A Myrinet firmware development experience

Marc Herbert

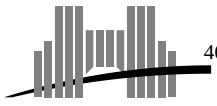
Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

A Myrinet Firmware development experience

Marc Herbert, RESAM / INRIA RESO March 2002

Technical Report N° TR2002-01



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

A Myrinet Firmware development experience

Marc Herbert, RESAM / INRIA RESO

March 2002

Abstract

This report describes the *technical* aspects of the developpement of a Myrinet LANai code on a Linux platform. It contains information and references that can be useful to people wishing to develop in a similar context, whether from the Myrinet side (embedded code) or from the Linux side (driver).

Keywords: Myrinet, LANai, Linux, driver, programming

Résumé

Ce rapport décrit les problèmes techniques liés à la programmation d'un code pour processeur LANai, embarqué sur carte de communication Myrinet. Il contient informations et références pertinentes d'un point vue LANai mais aussi concernant le développement de pilotes de matériel pour Linux.

Mots-clés: Myrinet, LANai, Linux, pilote, programmation

1 Introduction

This work took place as part of a study [Her01], whose goal was to sketch a distributed and decentralized switching architecture. The Myricom LANai processor was chosen as the processing leaf node in this "dumb-core" high speed switching architecture experiment. A partial goal for the thesis was to implement and benchmark embedded LANai programs, refered to in the following by the acronym MCP, short for Myrinet Control Program. This report describes the documentations, references and methods used for this development.

Myrinet hardware was picked out because it fitted well in the envisioned architecture. Myrinet is composed of an over-simplified, dumb and performant core, with potential "intelligence" at the edges of the network thanks to programmable processors. Moreover, the openness of the technology and the availability of numerous documentations, research papers and free software made this study possible.

Due to time and cost constraints, only common ("off the shelf") hardware parts were used: *i.e.* Myrinet PCI interface cards with their on-board LANai processor. The hosts for these cards were simple IBM-compatible PCs running the Linux 2.2 kernel (Debian operating system). The hosts acted as a necessary power supply for the LANai, and as a developpement and control platform thanks to and through the PCI bus.

The section 2 gives the project background. It introduces Myricom's hardware, documentations available and existing software projects. The section 3 presents the developed software and the design issues.

2 Context

At the very beginning of the Myrinet technology marketing, the software offer was naturally poor. But thanks to a very open policy from Myricom [Myr] and the immediate availability of a C compiler and a primitive myriAPI library, many research projects took advantage of the programmable feature of the network interfaces to experiment miscellaneous ideas about high-performance computing. The software and documentation offer grew up very quickly.

Today Myricom has achieved a full-featured, versatile, efficient and GPL'ed [GPL] low communication layer (MCP + drivers + communication library) called GM, as well as a MPI layer above GM. They have given up since 1998 their original myriAPI. They still actively maintain and distribute the LANai-gcc backend (version 2.95.2). The research activity around Myrinet communication layers has levelled off: aside GM, only a few projects are still maintained, generally because they are still used as a foundation for higher level communication software. The "customers project page" on Myricom's web site [Myr] gives an exhaustive list of pointers.

2.1 Myricom hardware

Myricom has a complete Myrinet offer. At the heart of their products is their edge-network processor, *i.e.* their successive LANai generations, that they sell as an OEM [1] to some high-performance computer vendors. Their main retail product is their PCI boards, now 64 bits/66 MHz capable. They also propose a smaller PMC (PCI Mezzanine Card) variation.

[1] Original Equipment Manufacturer

A wide range of wormhole switches are proposed, all built by interconnecting several of their basic XBar16 crossbar chip (bissection 35.2 Gb/s). Thanks to extremely low switch latency, Myrinet network diameter is only limited by financial considerations.

2.1.1 Notable steps in LANai evolution

All known LANai processors (until 9.x) have a 32 bits internal architecture.

Version	Important innovations (incremental list)
5.0	64 bits-*datapaths*. CRC32.
7.0	Supression of the dual-context capability[a]
7.2	66 MHz internal clock[b]
9.0	Increased clock rates, up to 200 MHz

[a] The Interrupt Status Register is renamed Interface Status Register.

[b] Previous versions were clocked at *once* thestandard bus frequency (33 MHz for PCI, up to 40 MHz with faster buses)

2.1.2 PCI boards

Retail PCI boards featured only LANai processors 3.x, 4.x, 7.x and now 9.x. Intermediate LANai versions were either only sold on the OEM market or just internal prototypes.

PCI revision	Introd. date	LANai	New features
1	?	3.x, 4.x	
2 (PCI64)	April 1999	7.0, 7.1	64 bits datapaths. PCI DMA linked lists.
3 (PCI64**A**)	Sept. 1999	7.2, 7.3	66 MHz PCI support, clock multiplier.
(PCI64**B, C**)	May 2000	9.x	

Every board is compatible with every kind of PCI bus. Thanks to the clock multiplier, the LANai is able to run at its max speed whatever the characteristics of the PCI bus. A, B and C boards feature processors running at respectively 66, 133 and 200 MHz maximum speed.

2.2 Documentation

2.2.1 Myrinet

The documentation freely available from Myricom is much more plentiful than from other similar hardware companies. All Myricom documents can be found on their web site [Myr].

A complete documentation of every LANai processor is available, detailed from the raw hardware (pinout and timings) to the *softer*-ware (interface registers semantics). A separate document presents the LANai 3.x instruction set, which has not changed much until and including LANai 9.0. Reading LANai machine code is rather easy since it is a real RISC architecture. Newer instruction sets are not and will not be documented ("*We* [Myricom] *strongly discourage people from writing their own MCPs since you need* LANai*, host-driver and host-user code. It is a non-trivial amount of work.*") The instruction set is chosen at compilation time thanks to the classical '`-m`' gcc option: `-m4.1`, `-m7.0` or `-m9.0`.

The LANai-gcc suite should be sufficient to satisfy every development need, even if it is no more documented than standard gcc.

An even-less documented '`-many`' option of the compiler may generate a verbose and cross-generation compatible code. In fact, giving no `-mXXX` option at all produces the same binary code. This kind of polyvalent code has not been tested, mainly because I have reused some GM header files that define generation-specific constants and that definitely do not allow this option. Morevover, some quick and dirty tests with the `-many` option succeeded in crashing the compiler ("*Internal compiler error in* '`output_store`',..."), so this looked like *not* the way to go.

The *processor* documentation is not sufficient to run the PCI *boards*. The documentation for the revision 2 and 3 PCI boards (PCI64) is also available. This PCI documentation is fairly complete, only some details are missing or confusing. For instance explanations about the hardware link speed (1280/2000 Mb/s) microswitch are missing, you have to explore some FAQ. It seems the software links speed setting is recommended and prefered. The code for this setting may be extracted from GM source.

There is no documentation for the older revision 1/PCI 32 boards. Their (simpler) driving and their hardware constants have to be digged out from sample code source.

Some FAQs, tutorials and old *MyriAPI* documentations can also be found on the Myricom web site, but are outdated: they do not consider LANai generations above 4.1. A tutorial based on this old API was published by the High Performance Computing Laboratory of the Mississippi State University [SHDM] some time ago. Some basic concepts exposed there may still be interesting to read.

For GM users, *i.e.* people *not* wanting to hack into Myrinet hardware, an extensive and up-to-date documentation is freely downloadable. Reactive support from Myricom is available through a mailing list.

2.2.2 PCI

A basic understanding of the PCI bus is necessary. This can be achieved for instance thanks to the following references [Tec] [RC01] or to any PCI chipset documentation.

2.2.3 Linux kernel

Debian/GNU Linux was the system running the hosts. It was taken among other open systems mainly for familiarity reasons, and for the assumption that a lot of documentation was available. This choice is retrospectively questionable. As a Linux application programmer or system administrator, the volume of documentation is indeed impressive. But when getting closer to the hardware, life is much less easy, due to the very quick evolution of the kernel code and structures; every documentation effort is doomed to fast obsolescence. There is a productive kernel and drivers documentation effort, but very scattered and often subtly outdated and inaccurate. It is rather easy to find inconsistent statements among documents. Often, the only really reliable reference is the source code[2]. Using a "company-supported" and professionaly documented open system (e.g., Solaris [SUN]) would probably have solved this documentation problem.

Probably the best structuration effort for Linux documentation is the Linux Documentation Project [LDP], but kernel programming is only a small part of this effort. The more important document is the Linux Kernel Mailing List FAQ [LKM], which among simple answers includes pointers to many (too many ?) pertinent documents and

[2]The Linux source includes a low volume but not to under-estimate "Documentation" directory

to list archive search engines. Another interesting document, and up-to-date at time of writing, is the list of pointers collected by Juan-Mariano de Goyeneche [dG]. The best book available on the subject of drivers is the one from Alessandro Rubini [RC01]. Unfortunately, the updated 2nd edition was published just after this work.

Other drawbacks of Linux were the monolithical approach (micro-kernel architectures are known for natively providing a fine granularity access control to hardware, see issues in section 3.2), and the lack of real-time capabilities, which could have been useful for control and monitoring tasks (see section 3.4).

On the other hand, the setup and administration of a development Linux system is probably among the easiest, thanks to the tight integration with standard tools. Most Linux distributions (in particular Debian) have a lead in this domain on all other systems, and this is really time spare.

2.3 Software resources

In order to avoid re-inventing the wheel, our code was designed with a constant eye on already available Myrinet open-source codes and a pair of scissors. Two low level message passing libraries, GM and BIP, were used as samples. Another one (PM/SCore [SCo]) was considered, but as the start-up cost of understanding a code is quite high, we did not think the information brought by this third code was worth the investment.

2.3.1 GM

GM is not a research project, it is designed with an "industrial" perspective. No research paper describing its architecture is available. On the other hand, its code has most of the characteristics of a professional project which renders it rather easy to approach for a C project of this size[3]. It also has many users, is continuously maintained and has very few bugs.

The big volume of GM is partially due to a very heavy and disseminated error-handling and debugging part, probably necessary for unsavvy customer support. Another reason is the great number of architectures it supports, and the multiple abstraction layers it is made of. The pecularities of the outdated C language increase the verbosity of this part and do some harm to readability. Anyway, a complete overview of GM design choices was not needed for this study. For instance, understanding the complex event-driven MCP automaton would have been a waste of time, since our goal was to evaluate our own.

Last but not least, GM is naturally developped in close collaboration with Myricom hardware developpers. In addition to a pool of hardware constants and small subroutines, GM has been in used in this project as the reference to solve any documentation hole or inaccuracy that has arisen during developpment.

2.3.2 BIP

BIP [PT98] was the first communication library bringing the full wire performance to the user application. As it has less high-level features and supports less host architectures than GM, its volume is much lower[4] and so it seems at first much quicker to grasp. But its highly tuned approach and hacking style counter-balances this first impression. Moreover, it is much less widely used than GM, less steadily maintained, and lacks

[3] about 260,000 lines
[4] 25,000 lines of C

a reliable support of the latest LANai generation (v9). One of the big advantages of studying BIP was a direct access to some of its (kindly available) main developpers.

2.3.3 Why not writing a patch ?

An alternative to the developpement of an isolated and autonomous Myrinet software would have been to patch an existing communication library until it meets our needs. This alternative has not been followed for the following reasons.

Very different goal Our aim was to evaluate the capabilities of a "network-edge" processor, and to find how to control and evaluate this processor in a non-intrusive way. In comparison, all available Myrinet software is designed to use the Myrinet *boards* exactly for what they are: *i.e. network interface cards*, as transparent as possible to the communication library that use them.

Big project management overhead Except maybe in a perfectly modular designed software architecture, it is always risky to make any change to the fragile balance of a big software project. Unless a very good global overview of the project is acquired (which is already time-consuming), any change may often have subtle side-effects that are very hard to understand, the simplest example being to trigger hidden bugs.

Performance issues The evaluation of embedded software costs implies no interferences, in particular from the hypothetical "guest" code. It may be hard to check that every unwanted feature has been disabled.

Re-usability The code developped to make our experiments is only a few thousands lines long and easy to read, and so it can be really quickly grasped by any person wishing to start a similar project or simply understand how to drive Myrinet boards.

3 Developed software components

3.1 Hardware used

The hardware and software used for this experience is presented on figure 1. The goal is to emulate a standard Ethernet switching equipment with a Myrinet network and LANai processors at the edge (*i.e.* on Ethernet ports) of this pseudo-equipment. LANai role is to handle the Ethernet into Myrinet encapsulation. In our prototype, the Ethernet ports are simulated by the PCI bus. The machines used featured old Pentium Pro 200 MHz processors, and the famous Intel 440BX PCI chipset, still quite performant by today's standards. Different LANai generations (4.1, 7.2, 9.0) have been tested.

3.2 PCI board Linux driver

In order to control myrinet boards (initialization, code upload, debugging, monitoring and measures) some software must be running on the hosts: we need an operating system. The Debian Linux (kernel 2.2) was taken (see why in section 2.2.3).

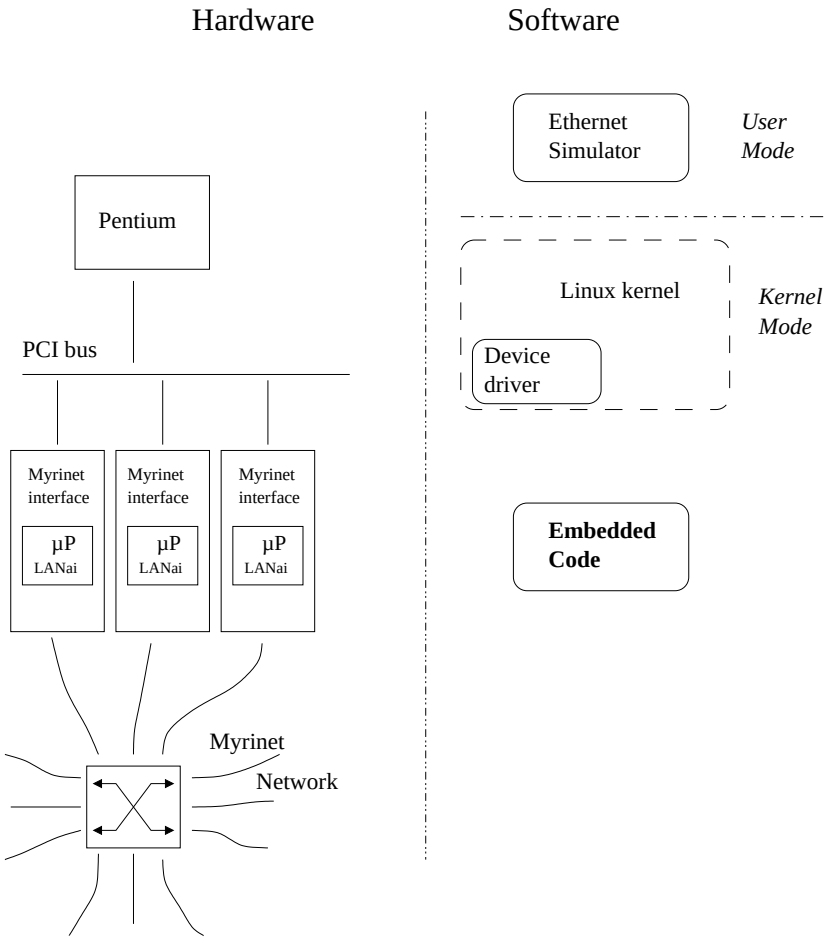


Figure 1: General architecture

Issues The main role of an operating system is to isolate users from the hardware, and thus provide security. This is especially useful during classical development tasks, when incomplete and buggy software is tested, since crashes may be contained in the process' sandbox. Unfortunately, we want here to access the Myrinet hardware through the PCI bus, and this normaly restricted to programs running in unprotected kernel mode, typically hardware drivers. Programming in kernel mode is really painful compared to user mode since a complete machine reboot is very long, and repetitive machine crashes may even corrupt the system stored on the hard drive.

Solution I've implemented a minimal Linux driver, which role is to detect and initialize the boards and to provide a memory mapping of the boards for normal user programs. This driver abstracts differences between boards, and exports to user mode four pseudo-files corresponding to the four memory zones of the board. Since the driver restrict hardware access to PCI adresses specific to a Myrinet board zone, the "damage zone" in user mode is extended to some sections of the Myrinet board. Naturally, it is easy to imagine a board crash that drags the whole host down thanks to the PCI bus, but this is extremely unlikely to happen since the LANai code needs an explicit DMA engine programmation to access the PCI bus. My experience is that a few machine crashes occurred during driver development, but none at all once the rest of the development took place in user mode. Moreover, this approach has allowed to test and debug some routines in user mode before easily transfering fixed code to the LANai, taking advantage of facilities like debugger and librairies. This is also useful from the evaluation point of view: this code transfer helps breaking down costs without loss of functionality. For instance the Ethernet address learning algorithm source code is 90% source compatible between the Pentium and the LANai versions, thanks to a few pre-processing instructions.

Implementation details The driver is naturally implemented as a module dynamically linked and unlinked with a running kernel, since this allows very fast testing: no need to reboot to test a new driver (unless a fatal error freezes the kernel). At loading time, the driver explores the PCI space, locating all Myrinet boards, and stores basic parameters for each one in its internal devices table. It also runs the revision-specific initialization procedure for each one, possibly setting the clock multiplier. PCI 64 revisions (2 and above) have a more complex configuration and initialization sequence. For instance, they have three different reset triggers (instead of only one for previous boards): one for the board, one for the PCI interface, and finally the LANai processor one. After the initialization by the driver, only the processor is maintained in "frozen" reset state, waiting for the user to upload his code. Nothing is done at driver unloading, leaving the user the freedom to let the LANai still running or not.

The `mmap(2)` system call is implemented with 4 different devices per physical Myrinet board, one for each board memory zone (ROM, RAM, board registers and processor registers). This provides abstraction and finer access granularity (and so better security) in user-mode.

3.3 MCP

Programming the LANai and the on-board SRAM is similar to classical standard C programming, but stripped to a bare minimum. No librairies are available, so operations on floating point numbers, strings, I/O and memory allocation are absent. Neither a

debugger nor memory protection nor preemptive multitasking are available. We will see in the next section some solutions to partially overcome these development difficulties thanks to host control. In addition to standard C programming, the LANai-gcc suite provides transparent access to LANai and board registers in a cross-generation compatible way[5], and thus provides programmable access to both external (PCI) and internal (Myri network) interfaces. Contrary to the host side, the access to the PCI bus is not mapped to processor adresses: every access must be explicitly programmed into the on-board DMA engine. This PCI space access from the board was not needed here.

The goal of this experience was to implement and evaluate a distributed Ethernet switch prototype [Her01]. The LANai processors role is to encapsulate incoming Ethernet frames into Myrinet worms, send ("switch") them to the right LANai "exit", where this destination LANai will decapsulate the Ethernet frame and forwards it to destination Ethernet segment. In order to break down the cost of this "switching" process between its different features, a simple conditional compilation technique was used at first. Then the synchronization procedure between the board and the host (described in the next section) was enriched to pass command line arguments to the embedded code. The inclusion of some code sections has to stay under preprocessing control for performance reasons.

The first and very basic feature implemented, after the link initialization, consists in bare send/receive operations. Then two independent buffer rings are defined in the board SRAM, with two on-board indexes per ring to handle queueing: one index is updated by the host control and the other by the LANai. First the LANai was evaluated running only one of the two functions at a time, then a simple send/receive automaton was tested, and remaining features (CRC check, variable frame length, address learning,...) have also been incrementally added.

In order to use 32 bits and 64 bits boards together on the same network, LANai 4 worm lengths are forced to 8 octets multiples, and the advanced CRC32 feature is unused.

3.4 Control and monitoring

Functions implemented in user-level take care of uploading LANai code in board memory, launching the desired test, simulating the Ethernet trafic/environment, collect data and compute and display results. In addition, the host debugger may be used to inspect and control LANai execution in a limited way through the PCI bus.

Handshake In the beginning both codes initialize separately then a handshake mechanism, very close to the GM one, is used to establish the communication channel (*i.e.* the shared structure), synchronize and pass parameters to the board before starting the test.

First the host polls on a well known "bootstrap" LANai register, which is garanteed to be initialized to zero. The LANai writes in this visible register the pointer to the shared structure located in its memory. Then it waits on another mailbox located in this shared structure. The host grabs this pointer and so can write in the just-located shared structure to transmit parameters to the board. Finally, it writes in the mailbox to signal the board it is ready. Then the board may possibly check the network connection, and if all clear, send a third and final handshake message to the host. The test starts.

[5] except of course for new registers

Traffic simulation As every type of host-LANai communication, the simulation of the incoming and outcoming Ethernet traffic goes through the host PCI bus. The host polls on the "new frames arrived" index, picking them up, and updates the "frames picked" index. And the same thing but reversed for the other independent outgoing frames ring.

The issue here, is that the PCI bus is performance limited compared to the Myrinet link. Wire PCI speed starts at 1 Gb/s (at 32 bits/33 MHz) *half-duplex*, whereas Myrinet link starts at 1 Gb/s full duplex. The main difference is that the PCI bus is shared and undeterministic, with an important arbitration cost leading to poor performance for small size transactions. The first solution to reduce the needed throughput is simply to read and write only frames *headers*[6], since we do not care about the payload content. Another is to optimize index updates by sending and receiving frames headers by bursts: this is a typical throughput/latency trade-off. But the performance on the bus still had to be evaluated to check that the host control was tight enough. The main worry comes from the operating system: the price to pay for the previously praised development facilities is that our control code gets no guarantee on the latency of its PCI accesses. In order to check this, I measured the fillness (or emptyness) of send (or receive) rings. Only with very long bursts could the send ring become empty, even in tests with very short payload frames and minimal LANai tasks. This is due to the fact that even latest Myrinet boards can not handle more than 6 millions of frames per second. So the PCI bus was proved performant enough to handle this kind of control/simulation task.

Another issue is the limited bandwith of the SRAM of old boards. On revision 1 boards, it was possible to stall the processor thanks to intensive other concurrent memory accesses. But this time, the high arbitration/transaction cost of the PCI bus plays with us, since we only transfer small chunks on it. By artificially pushing the polling from the host to the board to a maximum, I checked that the polling still not slowed down the LANai.

A possibility to optimize PCI performance could have been to exploit the complex cache memory possibilities of the Intel P6 architecture [Int01] to carefully use the *write-combining* optimization. This has not been tested. Another one would have been to use interrupts, or the PCI DMA capability of the board to make the user code poll on its local system RAM instead of the PCI bus space. Interrupts have a big cost and latency. Their interest relies in alleviating the main processor. As our main processor is dedicated to run the test, we would have seen only the drawbacks. The DMA possibility is more interesting, even if it would have added a parasite cost on the evaluated MCP. But as the simple PCI polling was proved successful, this was not investigated.

Evaluation Measuring software micro-costs can be tricky. The Myrinet on-board clock could have provided a solution. But this would have required an additional LANai cost disturbing the measure. Moreover, these clocks have been seen diverging up to 20% from their nominal frequency on old boards, so it is difficult to put trust on them. The only alternative is to time from the host, but the PCI latency prevents any micro-timing. The final solution is to time repetitive micro benchmarks. A variance inferior to $1/1000$ has been achieved this way.

Debugging Since the board memory is seen from the host, it is possible to do a sort of primitive "remote-debugging" by reading and writing distant memory locations while using a debugger on the host. For an unknown reason, it was not possible to use

[6] three 32 bits words at most

direct read/write `gdb` commands. The workaround is to compile simple read and write functions in the user code and to call them from the debugger.

Conditional compilation allows to get, in a manual but yet simple way, a pretty good quantity of debugging information without performance consequences. Possible MCP debugging techniques are — *tracing*, by asking the MCP to frequently write line numbers reached, or any useful information in dedicated memory locations; — *assertions*, where the MCP enters a infinite loop after writing down the crash cause. *Stepping* is also possible, but much less practical since breakpoints, where the MCP polls a pre-defined restart flag, must be recompiled. Myricom uses internally more advanced debugging techniques, but since LANai programmation is discouraged, it is difficult to know if it involves ad hoc hardware of only undocumented "software" LANai features.

References

[dG] Juan-Mariano de Goyeneche. Kernel links. `http://jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html`.

[GPL] GPL: GNU general public license. `http://www.gnu.org/philosophy/license-list.html`.

[Her01] Marc Herbert. Vers une architecture de commutation distribuée. Master's thesis, École Normale Supérieure de Lyon, Lyon, France, July 2001.

[Int01] Intel, editor. *IA-32 Intel Architecture Software developer's manual*, volume 3. 2001. Available at "Intel's Literature Center", `http://developer.intel.com/`.

[LDP] LDP: Linux documentation project. `http://www.linuxdoc.org/mirrors.html`.

[LKM] Linux kernel mailing list FAQ. `http://www.tux.org/lkml/`.

[Myr] Myricom web site. `http://www.myri.com/`.

[PT98] L. Prylli and B. Tourancheau. BIP: A new Protocol Designed for High Performance Networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, volume 1388 of *Lect. Notes in Comp. Science*, pages 472–485, Springer-Verlag, April 1998.

[RC01] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, 2nd edition, June 2001. Fully available on line `http://www.oreilly.com/catalog/linuxdrive2/`.

[SCo] SCore web site. `http://pdswww.rwcp.or.jp/`.

[SHDM] Anthony Skjellum, Gregory Henley, Nathan Doss, and Thomas McMahon. A guide to writing myrinet control programs for LANai 3.x. Available at `http://www.erc.msstate.edu/research/labs/hpcl/myrimpi/learn_mcp/`.

[SUN] SUN product documentation. `http://docs.sun.com/`.

[Tec] TechFest PCI local bus technical summary. `http://www.techfest.com/`.