Arrays,* Berkeley, California, February 1992.

3. A. Borg, R.E. Kessler, and D.W. Wall. Generation and Analysis of Very Long Address Traces. In *Proc. of 17th Annual Symposium on Computer Architecture*, Seattle, Washington, pages 270-279 (1990).

4. D.W. Clark, P.J. Bannon, and J.B. Keller. Measuring VAX8800 Performance with a Histogram Hardware Monitor, In *Proc. of 15th Annual Intl. Symposium on Computer Architecture*, Honolulu, May 1988, pages 176-185.

5. Digital Equipment Corporation. *TURBOchannel Hardware Specification*, EK-369AA-OD-007, Digital Equipment Corporation, Maynard, MA (April 1991).

6. Digital Equipment Corporation. *Alpha Architecture Handbook* (February 1992).

7. J.S. Emer and D.W. Clark. A Characterization of Processor Performance in the VAX-11/780, *Proc. of 11th Annual Intl. Symposium on Computer Architecture*, Ann Arbor, MI, May 1984, pages 176-185.

8. A. Goldberg and J. Hennessy. *MTOOL: A Method For Detecting Memory Bottlenecks*, WRL Technical Note TN-17, Digital Western Research Laboratory, Palo Alto, CA (1991).

9. MIPS Computer Systems. *Language Programmer's Guide* (1986).

10. M. Shand, P. Bertin, and J. Vuillemin. Hardware Speedups in Long Integer Multiplication, In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 138-145, ACM Press (1990).

11. Videk. *VIDEK Megaplus Camera Operator Manual*, Publication No. K00786, Videk, A Kodak Company (February 1990).

12. J. Vuillemin. Constant Time Arbitrary Length Synchronous Binary Counters, *10th IEEE Symposium on Computer Arithmetic*, Grenoble, France, June 1991.

13. D.W. Wall. *Systems for Late Code Modification*, WRL Technical Note TN-19, Digital Western Research Laboratory, Palo Alto, CA (1991).

14. Xilinx. *The Programmable Gate Array Data Book*, Product Briefs, Xilinx Inc. (1989).

PRL Research Reports

The following documents may be ordered by regular mail from:

Librarian – Research Reports
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France.

It is also possible to obtain them by electronic mail. For more information, send a message whose subject line is `help` to `doc-server@prl.dec.com` or, from within Digital, to `decprl::doc-server`.

Research Report 1: *Incremental Computation of Planar Maps*. Michel Gangnet, Jean-Claude Hervé, Thierry Pudet, and Jean-Manuel Van Thong. May 1989.

Research Report 2: *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic*. Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé. May 1989.

Research Report 3: *Introduction to Programmable Active Memories*. Patrice Bertin, Didier Roncin, and Jean Vuillemin. June 1989.

Research Report 4: *Compiling Pattern Matching by Term Decomposition*. Laurence Puel and Ascánder Suárez. January 1990.

Research Report 5: *The WAM: A (Real) Tutorial*. Hassan Aït-Kaci. January 1990.

Research Report 6: *Binary Periodic Synchronizing Sequences*. Marcin Skubiszewski. May 1991.

Research Report 7: *The Siphon: Managing Distant Replicated Repositories*. Francis J. Prusker and Edward P. Wobber. May 1991.

Research Report 8: *Constructive Logics. Part I: A Tutorial on Proof Systems and Typed λ-Calculi*. Jean Gallier. May 1991.

Research Report 9: *Constructive Logics. Part II: Linear Logic and Proof Nets*. Jean Gallier. May 1991.

Research Report 10: *Pattern Matching in Order-Sorted Languages*. Delia Kesner. May 1991.

Research Report 11: *Towards a Meaning of LIFE*. Hassan Aït-Kaci and Andreas Podelski. June 1991.

Research Report 12: *Residuation and Guarded Rules for Constraint Logic Programming*. Gert Smolka. June 1991.

Research Report 13: *Functions as Passive Constraints in LIFE*. Hassan Aït-Kaci and Andreas Podelski. June 1991.

Research Report 14: *Automatic Motion Planning for Complex Articulated Bodies*. Jérôme Barraquand. June 1991.

Research Report 15: *A Hardware Implementation of Pure Esterel*. Gérard Berry. July 1991.

Research Report 16: *Contribution à la Résolution Numérique des Équations de Laplace et de la Chaleur*. Jean Vuillemin. February 1992.

Research Report 17: *Inferring Graphical Constraints with Rockit*. Solange Karsenty, James A. Landay, and Chris Weikart. March 1992.

Research Report 18: *Abstract Interpretation by Dynamic Partitioning*. François Bourdoncle. March 1992.

Research Report 19: *Measuring System Performance with Reprogrammable Hardware*. Mark Shand. August 1992.

19

Measuring System Performance with Reprogrammable Hardware

Mark Shand

digital

PARIS RESEARCH LABORATORY

85, Avenue Victor Hugo
92563 RUEIL MALMAISON CEDEX
FRANCE